A FORMALISATION OF DESIGN METHODS

A $\lambda$-calculus Approach to System Design with an Application to Text Editing

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. ir. M. Tels, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op vrijdag 23 februari 1990 te 16.00 uur

door

LAURENTIUS MICHIJM GERARDUS FEIJS

geboren te Sittard
Dit proefschrift is goedgekeurd
door de promotoren

prof. dr. F.E.J. Kruseman Arets
en
prof. dr. J.A. Bergstra

The work described in this thesis has been carried out at the Philips Research Laboratories, Eindhoven as part of the Philips Research programme.
This work has been done in the context of ESPRIT project 432 (Meteor).
Aan mijn ouders
Contents

1 Overview and introduction .................................. 1
  1.1 Structure of this monograph .......................... 1
  1.2 The nature of software .............................. 2
    1.2.1 Software engineering .......................... 3
    1.2.2 Formality ...................................... 5
    1.2.3 The role of languages .......................... 9
    1.2.4 The role of tools .............................. 12
    1.2.5 Application domain evolution ................. 13
  1.3 Informal models of software development ............ 14
    1.3.1 General ....................................... 14
    1.3.2 Waterfall model ................................ 15
    1.3.3 Boehm's spiral model .......................... 15
    1.3.4 A layered behavioural model ................. 16
    1.3.5 Koenen's iteration, learning and detailing model .... 17
  1.4 Towards formal models of software development .... 18
    1.4.1 Modularisation ................................ 18
    1.4.2 Parameterisation ................................ 20
    1.4.3 Simple top-down and bottom-up models ........ 21
  1.5 A formalisation of design structures ................. 23
  1.6 Correctness-preserving transformations of designs .. 27
  1.7 Formal specification techniques ..................... 30
    1.7.1 General ....................................... 30
2 A Formalisation of Design Structures

2.1 Introduction ........................................ 51
2.2 Algebraic systems with preorder ................. 53
  2.2.1 Motivation .................................... 53
  2.2.2 Formalisation .................................. 55
  2.2.3 Examples ..................................... 58
2.3 Lambda calculus .................................... 59
  2.3.1 Introduction .................................. 59
  2.3.2 Definition of the Calculus ................. 61
  2.3.3 Derived rules ................................. 70
  2.3.4 Monotonicity ................................ 73
  2.3.5 Reduction ................................... 74
  2.3.6 Normalisation ................................ 76
  2.3.7 Confluence .................................. 80
2.4 Components ........................................ 83
  2.4.1 Introduction .................................. 83
  2.4.2 Formal definitions ............................ 84
2.5 Design .............................................. 85
  2.5.1 Introduction .................................. 85
  2.5.2 Syntax and Correctness ...................... 86
  2.5.3 Semantics .................................... 90
  2.5.4 Correctness-preserving Modifications .... 99
2.6 Looking back ....................................... 105
## 3 Correctness-Preserving Transformations of Designs

### 3.1 Introduction

### 3.2 Designs

3.2.1 Designs where the system is a sequence

3.2.2 Permuting Components

3.2.3 The conditions 'pf' and 'ds'

### 3.3 Algebraic Operations on Designs

3.3.1 The operation $\ast$

3.3.2 The operation $\circ$

3.3.3 The operations bot and top

3.3.4 The operation $\downarrow$

3.3.5 Summary

### 3.4 Design Creation

3.4.1 General

3.4.2 Top-down example

3.4.3 Design-programs

3.4.4 Top-down development

3.4.5 Bottom-up development

### 3.5 Design Evolution

3.5.1 General

3.5.2 Changing machine

3.5.3 Changing system user

### 3.6 Design Partition

3.6.1 General

3.6.2 Splitting into components

3.6.3 Splitting according to $\ast$
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6.4</td>
<td>Splitting according to $\circ$</td>
<td>188</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Splitting according to $|$</td>
<td>190</td>
</tr>
<tr>
<td>3.7</td>
<td>Looking Back</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td><strong>Bibliography</strong></td>
<td>194</td>
</tr>
<tr>
<td>A</td>
<td>$\lambda$-Calculus with sequences</td>
<td>195</td>
</tr>
<tr>
<td>B</td>
<td>Design-development language</td>
<td>202</td>
</tr>
<tr>
<td>C</td>
<td>List of symbols</td>
<td>209</td>
</tr>
<tr>
<td>4</td>
<td><strong>Formal Specification of a Text Editor</strong></td>
<td>215</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>215</td>
</tr>
<tr>
<td>4.2</td>
<td>Overview of the Formal Specification</td>
<td>217</td>
</tr>
<tr>
<td>4.3</td>
<td>Text and Algebraic Operations On Text</td>
<td>220</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Introduction</td>
<td>220</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Texts</td>
<td>222</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Strings</td>
<td>225</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Relating Texts and Strings</td>
<td>226</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Printability</td>
<td>227</td>
</tr>
<tr>
<td>4.3.6</td>
<td>Natural Operations on Text</td>
<td>229</td>
</tr>
<tr>
<td>4.3.7</td>
<td>The Reach of a Text</td>
<td>233</td>
</tr>
<tr>
<td>4.3.8</td>
<td>The Profile of a Text</td>
<td>237</td>
</tr>
<tr>
<td>4.3.9</td>
<td>Vertical and Horizontal Composition of Text</td>
<td>240</td>
</tr>
<tr>
<td>4.3.10</td>
<td>Operations for Searching</td>
<td>242</td>
</tr>
<tr>
<td>4.3.11</td>
<td>Procrustean Operations</td>
<td>244</td>
</tr>
<tr>
<td>4.4</td>
<td>Interfacing an Editor with its Environment</td>
<td>250</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Introduction</td>
<td>250</td>
</tr>
<tr>
<td>4.4.2</td>
<td>DISPLAY: an Abstract Display</td>
<td>250</td>
</tr>
<tr>
<td>4.4.3</td>
<td>'SEQ' and 'STRING': Data types for Interfacing</td>
<td>257</td>
</tr>
<tr>
<td>4.4.4</td>
<td>FILE: a File system</td>
<td>263</td>
</tr>
<tr>
<td>4.5</td>
<td>Text Editing</td>
<td>266</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Introduction</td>
<td>266</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Marked Texts</td>
<td>267</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Stating the Application Domain</td>
<td>269</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Spanning the State-space</td>
<td>270</td>
</tr>
<tr>
<td>4.5.5</td>
<td>The Text Invariant</td>
<td>272</td>
</tr>
<tr>
<td>4.5.6</td>
<td>Operations for Dot and Mark control</td>
<td>276</td>
</tr>
<tr>
<td>4.5.7</td>
<td>Operations for Text Modification</td>
<td>279</td>
</tr>
<tr>
<td>4.5.8</td>
<td>Operations for Marked-text Management</td>
<td>283</td>
</tr>
<tr>
<td>4.5.9</td>
<td>Operations for Searching</td>
<td>284</td>
</tr>
<tr>
<td>4.5.10</td>
<td>Operations for String Conversion</td>
<td>285</td>
</tr>
<tr>
<td>4.5.11</td>
<td>Termination Axioms</td>
<td>285</td>
</tr>
<tr>
<td>4.5.12</td>
<td>Connecting the Editor with the Display</td>
<td>286</td>
</tr>
<tr>
<td>4.5.13</td>
<td>MOREDOP: More Editing Operations</td>
<td>289</td>
</tr>
<tr>
<td>4.5.14</td>
<td>KEYBIND: Connecting the Editor with the Keyboard</td>
<td>291</td>
</tr>
<tr>
<td>4.6</td>
<td>Related Work</td>
<td>292</td>
</tr>
<tr>
<td>4.7</td>
<td>Looking Back</td>
<td>293</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>297</td>
</tr>
<tr>
<td>A</td>
<td>List of Symbols</td>
<td>299</td>
</tr>
<tr>
<td>B</td>
<td>Standard Class Descriptions</td>
<td>306</td>
</tr>
</tbody>
</table>

5 Systematic Design of a Text Editor 331

5.1 Introduction 331
5.2 The Top of the Editor Design 333
5.3 The Bottom of the Editor Design 335
  5.3.1 Introduction 335
  5.3.2 Specifying Instances 335
  5.3.3 Specifying Attributes 336
  5.3.4 Specifying Tables 336
  5.3.5 Specifying Blocks 343
  5.3.6 The Bottom of the Editor Design 345
5.4 Summary of the Top-down Approach 346
5.5 Implementing the System Components 348
  5.5.1 Introduction 348
5.5.2 Implementing KEYBIND .................. 349
5.5.3 Implementing MOREDOP .................. 351
5.5.4 Transforming the State space ............. 353
5.5.5 Specifying Pair-wise Attributes .......... 355
5.5.6 ATTR2: a Postulated Component .......... 356
5.5.7 Transforming the State space (continued) 357
5.5.8 Specifying Simple Programming Variables 360
5.5.9 SVAR: a Postulated Component .......... 361
5.5.10 'STRING': another Postulated Component 361
5.5.11 Transforming the State space (continued) 361
5.5.12 Specifying a Window-Invariant Package .... 370
5.5.13 WLPACKAGE: a postulated component ...... 373
5.5.14 Programming the Editing Operations ...... 373
5.5.15 Termination ................................ 396
5.6 Implementing the Internal Components ..... 399
5.6.1 Implementing WLPACKAGE ................. 399
5.6.2 Specifying Two-dimensional Arrays .......... 400
5.6.3 ARRAY2: a Postulated Component .......... 402
5.6.4 Implementing WLPACKAGE (continued) ...... 402
5.6.5 Specifying Display Handling ............... 408
5.6.6 DISPLAY.HANDLING: a Postulated Component 410
5.6.7 Implementing WLPACKAGE (continued) ..... 410
5.6.8 Implementing DISPLAY.HANDLING .......... 418
5.6.9 Implementing ATTR2 ....................... 424
5.6.10 Implementing SVAR ....................... 425
5.6.11 Implementing 'STRING' ................... 425
5.6.12 Implementing ARRAY2 ...................... 426
5.6.13 Arriving at an Editor Design .............. 427
5.7 Related work ................................ 428
5.8 Looking Back ................................. 429
5.9 Evaluation ................................. 432

Bibliography ................................ 437

A Lower Design Layer ...................... 438
A.1.1 Introduction ............................ 438
A.1.2 The Top of the Lower Design Layer .... 438
A.1.3 The Bottom of the Lower Design Layer ... 439
A.1.4 Implementing the System Components ... 439
A.1.4.1 Introduction .......................... 439
A.1.4.2 Implementing Sequences ............... 440
A.1.4.3 Implementing Tables .................. 444

B List of Symbols ............................ 452

C The C Program of the Editor ............... 457

D Reference Chart ............................ 466

Curriculum vitae ............................ 467

Samenvatting ................................. 468
Chapter 1

Overview and introduction

1.1 Structure of this monograph

This monograph is concerned with models of software development. The foundations for the models studied are based on a $\lambda$-calculus approach to design structures. It also contains an application of the concepts studied to text editing. This first chapter is an overview chapter.

The main body of the monograph is formed by the Chapters 2, 3, 4 and 5. This body is divided into two parts. The first part consists of the Chapters 2 and 3 which are about research of a fundamental nature, dealing with models of software development based on a $\lambda$-calculus approach to design structures. The second part consists of Chapter 4 and 5, which are about work of an engineering nature, viz. the application of the concepts studied in the first part to text editing. Each chapter is structured as a unit with its own introduction, bibliography and appendices.

Chapter 1 is self-contained and requires no preliminaries. Within Chapter 1 there are, amongst others, sections dedicated to the Chapters 2, 3, 4 and 5 — one section for each chapter. Chapter 1 serves to place the other chapters in a broader context and to make the relation between them explicit.

Chapter 2 is concerned with a formalisation of design structures. It is self-contained and its preliminaries are an elementary knowledge of mathematics and computer science as well as a certain familiarity with lambda calculus.

Chapter 3 is concerned with correctness-preserving transformations of designs. This chapter is the heart of the monograph in the sense that it is devoted entirely to the dynamic aspects of the process of software development. Chapter 3 is a natural continuation of Chapter 2. Results from Chapter 2 are used in Chapter 3, but no additional preliminaries are required.
Chapter 4 is concerned with the formal specification of a text editor. The text editor is introduced by way of illustration of the concepts from Chapter 2 and 3, but at the same time Chapter 4 can be viewed as a case study in the use of formal specification techniques. Chapter 4 is self-contained and its preliminaries are an elementary knowledge of mathematics and computer science as well as a certain familiarity with the COLD-K language.

Chapter 5 is concerned with the systematic design of a text editor and is a natural continuation of Chapter 4. The definitions from Chapter 4 are used in Chapter 5, but no additional preliminaries are required.

The remainder of this first chapter is organised as follows. Section 1.2 is entitled “the nature of software” and it serves to introduce and demarcate the field of software engineering, which is the branch of applied science in which the work presented in this monograph fits. Section 1.3 is about informal models of software development and in Section 1.4 we sketch a line of thought which gradually moves away from the informal models, leading to the formalisation of design structures and the correctness-preserving transformations of designs presented in the subsequent sections.

Section 1.5 and 1.6 refer to Chapter 2 and Chapter 3 respectively. These sections serve to present the main achievements of the corresponding chapters and to relate them to other work.

The results of Chapters 2 and 3 are generic in the sense that they can be instantiated with a particular formalism. In order to apply the results it is necessary to do this and therefore Section 1.7 is about formal specification techniques in general and Section 1.8 is about the particular formalism we have chosen to be used in Chapters 4 and 5.

Sections 1.9 and 1.10 refer to Chapters 4 and 5 respectively. These sections serve to present the main achievements of the corresponding chapters and to relate them to other work. Finally, Section 1.11 discusses a number of options for future work.

1.2 The nature of software

We approach the nature of software from the viewpoint that software construction is an engineering activity (see Section 1.2.1 below) with certain peculiar characteristics, viz. formality, the role of languages, the role of tools, and application domain evolution (Sections 1.2.2 to 1.2.5 below). The most important of these characteristics is formality which acts as a recurring theme in the discussion of the other characteristics.
1.2. THE NATURE OF SOFTWARE

1.2.1 Software engineering

The history of software engineering started in 1944 when Eckert, Mauchly and von Neumann adopted the idea of storing the instructions for a computer in electronic form internally in the memory of the computer [1]. The size of computer memories has grown exponentially since then and as a consequence there are hardly any physical limitations upon the size of the computer programs themselves. Also from a physical viewpoint, computer programs are extremely flexible in the sense that it is easy to copy and to modify them. Probably this situation has led to the adoption of the term 'software' as a synonym for 'computer program'. The first computer programs were just sequences of instructions, but it did not take long until program structuring mechanisms were introduced. As Turing [1] put it:

When we wish to start on a subsidiary operation we need only make a note of where we left off the major operation and then apply the first instruction of the subsidiary. When the subsidiary is over we look up the note and continue with the major operation.

... We have only to think how this is to be done once, and then forget how it is done.

Also it was soon realised that computers and computer programs were going to serve as tools for the programmer. As Turing [1] wrote:

This process of constructing instruction tables should be very fascinating. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself.

But despite these very promising aspects of software construction, the by now well-known problem of program correctness was present from the very beginning. Again we quote Turing [1]:

Delay there must be, due to the virtually inevitable snags, for up to a point it is better to let the snags be there than to spend such time in design that there are none (how many decades would this course take).

Although the above-mentioned characteristics of software guarantee its construction to be a fascinating activity, it takes two more characteristics to turn it into a real engineering activity.
CHAPTER 1. OVERVIEW AND INTRODUCTION

The first of these characteristics is the possibility to construct mathematical models of computers and of the calculations performed by them. The second characteristic is the existence of a wide range of practical applications.

We shall briefly discuss the latter two characteristics now. Certain mathematical models of computers and of the calculations performed by them were already available before the actual construction of computers took place. Church's λ-calculus model and the Turing-machine model already existed in 1936 and they provided very abstract mathematical models of computation. Although these models were tuned to the analysis of a certain theoretical notion of computability, they mark the beginning of fruitful research which has produced an enormous amount of results and concepts. Let us highlight just a few areas. Of course the following list is a gross simplification of a huge field of research.

- The invention and analysis of numerous efficient algorithms for searching, sorting and numerical calculations (see e.g. [2]).
- The theory of program correctness started by Floyd, Hoare, Dijkstra and others (see [3,4,5]).
- Specification techniques and denotational semantics, which have led to the conception of special specification languages [6,7,8] and to a rich collection of techniques for describing semantics [9,10,4].
- Type theory [13,14,15,16], providing theoretical foundations for the concept of type-checking.
- Models of communicating processes starting with Petri nets and Milner's CCS [17,18].
- Programming language concepts, embodied in a never-ending stream of new programming languages: parameter mechanisms, programming variables, scoping, etc. Let us mention just a few important languages: FORTRAN, ALGOL, COBOL, LISP, Pascal, Modula, C, SETL, Ada, FP, ML, OCCAM, POOL and Prolog.

Programmed computers turn out to have many practical applications. Initially the applications were restricted to numerical computations, but soon they included also bookkeeping, administration, language processing, data manipulation and process control. Programmed computers have become an integral part of almost all technical systems. Completely new applications have come into existence in areas such as telecommunications, entertainment, manufacturing, safety, medicine and education, but regrettfully also in warfare. By now, programmed computers have deeply penetrated all systems of infrastructure upon which our society is built.
1.2. THE NATURE OF SOFTWARE

As shown above, the term 'software engineering' is quite justified. Let us mention two other terms as well: 'computer science' and 'computing science'. The former term is chosen somewhat unfortunately whereas the latter is not generally accepted (yet). Both 'computer science' and 'computing science' are often used in order to stress the role of the mathematical discipline. The term 'software engineering' is also used in connection with the project management and the organisational aspects of software construction.

It is easy to think that software engineering is related to electronic engineering, but in fact this is not really the case at all. It is true that computers are built up from electronic components; yet programmers need not know about the actual construction of the computer. The situation becomes even clearer when programming languages are employed which abstract more and more from the underlying machine.

Software engineering has a significant overlap with mathematics, although of course there are many branches of mathematics which are just not applicable. The following branches of mathematics are relevant [19]:

- formal logic,
- set theory,
- theory of relations, maps and functions,
- $\lambda$-calculus,
- automata theory,
- theory of formal languages,
- discrete mathematics.

Also queueing theory, category theory, graph theory and the theory of partial orders and lattices should be mentioned. The following definition of software engineering is given by Boehm [20]. It is quite consistent with the discussion presented above:

the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them.

1.2.2 Formality

There is one characteristic which distinguishes software engineering from most other engineering areas: formality. Every task to be performed by a computer must be formulated as a program. The computer executes such a program precisely as it is. In particular, the computer will not complain if the program is wrong, or if the program solves another problem than the one
meant by the programmer. Computer programs must be constructed with extreme care. One mistake may cause a program to produce nonsensical results. With respect to the results, there is no difference between small mistakes and large mistakes: forgetting a semicolon can be as disastrous as a collection of huge design errors.

The relevance of this characteristic is amplified by the size and the complexity of the programs. For simple programs there is a possibility that testing or even trial and error methods will eventually lead to acceptable results. For large software products such an approach is fatal. When the complexity of software products increases, program correctness becomes a very serious issue. Remember the fact that one mistake may cause the program to produce nonsensical results. Furthermore, this complexity may be such that it is impossible for one person to understand and remember all details of a program. It is a consequence of this complexity that communication, documentation, modularisation, specification and abstraction play an increasingly important role in the production of large software systems.

Many software projects exhibit phenomena which can easily be interpreted as the symptoms of inadequately approaching the formality and the complexity characteristics sketched above. One such phenomenon is a strong emphasis on project management, risk analysis, planning, baseline documents and milestones, whereas on the other hand it turns out to be very hard to put the planning into practice. Some authors even introduced the term 'software crisis'. We quote Sommerville and Morrison [21].

... in the late 1960s after the so-called software crisis was identified. This was the name given to the failure of the software developers to build large systems which were required to run on the then-new third-generation computer hardware.

Very much related to this, there is an increasing interest for modelling the software development process. We shall come back to these models later and in fact we shall construct a special kind of such models ourselves (this will be discussed in Section 1.6). We view the 'software crisis' simply as a symptom of poorly understood deeper causes such as the lack of formality and the complexity of software.

The formality issues mentioned above can be explained better by referring to the notion of a formal system [19]. A formal system consists of: (1) a syntax, by which we mean an alphabet and rules which define a set of formulae, and (2) a number of rules which can be used for the derivation of new formulae from some given formula. The rules of a formal system are given without any reference to their intuition or meaning. The formal system just defines
1.2. THE NATURE OF SOFTWARE

A computer program can be viewed as a formal system and the computer executing it can be viewed as the player of the game. The number of steps performed in a calculation may become so large that one person cannot imagine and understand the calculation as a sequence of steps.

Instead of that, reasoning about program execution must be based on more powerful (mathematical) approaches, using some implicit or explicit induction principle. If we want to consider the formal system as a black box and try to verify its correctness just by testing, then in general we can never be sure that the formal system is correct. Program testing can be used to show the presence of errors, but not their absence.

There is a second point where formal systems enter into software engineering, viz. as systems of reasoning. For this purpose many relatively simple formal systems have been proposed. The following list is of course not complete.

- Equational logic for specification of data types [22]. These specifications are often called 'algebraic specifications'. They work well for data types such as natural numbers, Boolean values, stacks and queues. In the typical 'stacks' example, one uses equations like $\text{pop}(\text{push}(s,x)) = s$ and $\text{top}(\text{push}(s,x)) = x$. Efforts have been made to push the approach further by describing more complex systems, such as simple programming languages [23], file systems [24] and data base systems [25].

- First-order predicate logic for describing the various states involved in a program execution. Imperative programs can be viewed as state transformers and hence there is a need to describe properties of states. A typical example of such a property is that in some state an array $a[0..n-1]$ is sorted. The state itself remains implicit and to express this property we would have a first-order predicate logic formula $\forall i, j : \text{N}(i < n \land j < n \land i < j \Rightarrow a[i] \leq a[j])$. This logic is already classic [26], but in the 1970s it has received much attention in connection with Hoare’s logic and Dijkstra’s wp calculus (see below).

- Hoare’s logic for reasoning about the correctness of sequential programs. This is a formal system for reasoning about triples $\langle P, s, Q \rangle$ where $P$ is a precondition, $s$ is a program and $Q$ is a postcondition. This logic can be interpreted as dealing with partial correctness of programs. It was one of the first formal systems in software engineering which was really presented and widely recognised as a formal system. By now it has been incorporated into the field of mathematical logic, where its generalisations are known as dynamic logic [27].

- Dijkstra’s wp calculus [5] is a formal system which can be interpreted as dealing with total correctness. One writes $P \Rightarrow wp(s, Q)$ where $P$ is a precondition, $s$ is a program and $Q$ is a postcondition.
There are several related reasons for the introduction of these formal systems of logic into software engineering. The first is the program correctness problem: formal systems of logic provide mental tools for reasoning about programs and data types. We could express this as follows: *one way of solving the problems caused by the introduction of a formal system, is to introduce another formal system, viz. a logic for reasoning.*

Another closely related reason is that formal systems of logic make it possible to construct tools that assist in reasoning about programs – at least in principle. The current state of the art is that the reasoning power of theorem provers is too limited to take full advantage of this possibility; this situation might change in the future. Even if there are no tools for reasoning yet, syntax checking and type checking the formulae of some system of logic already give useful results.

The size and complexity of programs make it necessary to split programs into modules which have a specification associated with them. Such specifications should contain precise properties of data types and of programs; the formulae of some system of logic are very suited for expressing these properties.

A question which has until now not been solved entirely satisfactorily is the integration of these formal systems of logic. One promising approach to this integration leads to the so-called wide-spectrum languages. These are languages combining an expressive logic with programming language constructs. Examples of these are VDM [6], Z [7] and COLD-K [8]. A further integration would have to incorporate models of communicating processes.

A somewhat different approach is taken in the Genesis project [28], where tools are constructed which are parameterised over systems of logic.

An interesting option which comes with the introduction of formal systems of logic is the possibility to provide the proofs and have them checked automatically. The proofs must be coded as expressions of a special version of λ-typed λ-calculus and then proof-checking becomes a matter of type-checking – which turns out to be decidable. The checkers are called verification systems or justification systems; the Automath checker [16] is one of the first examples.

An approach which might become important in the long term is to have a formalisation of typical software development processes. This formalisation then might be turned into an automation of (part of) the software development process [29].
1.2. THE NATURE OF SOFTWARE

1.2.3 The role of languages

There are many different languages in use in software engineering and much progress is language-driven. Each language $L$ is a set of sentences and when e.g. $L$ is a programming language, these are the well-formed programs that can be compiled and executed. To characterise a language one usually employs a grammar $G$ — which is a formal system — and we have $L = L(G)$. In order to structure our discussion, we adopt the following classification scheme:

- programming languages,
- specification languages,
- special-purpose languages.

We begin with a discussion of programming languages. These have always played an important role throughout the history of software engineering, which started with the machine language of the early computers. A machine-language program contains precisely the sequences of 0s and 1s that can be interpreted directly by the computer. For example, an instruction to load the accumulator register with a binary number 111, for example, could look as follows:

10000110 00000111.

Clearly this was a user-unfriendly approach and the wish to abstract from these bit sequences soon gave rise to a following generation of languages. These were called assembly languages; an assembly-language program has precisely the same structure as the corresponding machine-language program, but the sequences of 0s and 1s have been replaced by symbols. For example one could write

LDA A #7

rather than 10000110 00000111. In the 1950s FORTRAN [30] was introduced, in which one could write down formulae such as

$$\begin{align*}
X_1 &= \frac{-b + \sqrt{b^2 - 4.0*a*c}}{2.0*a}
\end{align*}$$

which is relatively close to the usual mathematical notation. This was a great step forward, although control-flow constructs and parameter passing were not very well-chosen from today's point of view.

Let us interrupt our historical survey now to make a few observations; these apply to the historical development sketched above, but in fact they apply to
the later developments as well. First, there has been a tendency to make the languages more abstract in the sense that the programs have more in common with the mathematical descriptions of the underlying models and less with the corresponding machine-language programs. The second observation is that many software developers care a lot about the language they use. The language in use serves as a medium for thinking about the problem at hand and the program to solve it.

The last observation is that the introduction of new concepts into the practice of software engineering can be done by embodying each concept as a language construct in a new language. But somehow the converse of this statement seems to be the case as well: it is hard to introduce methodological concepts unless these are very concretely available as constructs in a programming language. To give one example, the concept of parameterisation was from a mathematical point of view already well-described and modelled since the formulation of the \( \lambda \)-calculus; still it required effort and discipline to have well-identified parameters when writing subroutines in an assembly language. FORTRAN provided for functions and subroutines with parameters which made it much easier to use the concept of parameterisation – although of course the FORTRAN and the \( \lambda \)-calculus parameter mechanism are quite different.

In the 1960s and 1970s the historical development has been more complex. Several new concepts were introduced, but not in such a way that there is precisely one sequence of languages which is monotonically increasing with respect to abstractness and expressiveness. Instead of trying to provide a complete historical survey, we shall just mention a number of important concepts:

- modularisation,
- parameterisation,
- concurrency,
- higher-order functions,
- types,
- abstract data types,
- logic programming,
- term rewriting,
- inheritance.

For a systematic comparison of a number of languages, see [31] and [32].

A problem with the introduction of new concepts via a programming language is that usually several concepts are presented in a somewhat intertwined fashion. Typically the intertwining is between concepts of a method-
1.2. THE NATURE OF SOFTWARE

ological nature and concepts dealing with efficient program execution mechanisms. For example, the modules in [33] are primarily intended as a rather syntactic means of ordering a sequence of procedures, data types and variables, together with static ways of prohibiting access to constructs which should be considered local. But at the same time modules can have initialisation statements; moreover there are special kinds of modules, e.g. the so-called interface modules which offer solutions to the mutual exclusion problems of communicating processes.

Much progress of a methodological nature has been achieved by introducing new concepts such as modularisation, parameterisation, types, concurrency, and higher-order functions into programming languages. However, the problem with programming languages is that all constructs must be executable and as a consequence a need arose for another kind of languages which are optimal with respect to expressivity and abstractness. The resulting languages are called specification languages. This term may be somewhat misleading because these are not only useful for specifying programs to be constructed, but they can also be used for describing programs and systems which are already available. This need for another kind of languages has been expressed elegantly by Hoare [34]:

So in the specification you should take advantage of the full range of concepts and notations of mathematics, including concepts that cannot be represented on a computer and operations that could not be implemented in a programming language.

Many of the issues occurring in relation with modularisation, parameterisation, typing, concurrency, higher-order functions, etc. apply also to specification languages. For most of these concepts it seems a good idea to make them available in specification languages as well — although their integration still is a major technical problem. In [35] a specification language is defined as a language satisfying the following requirements:

- the language concepts enable the description of digital systems,
- the language constructions are derived from mathematical logic,
- the language has a precise syntax,
- the language has a precise semantics.

If furthermore the language allows for descriptions at several levels of abstraction, it is called a wide-spectrum specification language. Typically the wide-spectrum languages have a subset which can be considered as executable. However it is a misconception to postulate that as an ideal it should be tried to make all constructs in the language executable. By doing so, one just
gets another programming language. It is the very purpose of a specification language to avoid the conflicts between executability on the one hand and expressivity and abstractness on the other hand; in a specification language the choice should be in favour of expressivity and abstractness. In [35] a classification of specification languages is presented. We summarise it here:

- thematic languages, tuned to feature a theoretical approach,
- combination languages,
- mid-spectrum languages,
- wide-spectrum languages.

In this monograph, the main focus will be upon the mid-spectrum and wide-spectrum languages since these are currently reaching a level of maturity at which they can be applied in an industrial context. Furthermore in section 1.5 we shall propose a class of syntactic structures called ‘designs’ and hence the set \( D \) of all designs can be viewed as a language. Yet, \( D \) does not fit in the classification programming/specification languages; instead of that, it provides a kind of ‘missing link’ between these two language-classes because it serves for putting programs and their specifications together in an orderly fashion.

There is a great diversity of special-purpose languages. Because these can be tuned to very specific needs, they are less subject to conflicting requirements than the general-purpose programming languages and the general-purpose specification languages. As examples we have data-base query languages, job-control languages, input languages for parser-generators etc.

### 1.2.4 The role of tools

In this section we shall briefly discuss the role of tools, i.e. computer-programs which are meant to be of help in software engineering. The role of tools is to relieve the software designer from certain dull and error-prone subtasks — as already indicated by the second citation of Turing in Section 1.2.1. Of course only those subtasks can be automated which are completely understood and which are formalised. We list a number of tools.

- Syntax checkers which serve to determine whether an input text belongs to the language for which the checker has been built.
- Type checkers which serve to determine whether an input text is correct according to the type system for which the checker has been built.
- Code generators which serve to perform meaning-preserving translations from a source language to a target language.
1.2. THE NATURE OF SOFTWARE

- Compilers for programming languages. Typically each compiler contains a syntax checker, a type checker and a code generator.
- Text editors, structure editors and graphics editors which serve to enter and modify programs, specifications and other documents which play a role in software engineering.
- Data-base systems, file systems, and configuration-management systems which provide systematic ways of storing programs, specifications and other documents.
- Mail systems, calendars, news systems etc. which serve to structure the communication between software engineers.
- Text-processing tools and type-setters which serve to produce nice-looking and well-structured documents.
- Interactive program transformation systems which assist the software engineer in transforming specifications into programs using correctness-preserving transformations.
- Theorem provers, proof assistants and proof checkers which assist the software engineer in constructing and checking proofs in a formal system of logic.

The full potential of possibilities has by no means been fully exploited so far. A potential bottleneck is the lack of integration of all these tools [36]. This integration is complicated by the existence of many programming/specification languages and systems of logic. Also the diversity of available computers and operating systems is a complicating factor in practice.

Each tool must be based on a formal system and/or an underlying theory; for example, a syntax checker is based on a particular grammar and the general theory of languages and parsing whereas a type checker is based on a particular version of type theory. In general, the construction of tools is a natural follow-up of language definition or of methodological and theoretical advancement.

1.2.5 Application domain evolution

The characteristics of the process of software development are not the same for all application domains. It is not possible to give an overview of all application domains, but some useful remarks can be made by distinguishing application domains with respect to their maturity. In particular, the degree of achieved formality turns out to be a dominant factor. This section serves to provide a slightly relativising context for the discussions of the preceding sections and it also plays a role in positioning our own work.
CHAPTER 1. OVERVIEW AND INTRODUCTION

We summarise the idea as worked out in [37], where Sikkel and Van Vliet discuss domain-oriented reuse of software. They state that the possibilities for reusing software depend on the maturity of the application domain.

Application domains are subject to a certain development and there are four distinct phases:

- no reuse,
- ad-hoc reuse,
- structured reuse,
- automation of the domain.

It is argued that the introduction and standardisation of useful ideas, concepts and structures starts in the second phase. This introduction is an iterative process which takes quite a lot of time. Only after that is it possible to make the transition to the third phase. Formalisation is viewed as a tool which can help in establishing this transition.

Sikkel and Van Vliet focus attention on this transition from the second phase to the third phase. Indeed, for the first and second phases it is hardly possible to provide useful models of software development, whereas in the third phase the software development itself can proceed in an orderly fashion. Therefore it seems worthwhile to speed-up this transition whenever possible. Just as Sikkel and Van Vliet, we believe in formal specification techniques as a tool for doing this. In the third phase, the software development itself is to a certain extent already amenable to mathematical analysis.

In Section 1.6 we shall discuss certain models of the software development process, which can indeed be viewed as examples of such a mathematical analysis. In the fourth phase the formalisation of the software development process leads to replacing an entire design activity by a simple push-button operation. Furthermore, when this push-button operation has become sufficiently efficient, the application domain will have become quite unproblematic from a software engineering viewpoint.

1.3 Informal models of software development

1.3.1 General

Below we shall summarise some informal models of software development. Most of these informal models of software development are not based on a deep analysis of formal systems, but rather they are based on common sense, social observations and empirical data. Each of these models conveys some
1.3. INFORMAL MODELS

useful insight about software development. There is a large variety of such models and quite arbitrarily we selected a few representative models for a brief discussion. No attempt has been made to give a complete overview of this type of models. For an overview we refer to [38] or [39].

1.3.2 Waterfall model

The waterfall model [20], also called the life-cycle model, states that, roughly speaking, the software development process goes through the following phases:

- system feasibility,
- software plans and requirements,
- product design,
- detailed design,
- code,
- integration,
- implementation,
- operations and maintenance.

At the end of each phase a kind of verification or validation takes place and if its outcome is unsatisfactory, a return to a previous phase may take place. This model and variants of it have been used in many projects and these models have a certain value by providing guidance with respect to the order in which a project should carry out its major tasks.

1.3.3 Boehm's spiral model

Boehm [40] states that many software life-cycle models of software development are inappropriate in many situations. He proposes a so-called spiral model of software development and enhancement. This model is an elaboration of the waterfall model of Section 1.3.2. The spiral model adopts a similar sequence of phases as the waterfall model, but the sequence begins with having a prototype. Furthermore, this model has the additional feature that this sequence should be done several times as an iterative process. Every time before the sequence is entered, a risk-analysis must take place. The model can be viewed as a kind of repetitive statement and our own presentation of it below shows it as a while ... do ... loop.
Boehm has a pictorial notation for it where the total execution sequence is shown as a spiral.

\[
\textbf{while} \ \text{risk} < \text{max} \ \textbf{do} \\
\quad \bullet \ \text{prototype,} \\
\quad \bullet \ \text{software plans and requirements,} \\
\quad \bullet \ \ldots \\
\quad \bullet \ \text{operations and maintenance.} \\
\textbf{end}
\]

Every iteration cycle yields some product which enters the next cycle as a prototype. The idea of risk-analysis gets a lot of attention in this model and it is extensively discussed in [40].

### 1.3.4 A layered behavioural model

Curtis, Krasner and Iscoe [41] report about a field study of the software design process for large systems. It is argued that human and organisational factors seriously influence the execution of software development tasks. A layered behavioural model of software development is given. It has five layers, each of which has a certain influence on the software development process. The model focuses on the behaviour of the people creating the software, rather than on the software itself. The five layers are as follows:

- individual,
- team,
- project,
- company,
- business milieu.

By means of a field study based on interview techniques the three most salient problems of software development were identified. Each problem is extensively discussed for each of the layers of the model:

- the thin spread of application domain knowledge,
- fluctuating and conflicting requirements,
- communication and coordination problems.

These problems have survived for several decades despite serious efforts to improve software productivity and quality. Software development tools and practices are said to have disappointingly small effects in earlier studies,
because they did not improve the most troublesome processes in software development.

The proposition that human and organisational factors affect the execution of software development tasks is convincingly shown in [41]. However there are opportunities to make progress by studying simplified models of software development at a technical level. Also the reported communication problems suggest that it is useful to improve upon the specification techniques employed for the products.

Let us say a few words about the relation of the model and the work on models of software development of this monograph. We do not contribute to the ecological research suggested in [41] nor to the studies of processes such as learning, requirements negotiation, etc. Instead we focus on the software product and its descriptions and on the evolutionary behaviour of these through certain development stages.

1.3.5 Koomen's iteration, learning and detailing model

In [42] the software development model is viewed as a step from a specification $S$ to an implementation $I$. In general, the implementation is more complex than the specification; this is because of decomposition and the introduction of additional information. Koomen explicitly mentions the knowledge $K$ which is needed by the developer to perform the step $S \rightarrow I$. This is denoted as follows:

$$S \rightarrow_I \ K$$

It is argued that the developer must acquire the knowledge $K$ by learning, which in most cases takes place by means of iterations and making errors. Furthermore the transformation process is not just one step, but at another level this step consists of iterations and recursions.

Also in [42] a distinction is made which in our opinion is quite useful, viz. the distinction between an idealised design process (IDP) and an actual design process (ADP). It seems worthwhile to study IDPs and even to formalise them, but because of the errors which occur during the learning process, it is inevitable that there is a difference between ADP and IDP.

Before setting up our own line of development aiming at certain formal models of software development, let us explicitly mention some useful ideas that can be taken from the above informal models and that also act as ingredients of the formal models. From the waterfall and spiral models, we take the use of sequential composition and repetition as structuring mechanisms at
process level. From Curtis, Krasner and Jacobs' report we recall that there are many aspects of software development that seem not amenable to formalisation (yet). But we also mention the reported communication problems which we shall translate into the need for having 'redundant' specifications (see Section 1.6). From Koomey's model we adopt the idea that software development is a process of detailing and adding information; this will come back in our top-down and bottom-up models (see Section 1.5) which are strategies of growing (= adding information to) structures called designs.

1.4 Towards formal models of software development

1.4.1 Modularisation

Although the informal models of software development of Sections 1.3.2, 1.3.3, and 1.3.4 convey certain information about software and its construction, they are still somewhat unsatisfactory. We view it as a weakness of these and similar models that they impose a management-oriented and project-oriented structure upon a process of software development which is in fact still poorly understood. Indeed, the situation is even worse in practice, for often the software itself is poorly understood.

We shall study models of software development in this monograph, but we want to start from a position where we can understand the structure of the software. Then we take this structure as a starting point for making meaningful statements about models of software development. Our main mental tool will be that of formalisation. We begin with having a look at modularisation, since this seems to be the right way ahead to impose meaningful structure upon the software itself.

To get started, it is sufficient to have an informal understanding of the question what is a module?. A module is a piece of software which more or less constitutes a coherent unit. One may think of it as a bunch of procedures which together provide a certain functionality; one may also think of it as an abstract data type.

Let us assume that modules can be fitted together to form larger modules by composition mechanisms such as import and that it is possible to associate names to modules. Then it becomes possible to represent the module structure of a software product as a directed acyclic graph (DAG) where the nodes are modules and where the edges correspond with the 'is part of' relation.
For example, consider a module structure which is as follows:

- let m1 := module ... end;
- let m2 := module ... end;
- let m3 := import m1 into module ... end;
- let m4 := import m1, m2 into module ... end;
- let m5 := import m2, m3, m4 into module ... end;

This is represented by the figure below, where an arrow indicates that one module is part of another module.

![DAG module structure diagram]

**Fig 1.1.** Example of DAG module structure.

To have a well-chosen module structure in a software product is the key to avoiding many problems in later stages of the software development process. For a discussion of the criteria for choosing a module structure, we refer to [43]. Although DAG structures as sketched above are attractive because of their simplicity, they are in practice hardly adequate for describing the structure of real software systems. This is because they can not cope with parameterisation (generic modules) and specification ('redundant' modules for specification purposes only).
1.4.2 Parameterisation

Parameterisation of modules is a technique for improving the reusability of modules. Software reuse is currently a hot topic in software engineering literature [44] and it is important indeed, for at least the following reasons.

- It is hoped that more frequent reuse of software will lead to higher productivity of software developers and to significant cost reductions.
- Reuse of modules within one and the same design may lead to significant reductions with respect to the complexity of the design.
- Frequently used software packages could be better certificated for proper construction and operation. For these packages, it is worthwhile to put more effort in obtaining optimal results.

Parameterisation comes from the insight that often modules could be reused, provided they have suitable parameters. E.g. the Ada language provides a mechanism to indicate that a package (= module) is generic, which means that the package has one or more parameters. Both types and functions can occur as parameters. We quote Sommerville and Morrison [21].

One classic example of how generics are useful is in sorting procedures. In most languages two different procedures are required to sort an array of integers and an array of real numbers. Indeed, the situation is much worse than that since we require a different sort procedure for every different type.

When a generic package is used, the actual parameters must be chosen and the substitution of the actual parameters for the formal parameters is supposed to be done at compile time.

Reusability of modules can be improved by providing them with parameters but one can conceive of a large number of options for such a parameter mechanism. Let us have a look at two approaches, viz. the parameterisation of the algebraic specification languages CLEAR and ASL.

In CLEAR, the parameterisation is uniform in terms of modules: one can parameterise a module with respect to a parameter (x say) ranging over modules. In the design of CLEAR [50] it has been recognised that the formal parameters of modules must satisfy certain requirements before an instantiation can take place. There is no full λ-calculus, but there is a special construct called theory procedure. For example in CLEAR one writes

\[
\text{procedure Sorting}(x : R) = M
\]
to introduce a parameterised module named Sorting. $M$ is the actual specification of sorting which of course must be parameterised with respect to the sort of things to be sorted and the relevant ordering relation. The latter sort and relation are described in the parameter restriction $R$. The colon is somewhat misleading since it does not refer to a classical (irreflexive) typing relation but to the implementation relation.

In ASL [52] there is an explicit lambda notation for parameterisation, but the connection with the formal implementation relation is not made. In ASL one writes

$$\lambda \text{spec } x, \text{sort } s, \text{opn } p : R \cdot M$$

to parameterise the module $M$ with respect to a module (spec) $x$ which contains a sort $s$ and an operation $p$. Again $R$ is a parameter restriction, but in ASL it can be either a module or a Boolean expression.

It is not obvious how to reflect parameterisation and instantiation of modules in the DAG structures discussed before, which means that actually the DAG approach is somewhat too naive in practice. In Section 1.5 improved design structures and a parameterisation mechanism are proposed.

### 1.4.3 Simple top-down and bottom-up models

When the DAG model for the modular software products applies, this gives an opportunity to impose structure upon the development process. In this way one easily arrives at simple versions of top-down and bottom-up models and these are nothing but ways of letting the DAG grow.

We depict a few steps from a typical bottom-up development process.

---

![Diagram](image)

Fig 1.2. The simple bottom-up model.
The three distinguishing properties of this simple bottom-up development model are the following.

- Initially there are one or more unrelated modules. Let us call these the *machine* modules.
- Modules are only added, and never removed.
- All modules are always constructed in terms of the machine modules - again either directly or indirectly.

We depict a few states from a typical top-down development process.

![Diagram](image)

**Fig 1.3.** The simple top-down model.

The three distinguishing properties of this simple top-down development model are the following.

- Initially there are one or more unrelated modules. Let us call these the *system* modules.
- Modules are only added, and never removed.
- All modules are always either directly or indirectly 'part of' at least one system module.

It is not hard to formalise the DAG structure and the simple top-down and bottom-up models discussed above. Similar discussions are presented in [46] and [47]. The approach can also be pushed further by including middle-out and outside-in models.

However, the approach is not entirely satisfactory, mainly because the need for specifications is not taken into account. Especially the simple top-down

---

3It is interesting to compare this with M. Feather's [45] view of growing specifications as a process of applying elaborations e.step1, e.step2 etc. upon an initial specification m0. Feather explores the combination of parallel elaborations e.step1(m0) and e.step2(m0) say. which e.g. for independent steps could mean to take e.step2(e.step1(m0)).
1.5. A FORMALISATION OF DESIGN STRUCTURES

approach can hardly work in practice, because it is problematic to use a non-existing module unless a sufficiently precise specification of it is available. This defect becomes even more clear when we aim at reuse of modules. Parameterisation is a mechanism for improving module reusability, but at the same time it leads to an increasing need for specifications. Even when the functionality of a reusable module is sufficiently known, there is still a need to have clear (formal) specifications of the parameters and of their effect upon module behaviour.

The simple top-down and bottom-up models are too poor to describe this, but there is a clear route for improving this situation. The main extension is to associate specifications with modules. In order to make this precise, a formalisation of the resulting design structures (= structure of modules + specifications) will be needed. This is the subject of Chapter 2, summarised in Section 1.5. After that we shall be in a better position to have a second look at the dynamic aspects of the process of software development and this is the subject of Chapter 3, summarised in Section 1.6.

1.5 A formalisation of design structures

This section is just a very informal and incomplete introduction to the contents of Chapter 2. The current section is meant to fit logically in the main lines of thought presented in the current chapter, but the detailed technical work involved is done in Chapter 2.

Chapter 2 begins with a formalisation of the module structure of software products. A mathematical model is introduced, which covers from an abstract point of view several distinct languages. According to the model, modules are nothing but terms and the ways of fitting modules together are algebraic operators in the language of an algebraic system.

This algebraic approach to module composition is investigated by several authors. For example, in Module Algebra [48] modules are terms consisting of module constants/variables and the operators Σ (the visible signature of a module), . (renaming of a module), T (conversion of a signature to a module without axioms), + (combination/union of modules), and ⊓ (restriction of the visible signature of a module). The ACT ONE MOD language [49] provides algebraic operators for basic module specification, extension, renaming, union, composition and actualisation. The CLEAR language [50] provides algebraic operators for basic module specification (either with loose semantics or initial-algebra semantics), enrichment and derivation (= hiding + renaming). Closely related to Module Algebra is the Class Algebra underlying the COLD-K design language [8], the main difference from Module
Algebra being related to the fact that in COLD-K one can describe systems having a state.

In Chapter 2, there is no fundamental difference between a specification module and an implementation module. The only difference is a matter of role and this is formalised by assuming a binary relation \( \subseteq \) on modules where

\[
M_1 \subseteq M_2
\]

means that \( M_1 \) is an implementation of \( M_2 \). This relation is required to be a preorder (= reflexive + transitive). Starting from this simple model, a \( \lambda \)-calculus approach to parameterisation is developed in Chapter 2. However, unlike classical (untyped) \( \lambda \)-calculus, there are restrictions to the actual parameters which may be provided for a parameterised term. The resulting calculus is called \( \lambda \pi \)-calculus. In classical \( \lambda \)-calculus [51] one would write \( \lambda x.M \) to indicate that \( x \) is a formal parameter, which may occur in \( M \).

Instead of that, in \( \lambda \pi \)-calculus we get

\[
\lambda x \subseteq R.M
\]

where \( R \) is a so-called parameter restriction. The calculus is formalised by means of a collection of rules, where the most important rule is called (\( \pi \)), because it is a kind of partial version of the well-known rule of classical \( \lambda \)-calculus, which is called (\( \beta \)). Somewhat simplified, the rule (\( \pi \)) is as follows:

\[
(\lambda x \subseteq R.M)A \rightarrow M[x := A], \quad \text{provided } A \subseteq R.
\]

This calculus can be put on top of an arbitrary algebraic system with preorder, which means that it works for any formalism with an algebraic approach to module composition.

It is interesting to compare this \( \lambda \pi \)-calculus with the parameterisation of CLEAR and ASL as mentioned in Section 1.4.2. CLEAR does not provide a set of rules as in \( \lambda \pi \)-calculus but instead of that the parameterised specifications get a semantics directly in terms of mappings from algebras to algebras. Just as CLEAR, ASL does not provide a set of rules and the parameterised specifications get a semantics in terms of mappings from algebras to algebras. Furthermore the ASL mechanism is not uniformly in terms of modules: one can parameterise a module with respect to one or more sorts and/or operations.

The \( \lambda \pi \)-calculus can be provided with a simple system of types and then it is shown to have the Church-Rosser property (also called confluence) and the strong normalisation property.
Using $\lambda\pi$-calculus, Chapter 2 describes a formalisation of design structures by introducing the concept of a \textit{design}. Intuitively, a design is a (possibly large) hierarchically structured and component-wise specified software system. Precisely the fact that a design is component-wise specified implies that issues of information-hiding arise, which cannot adequately be dealt with by just the algebraic approach to module composition alone; as it turns out, they can however be dealt with by the $\lambda\pi$-calculus.

The designs of Chapter 2 are related to the so-called books in Automath [54] and we shall sketch the relation here using De Bruijn's terminology — which is quite recent. Every lambda term can be viewed as a tree, having nodes of two kinds.

- Application (a) nodes, where a parameterised term gets an argument,
- Typing nodes (t), which are just $\lambda$-abstractions. Let us adopt this term, though formally the parameter restrictions of $\lambda\pi$-calculus are not types.

Now one can organise lambda terms so that they become a special kind of slanted trees, built up from typing-with-immediate-application pairs (at pairs) and single typing nodes (t nodes). Hence the trees are named at&t systems. This is precisely the way Automath books are organised and it can be used for coding and type-checking mathematics. De Bruijn noted this 'books-as-lambda-terms' analogy.

Now it turns out that at&t systems in $\lambda\pi$-calculus can be used for giving a meaning to a formal notion of design, corresponding to the intuition sketched above. In this way there is a 'designs-as-lambda-terms' analogy.

A design can be represented by a structure which looks like this:

\[
\begin{align*}
x_1 & := P_1 \sqsubseteq Q_1 \\
x_2 & := P_2 \sqsubseteq Q_2 \\
\cdots \\
x_n & := P_n \sqsubseteq Q_n \\
\text{system } S
\end{align*}
\]

where each line is called a component, each $P_i$ is called a \textit{glass-box description} (= implementation module) and each $Q_i$ is called a \textit{black-box description} (= specification module).

It is optional to omit the glass-box descriptions for certain components and in that case the corresponding $P_i$'s are just prim, which is a dummy placeholder. Such components are called \textit{primitives}. Primitive components act as the parameters of the design in which they occur; typically something will be filled-in for the prims in a later stage of the development.
In Chapter 2 two notions of correctness for designs are defined, called glass-box correctness and black-box correctness. Both forms of correctness are very important from a methodological point of view. Glass-box correctness is based on the principle that all details of the implementation modules may be known when using these modules. No information-hiding is required.

Black-box correctness is based on the principle that no details of the implementation modules may be known when using these modules. Instead of that, the user of a module must rely exclusively on specifications - called black-box descriptions in our terminology. The principle of black-box correctness is probably already old and has for example been expressed in [55].

We feel it is essential that the user of an interface should not have to know anything about the details of the implementation. In particular, the fact that this interface may be formed by a single body or a set of several bodies should not make any difference.

However, to the best of our knowledge, it has never been formalised adequately and analysed with the techniques of mathematical logic, as in Chapter 2.

The definitions of these correctness notions provide a starting point for a systematic study of transformational development. In particular, a design is built up from components and the correctness of a complete design follows from the correctness of its components. In Chapter 2 we investigate the precise conditions so that when modifying one component, the correctness of the resulting design follows from the correctness of the modified component. Black-box correctness gives rise to a certain locality principle with respect to modifications of implementations. Such modifications of implementations occur very frequently in practice, typically because of requirements to achieve certain efficiency improvements. For large designs, a complete re-verification of the (black-box) correctness is out of the question in practice, and as a consequence it is of key importance to have locality principles. In Chapter 2 a mathematical basis is provided for this and a number of locality principles and a number of pitfalls are formulated and demonstrated in a very precise way.

In Chapter 2 it is shown that several interesting properties of designs have simple analogons in the $\lambda\sigma$-calculus. The most striking analogy is that black-box correctness holds in a design if and only if the corresponding lambda term in $\lambda\sigma$-calculus can be reduced so that no candidate redex remains uncontracted and the contractions are performed from right to left. The new and innovative aspect of these analogies is that reduction strategies of lambda terms are used to characterise methodological notions.
It is important to note that in this monograph, both the design structures and a number of methodological notions are defined formally and are analysed using techniques from mathematical logic. This is a great step forward with respect to the usual informal and intuitive approach to the same methodological notions (as e.g., in the above quotation from [55]).

1.6 Correctness-preserving transformations of designs

This section is an overview of the contents of Chapter 3. In view of the achievements mentioned in Section 1.5, we are now in a better position to continue the line of thought presented in Section 1.4. We aim at a precise description of the top-down and bottom-up models of software development, but now we consider them as ways of growing designs rather than just DAGs.

In order to have a systematic approach, Chapter 3 begins with an examination of algebraic operations at the level of designs. So given two designs $d_1$ and $d_2$, we investigate the mechanisms to combine these to another design, $d_3$ say. There are two operations of this type:

- a binary operation $*$ to be used for combining unrelated designs,
- a binary operation $\circ$ which can be used for combining designs where the system provided by one design is plugged into the list of assumed primitives of another design.

Of course the question arises under which circumstances these operations preserve black-box correctness. This leads to a formal notion called validation. As it turns out, the question of preservation of black-box correctness can be answered satisfactorily. Again this question involves non-trivial issues of information hiding, requiring an analysis in the tradition of mathematical logic, using results from Chapter 2.

After that, a formalisation of the top-down model and the bottom-up model is undertaken, employing techniques which are taken from the product level: first the pre- and postconditions are made explicit and after that an iteration construct is introduced based on an invariant assertion. In particular, the top-down model is based on an invariant assertion TD.INV which asserts that the current design is black-box correct and that all components play a role in the system of the design. In Chapter 3 we describe the models as design-programs and by way of example we show a part of the top-down design-program, which is called $td$. The input-parameters of $td$ are denoted as $d_b$ and $d_i$ which – roughly speaking – are the external interfaces of the
design to be constructed. Below, \( d \) corresponds with a variable design (the 'current design') which is modified in a step-wise manner,

\[
\begin{align*}
td & := \text{technique } d_s, d_t \\
def & := d_t; \\
& \text{while not bot}(d) = d_s \text{ do } d := \text{td}_{\text{step}}(d); \text{ od}; \\
& d.
\end{align*}
\]

This \( \text{td}_{\text{step}} \) is such that it preserves TD.INV. The design-development language used to express the design-programs has a well-defined syntax and semantics which is given in an appendix of Chapter 3. For the formal details we refer to Chapter 3.

It is remarkable that the top-down and bottom-up models of the development process arise in a systematic manner by applying (at the design-program level) an approach which comes from the field of classical sequential programming. The possibilities for deriving invariants and design-programs describing design creation by this approach have by no means been exhausted in Chapter 3. It certainly is interesting to investigate other possibilities.

The fact that it is possible to split designs and reassemble them again makes it possible to discuss models of the development process where two (or more) developers each operate on a part of a design so that when each of them has finished his part, their results are fitted together to yield a new design again, which furthermore is black-box correct.

The formal notion of validation makes it possible to construct also several simple models of design evolution. Summarising, Chapter 3 provides three types of models of the software development process:

- models of design creation,
- models of design partition,
- models of design evolution.

In each case the models are very simple – at least from the viewpoint of practical industrial applications. But they are formal and they are based on a non-trivial approach to modularisation and information hiding and in that sense they constitute a step forward with respect to the DAG-growing models of Section 1.4.3. Also, it is very important that these models are characterised formally and that now we know very precisely the invariant assertions on which the top-down and bottom-up design programs are based. The fact that our models are formal can be viewed as an advantage over informal and semi-formal process models, as in Section 1.3.
The important question of course is whether these models can be made useful for the practice of developing non-trivial software products. Actually, a very similar question applies to λ-calculus and the formal notion of design of Chapter 2. Turning this theory into practice is a large undertaking of an engineering nature. This has been undertaken indeed, viz. in the context of a 'project' around the COLD-K language. This project aimed (and aims) at achieving a significant development in formal specification techniques and at industrialising the results. Clearly this is a task involving many people, the author being just one of them. At the present time not much tool-support related to the results of Chapters 2 and 3 is available, but as argued in Section 1.2.4, the construction of these tools may be a natural follow-up of the methodological and theoretical results of Chapter 2 and 3. Presently only the well-formedness condition 'wf' on designs can be checked, but one can imagine more sophisticated checks. Also a tool to list all proof-obligations related to the black-box correctness of a given design could be useful.

Next we would like to relate and compare the achievements from Chapter 3 with another, somewhat similar approach, viz. L. Williams' approach [56]. In [56] a software process model (SPM) describes software development as a sequence of activities. An activity is defined as a 4-tuple consisting of a set of preconditions, an action, a set of postconditions, and a set of messages. Williams describes by means of a kind of grammar which sequence of activities are possible for a certain approach. In particular, his design programs (to use our own terminology) take the shape of regular expressions with sequential composition, $\lor$ (choice), $\Lambda$ (interleaved parallelism) and $*$ (repetition). Using this as a description technique, he gives design programs for several models, such as Boehm's spiral model, Lehman's contractual model and the JSD method.

Although at the process level a certain formal machinery is used (regular expressions), at a lower level the activities (forming the alphabet for the regular expressions) are described informally and do not refer to product artifacts with a formal definition. When we compare this with the process models (design-programs) of Chapter 3, we see that the formality of [56] is lacking a basis, which is present for our approach — viz. in Chapter 2.

It is also interesting to compare our design programs with the DEVA approach of [29]. DEVA can be used for (a.o.) describing functions, statements, programs, contexts, modules, etc. and also the transformations and tactics operating on these. The design-development language of Chapter 3 is of a much less general-purpose nature and the design programs of Chapter 3 describe strategies for growing and combining the design structures of Chapter 2. The kernel of DEVA is an applicative language whereas we employ an imperative design-development language. Probably the most important dif-
ference is that DEVA design programs can (at least in principle) be executed automatically, whereas the semantics of our design programs is relational to reflect the phenomenon of human 'creative freedom' and there is hardly any sense in letting a computer evaluate the top-down design program, for example.

The work presented in Section 1.5 (which is a summary of Chapter 2) and Section 1.6 (which is a summary of Chapter 3) has served as a contribution to the design of COLD-K. As a result, the parameterisation of COLD-K modules is based on λσ-calculus. Furthermore, COLD-K provides constructs for components and designs, based on the concepts developed in Chapter 2 and Chapter 3.

The next two sections (1.7 and 1.8) are about formal specification techniques and they discuss several specification languages, with a special emphasis on COLD-K. Section 1.8 should be viewed as a kind of import at the level of this monograph: we import COLD-K because it is used in Section 1.9 (which is a summary of Chapter 4) and Section 1.10 (which is a summary of Chapter 5). Chapters 4 and 5 illustrate the applicability and usefulness of the theory developed in Chapter 2 and Chapter 3. They contain an example of a large formal specification and a subsequent development process based on the formal models of software development.

### 1.7 Formal specification techniques

#### 1.7.1 General

In Section 1.2.3 we have already discussed the role of specification languages. In this section we will have a closer look at a few approaches to formal specifications, each with its own specification language.

The discussion in Section 1.2.3 suggests that in many respects the practical progress in software engineering is language-driven: it is relatively hard to introduce new methodological concepts unless there are very concretely available as constructs in the language in use. Probably this is a major motivation behind the need for formal specification languages. In practice it is not enough to have a number of good concepts dealing with the methodology of writing formal specifications: in order to make these concepts transferable to a large audience of potential users, one needs a language as a vehicle. The design of such a language involves many choices and clearly the outcome of these choices will determine the restrictions to be imposed upon the users — who may or may not like that. But this seems by far outweighed by the
1.7. FORMAL SPECIFICATION TECHNIQUES

benefits of having a very concrete, more or less standardised language.

We do not attempt to provide a complete overview of all relevant languages; for that we refer the reader to [57] or [35]. The main purpose of this section is just to mention some specification languages which are relatively close to COLD-K with respect to their aims and their technical contents. In particular this means that we shall restrict ourselves to so-called mid-spectrum and wide-spectrum languages. Quite arbitrarily we have chosen to discuss VDM and Z, but there are other important approaches as well, notably CIP-L and Larch. For a comparison with these we refer to [58].

1.7.2 VDM

VDM is a formal-specification approach which has led to the design of several related specification languages, most of which are often just called VDM. The roots of VDM lie in the formal specification of programming languages (PL/I) and in the early days one would refer to the specification language as the meta-language, just to distinguish it from the programming language being specified.

The VDM version Meta-IV [6] (1978) has attracted considerable attention and it has been used to specify many complex systems. Its strong points are higher-order functions, a rich collection of built-in data types and the existence of a large body of pragmatics and examples. The built-in data types include sets, pairs, maps, tuples, trees. Each of these data types comes with a rich collection of operators, most of which are denoted in a mathematical style. E.g. for sets one can use symbols such as \( \cup, \cap, \subseteq \) and set comprehension \( \{ x \in S \mid p(x) \} \). For functions there is even a \( \lambda \)-calculus notation, including a fixed-point operator \( Y \).

In [59] an improvement of the VDM language is described. It reflects some influence of both the achievements in the theory of program correctness and the work on algebraic specifications. There is an explicit treatment of the problem of partial functions. This topic arises naturally in many practical situations and it is closely related to the idea of associating preconditions with functions. Jones uses a pre- and postcondition style for specifying functions. The following toy-example of a partial function has been taken from [59] (p. 74).

\[
\text{subp} \{ i : \mathbb{N}, j : \mathbb{N} \} \ r : \mathbb{N} \\
\text{pre } j \leq i \\
\text{post } r + j = i 
\]

The problem of partial functions is solved by the introduction of the logic
of partial functions (LPF). In this way it becomes possible to treat the set of truth-values as any set and to view a predicate $P$ as a Boolean-valued function. This leads to the situation that $P(x)$ fails to denote a Boolean value when $x$ does not satisfy the precondition of $P$. In that case neither $P(x) = \text{true}$ nor $P(x) = \text{false}$ holds.

VDM is based on set-theory which is very flexible; one can freely construct subsets of given sets and each of these subsets can occur again as the domain or range of a function. This implies that there is no concept of typing – except of course that the number of arguments of a function can be viewed as a simple type.

VDM has special constructs for so-called operations which are similar to functions, except for the fact that operations can have side-effects. The following example of an operation has been taken from [59] (p. 86).

\[
\text{LOAD} (i : \mathbb{N}) \\
\hspace{1cm} \text{ext wr reg : } \mathbb{N} \\
\hspace{1cm} \text{post reg = i}
\]

\text{LOAD} is introduced as an operation which has write-read access (wr) with respect to a so-called external variable (ext) called \textit{reg}. This variable can contain natural numbers. The availability of these operations and algorithmic definitions for them in addition to the specification techniques for mathematical functions makes VDM into a true mid-spectrum or wide-spectrum language (see [35]). Currently there is still a lot of development going on around VDM and the language has reached a degree of maturity and acceptance exceeding that of most other specification languages. An interesting development is Middelburg's VVSL language [60] where the COLD-K modularisation constructs and \textlambda\pi-based parameterisation constructs have been put on top of an (enriched) version of VDM.

1.7.3 Z

The specification language Z has emerged from a certain style of specification developed at the Oxford Programming Research Group. Z was first proposed by Abrial and it has been evolving for several years, but the recent book of Spivey has set a rigorous standard now [61]. Z is essentially based on Zermelo-Fraenkel set-theory described in first-order predicate logic. It offers a rich collection of notations for operations on sets such as $\cup, \cap, \subseteq$ and $\{x \in S \mid p(x)\}$. Relations, maps and functions are considered as special kinds of sets and they all have their own special notations. It is possible to introduce new primitive sets and if one does so, this means that the remainder of the
1.7. FORMAL SPECIFICATION TECHNIQUES

specification is parameterised with respect to these primitive sets.

We show an example in Z below. It introduces a so-called schema `subp` which can be interpreted as an operation from \( \mathbb{N} \times \mathbb{N} \) to \( \mathbb{N} \) satisfying the specification of our earlier VDM example.

\[
\text{\texttt{subp}} \\
\text{\texttt{i?, j? : \mathbb{N}}} \\
\text{\texttt{r! : \mathbb{N}}} \\
\text{\texttt{j? \leq i?}} \\
\text{\texttt{r! + j? = i?}}
\]

Although formally \( i? \), \( j? \) and \( r! \) are just logical variables, it is a convention that variables ending in \( ? \) are inputs and those ending in \( ! \) are outputs. The entire double-box construction is called a schema. A schema consists of a signature part, declaring one or more variables and an optional axiom part which serves as a predicate relating these variables.

It is also possible to describe operations having side-effects. This is done by formally introducing the two states as logical variables in the signature part of the schema of the operation.

Let us try the second VDM example as well. The only interesting state component in this example is a register which we formally introduce by a schema having a signature part only.

\[
\text{\texttt{REG}} \\
\text{\texttt{\texttt{reg} : \mathbb{N}}} \\
\]

Now we can mention both \( \texttt{REG} \) and \( \texttt{REG}' \) in the signature part of the schema of \( \texttt{LOAD} \) in order to indicate that \( \texttt{LOAD} \) has write-read access with respect to the register.
By way of convention, the dashed version refers to the new state.

```
LOAD
REG
REG'
i? : IN

reg' = i?
```

$Z$ provides a collection of operators on schemas, some of which are best understood in a rather syntactic way. By way of example we mention two such operators.

- Inclusion: if $S$ is a schema, it is allowed to mention $S$ in the signature part of another schema $T$. This is semantically equivalent to combining the signature parts of $S$ and $T$ and taking the conjunction of their axiom parts.

- Hiding: in order to hide a variable $v$ from a schema $S$ we can write $S \setminus \{v\}$. It is semantically equivalent to removing the variable from the signature part and prefixing the axiom part with a quantification $\exists v : ?$ where $?$ is the type of $v$.

Spivey [61] gives a formal definition of $Z$, using $Z$ as a metalanguage. As it turns out, the meaning of a schema with signature part $\Sigma$ and axiom part $\varphi$ is a so-called variety. For $(\Sigma, \varphi)$ this variety is the collection of structures (= models or heterogeneous algebras) having the signature $\Sigma$ and which satisfy $\varphi$. So for a $\Sigma$-structure $\mathcal{A}$ we have $\mathcal{A} \in \text{meaning}(\Sigma, \varphi) \iff \mathcal{A} \models \varphi$. There is also a concept of typing in $Z$, but this is set-oriented rather than function-oriented and as a result there is only a limited kind of type checking possible. We shall come back to that later.

## 1.8 COLD-K

COLD-K is a wide-spectrum specification language developed at the Philips Research Laboratories in Eindhoven within the framework of the Meteor project. The language has been designed mainly by H.B.M. Jonker, with technical contributions from C.P.J. Koymans, G.R. Renardel de Lavalette,
the author, and to a lesser degree also from J.H. Obst and P.H. Rodenberg. COLD-K is meant to be used throughout several stages of the software development process, including the specification and implementation stages. Actually there is not just one language COLD-K but there is a sequence of subsequent versions, such as COLD-S, COLD-flat, COLD-K, COLD-K-RTL and COLD-1. Amongst these, the COLD-K language plays a special role, since it is a kernel language which is meant to serve as a fixed point in the development of the language. There is a mathematically defined syntax and semantics for COLD-K [53]. Other language versions (notably COLD-1) can be defined in terms of COLD-K, just by adding constructs of a purely syntactic nature. In the remainder of this chapter we shall restrict ourselves to COLD-K.

In order to give an impression of COLD-K, let us do the VDM and Z examples in COLD-K. We assume that we have the sort of natural numbers denoted by Nat and operations on Nat such as leq and add.

\[
\text{FUNC } \text{sub}: \text{Nat} \times \text{Nat} \rightarrow \text{Nat}
\]

\[
\text{AXIOM FORALL } i: \text{Nat}, j: \text{Nat}
( \text{leq}(j,i) \Rightarrow \text{add}(<i,j>) = i )
\]

The second example involves a register which is introduced as a so-called variable function (keyword VAR).

\[
\text{FUNC } \text{reg}: \rightarrow \text{Nat} \text{ VAR}
\]

The LOAD operation becomes a procedure, because of its side-effect. This procedure has modification rights (keyword MOD) with respect to the register reg. In COLD-K there is no built-in construct for indicating a postcondition, but the assertion sub-language includes Harel’s dynamic logic [27]. This logic is employed in the axiom given below. This axiom should be read as follows: in all states and for all i, whenever we reach a new state by executing LOAD(i), the assertion reg = i holds in this new state.

\[
\text{PROC LOAD: Nat } \rightarrow
\]

\[
\text{MOD } \text{reg}
\]

\[
\text{AXIOM FORALL } i: \text{Nat}
( ( \text{LOAD}(i) ) \text{ reg = i } )
\]

In fact the examples above only show definitions which could occur within a COLD-K scheme (= module) which in turn could be part of an entire design. It is outside the scope of this monograph to give a more or less complete introduction to COLD-K; for that we refer to [8], [53] and [62].
Here we would like to point out a few differences with respect to the versions of VDM and Z as discussed in the sections 1.7.2 and 1.7.3.

First of all the COLD-K solution to the problem of functions being partial differs from the solutions of VDM and Z. The assertion language of COLD-K is based on a special version of typed predicate calculus called MPL [53] which at its turn is derived from Scott's E-logic. It is possible to state that an expression $e$ is defined and this is done by means of the postfix operator $!$ which can be used to write $e!$. There are rules which relate this notion of definedness to the built-in equality and to the quantifiers $\forall$ and $\exists$. The problem of the undefined truth-values vanishes because one of these rules says that a predicate applied to an undefined expression just yields $\text{FALSE}$. The point is that predicates are normal mathematical predicates and not Boolean-valued functions.

It is also interesting to note how programming variables and operations with side-effects are described in COLD-K. Programming variables are modelled in COLD-K by having variable functions and variable predicates. The operations which modify them are called procedures and there are mechanisms that regulate the modification rights of procedures with respect to variables. There are several ways of defining procedures: they can be given algorithmically or they can be described axiomatically because dynamic logic is built in to the assertion sub-language. This is a difference with respect to VDM, where there are special constructs with keywords pre and post, rather than a more powerful logic. In Z there is nothing special about states and side-effects at all: most of this is dealt with by having a certain pragmatic style relying on conventions about identifiers such as $i\text{?}$, $i!$ and $\text{reg}'$.

The expressions in COLD-K are strongly typed, which has the important advantage that mechanical type-checking is possible. A syntax- and type-checker exists. This is a difference with VDM and Z, since in the latter languages one can just define any set, e.g. $E \triangleq \{ i : \mathbb{N} \mid \exists j : \mathbb{N} \cdot i = 2 \times j \}$ introduces $E \subseteq \mathbb{N}$ as the set of even numbers. Now one could introduce a function $f : E \to E$ and then to find out that $f(77)$ is wrong requires reasoning about the assertion $\exists j : \mathbb{N} \cdot i = 2 \times j$ which was used to define $E$. In Z there is a rather coarse concept of type checking which cannot detect the problem with $f(77)$ but which could at least find out that $f(-3)$ is nonsense.

Probably the most important difference between COLD-K on the one hand and VDM and Z on the other hand is the presence of powerful constructs for modularisation, parameterisation and designs in COLD-K. The modularisation of COLD-K is based on an algebraic approach to modularisation. The basic module construct is called a class scheme and it consists of a list of definitions of sorts, functions, predicate, procedures and axioms. These can be
combined into larger schemes (= modules) by means of import, export, renaming and by using names for schemes. Furthermore COLD-K provides for parameterisation of schemes over schemes. The work presented in Chapters 2 and 3 has served as a contribution to the design of COLD-K in the sense that the parameterisation of COLD-K schemes is based on \( \lambda \pi \)-calculus and that COLD-K provides constructs for components and designs. In COLD-K one writes \( \text{LAMBDA} \ x \ : \ \text{ITEM OF BODY} \), corresponding with a \( \lambda \pi \)-calculus term \( \lambda x \in \text{ITEM.BODY} \), and one writes \( \text{APPLY PARAM TO ARG} \) corresponding with a \( \lambda \pi \)-calculus term \( \text{(PARAM ARG)} \). The top-level construct which can be denoted in COLD-K is a design. In Jones’ VDM there is nothing comparable to the COLD-K modularisation, parameterisation and designs. In \( Z \) there are schemas comparable to the COLD-K schemes. The parameterisation of \( Z \) is just over sets rather than over entire schemes, whereas there are no such things as components and designs in \( Z \).

Another difference between COLD-K and VDM/\( Z \) is the absence of fixed built-in notions like Booleans, natural number, tuples, sequences, sets and maps in COLD-K. Instead of that, the language provides facilities to define them. In practice such data types are taken from standard library – written in the language, cf. Appendix B of Chapter 4.

COLD-K can be used for expressing programs and specifications, but it is not meant for denoting proofs. Of course, languages to denote proofs exist, ranging from plain English for convincing arguments to \( \lambda \)-typed \( \lambda \)-calculus for automated checking – as in the Automath approach. Yet COLD-K gives rise to proof obligations; more precisely, the principle of black-box correctness requires for each component \( \text{COMP} X : X \_\text{SPEC} = X \_\text{IMPL} \) that \( \Gamma \vdash X \_\text{IMPL} \subseteq X \_\text{SPEC} \) for a certain well-defined \( \Gamma \). In the editor case study of Chapter 5, some proof obligations are treated informally, whereas many others are tacitly dealt with.

After the intermezzo about formal specification techniques of Sections 1.7 and 1.8, we can proceed with the main line of this monograph, viz. the formalisation of design structures (= structure of modules + specifications) and the models of software development based on that. We are now in a position where we can indeed attach ‘redundant’ formal specifications to modules, viz. by using VDM, \( Z \) or COLD-K. Using one of these, we shall present a case study concerning a text editor, which has been developed as an application and illustration of the results of Chapter 2 and Chapter 3.
CHAPTER 1. OVERVIEW AND INTRODUCTION

1.9 Formal specification of a text editor

This section is an overview of Chapter 4 which concerns the formal specification of a multi-buffer and display-oriented text editor. This specification illustrates the use of COLD-K as a specification technique. In particular, the editor specification is modularised and certain parts of it are generic and as such it illustrates the COLD-K algebraic approach to module composition and the λ-calculus. There is no real development process yet in Chapter 4, except for the bottom-up construction of the formal specification, where we mean bottom-up in the DAG sense of Section 1.4.3.

The specification of the editor consists of four parts:

- a library,
- an application domain formalisation: texts and operations on texts,
- models of the available primitives, such as a file system and a video display unit,
- specification of the actual editor including features such as buffer management and keybinding.

When the construction of this specification began, there was already a small library of data type specifications, containing Booleans, natural numbers, sets, sequences, bags etc. Most of the library had been constructed earlier, mainly by H.B.M. Jonkers and the author. The library is given in an appendix of Chapter 4.

The need to formalise the application domain is very typical: almost every specific application domain has its own concepts, notations, conventions and jargon. Before any system specification can be written down, these concepts must be introduced formally. As argued already in Section 1.2.4, each tool must be based on a formal system and/or an underlying theory and for the text editor – considered as a tool – the underlying theory is a formalisation of ‘text’ and operations on texts. For example, in Chapter 4 there is a sort of texts, denoted as Text where each text consist of a sequence of lines. Consider the following text:

```
first line of text
second line
```

Fig 1.4. Example of text.

which is viewed as a sequence of lines where the first line has 18 characters viz. "first line of text" and where the second line has 11 characters, viz.
"second line". An equally valid, but somewhat more abstract approach is to model texts by focusing on their 'contour' only. In this approach the above text is modelled by just the sequence (18,11) alone. Of course there is a kind of forgetful mapping from the first model to the second model in the sense that the information conveyed by the actual characters in the text is lost in the second model. Such approaches at distinct levels of abstraction play a role in the formalisation of the notion of text.

The sort of lines is called Line and each line consists of a sequence of characters (sort Char). The relevant sorts can be denoted in GOLD-K as

\[
\text{SORT Char} \\
\text{SORT Line} \\
\text{SORT Text}
\]

and there are operations to select a line from a text and a character from a line.

\[
\text{FUNC sel: Text \# Nat \rightarrow Line} \\
\text{FUNC sel: Line \# Nat \rightarrow Char}
\]

Starting from this very simple model, a rich collection of operations on texts is defined, including a variety of cut and paste operations. For example, there is a paste operation such that paste(t, u, k, l) means to take a text t and to insert another text u into it immediately before the position with given coordinates (k, l). As a kind of inverse of paste there is an operation cut such that cut(t, k, i, j, l) means taking a text t and cutting out the piece of text beginning at position (k, l) and ending at position (i, j). It yields a pair \((t_1, t_2)\) where \(t_1 = \text{'remaining text'}\) and \(t_2 = \text{'deleted text'}\).

\[
\text{FUNC paste: Text \# Text \# Nat \# Nat \rightarrow Text} \\
\text{FUNC cut: Text \# Nat \# Nat \# Nat \# Nat \rightarrow Text \# Text}
\]

The following picture may give an idea of both cut and paste.
Algebraic laws for the operations are investigated and an example is the equation $\text{paste}(\text{cut}(i, k, l, i, j), k, l) = t$, which holds conditionally. It is shown that this helps in establishing a set of well-understood operations with useful notations. In this way an elegant and useful formalization of the application domain is obtained.

To interface the editor with its environment it is necessary to model the available primitives such as a file system and a video display unit. By way of example we shall have a closer look at the video display unit. Its state space is spanned by two variable functions

\begin{verbatim}
FUCN screen: -> Text VAR
FUCN cursor: -> Nat # Nat VAR
\end{verbatim}

where \texttt{Nat} refers to the natural numbers employed to represent the vertical and horizontal co-ordinates of the cursor. All display operations are described by their effect on either \texttt{screen} or \texttt{cursor} or both. For example \texttt{PROC n1: -> MOD screen, cursor serves for sending a new-line command to the display, thereby possibly modifying the screen and the cursor.}

To describe the operation of the actual editor, the notion of ‘marked text’ is introduced first. A marked text is a composite object that consists of a text and two co-ordinate pairs called mark and dot. The \texttt{dot} serves a kind of ‘current’ location in the text whereas as the \texttt{mark} is a marker that can be put on any position in the text. The specification of the editor is based on a variable map from buffer names to marked texts (sometimes also called ‘buffers’) and on a notion of ‘current’ marked text.

On several occasions in Chapter 4, the technique of invariants is used to describe essential aspects of the editor. There is a so-called ‘text-invariant’ that says that for each marked text in the editor, both the dot and mark are
1.9. SPECIFICATION OF A TEXT EDITOR

positions that exist in the text of that marked text. Another interesting example is the so-called `window-invariant', which is introduced as a predicate \texttt{PRED \textsc{wi}}. It describes the relation between the current marked text on the one hand and the screen and the cursor of the display on the other hand. What \textsc{wi} states in formal terms boils down (in informal terminology) to the statement that the window should correspond with a `look' to the text, if necessary filled with blanks, such that the dot is visible as the cursor. The following picture sketches part of the situation.

![Diagram of window and text relationship]

\textit{Fig 1.6. The relation between window and text.}

The editor supports a complete set of editing commands like \texttt{insert.character}, \texttt{set.mark}, \texttt{beginning.of.line}, \texttt{backward.character}, \texttt{delete.next.character}, \texttt{search.forward}, \texttt{write.named.file}, \texttt{delete.to.killbuffer} etc. The editor specification is completed by a simple keybinding to associate command-invocations with key-strokes.

In fact the specification of the editor itself is only a fragment of the total specification. The relative fractions of the number of lines in each of the four parts mentioned above are about 35%, 20%, 20% and 25% respectively (from these the first 35% are general purpose and usable in any specification case study; the next 20% are reusable within the same application domain, i.e. when `text' plays a role, whereas the remaining 45% are dedicated to this application).

Chapter 4 covers a number of important aspects of a text editor, although the editor described is relatively poor in its bells and whistles. However it is far from trivial and its functionality makes it a usable editor. It covers several interesting features also present in other text editors and in particular in EMACS. For a more detailed discussion and evaluation we refer to the relevant sections at the end of Chapter 4.

Chapter 4 shows how \texttt{\lambda}-calculus and \textsc{COLD-K} can be used as an instrument for the description of a relatively large and complex software system. At the
same time it illustrates a number of general-purpose specification techniques and as such, it can be viewed as a contribution to the advancement of formal specification techniques in general. In the context of the main line of thought of this monograph, the role of Chapter 4 is twofold: firstly, it shows a number of examples of parameterised modules, using the $\lambda r$ mechanism of Chapter 2. Secondly, the resulting specification serves as a starting point for a systematic development process (using the results of Chapter 3) which is described in Chapter 5.

1.10 Systematic design of a text editor

This section is a short summary of Chapter 5 which is a continuation of the editor case study of Chapter 4. The formal specification is taken as a specification and a systematic development process aiming at the actual construction of the editor is undertaken. The top-down model of Chapter 3 is employed. This model turns out to be workable and because the editor design is quite complex, it is shown how the top-down model helps in mastering this complexity. Realistic efficiency considerations and data-reification issues are taken into account. In particular the design is based on the assumption that it is important to economise with respect to memory usage and with respect to communication with the video display unit.

In Chapter 5 there is not just one editor design, but two:

- a design $d_{editor}$ which is mainly concerned with topics such as buffer management, file handling, window management and key binding in terms of a.o. sequences and maps,
- a design $d_{basic}$ which is concerned with efficient implementations of sequences and maps.

These designs can be fitted together with the $\circ$ operation on designs as mentioned in Section 1.6. In this way Chapter 5 illustrates also one of the models of design partition. Chapter 5 focuses on the top-down development of $d_{editor}$, whereas $d_{basic}$ can be found in an appendix of Chapter 5.

In the development of $d_{editor}$ many classical efficiency considerations concerning memory usage, execution time and communication overhead play a guiding role. The important issue of data reification is addressed on several occasions e.g. when choosing a representation of the marked texts. Each text buffer is represented as an array with a gap and a group of pointers and co-ordinate pairs. To describe this, the well-known machinery of abstraction functions and representation invariants is employed. The top-down approach
1.10. DESIGN OF A TEXT EDITOR

is exploited to achieve a separation of concerns on several occasions - e.g.
to separate the buffer-management from the window-management. Let us
have a closer look at the latter example. When programming the editor
operations insert_character, beginning_of_line, backward_character etc. no
details of the window-management play a role. This is made possible by a
postulated component named WI_PACKAGE which is specified to offer a proce-
dure mod_text_restore which re-establishes the window-invariant WI after
an arbitrary modification of the current text buffer. In a later stage of the
development process this postulated component is implemented.

The resulting editor, viewed as a product plus its documentation is in a
number of aspects quite satisfactory. In particular, the editor design is with
respect to most operations, reasonably efficient. It should be noted that
the editor design is quite complex, especially because it is a display-oriented
multi-buffer editor, which means that it is more than just a small toy ex-
ample.

It can be expected that due to its completeness and its component structure,
the editor design will be 'robust' for various forms of design evolution and ef-
ficiency improvements. The theory of correctness-preserving transformations
on designs of Chapter 2 and Chapter 3 is applicable here: in particular, since
the principle of black-box correctness has been adopted, many efficiency im-
provements can take the shape of black-box correctness preserving glass-box
modifications - meaning that there is a 'locality principle' which can yield a
significant reduction of the verification task.

The algorithmic COLD-K texts are translated manually to C to get a working
editor. The details of this translation process are given in an appendix of
Chapter 5.

The editor case study is quite large in view of the fact that it was meant as an
example. This is justified by the need to have a non-toy example illustrating
how the theoretical concepts of \(\lambda\)-calculus, components, designs, algebraic
operations on designs and formal development models can be turned into real
engineering concepts. The fact that the example grew large is a direct con-
sequence of the level at which our concepts apply, viz. programming-in-the-
large, rather than programming-in-the-small. After reaching the milestone
in the evolution of COLD of finishing COLD-K, it was important to have this
kind of an exercise before the design of yet another language version started.
The example has served already for educational purposes in the COLD-K
workshop of the Philips Centre for Software Technology (CST) in November

In Chapters 4 and 5 we employed the COLD-K language (which certainly
was not the only option) and we showed that this is useful as an instrument
when constructing software following a systematic approach and employing formal specifications.

For a more detailed discussion and evaluation we refer to the relevant sections at the end of Chapter 5.

In the context of the main line of thought of this monograph, the role of Chapter 5 is twofold: firstly, it shows a number of examples of designs, thereby illustrating the notions of component and design from Chapter 2. Secondly, Chapter 5 is an application of the results from Chapter 3 and it demonstrates the applicability and usefulness of these results.

1.11 Options for future work

In this section we shall mention a number of topics which could be the subject of further research or of further development.

The first topic concerns the definition of a number of useful language features which possibly could be added to wide-spectrum languages such as COLD-K and which could make the task of constructing large designs easier. Amongst these we have higher-order logic, concurrency and – at a more syntactical level – flexible mechanisms for dealing with user-defined operations and binders.

The second topic concerns the precise definition of the implementation relation $\subseteq$ for wide-spectrum languages – and for COLD-K in particular. For the editor case study of Chapter 4 and 5 we adopted a relation based on signature inclusion and theory inclusion. This is certainly not the only possibility and it would be interesting to have a look at other notions of implementation. Furthermore, the precise conditions that guarantee that the module composition operations of the so-called ‘class-algebra’ of COLD-K are monotonic with respect to $\subseteq$ have not been formulated yet. For an analysis of the problem we refer to [63].

The third topic is the construction of tools supporting the development process based on the instantiation of the approach from Chapters 2 and 3 with a given formalism. We have already given a list of tools in Section 1.2.4 and most of these are relevant when using a wide-spectrum language. We can not give a minimal list of tools which are absolutely necessary, but clearly the attractiveness of a certain formalism is increased by providing a syntax checker, type checker, module library, code generator and a tool to create/manipulate graphical representations.

Finally we mention the topic of practical applications. In view of the fact
that formal techniques are meant to be a mental tool, it is not enough just to develop the mathematical foundations of a language and the theoretical models of software development based on them. It is also necessary to build up experience in using formal methods and although it is inevitable that the first case studies are small toy examples, the next step is to scale them up and seriously attack complex systems. Performing large case studies is a time-consuming and resource-consuming activity but there is more than one yield. Case studies yield feedback on the language, the method and the theory and insight in the nature of the problems which are relevant in practice.

By way of a concluding remark we state that the theory of Chapter 2 and Chapter 3 in combination with a particular formalism (such as the COLD-K language) provides a potential starting point for improving the process of software development. It is also clear that this involves much work of an engineering nature. Chapter 4 and Chapter 5 are a contribution to that, but much remains to be done.
Bibliography

[16] N.G. De Bruijn. A survey of the project Automath, in: J.P. Seldin,


CHAPTER 1. OVERVIEW AND INTRODUCTION


Chapter 2

A Formalisation of Design Structures

2.1 Introduction

This chapter presents a formal theory dealing with the component-wise construction and specification of complex systems, addressing issues of modularisation, parameterisation, abstraction and information hiding. The theory consists of two parts where the first part mainly serves to introduce an extensive formal machinery which is employed in the second part.

The first part of this theory introduces the notion of an algebraic system with associated preorder relation. Furthermore a special version of lambda-calculus is developed which is called $\lambda x$-calculus. More precisely, there is an instance of this calculus for every algebraic system with preorder. The most important characteristic of the calculus is that associated with every abstraction ($\lambda x$ say) there is a so-called parameter-restriction.

The motivation for the introduction of these notions is as follows. An algebraic system can be used to model the modular structuring of systems. In particular, we view module-composition mechanisms as operators of an algebraic system — following [1]. The associated preorder (denoted by $\sqsubseteq$) is used to model the so-called implementation-relation — which is an important relationship between modules and their specifications. We do not partition the modules into two kinds (implementation/specification) but we adopt a single-sorted approach where $\sqsubseteq$ is used to model the implementation and specification roles of modules. This allows for a smooth transition from specification to implementation and in particular we can have the situation that $m_1 \sqsubseteq m_2$ and $m_2 \sqsubseteq m_3$. The reflexivity and transitivity properties of this relation are adopted for a-priori and intuitive reasons [2]. The purpose of
the $\lambda\pi$-calculus is to describe the introduction of names and the abstraction-step taking place when introducing specifications for yet unknown modules. At the same time this $\lambda\pi$-calculus serves as a parameterisation mechanism for modules (this calculus was used for the parameterisation of the design language COLD-K [3], [4]). In this calculus we have abstractions of the form $\lambda x \subseteq R.M(x)$ where $R$ is meant as a restriction upon the possible actual parameters $A$ to which $\lambda x \subseteq R.M(x)$ can be applied. To make the restriction $R$ effective, we adopt a certain mechanism regulating the equalities that can be derived about application-expressions. In particular, there is a partial version of the classical rule ($\beta$) in the sense that $(\lambda x \subseteq R.M(x))A = M(A)$ can only be concluded when it has been shown that $A$ is an implementation of $R$, i.e. $A \subseteq R$. This $\lambda\pi$-calculus is a theory where equations play an important role and just as for classical lambda calculus [5], it is useful to study reduction. In particular the reduction relations $\rightarrow$ and $\rightarrow\rightarrow$ allow for a detailed study of the (modified) rule ($\beta$).

The second part of this theory introduces the notion 'design of a system'. The intuition for this is that a design is a hierarchically-structured and component-wise specified software system. We shall define a black-box description as the specification part of a component and a glass-box description as the implementation part of a component. Although it may be redundant, a black-box description can be useful when it is less complex or easier to read than the glass-box description. Even when the black-box description is not less complex, there may be advantages for having an alternative representation. There is a correctness criterion for components because the two descriptions must be related by the preorder $\subseteq$. We define a design as a collection of components, where some components serve as building blocks for other components. A design need not be finished, but it may correspond to an intermediate stage in a development. In particular, a design may contain components which have no glass-box description yet.

There are non-trivial issues of information hiding that arise in connection with designs. This leads to definitions of black-box correctness (based on the exclusive use of specifications) and glass-box correctness (using implementation knowledge). As it turns out, $\lambda\pi$-calculus can be used to give an interpretation of designs. The reduction relations can be used to give alternative characterisations of black-box correctness and glass-box correctness.

The second part of this theory provides a starting point for a systematic study of transformational development. In particular, a design is built-up from components and the correctness of a complete design follows from the correctness of its components. We shall investigate the precise conditions such that when modifying one component, the correctness of the resulting design follows from the correctness of the modified component. As it
2.2. ALGEBRAIC SYSTEMS WITH PREORDER

turns out, this depends both on the chosen notion of correctness and on
the question whether an implementation module or a specification module is
modified.

In view of the above introduction, we adopt the following structure for this
chapter. In Section 2.2 we shall introduce algebraic systems with preorder. In
Section 2.3 we introduce the $\lambda$-calculus and we study some of its properties.
This covers the first part of the theory. For a given algebraic system with
preorder, we can define the notion of a component. Furthermore we shall
define what it means that a component is correct - in a given context. This
will be done in Section 2.4. Using the notions of component and correctness,
we can define the notion of a design and its correctness. In Section 2.5 we
introduce designs and we study their properties. Also in Section 2.5 we
study a certain class of correctness-preserving transformations of designs.
This covers the second part of the theory.

In Chapter 3 the study of correctness-preserving transformations of designs
will be continued in a somewhat more general setting and with a focus on
the dynamic aspects of the software development process.

2.2 Algebraic systems with preorder

2.2.1 Motivation

We adopt the following minimal set of notions to begin the development of
our theory.

- a set of constructs which are the objects to be created and/or used by
  software developers. For each construct there are two possibilities. A
  construct can be given as primitive, by which we mean that the de-
  veloper can not or should not investigate how it was constructed. So
  a construct is primitive if it can be fitted together from zero compo-
  nents. Alternatively, a construct can be built by fitting together a finite
  nonzero number of constructs.
- composition mechanisms (or 'combinators'), by which we mean ways of
  combining existing constructs into new constructs.
- an implementation relation, corresponding to the possibility that one
  construct is viewed as an implementation of another construct.

As an example of these notions we consider (software) modules. A primitive
module can contain several sort definitions, function definitions, procedure
definitions and axioms. At this level we have module composition mecha-
nisms such as import. To illustrate the implementation relation, we consider two modules $m_1$ and $m_2$ where $m_1$ is an algorithmic description which is characterised axiomatically by $m_2$; then the pair $(m_1, m_2)$ is in the implementation relation. The motivation for the choice of these notions is as follows. First of all it is clear that without constructs or without composition mechanisms no constructive activity is possible at all. We adopt an implementation relation because it seems a very general starting point for a discussion of abstraction and information hiding. It is clear that these notions are essential for applying the paradigm of hierarchical decomposition. The latter paradigm appears frequently in literature dealing with the manageability of the software development process [6] and it certainly is of great value.

The notions of constructs, composition mechanisms and an implementation relation can be formalised by introducing the mathematical concept of an algebraic system. Constructs correspond to terms and composition mechanisms to function symbols. This approach is also chosen in [1], where a so-called 'module-algebra' is studied. We shall choose one kind of algebraic system which exhibits precisely the kind of implementation relation in which we are interested. It would be nice if we could give a single formal definition of some kind of mathematical structure (which we might call a construction system) in which we formalise these notions once and for all. However, one can imagine several kinds of implementation relations and they differ strongly, both in their mathematical properties and in their role. Therefore it seems better to focus on one kind of implementation relation first.

At this point we must be more precise about the nature of the implementation relation. Therefore we must be specific about the kind of constructs we have in mind. As an important case, we think of constructs at the level of (software) modules. A module can be used as a specification but certain modules can also be used as executable implementations. If $m_1$ and $m_2$ are modules such that - viewing $m_1$ as an implementation and viewing $m_2$ as a specification - $m_1$ implements $m_2$, then we say that the pair $(m_1, m_2)$ is in the implementation relation. We can make a list of properties which from an intuitive point of view should hold for this implementation relation [2].

- Each module implements itself, i.e. the implementation relation should be reflexive.
- If $m_1$ implements $m_2$ and $m_2$ implements $m_3$, then $m_1$ should implement $m_3$, i.e. the implementation relation should be transitive.
- Suppose that $m_1$ implements $m_2$. Now if $m_2$ occurs as a subconstruct in a larger construct, $m(m_2)$ say, then $m(m_1)$ - i.e. the construct $m$ containing $m_1$ instead of $m_2$ - should implement $m(m_2)$. It follows that we
shall prefer composition mechanisms which are monotonic with respect
to the implementation relation. Alternative terms for 'monotonic' are:
'compatible' and 'compositional'.

There exist algebraic systems for which a precise definition of the implement-
tation relation has been given. In [7] it has been defined what it means that
\( m_1 \) implements \( m_2 \) for the case where \( m_1 \) is a sequential program and \( m_2 \)
is a pair \((\varphi, \psi)\) consisting of a precondition and a postcondition (the usual
notation for this is \( (\varphi)m_1;\psi)\)). In [8] this has been done for the case where
\( m_1 \) is an implementation of a data type and \( m_2 \) is a specification of a data
type.

2.2.2 Formalisation

We have gathered enough properties of our minimal set of notions in order to
formalise them. After we shall have done the formalisation we shall often not
use the above terminology, but rather adopt a terminology which fits best
to the results of the formalisation. In particular, instead of constructs we
shall have terms and instead of combinatorics we shall have constants (which
act as the primitive constructs) and function symbols. We shall often give
examples which are extremely small, e.g. by taking symbols denoting natural
numbers rather than realistic software modules. Nevertheless, such examples
are related to problems which occur in connection with realistic software
modules also. We shall describe an algebraic system as a structure or model
and we shall adopt several notations and conventions from logic [9].

Definition 2.2.1 An algebraic system with preorder is a quadruple

\[ \mathcal{R} = (A, R, \{F_j | j \in J\}, \{C_i | i \in I\}) \]

\((J, I \text{ index sets})\) where \( A \) is a set (called the domain of \( \mathcal{R} \)), \( R \) is a preorder,
each \( F_j \) is a function and each \( C_i \) is an element (a constant) of \( A \). Recall that a
preorder is a relation which is reflexive and transitive. We assume that the
arity of each \( F_j \) is given as a natural number \( a_j \).

We also allow for many-sorted algebraic systems with preorder. However we
require that there is one domain of interest (\( A \)), on which the preorder \( R \) is
defined. In such a case we shall say that there are secondary domains. In
order to keep things simple, we shall focus our discussion on the single-sorted
case. When necessary, we can always deal with secondary domains somewhat
informally. Often we shall refer to \( R \) as the 'implementation relation'. Clearly
we shall prefer algebraic systems in which each \( F_j \) is a monotonic function,
but in general we shall not require this. The advantage of a monotonic function is that knowledge about the constituents of a composite construct, allows one to infer properties of the composite construct itself. Monotonicity is a relevant issue in this context - but we have no reason here to consider some notion of continuity of functions.

We shall distinguish between the elements of the domain of an algebraic system with preorder, and the terms (consisting of symbols) used to denote these elements.

Definition 2.2.2 The alphabet to be used for constructing the set of terms for a \( R \) as above, consists of the following symbols:

1. function symbols: \( f_j \) (one for each \( F_j \)),
2. constant symbols: \( c_i \) (one for each \( C_i \)),
3. variables: \( x_i \) (one for each \( i \in \mathbb{N} \))

where \( \mathbb{N} \) denotes the set of natural numbers. For this alphabet, the collection of symbols \( \{ c_i \} \cup \{ f_j \mid j \in J \} \cup \{ x_i \mid i \in I \} \) is called the signature of \( R \). We denote it by \( \text{Sig}(R) \).

The function symbols and constant symbols from \( \text{Sig}(R) \) can be combined with variables to build terms in the usual way. Recall that the arity of each function \( F_j \) with function symbol \( f_j \) is given as \( a_j \).

Definition 2.2.3 The set of terms for \( R \), denoted as \( T_R \) is inductively defined by:

1. \( c_i \in T_R \) for all \( i \in I \),
2. \( x_i \in T_R \) for all \( i \in \mathbb{N} \),
3. if \( t_1, \ldots, t_{a_j} \in T_R \) then \( f_j(t_1, \ldots, t_{a_j}) \in T_R \) for all \( j \in J \).

The terms at their turn can be used to build formulae.

Definition 2.2.4 \( (T_R\text{-formulae}) \) The set of atomic \( T_R\text{-formulae} \) is defined as the set of formulae \( P = Q \) and \( P \sqsubseteq Q \) for \( P, Q \in T_R \).

The set of \( T_R\text{-formulae} \) is the smallest set containing \( \top, \bot \), all atomic \( T_R\text{-formulae} \) and which is closed under the usual connectives \( \sqcap, \sqcup, \forall, \rightarrow \) and the quantifier \( \forall \). More precisely, the set of \( T_R\text{-formulae} \) is inductively defined by:

1. \( \top \) and \( \bot \) are \( T_R\text{-formulae} \),
2. if \( \varphi \) is an atomic \( T_R\text{-formula} \), then \( \varphi \) is a \( T_R\text{-formula} \),
3. if \( \varphi \) is a \( T_k \)-formulae, then so is \( \neg \varphi \).
4. if \( \varphi \) and \( \psi \) are \( T_k \)-formulae, then so are \( \varphi \land \psi \), \( \varphi \lor \psi \) and \( \varphi \rightarrow \psi \).
5. if \( x \) is a variable and \( \varphi \) is a \( T_k \)-formula, then \( \forall x(\varphi) \) is a \( T_k \)-formula. \( \Box \)

An (atomic) \( T_k \)-formula is said to be closed if no free variables occur in it. Note that (atomic \( T_k \)-formulae) \( \subseteq \) (\( T_k \)-formulae). Sometimes we shall write just ‘formula’ instead of ‘\( T_k \)-formula’. Now we have the syntactic definition of (atomic) \( T_k \)-formulae, the next step is to define what it means that such a formula is true.

**Definition 2.2.5 Truth in \( \mathbb{R} \)** defined as usual where it is understood that \( \models \) corresponds to \( \models \). When the \( T_k \)-formula \( \varphi \) holds in \( \mathbb{R} \), we denote this by \( \mathbb{R} \models \varphi \). See [9] for the details. In particular, the symbol \( = \) is interpreted by (mathematical) equality on \( A \). Furthermore \( \forall, \exists, \neg, \land, \lor, \rightarrow \) and \( \forall \) correspond with truth, absurdity, negation, conjunction, disjunction, implication and universal quantification respectively. \( T_k \)-formulae which are not closed, are interpreted as implicitly universally quantified. \( \Box \)

In the following table we put together some of the notions introduced so far. We simplify the presentation in the table, in the sense that the entry ‘symbols’ does not show the variables \( x \). The entries for ‘terms’ and ‘atomic formulae’ speak for themselves. The entry ‘statements’ shows the notation for the statement that a \( T_k \)-formula \( \varphi \) holds in \( \mathbb{R} \). The entry ‘model’ is trivial, but we introduce it here already because later we shall add a second column to this table where the entry ‘model’ will be less trivial.

<table>
<thead>
<tr>
<th>symbols</th>
<th>( \text{Sig}(\mathbb{R}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>terms</td>
<td>( T_k )</td>
</tr>
<tr>
<td>atomic formulae</td>
<td>( { P = Q, P \subseteq Q \mid P, Q \in T_k } )</td>
</tr>
<tr>
<td>statements</td>
<td>( \mathbb{R} \models \varphi )</td>
</tr>
<tr>
<td>model</td>
<td>( \mathbb{R} )</td>
</tr>
</tbody>
</table>

Recall that we distinguish between the elements of the domain of an algebraic system with preorder, and the terms, (consisting of symbols) used to denote these elements. We adopt the point of view that a typical software developer manipulates symbols. In fact the developer might not be able to find out for all \( \varphi \) whether \( \mathbb{R} \models \varphi \) or \( \mathbb{R} \not\models \varphi \). Therefore it might seem strange that we nevertheless introduce the actual algebraic system itself. The reason is that for an understanding of the notions of component, black-box description and design we need not to know the precise rules by which the developer reasons
for finding out if $\mathcal{R} \models \varphi$. We simply introduce the algebraic system as a source of true facts. This is somewhat similar to the way in which one simply imports all valid assertions about the basic data types into a proof system for imperative programs [10] (p.61).

### 2.2.3 Examples

In this section we give two extremely simple examples and a realistic, relevant example of an algebraic system with preorder.

As an extremely simple example, consider $\mathcal{R}_1 := \langle \mathbb{N}, \leq, \{+\}, \{0, 1, 2, \ldots\} \rangle$ where $\mathbb{N}$ is the set of natural numbers, $\leq$ is the usual “less than or equal” ordering on $\mathbb{N}$, and $+$ is binary addition. Trivially $\leq$ is a preorder and $+$ is monotonic in both arguments. We assume $\text{Sig}(\mathcal{R}_1) = \{\leq, +, 0, 1, 2, \ldots\}$ and use infix notation for $\times$. We can consider a statement like $\mathcal{R}_1 \models x_0 + 1 \leq x_0 + 2$ (which holds).

As a better example, consider some set $S$ and take its power set as a domain. Assume constants for the subsets of $S$. Let $\mathcal{R}_2 := \langle P(S), \subseteq, \{\cap\}, \{c_i | c_i \subseteq S\} \rangle$. Notice that $\subseteq$ is a preorder and $\cap$ is monotonic in both arguments. This example can have many faces, depending on the choice for the symbols. If we adopt (infix) $\land$ for $\cap$, $\top$ for $S$ and $\bot$ for $\emptyset$ then the example is about proposition logic and $\subseteq$ becomes logical implication. But we can also adopt (mixfix) import $..$ into $..$ for $\cap$, module end for $S$ and module axiom false end for $\emptyset$, making the example look as a simplified kind of module algebra.

Next we shall discuss an example of an algebraic system which is interesting for practical applications, viz. Jonkers’ so-called class algebra CA underlying COLD-K [4] which is related to Bergstra’s Module Algebra [1]. This CA only serves as an example here; in particular, no knowledge about CA or COLD-K is required to read or apply the theory presented in this chapter. The reader who is not interested in COLD-K may very well decide to skip the current section and proceed with Section 2.3. The fact that the theory presented in this chapter is independent of COLD-K has several advantages. We mention two advantages. First, by abstracting from COLD-K, it is easier to study the mathematical properties of the notions in which we are interested. Secondly, the notions component, black-box description and design are applicable to a wider class of design languages.

In COLD-K [4] one has a set of (schemes denoting) so-called class descriptions, which we simply view as constant symbols (the $c_i$). There are a number of function symbols (among which one for import) by which one can combine class descriptions to build larger class descriptions. For class descriptions $P$
and \( Q \) one can define what it means that \( P \) implements \( Q \) and we view such a fact as a pair in the implementation relation. So \( \text{CA} \models P \sqsubseteq Q \) means that \( P \) is an implementation of \( Q \). Formally we view \( \text{CA} \) as an algebraic system with preorder

\[
(\text{CDescription}, \sqsubseteq, \{\Sigma, T, \Box, \bullet, +\}, \{C_i | i \in \text{CDescription}\})
\]

with secondary domains \( \text{CSignature} \) and \( \text{CRenaming} \) containing class signatures and class renamings respectively. We sketch \( \text{CA} \) briefly. \( \text{CDescription} \) is the set of class descriptions modulo semantic equivalence. Each \( C_i \) is the equivalence class of \( i \). \( \Sigma \) is a unary function which takes an element of \( \text{CDescription} \) and yields its class signature. \( T \) is an embedding of class signatures in \( \text{CDescription} \). \( \Box \) (for export) is a binary function which takes a class signature and an element of \( \text{CDescription} \) and yields another (restricted) element of \( \text{CDescription} \). \( \bullet \) (for renaming) is a binary function which takes a class renaming and an element of \( \text{CDescription} \) and yields an element of \( \text{CDescription} \). \( \sqsubseteq \) is the implementation relation. Of course the above presentation of \( \text{CA} \) is somewhat simplified.

### 2.3 Lambda calculus

#### 2.3.1 Introduction

By way of preparation for the introduction of components and designs, we shall extend an arbitrary algebraic system with preorder such that we can deal with abstraction and with the introduction of names in a formal way. We do this by putting a version of lambda calculus 'on top of' the algebraic system.

At first sight it is not obvious that one benefits from introducing lambda calculus in order to describe components and designs. However, we can already at this point indicate that certain connections can be made. First of all lambda calculus offers the possibility to introduce names which may occur within terms and which become bound to some other term (by application). Also for the formalisation of the notion of a design we shall need names, because we want to be able to refer to components. The act of providing the specification of a term instead of the term itself can be viewed as a kind of abstraction. In classical lambda calculus \([8]\), abstraction is done by putting \( \lambda x. \) in front of a term. It is possible to combine both kinds of abstraction in an elegant way, viz. by having lambda abstraction where one puts \( \lambda x \sqsubseteq P \). in front of a term, where \( P \) is a specification of the term to which \( x \) is going to be bound by application.
It will turn out that one such version of lambda calculus can be used for several purposes, viz. describing parameterisation (e.g. of software modules) and for describing the notions of component and design. Furthermore we shall see in Section 2.5 that certain methodologically relevant properties of designs have very simple counterparts in this lambda calculus. The calculus is named $\lambda\pi$-calculus, where the $\pi$ refers to a rule ($\pi$) which resembles the rule ($\beta$) of classical lambda calculus, but which is partial.

Before we embark on the detailed definition of the $\lambda\pi$-calculus, let us give a preview of it by means of a table. We have the same entries as before, but in addition to the column for $\mathbb{R}$, there is a second column for $\lambda\pi$-calculus.

<table>
<thead>
<tr>
<th>symbols</th>
<th>$\text{Sig}(\mathbb{R})$</th>
<th>$\text{Sig}(\mathbb{R}) \cup {\lambda,} }$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terms</td>
<td>$\Lambda_\mathbb{R}$</td>
<td>$\Lambda_\mathbb{R}$</td>
</tr>
<tr>
<td>atomic formulae</td>
<td>${P \equiv Q, P \subseteq Q$ \mid $P, Q \in \mathbb{R}}$</td>
<td>${P \equiv Q, P \subseteq Q$ \mid $P, Q \in \Lambda_\mathbb{R}}$</td>
</tr>
<tr>
<td>statements</td>
<td>$\mathbb{R} \models \varphi$</td>
<td>$\vdash \varphi$</td>
</tr>
<tr>
<td>model</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}^+$</td>
</tr>
</tbody>
</table>

For the entries 'symbols', 'terms' and 'atomic formulae' we have a simple inclusion. This reflects the fact that we shall add new symbols which are used to construct new terms and new atomic formulae. In particular, $\mathbb{R}_\mathbb{R} \subseteq \Lambda_\mathbb{R}$, where $\Lambda_\mathbb{R}$ is a set of so-called lambda terms. The entry 'statements' requires more explanation. The $\lambda\pi$-calculus introduces a collection of rules which have an axiomatic status, and that are adopted for a-priory and intuitive reasons. The simplest statements in this calculus are of the form $\vdash \varphi$. The calculus is such that true facts $\varphi$ from $\mathbb{R}$ automatically yield a statement $\vdash \varphi$ in $\lambda\pi$. This will be achieved, roughly speaking, by adopting the principle $(\mathbb{R} \models \varphi) \Rightarrow (\vdash \varphi)$ as one of the rules. In order to avoid confusion, we must clearly state that $\lambda\pi$-calculus is not about some proof system for $\mathbb{R}$ in the tradition of propositional logic or first-order predicate-logic, say – as the notation $\vdash \varphi$ might suggest. Of course many such proof systems exist already and various soundness and completeness results are known [9]. Instead of that, the theme of $\lambda\pi$ is a certain non-conventional approach to reasoning about lambda abstraction. Because the rules of $\lambda\pi$ have an axiomatic status, the question whether they have a model arises naturally. This question is not completely trivial. It will be addressed in Appendix A where we shall construct a model which will be denoted by $\mathbb{R}^+$. This explains the entry 'model' in the table.

In subsection 2.3.2 we shall define the set of terms and the rules of the $\lambda\pi$-calculus. In subsection 2.3.3 we shall give some derived rules. In subsection
2.3.4 we shall briefly discuss monotonicity and in subsection 2.3.5 we shall define reduction for the $\lambda \pi$-calculus. Finally subsections 2.3.6 and 2.3.7 are about normalisation and confluence for this reduction. In Appendix A we shall construct a model for the $\lambda \pi$-calculus.

2.3.2 Definition of the Calculus

We assume an algebraic system with preorder

$$\mathcal{R} = (A, R, \{F_j \mid j \in J\}, \{G_i \mid i \in I\})$$

with its associated signature $\text{Sig}(\mathcal{R})$ etc. as before. We shall put a version of lambda calculus 'on top of' $\mathcal{R}$ to get the $\lambda \pi$-calculus and in this way the algebraic system $\mathcal{R}$ acts as a parameter of the calculus! The detailed definition of the calculus begins with the definition of the type symbols and the lambda terms.

**Definition 2.3.1** The set of type symbols is inductively defined by

1. $0$ is a type symbol,
2. if $\sigma, \tau$ are type symbols, then so is $(\sigma \rightarrow \tau)$.

These type symbols are usually known as 'simple types'. Roughly speaking, there are two classical approaches for associating such type symbols with terms. The first approach is Curry's approach where the types are not textually part of the terms. Instead of that there is a derivation system for associating types with terms. E.g. $\lambda x.x$ has type $(0 \rightarrow 0)$, but also e.g. $((0 \rightarrow 0) \rightarrow (0 \rightarrow 0))$, whereas $(\lambda z.(xz))(\lambda z.(xz))$ has no type at all. The second approach is Church's approach where type-information is an essential part of the terms. One way of achieving this is to associate explicitly one type with each variable. E.g. $x^0$ and $x^{0\rightarrow 0}$ could be distinct variables of types $0$ and $(0 \rightarrow 0)$ respectively. In this approach $\lambda x^0.x^0$ has type $(0 \rightarrow 0)$ only. Each term has one type and the problematic $(\lambda z.(xz))(\lambda z.(xz))$ is excluded from the set of terms. We use the type symbols along the lines of Church's approach. In order to avoid confusion, we must state explicitly that these simple types are not the main issue of $\lambda \pi$; the types are used in a classical way and their only purpose is to exclude certain problematic terms.

**Definition 2.3.2** We assume infinitely many variables $x^i$ of each type $\tau$ ($i \in \mathbb{N}$). The alphabet to be used for constructing the set of lambda terms consists of the following symbols:
1. function symbols: $f_i$ (one for each $F_i$),
2. constant symbols: $c_i$ (one for each $C_i$),
3. variables: $x^r_i$ (one for each $(r, i)$ with $r \in \text{type symbols}$, $i \in \mathbb{N}$),
4. auxiliary symbols: $\cdot$, $\cdot$, $\lambda, \subseteq$.

The type superscript $x^r_i$ is used with variables only when necessary to avoid ambiguity. Recall that the arity of each function $F_i$ with function symbol $f_j$ is given as $a_j$.

Definition 2.3.3 The set of lambda terms for $\mathbb{R}$, denoted as $\Lambda_\mathbb{R}$, and the type of each lambda term are inductively defined by

1. $x^r_i \in \Lambda_\mathbb{R}$ ($i \in \mathbb{N}$) with type $r$,
2. $c_i \in \Lambda_\mathbb{R}$ ($i \in I$) with type 0,
3. if $P_1, \ldots, P_{a_j} \in \Lambda_\mathbb{R}$ with types 0, then $f_j(P_1, \ldots, P_{a_j}) \in \Lambda_\mathbb{R}$ with type 0,
4. if $P, Q \in \Lambda_\mathbb{R}$ and $P$ is of type $(\sigma \rightarrow r)$ and $Q$ is of type $\sigma$, then $(PQ) \in \Lambda_\mathbb{R}$ with type $r$,
5. if $P, Q \in \Lambda_\mathbb{R}$ where $P$ is of type $\sigma$, $Q$ is of type $r$ and $x^r_i$ does not occur in $P$,
   then $(\lambda x^r_i \subseteq P, Q) \in \Lambda_\mathbb{R}$ with type $(\sigma \rightarrow r)$.

Note that clauses 1., 2. and 3. yield the terms in $T_\mathbb{R}$ if we restrict ourselves to variables of type 0, identifying $x_i$ and $x^0_i$. Also note that in the last clause, $\subseteq$ occurs as a symbol in lambda terms.

We use $\equiv$ to denote syntactical equality. For $P \in \Lambda_\mathbb{R}$, $FV(P)$ denotes the set of free variables of $P$ which is defined as usual.

We shall identify $\alpha$-congruent terms, as is usually done in classical lambda calculus; see e.g. [8] 2.1.12. We adopt the usual variable convention [5] 2.1.13 which is as follows. If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables. To work according to the variable convention means that sometimes a systematic renaming of bound variables (a so-called $\alpha$-conversion) has to take place.

There is an option to simplify the calculus by strengthening the last clause to exclude all variables from the parameter restriction $P$ in $(\lambda x^r_i \subseteq P, Q)$. We have chosen not to adopt this simplification and we shall explicitly study some of the consequences of the fact that such $P$ may contain variables. When it comes to applications of the calculus where these consequences are felt as an unnecessary complication, it is straightforward to adapt the calculus...
2.3. LAMBDA CALCULUS

correspondingly. Unless stated otherwise, we shall assume that parameter
restrictions may contain variables.

As typical elements of $\Lambda_\mathbb{R}$ we shall use $\lambda, B, C$ etc. and their indexed versions.
We shall sometimes be somewhat sloppy in the distinction between actual
variables $(x_0, x_1, \ldots)$ and the typical elements of the set of variables $(x, y, z$
and their indexed versions).

Example 2.3.4 Consider $\mathbb{R}_1 = (\mathbb{N}, \leq, \{+\}, \{0, 1, 2, \ldots\})$ where $\mathbb{N}$ is the set
of natural numbers, $\leq$ is the usual "less than or equal" ordering on $\mathbb{N}$, and 
$+$ is binary addition. Assume an alphabet containing the symbols $+, 0, 1, \ldots$
with the obvious interpretation. The function symbol $+$ is used in infix
notation. We use the same notation for the symbols and for the elements
of the domain of the algebraic system with preorder, but this cannot cause
confusion, since lambda terms can contain symbols only.

(i) $1 + (x_0 + x_1)$ is a lambda term with type 0, provided $x_0, x_1$ of type 0.
(ii) $\lambda x_1 \subseteq 2.x_1$ is a lambda term of type $(0 \rightarrow 0)$, provided $x_1$ of type 0.
(iii) $\lambda x_0 \subseteq (\lambda x_1 \subseteq 2.x_1), (x_01)$ is a lambda term of type $(0 \rightarrow 0) \rightarrow 0)$,
provided $x_0$ of type $(0 \rightarrow 0)$, $x_1$ of type 0.

We must make a remark with respect to the above example where we used
$\mathbb{R}_1$ as an example of an algebraic system with preorder. We shall use this $\mathbb{R}_1$
very often to illustrate some formal and technical detail of our theory and
for that purpose this trivial $\mathbb{R}_1$ is usable. But we must warn the reader that
from a software-engineering point of view it is a misleading example and that
it even may suggest an intuition for $\mathbb{R}$ which is quite wrong. The examples
$\mathbb{R}_2$ and $\mathbb{R}_3$ from Section 2.2.3 are much better in this respect.

Recall that the algebraic system $\mathbb{R}$ has the implementation relation $\mathbb{R}$ and
that we write $\mathbb{R} \models P \subseteq Q$ if the elements denoted by $P$ and $Q$ are in this
relation. Now we want to extend this relation, or more precisely, we want
to compare lambda terms as well. We refer to $\subseteq$ as the 'implementation
relation'. From the terms one can again build formulae $P = Q$ and $P \subseteq Q$.
The following should be compared with definition 2.2.4.

Definition 2.3.5 ($\Lambda_\mathbb{R}$-formulas) The set of atomic $\Lambda_\mathbb{R}$-formulas is defined
as the set of formulae $P = Q$ and $P \subseteq Q$ for $P, Q \in \Lambda_\mathbb{R}$, where we require
that $P$ and $Q$ have the same type.

The set of $\Lambda_\mathbb{R}$-formulas is the smallest set containing $\top, \bot$, all atomic $\Lambda_\mathbb{R}$-
formulas and which is closed under the usual logical connectives $\neg, \wedge, \vee, \rightarrow$
and the quantifier $\forall$.  \[ \square \]
An (atomic) $\Lambda_\mathbb{R}$-formula is said to be closed if no free variables occur in it. Sometimes we write just ‘formula’ instead of ‘$\Lambda_\mathbb{R}$-formula’. Let us say a few words about the interpretation of formulae. Because $T_\mathbb{R} \subseteq \Lambda_\mathbb{R}$, also $(T_\mathbb{R}$-formulae) $\subseteq (\Lambda_\mathbb{R}$-formulae). Clearly $T_\mathbb{R}$-formulae $\varphi$ can be interpreted in $\mathbb{R}$. These are precisely the formulae where all (sub)terms have type 0 and do not contain the symbol $\lambda$. For these formulae the statement $\mathbb{R} \models \varphi$ makes sense and has been introduced already in definition 2.2.5. For other formulae, no interpretation in $\mathbb{R}$ is defined.

**Definition 2.3.6 (Context)**

(i) An assumption is an atomic $\Lambda_\mathbb{R}$-formula, put in square brackets. In particular, if $P, Q \in \Lambda_\mathbb{R}$ where $P$ and $Q$ have the same type then $[P \subseteq Q]$ is an assumption (viz. an inequality), and $[P = Q]$ is an assumption (viz. an equality).

(ii) A context is a finite set set of assumptions, e.g. $\{(P_1 \subseteq Q_1), \ldots, (P_n \subseteq Q_n)\}$. We shall use $\Gamma, \Gamma'$ etc. to denote contexts.

(iii) If $\Gamma = \{\varphi_1, \ldots, \varphi_n\}$ with $\Gamma \neq \emptyset$, then $\Lambda \Gamma$ abbreviates $\varphi_1 \wedge \ldots \wedge \varphi_n$. If $\Gamma = \emptyset$, then $\Lambda \Gamma$ abbreviates $\top$.

(iv) We shall write $\Gamma, [\varphi]$ for $\Gamma \cup \{[\varphi]\}$ and we shall write $[\varphi]$ for $\{[\varphi]\}$.

(v) As a convenient notation we shall write $x \in \Gamma$ when $x$ occurs freely in $\Gamma$. More precisely $x \in \Gamma :\iff \exists P, Q \in \Lambda_\mathbb{R} \cdot ([[P = Q]] \in \Gamma \lor ([P \subseteq Q]) \in \Gamma) \wedge x \in \text{FV}(P) \cup \text{FV}(Q)$. $\square$

In our preview above, we indicated already that the simplest statements in $\lambda \pi$ are of the form $\Gamma \vdash \varphi$. The general form is $\Gamma \vdash \varphi$ which intuitively corresponds with "$\varphi$ follows from the assumptions in $\Gamma$". The technical term for such a statement is a ‘sequent’. We shall adopt a rather syntactic point of view and we shall define a notion of derivability for these sequents, based on a set of rules, which will be given in a Gentzen-style formulation. We shall simultaneously define derivability for sequents of the form $\Gamma \vdash P = Q$ and of the form $\Gamma \vdash P \subseteq Q$.

**Definition 2.3.7** A sequent is a pair $(\Gamma, \varphi)$, written as $\Gamma \vdash \varphi$, where $\Gamma$ is a context and where $\varphi$ is an atomic $\Lambda_\mathbb{R}$-formula. So $\varphi$ is either $P \subseteq Q$ or $P = Q$, $(P, Q \in \Lambda_\mathbb{R}$ and of equal type). We shall write $\vdash \varphi$ for $\emptyset \vdash \varphi$. $\square$

The first group of rules serves for importing facts about $\mathbb{R}$ into the calculus. There is a rule denoted by $(=1)$, expressing that for each monotonic function $F_j$ of $\mathbb{R}$, this monotonicity can be used in the calculus. To state this somewhat more clearly: the algebraic system with preorder has an indexed collection of functions $\{F_j \mid j \in J\}$. Some of these functions may
be monotonic, others are not. As usual, a function $F_j$ is said to be monotonic if $\forall x, y, m, n \left( x \sqsubseteq y \Rightarrow F_j(m, x, n) \sqsubseteq F_j(m, y, n) \right)$. We do not require that all functions $F_j$ are monotonic, but we assume that somehow we can tell the monotonic functions from the non-monotonic ones. Then the rule applies to the monotonic functions only. To simplify the notation, we write $f_j(\ldots, P, \ldots) \sqsubseteq f_j(\ldots, Q, \ldots)$ instead of $f_j(\tilde{M}, P, \tilde{N}) \sqsubseteq f_j(\tilde{M}, Q, \tilde{N})$.

The second rule can be read as follows: whenever a closed formula $\varphi$ is true in the algebraic system $\mathcal{R}$, then $\varphi$ can be considered derivable in the calculus. The statement $\mathcal{R} \models \varphi$ is in the upper part of this rule, which has the effect that it acts as a premise for the applicability of the rule. The sequent $\Gamma \vdash \varphi$ is in the lower part of the rule, which has the effect that it acts as the conclusion of the rule.

**Definition 2.3.8 (Rules $\models_1, \models_2$)**

\[
\begin{align*}
\mathcal{R} \models F_j & \\
\Gamma \vdash P \sqsubseteq Q & \\
\Gamma \vdash f_j(\ldots, P, \ldots) \sqsubseteq f_j(\ldots, Q, \ldots) & (\models_1)
\end{align*}
\]

\[
\begin{align*}
\varphi \text{ closed} & \\
\mathcal{R} \models \varphi & \\
\Gamma \vdash \varphi & (\models_2)
\end{align*}
\]

Note that the well-formedness of the sequents involved in rule $(\models_1)$ requires that both $P$ and $Q$ have type $\emptyset$. By these rules $(\models_1)$ one can import certain relevant facts about $\mathcal{R}$ into the calculus. In this way we need not include a rule which expresses monotonicity of all functions. In subsection 2.3.4 we shall see that a rule expressing monotonicity for all lambda terms would not even be acceptable. The next rule is just necessary for manipulating contexts in the obvious way.

**Definition 2.3.9 (Rule context)**

\[
\begin{align*}
\quad & \\
\Gamma, \varphi & \vdash \varphi
\end{align*}
\]

\[
\quad
\]

Now we proceed with some more rules. Reflexivity and transitivity have
been added as rules; for those closed formulae which can be interpreted in \( \mathcal{R} \), this reflexivity and transitivity are already given by rule \( (\models) \). But for the other formulae, these rules are essentially new. E.g., for \( \mathcal{R}_1 \), as in example 2.3.4, \( \vdash 1 \sqsubseteq 1 \) follows from \( \mathcal{R}_1 \models 1 \sqsubseteq 1 \), but \( \vdash x_0^{(0\to 0)} \sqsubseteq x_0^{(0\to 0)} \) does not. In general \( \mathcal{R} \) does not tell how to compare all lambda terms. But reflexivity and transitivity are properties which are associated with a implementation relation on a-priory grounds and this is formalised by the rules (refl.) and (trans.).

For the '\( \lambda \)-introduction' rules \((\lambda I_1)\), \((\lambda I_2)\) and the 'application rule' \((\text{ap.})\) we add some remarks. Abstraction is covariant with respect to \( \sqsubseteq \) in the second argument and it is contravariant with respect to \( \sqsubseteq \) in its first argument (!). Application is covariant in its first argument. There is no general covariance of application in its second argument which would be monotonicity (we shall discuss this later). These covariance and contravariance issues are important characteristics of the calculus.

**Definition 2.3.10** (Rules refl. trans. \( \lambda I_1, \lambda I_2, \text{ap.} \))

\[
\begin{align*}
\Gamma \vdash P & \quad \text{(refl.)} \\
\Gamma \vdash P_1 & \sqsubseteq P_2, P_2 \sqsubseteq P_3 \quad \text{(trans.)} \\
\Gamma \vdash P_1 & \sqsubseteq P_2 \\
\Gamma, [x \sqsubseteq P] & \vdash Q_1 \sqsubseteq Q_2 \quad (x \notin \Gamma) \\
\Gamma \vdash (\lambda x \sqsubseteq P.Q_1) & \sqsubseteq (\lambda x \sqsubseteq P.Q_2) \quad (\lambda I_1) \\
\Gamma \vdash P_1 & \sqsubseteq P_2 \quad \text{(\lambda I_2)} \\
\Gamma \vdash (\lambda x \sqsubseteq P_2.Q) & \sqsubseteq (\lambda x \sqsubseteq P_1.Q) \\
\Gamma \vdash P_1 & \sqsubseteq P_2 \quad \text{(ap.)} \\
\Gamma \vdash (P_1.Q) & \sqsubseteq (P_2.Q) \\
\end{align*}
\]

\( \square \)

Note that the well-formedness of the sequents involved in rule \((\lambda I_1)\) requires that \( x \notin FV(P) \). Similarly for \((\lambda I_2)\) we must have \( x \notin FV(P_i) \) for \( i = 1, 2 \). The definition of the next rule requires the notion of substitution.
2.3. LAMBDA CALCULUS

Definition 2.3.11 (Substitution) Let \( x_i \) be a variable and \( P \) a term such that \( x_i \) and \( P \) have the same type. Then we define the substitution operator \([z_i := P] \) inductively by

1. \( x_i[x_i := P] \equiv P \)
2. \( x_i[x_j := P] \equiv x_i \ (i \neq j) \)
3. \( f_j(P_1, \ldots, P_n)[x_i := P] \equiv f_j(P_1[x_i := P], \ldots, P_n[x_i := P]) \)
4. \( (Q_1 Q_2)[x_i := P] \equiv ((Q_1[x_i := P])(Q_2[x_i := P])) \)
5. \( (\lambda x_i \subseteq Q_1 Q_2)[x_i := P] \equiv (\lambda x_i \subseteq Q_1[x_i := P], Q_2[x_i := P]) \)

Note that for clause 5. above \( x_i \neq x_j \) by the variable convention and that \( x_i \not\in \text{FV}(P) \) also by the variable convention. In this way we avoid clashes of free and bound variables and violation of the restriction that a variable bound by a \( \lambda \) does not occur in the parameter restriction of that same \( \lambda \).

The next rule is called \((\pi)\) and it resembles the well-known rule \((\beta)\) of classical lambda calculus, but is partial, by which we mean that contraction is conditional. In the applicability condition of the rule \((\pi)\) an actual parameter is compared with a parameter restriction. Here is the very essential difference between \( \lambda \pi \) and classical typed \( \lambda \)-calculi: the parameter restrictions are not used to restrict the set of well-formed terms, but instead of that they are used to regulate the contractions.

Definition 2.3.12 (Rule \( \pi \))

\[
\frac{\Gamma \vdash P_1 \subseteq P_1}{\Gamma \vdash (\lambda x \subseteq P_1) Q_2 = Q[x := P_2]} \quad (\pi)
\]

Note that the property that a variable bound by a \( \lambda \) does not occur in the parameter restriction of that same \( \lambda \), is preserved by the conversion from \((\lambda x \subseteq P_1 Q_2)[x := P_3] \) to \( Q[x := P_3] \). Finally we have a rule that makes \( \subseteq \) into a partial order and a general substitution rule. We write \( \psi(P) \) to denote \( \psi[x := P] \) and \( \psi(Q) \) to denote \( \psi[x := Q] \). We add a remark about the substitution rule below. In the transition from \( \psi(P) \) to \( \psi(Q) \), it is not required that all occurrences of \( P \) are replaced by \( Q \).
CHAPTER 2. A FORMALISATION OF DESIGN STRUCTURES

Definition 2.3.13 (Rules $\Rightarrow I$, subst.)

\[
\begin{align*}
\Gamma \vdash P_1 \subseteq P_2 \\
\Gamma \vdash P_2 \subseteq P_1 \\
\hline
\Gamma \vdash P_1 = P_2 \\
\Gamma \vdash \varphi(P) \\
\Gamma \vdash P = Q \\
\hline
\varphi(Q) \\
\hline
\end{align*}
\]

Now we present the intuition behind the rules $(\lambda I_1)$, $(\lambda I_2)$ and $(\pi)$. Each lambda term $(\lambda x \subseteq P.Q)$ can be viewed as a function having a restriction concerning its argument. Therefore it is reasonable that the evaluation of an application term cannot take place unless the argument provably meets this restriction (rule $\pi$). The rules $(\lambda I_1)$ and $(\lambda I_2)$ describe the conditions under which one function can be viewed as the implementation of another function. First of all, two functions with the same argument restriction are in the implementation relation if for every acceptable argument their results are in the implementation relation (rule $(\lambda I_1)$). Secondly, if two functions have equal function bodies but the restriction of one function is weaker than the restriction of the other function, then the function with the weakest restriction implements the other function (rule $(\lambda I_2)$). The fact that in the rule $(\lambda I_1)$ an assumption $[x \subseteq P]$ is discharged is motivated as follows: by the condition in the rule $(\pi)$ we know that whatever is going to be substituted for $x$ will meet the restriction $x \subseteq P$ and therefore it is reasonable that the assumption $[x \subseteq P]$ can be used when comparing the function bodies $Q_1$ and $Q_2$.

Definition 2.3.14 $(\lambda \pi)$ Derivability for sequents is defined inductively by:

1. If $\varphi$ is closed and $\mathcal{R} \models \varphi$, then $\Gamma \vdash \varphi$ is derivable (by rule $(\rho_\pi)$). If $\Gamma \vdash \varphi$ is the conclusion of a rule from (context, refl.), then $\Gamma \vdash \varphi$ is derivable.

2. If $\Gamma \vdash \varphi$ is the conclusion of a rule from ($\Rightarrow I$, trans., $\lambda I_1$, $\lambda I_2$, ap., $\pi$, $\Rightarrow I$, subst.) and all premises of this rule are derivable, then $\Gamma \vdash \varphi$ is derivable, provided that the monotonicity condition and the variable condition are satisfied for the cases of ($\Rightarrow I_1$ and $(\lambda I_1)$ respectively.

We write $\Gamma \vdash \varphi$ if $\Gamma \vdash \varphi$ is derivable.
2.3. LAMBDA CALCULUS

The typing of terms should not be confused with the restrictions associated with each \( \lambda \). Actually the correctness with respect to the typing has been dealt with by the definition of the set of lambda terms, whereas the restrictions associated with each \( \lambda \) play a role in the calculus. E.g. for \( \mathbb{N}_1 \) as in example 2.3.4, \((\lambda x_1 \subseteq 2.x_1)(\lambda x_1 \subseteq 1.x_1 \rightarrow x_1)\) is not a lambda term but \((\lambda x_1 \subseteq 2.x_1)3\) is a lambda term. In the latter term the effect of the restriction \( \subseteq 2 \) is that we cannot apply the rule \( (\pi) \) to this term.

Let us point out explicitly that \( \lambda \omega \)-calculus and \( \lambda \)-typed \( \lambda \)-calculus are not just almost the same. There are fundamental differences between the nature of the implementation relation \( \subseteq \) of \( \lambda \omega \)-calculus and the typing relation \( \vdash \) of \( \lambda \)-typed \( \lambda \)-calculus. One might be tempted to think that it is just a matter of notation and that instead of writing \( \lambda x \subseteq P.Q \) we could as well replace the symbol \( \subseteq \) by the symbol \( \vdash \) to get \( \lambda x : P.Q \) which looks the same as a term of \( \lambda \)-typed \( \lambda \)-calculus. The difference is that \( \subseteq \) as a relation between terms is reflexive and transitive, whereas in \( \lambda \)-typed \( \lambda \)-calculus the typing relation \( \vdash \) is neither reflexive nor transitive.

**Example 2.3.15** Consider two terms \( P_1, P_2 \) of equal type. Let \( \Gamma \) be a context. Then we have

\[
\Gamma \vdash P_1 \equiv P_2 \Rightarrow \Gamma' \vdash P_1 \equiv P_2,
\]

which we show as follows. First, by rule (refl.), \( \Gamma \vdash P_1 \equiv P_1 \) and this can be combined with \( \Gamma \vdash P_1 = P_2 \) by rule (subst.) when we replace the second occurrence of \( P_1 \) in \( P_1 \equiv P_2 \) by \( P_2 \). This yields \( \Gamma \vdash P_1 \equiv P_2 \). The result of this example can be viewed as a derived rule of the calculus. Similarly

\[
\Gamma \vdash P_1 = P_2 \Rightarrow \Gamma \vdash P_3 \equiv P_3,
\]

which we show again with (refl.) and (subst.). The latter result can be viewed as a second derived rule of the calculus. \( \square \)

**Example 2.3.16** Consider \( \mathbb{N}_1 = (\mathbb{N}, \leq, \{+, \}, \{0, 1, 2, \ldots \}) \). Define the term \( P \) by

\[
P \equiv \lambda x_0 \subseteq 1.((\lambda x_1 \subseteq x_0 + 1.x_1)(x_0 + x_0)).
\]

By rule (context) we have \([x_0 \subseteq 1] \vdash x_0 \subseteq 1\). Because in \( \mathbb{N}_1 \) the function \(+\) is monotonic, we have from \((\preceq)\) that \([x_0 \subseteq 1] \vdash x_0 + x_0 \subseteq x_0 + 1 \) and therefore we can apply rule \((\pi)\) to get \([x_0 \subseteq 1] \vdash (\lambda x_1 \subseteq x_0 + 1.x_1)(x_0 + x_0) = x_0 + x_0\). By the first derived rule of the previous example this yields \([x_0 \subseteq 1] \vdash (\lambda x_1 \subseteq x_0 + 1.x_1)(x_0 + x_0) \equiv x_0 + x_0\) and similarly by the second derived rule of the previous example \([x_0 \subseteq 1] \vdash x_0 + x_0 \subseteq (\lambda x_1 \subseteq x_0 + 1.x_1)(x_0 + x_0)\).
Now we can apply rule \((\lambda I_1)\) twice and finally use rule \((-I)\) to get \(\vdash P = \lambda x_0 \in 1. x_0 + x_0\). \(\square\)

In Appendix A we study a model for the calculus of this section. To summarise the results of that appendix: assuming certain restrictions upon \(\mathcal{R}\), there is a model \(\mathcal{R}^+\) such that for atomic formula \(\varphi\) in the language of \(\mathcal{R}\) we have \(\mathcal{R} \models \varphi \iff \mathcal{R}^+ \models \varphi\) and furthermore for arbitrary \(\varphi\)

\[\Gamma \vdash \varphi \iff \mathcal{R}^+ \models \bigwedge \Gamma \rightarrow \varphi.\]

### 2.3.3 Derived rules

We shall list a number of lemmas that can be viewed as derived rules. The first lemma is very simple and it shows that we can perform a kind of \(\alpha\)-conversion at the level of sequents.

**Lemma 2.3.17** If \(\Gamma \vdash \varphi\) then for fresh \(y\), \(\Gamma[x := y] \vdash \varphi[x := y]\).

**Proof.** The proof is by induction over the length of the derivation of \(\Gamma \vdash \varphi\).

- If \(\Gamma \vdash \varphi\) is a direct consequence of rule \(\models\) with \(\varphi \equiv f(A) \sqsubseteq f(B)\) and \(\Gamma \vdash A \sqsubseteq B\), then by (i.h.) \(\Gamma[x := y] \vdash A[x := y] \sqsubseteq B[x := y]\). Apply rule \(\models\) again.
- If \(\Gamma \vdash \varphi\) is a direct consequence of \(\mathcal{R} \models \varphi\) by rule \(\models\), then \(\varphi\) is closed and hence \(\varphi \equiv \varphi[x := y]\). Apply rule \(\models\).
- If \(\Gamma \vdash \varphi\) is \(\Gamma \vdash (\lambda x \sqsubseteq R. S_1) \sqsubseteq (\lambda x \sqsubseteq R. S_2)\) as a direct consequence of rule \((\lambda I_1)\) with \(\Gamma, \{x \sqsubseteq R\} \vdash S_1 \sqsubseteq S_2\) and \(x \notin \Gamma\), then we distinguish two cases. If \(x \equiv x\), then \(x\) does not occur in \(\Gamma \vdash \varphi\) and we are done immediately. Otherwise use (i.h.) and apply rule \((\lambda I_1)\) again.
- Other rules: analogously. \(\square\)

**Lemma 2.3.18** (Weakening) If \(\Gamma \vdash \varphi\), then \(\Gamma, [\psi] \vdash \varphi\).

**Proof.** The proof is by induction over the length of the derivation of \(\Gamma \vdash \varphi\).

- If \(\Gamma \vdash \varphi\) is a direct consequence of rule \(\models\) with \(\varphi \equiv f(A) \sqsubseteq f(B)\) and \(\Gamma \vdash A \sqsubseteq B\), then by (i.h.) \(\Gamma, [\psi] \vdash A \sqsubseteq B\). Apply rule \(\models\) again.
- If \(\Gamma \vdash \varphi\) is a direct consequence of \(\mathcal{R} \models \varphi\) by rule \(\models\), then so is \(\Gamma, [\psi] \vdash \varphi\).
- If \(\varphi \in \Gamma\), then \(\Gamma, [\psi] \vdash \varphi\) by rule \((\text{context})\) again.
2.3. LAMBDA CALCULUS

- If \( \varphi \equiv (\lambda x \in R.S_1) \subseteq (\lambda x \in R.S_2) \) and \( \Gamma \vdash \varphi \) by rule \((\lambda I)\), then we know \( \Gamma, [x \in R] \vdash S_1 \subseteq S_2 \) and \( x \notin \Gamma \).

If \( x \in \text{FV}(\psi) \) then we substitute a fresh \( z \) for \( x \) (using lemma 2.3.17), yielding \( \Gamma, [x \in R] \vdash S_1[z := z] \subseteq S_2[z := z] \). Apply (i.h.) and rule \((\lambda I)\).

Otherwise we have \( x \notin \text{FV}(\psi) \) and by (i.h.) \( \Gamma, [\psi], [x \in R] \vdash S_1 \subseteq S_2 \), so by rule \((\lambda I)\) again we have \( \Gamma, [\psi] \vdash (\lambda x \in R.S_1) \subseteq (\lambda x \in R.S_2) \).

- Other rules: analogously.

\[ \square \]

Lemma 2.3.19 If \( \Gamma \vdash \varphi \), then \( \Gamma, [x := P] \vdash \varphi|x := P| \).

Proof. The proof is by induction over the length of the derivation of \( \Gamma \vdash \varphi \).

- If \( \Gamma \vdash \varphi \) is \( \Gamma \vdash f(A) \subseteq f(B) \) because \( \Gamma \vdash A \subseteq B \) for \( f \) corresponding with a monotonic function, then by (i.h.) \( \Gamma, [x := P] \vdash A[x := P] \subseteq B[x := P] \). Use rule \((=I)\) again.

- If \( \Gamma \vdash \varphi \) is a direct consequence of \( \mathbb{R} \vdash \varphi \), then note that (since \( \varphi \) closed) \( \varphi \equiv \varphi|x := P| \). Use rule \((=I)\) again. ¹

- If \( \varphi \in \Gamma \), then \( \varphi| [x] \in \Gamma| [x] \), so apply rule (context) again.

- If \( \varphi \equiv (R \subseteq R) \), apply rule (refl.) again.

- If \( \varphi \equiv (P_1 \subseteq P_2) \) because \( \Gamma \vdash P_1 \subseteq P_1 \) and \( \Gamma \vdash P_2 \subseteq P_3 \) by rule (trans.), then by (i.h.) \( \Gamma| [x] \vdash P_1[x] \subseteq P_2[x] \) and \( \Gamma| [x] \vdash P_2[x] \subseteq P_3[x] \). Apply rule (trans.) again.

- If \( \Gamma \vdash \varphi \) is \( \Gamma \vdash (\lambda y \in R.Q_1) \subseteq (\lambda y \in R.Q_2) \) as a direct consequence of rule \((\lambda I)\) with \( \Gamma, [y \in R] \vdash Q_1 \subseteq Q_2 \) and \( y \notin \Gamma \), then we distinguish two cases.

  If \( x = y \) then \( x \notin \Gamma \) and \( x \) does not occur freely in \( \Gamma \vdash \varphi \). Therefore \( \Gamma, [x := P] \vdash \varphi|x := P| \) is just the same as \( \Gamma \vdash \varphi \).

  If \( x \neq y \) then two possibilities arise. The simplest possibility is that \( y \notin P \). Then we can apply (i.h.) to get \( \Gamma, [x := P], [y \in R|x := P] \vdash Q_1[z := P] \subseteq Q_2[z := P] \). Now apply rule \((\lambda I)\) again.

  If \( y \in P \) then we first substitute some fresh \( x \) for \( y \) to get \( \Gamma, [x \in R] \vdash Q_1[y := x] \subseteq Q_2[y := x] \). Now use (i.h.) and apply rule \((\lambda I)\) again.

- Other rules: analogously.

\[ \square \]

Theorem 2.3.20 (Cut-rule) If \( \Gamma \vdash \varphi \), and \( \Gamma, [\varphi] \vdash \psi \), then \( \Gamma \vdash \psi \).

¹here is a reason why we require \( \varphi \) closed in \( \vdash \)
Proof. Induction over the length of the derivation of $\Gamma_1[\varphi] \vdash \psi$.

- If $\Gamma_1[\varphi] \vdash \psi$ is $\Gamma_1[\varphi] \vdash f(A) \subseteq f(B)$ because $\Gamma_1[\varphi] \vdash A \subseteq B$ for $f$ corresponding with a monotonic function, then by (i.h.) $\Gamma \vdash A \subseteq B$. Use rule (\[eq3\]) again.
- If $\Gamma_1[\varphi] \vdash \psi$ is a direct consequence of $\mathcal{R} \models \psi$, then use (\[eq2\]) again.
- If $\psi \in \Gamma$, then $\Gamma \vdash \psi$. If $\psi \equiv \varphi$ then $\Gamma \vdash \varphi$.
- If $\psi \equiv (\lambda x \subseteq R.S_1) \subseteq (\lambda x \subseteq R.S_2)$ and $\Gamma_1[\varphi] \vdash \psi$ by rule (\[eq1\]), then we know $\Gamma_1[\varphi], [x \subseteq R] \vdash S_1 \subseteq S_2$ and $x \not\in (\Gamma_1[\varphi])$ so certainly $x \not\in \Gamma$. By (i.h.) $\Gamma, [x \subseteq R] \vdash S_1 \subseteq S_2$ so by rule (\[eq1\]) we have $\Gamma \vdash (\lambda x \subseteq R.S_1) \subseteq (\lambda x \subseteq R.S_2)$.
- If $\psi \equiv \psi(Q)$ and $\Gamma_1[\varphi] \vdash \psi(Q)$ by rule (subst.), then we know $\Gamma_1[\varphi] \vdash \psi(P)$ and $\Gamma_1[\varphi] \vdash P = Q$ for some $P$. By (i.h.) $\Gamma \vdash \psi(P)$ and $\Gamma \vdash P = Q$ so by rule (subst.) again we have $\Gamma \vdash \psi(Q)$.
- Other rules: analogously.

Remark 2.3.21 In earlier versions of the calculus we employed a much more powerful rule (\[eq1\]), saying $\mathcal{R} \models \Lambda \Gamma \rightarrow \varphi \Rightarrow \Gamma \cup \Gamma' \vdash \varphi$ but this caused problems when proving the lemmas of this section. In particular, for the proof of the cut-rule we would have to use the model construction $\mathcal{R}^\uparrow$.

Lemma 2.3.22 If $\Gamma \vdash P = Q$, then $\Gamma \vdash R[x := P] = R[x := Q]$.

Proof. We want to use the rule (subst.) and we take $\varphi(P) \equiv (R[x := P] = R[x := P])$ which we denote as $\varphi(P) \equiv (R(P) = R(P))$. Clearly $\Gamma \vdash \varphi(P)$. Now consider the second occurrence of $P$ and apply rule (subst.). We obtain $\Gamma \vdash \varphi(Q)$, which is $\Gamma \vdash R(P) = R(Q)$.

Lemma 2.3.23 (Generalised cut-rule). If $\Gamma \vdash \varphi_1, \ldots, \Gamma \vdash \varphi_n$ and $\Gamma, [\varphi_1], \ldots, [\varphi_n] \vdash \psi$ then $\Gamma \vdash \psi$.

Proof. By iterated application of the cut-rule.

We listed a number of lemmas about the $\lambda\pi$-calculus above. For our discussion of components and designs in later sections of this chapter, one can simply accept the propositions of these lemmas as additional rules of the $\lambda\pi$-calculus.
2.3. LAMBDA CALCULUS

2.3.4 Monotonicity

We now give a small derivation that shows that even if the functions \( f_j \) are monotonic, there is a kind of consistency problem if we would adopt a rule expressing monotonicity for all lambda terms. The rule \((\lambda I_2)\) plays a key role in this matter. By monotonicity we mean the following rule.

**Definition 2.3.24** (Rule mon.)

\[
\frac{\Gamma \vdash Q_1 \sqsubseteq Q_2}{\Gamma \vdash (PQ_1) \sqsubseteq (PQ_2)} \quad (\forall \lambda \pi!)
\]

\[\square\]

**Example 2.3.25** Consider the lambda term \((\lambda x_0 \sqsubseteq P.(\lambda x_1 \sqsubseteq x_0.x_1))\), then we would have the following derivation - when adopting (mon.). Assume \( Q_1, Q_2 \in A_\otimes \) such that \( \vdash Q_1 \sqsubseteq Q_2 \sqsubseteq P \). By assuming monotonicity we have

\[
\vdash (\lambda x_0 \sqsubseteq P.(\lambda x_1 \sqsubseteq x_0.x_1))Q_1 \sqsubseteq (\lambda x_0 \sqsubseteq P.(\lambda x_1 \sqsubseteq x_0.x_1))Q_2
\]

and from this we obtain by the rule \((\pi)\)

\[
\vdash (\lambda x_1 \sqsubseteq Q_1.x_1) \sqsubseteq (\lambda x_1 \sqsubseteq Q_2.x_1).
\]

But by the rule \((\lambda I_2)\) we have

\[
\vdash (\lambda x_1 \sqsubseteq Q_1.x_1) \sqsubset (\lambda x_1 \sqsubset Q_2.x_1)
\]

and hence

\[
\vdash (\lambda x_1 \sqsubseteq Q_1.x_1) = (\lambda x_1 \sqsubseteq Q_2.x_1).
\]

This shows that if we assume monotonicity for all terms, we would have that \( Q_1 \) plays no role any more in \( \lambda x_1 \sqsubseteq Q_1.x_1 \).

\[\square\]

Under certain conditions and with certain modifications it is possible to adopt the rule (mon.) however. We sketch one workable approach, which yields a slightly different calculus, \( \lambda \pi_{mon} \), say. It can be done only with an obvious additional requirement on \( \mathbb{R} \).

**Definition 2.3.26** \( \lambda \pi_{mon} \) Assume an algebraic system as before \( \mathbb{R} \) where furthermore all functions \( f_j \) are monotonic with respect the preorder \( R \). Define a restricted set of lambda terms \( A_{\mathbb{R}_{\text{mon}}} \) which is like \( A_\otimes \) except for the additional restriction that parameter restrictions contain no variables at
CHAPTER 2. A FORMALISATION OF DESIGN STRUCTURES

all. The calculus $\lambda\pi_{\text{mon}}$ is about formulae and sequents based on $\lambda\pi_{\text{mono}}$. It has the same rules as $\lambda\pi$ and in addition to these also the rule (mon.).

A model $\mathcal{M}_{\text{mon}}^+$ can be constructed along the same lines as $\mathcal{M}^+$, except that the function domains may contain monotonic functions only. For most of the theory about components and designs later in this chapter, there is hardly any difference between $\lambda\pi$ and $\lambda\pi_{\text{mon}}$.

2.3.5 Reduction

In classical lambda calculus every term $(\lambda x.Q)R$ can be contracted and therefore such a term is called a redex. In the $\lambda\pi$-calculus it is not the case that every term $(\lambda x \in P.Q)R$ can be contracted. It follows that for the $\lambda\pi$-calculus some care is needed in using the word redex. We reserve the word redex for those terms which can be contracted (in a given context). We also introduce the term candidate-redex. The following should be contrasted with [5] 3.1.8.

Definition 2.3.27 A candidate-redex is a term $(\lambda x \in P.Q)R$. A candidate-redex $M \equiv (\lambda x \in P.Q)R$ can be contracted (is a redex) in context $\Gamma$ if $\Gamma \vdash R \in P$. In this case $Q[x := R]$ is called a contractum of $M$.

The following definition can be viewed as a reformulation of [5] 3.1.5 and 3.1.17 for the $\lambda\pi$-calculus.

Definition 2.3.28 The relation $\rightarrow$ is defined inductively by:

1. $\Gamma \vdash R \in A \Rightarrow \Gamma \vdash (\lambda x \in A.B)R \rightarrow B[x := R]$,
2. $\Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash f_1(\ldots, M, \ldots) \rightarrow f_1(\ldots, N, \ldots)$,
3. $\Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash ZM \rightarrow ZN$,
4. $\Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash MZ \rightarrow NZ$,
5. $\Gamma \vdash P \rightarrow Q \Rightarrow \Gamma \vdash (\lambda x \in P.M) \rightarrow (\lambda x \in Q.M)$,
6. $\Gamma, [z \in P] \vdash M \rightarrow N, z \notin \Gamma \Rightarrow \Gamma \vdash (\lambda x \in P.M) \rightarrow (\lambda x \in P.N)$.

A reduction (path) is a sequence $\Gamma \vdash M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow \ldots$

The following definition can be viewed as a reformulation of (part of) [5] 3.1.5 for the $\lambda\pi$-calculus.

Definition 2.3.29 The relation $\rightarrow$ is defined inductively by:
2.3. LAMBDA CALCULUS

1. $\Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash M \rightarrow N$,
2. $\Gamma \vdash M \rightarrow M$,
3. $\Gamma \vdash M \rightarrow N$, $\Gamma \vdash N \rightarrow L \Rightarrow \Gamma \vdash M \rightarrow L$,

and the relation $=_\sigma$ is defined inductively by:

1. $\Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash M =_\sigma N$,
2. $\Gamma \vdash M =_\sigma N \Rightarrow \Gamma \vdash N =_\sigma M$,
3. $\Gamma \vdash M =_\sigma N$, $\Gamma \vdash N =_\sigma L \Rightarrow \Gamma \vdash M =_\sigma L$. $\square$

The following lemma can be viewed as a justification of the definitions of $\rightarrow$, $\rightarrow$ and $=_\sigma$.

Lemma 2.3.30

(i) $\Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash M = N$,
(ii) $\Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash M = N$,
(iii) $\Gamma \vdash M =_\sigma N \Rightarrow \Gamma \vdash M = N$.

Proof. (i) The proof is by induction over the definition of $\rightarrow$.

1. $\Gamma \vdash (\lambda x \subseteq P.Q)R \rightarrow Q[x := R]$. Use rule $(\pi)$.
2. $\Gamma \vdash f_j(\ldots, M, \ldots) \rightarrow f_j(\ldots, N, \ldots)$ because $\Gamma \vdash M \rightarrow N$. By (i.h.)
   $\Gamma \vdash M = N$. Use rule (subst.).
3. $\Gamma \vdash ZM \rightarrow ZN$ because $\Gamma \vdash M \rightarrow N$. By (i.h.) $\Gamma \vdash M = N$. Use rule (subst.).
4. $\Gamma \vdash ZM \rightarrow ZN$ because $\Gamma \vdash M \rightarrow N$. As 3.
5. $\Gamma \vdash (\lambda x \subseteq P.M) \rightarrow (\lambda x \subseteq Q.M)$ because $\Gamma \vdash P \rightarrow Q$. By (i.h.)
   $\Gamma \vdash P = Q$. Use rule (subst.).
6. $\Gamma \vdash (\lambda x \subseteq P.M) \rightarrow (\lambda x \subseteq P.N)$ because $\Gamma,[x \subseteq P] \vdash M \rightarrow N$. By
   (i.h.) and the rules (refl.) and (subst.) $\Gamma,[x \subseteq P] \vdash M \subseteq N$. By
   rule $(M_f)$ $\Gamma \vdash (\lambda x \subseteq P.M) \subseteq (\lambda x \subseteq P.N)$. In the same way we get
   $\Gamma \vdash (\lambda x \subseteq P.N) \subseteq (\lambda x \subseteq P.M)$. Finally use rule ($\equiv I$).

(ii) The proof is by induction over the definition of $\rightarrow$.

1. Use (i),
2. Use reflexivity.
3. Use (i.h.) and transitivity.
(iii) The proof is by induction over the definition of $=_r$.

1. Use (ii),
2. Use (i.h.) and rule (subst.),
3. Use (i.h.) and transitivity.

\[ \square \]

2.3.6 Normalisation

A very desirable property of a notion of reduction is the strong normalisation property, denoted by SN. We write $\text{SN}(M)$ if $M$ strongly normalises by which we mean that it does not have an infinite reduction path. We say that SN holds if $\text{SN}(M)$ for all terms $M$. Classical untyped $\lambda$-calculus does not have this property, and the famous counter-example is $(\lambda x. (xx))(\lambda x. (xx))$ which reduces to itself. In $\lambda \pi$-calculus there is a system of simple types which excludes such problematic terms and as it turns out, this is sufficient to get SN for our notion of reduction. Note that this is not completely trivial, because of the presence of parameter restrictions: in these parameter restrictions also reduction steps can be done. Although typically most of these parameter restrictions will eventually disappear themselves by reductions, this idea cannot be considered as a proof of the fact that the parameter restrictions are harmless.

The remainder of this section is about proving SN. The reader may want to skip this proof and in that case he can proceed with Section 2.3.7. We write $\Lambda_\sigma$ to denote the set of terms from $\Lambda_\pi$ with type $\sigma$. We write $\Lambda_\sigma^c$ to denote the closed terms from $\Lambda_\sigma$.

Definition 2.3.31 We define $\lambda \beta$ as the calculus with terms from $\Lambda_\pi$ and with all rules of $\lambda \pi$-calculus but for the rule $(\pi)$ which has been replaced by the rule $(\beta)$:

\[ (\beta) \quad \vdash (\lambda x \subseteq P_1 Q) P_2 \rightarrow Q[x := P_2] \]

We adopt the obvious notion of reduction for $\lambda \beta$. We shall prove strong normalisation SN for $\lambda \beta$, using Tait's so-called 'computability' argument as given in e.g. [5] A2. Then SN for $\lambda \pi$ follows easily from SN for $\lambda \beta$.

Let us explain the basic structure of this computability argument. As a kind of naive approach one could try to prove SN by induction on the structure of the terms. Indeed, $\text{SN}(x), \text{SN}(a), \text{SN}(P_1) \land \ldots \land \text{SN}(P_n) \Rightarrow \text{SN}(f(P_1, \ldots, P_n))$ and $\text{SN}(P) \land \text{SN}(Q) \Rightarrow \text{SN}(\lambda x \subseteq P Q)$, but the approach fails when it comes to application. From the fact that $\text{SN}(P)$ and $\text{SN}(Q)$ hold one can not
conclude \( SN(PQ) \) – as is easily seen from the counterexample of the (not well-formed) term \( P \equiv Q \equiv \lambda x.(x) \).

Therefore we have to strengthen the property \( SN \) to another property with will be denoted by \( C_r \) (one for each \( r \)). This strengthening is done by way of \textit{induction loading}, i.e. \( M \in C_r \) implies \( SN(M) \). The property \( C_r \) is sometimes referred to as ‘computable’, ‘reducible’ or ‘stable’.

**Definition 2.3.32** Define the following classes of terms inductively:
\[
C_0 = \{ M \in A^0_s \mid SN(M) \}, \\
C_{s \rightarrow r} = \{ M \in A^0_s \mid \forall N \in C_s (MN \in C_r) \}.
\]

To prove that \( M \in C_o \) implies \( SN(M) \), requires an induction on the structure of \( \sigma \). In fact we have another very simple induction loading at this level, which is the reason for adding the proposition (ii) below. As usual we write \( \sigma \rightarrow \tau_1 \rightarrow \tau_2 \) to denote \( \sigma \rightarrow (\tau_1 \rightarrow \tau_2) \), i.e. we adopt the convention that type construction is right-associative. Note that every type \( \sigma \) is of the form \( \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow 0 \) which is \( \sigma_1 \rightarrow (\ldots \rightarrow (\sigma_n \rightarrow 0) \ldots) \).

**Lemma 2.3.33** (i) \( M \in C_o \Rightarrow SN(M) \),
(ii) \( \lambda x^{\sigma_1} \ldots x^{\sigma_n}.c \in C_o \) if \( \sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow 0 \), where \( c \) is some constant.

**Proof.** The proof is by simultaneous induction on \( \sigma \).

**Basis:** obviously \( M \in C_0 \Rightarrow SN(M) \) and \( c \in C_0 \). For the induction step we consider the type \( \sigma \rightarrow \tau = \sigma \rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow 0 \) and we assume (i) and (ii) for \( \sigma, \tau, \tau_1, \ldots, \tau_m \) (l.h.). We must prove (i) \( M \in C_{\sigma \rightarrow r} \Rightarrow SN(M) \) and (ii) \( \lambda x^{\sigma_1} \ldots x^{\sigma_n}.c \in C_{\sigma \rightarrow r} \).

First we treat (i). Let \( M \in C_{\sigma \rightarrow r} \), so \( \forall N \in C_s (MN \in C_r) \). We have \( \sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow 0 \) for some \( \sigma_1, \ldots, \sigma_n \). Hence \( M(\lambda x^{\sigma_1} \ldots x^{\sigma_n}.c) \in C_r \), and by l.h. has no infinite reduction. So certainly \( M \) has not, i.e. \( SN(M) \). In fact we only use \( \lambda x^{\sigma_1} \ldots x^{\sigma_n}.c \) to show that \( C_r \) is inhabited. The proof-step works for any term \( N \) in \( C_o \). Next we treat (ii), and we reason as follows:

\[
\lambda x^{\sigma_1} \ldots x^{\sigma_n}.c \in C_{\sigma \rightarrow r} \text{ if } \\
\text{for } N \in C_s, (\lambda x^{\sigma_1} \ldots x^{\sigma_n}.c)N \in C_r \text{ if } \\
\text{for } P_1 \in C_{\tau_1}, \ldots, P_m \in C_{\tau_m}, (\lambda x^{\sigma_1} \ldots x^{\sigma_n}.c)N P_1 \ldots P_m \in C_0 \text{ which holds since one can do at most } m+1 \text{ reductions + finitely many ones in } N \text{ and } P_1, \ldots, P_m. \text{ In the last step we used the induction hypothesis which gives us } SN(N), \text{ and } SN(P_1), \ldots, SN(P_m). \square
Definition 2.3.34 Define the following classes of terms:

\[ C = \bigcup \{ C_\sigma \mid \sigma \text{ is a type} \} , \]
\[ C_\sigma^* = \{ M \in \Lambda_\sigma \mid M[\vec{\bar{x}} := \vec{F}] \in C \text{ for } \vec{F} \subseteq C \} . \]

where \( \vec{F} \subseteq C \) denotes an arbitrary sequence of terms in \( C \) and where \( \vec{\bar{x}} \) should always contain all free variables of \( M \).

The intuition behind \( C_\sigma^* \) is the set of terms \( M \) in \( \Lambda_\sigma \) such that every instance of \( M \) with elements in \( C \) is in \( C \). In particular, if \( M^* \equiv M[\vec{\bar{x}} := \vec{F}] \) then \( M \in C_\sigma \Rightarrow M^* \in C_\sigma^* \). By introducing this \( C_\sigma^* \) we pushed the induction loading somewhat further and this will allow us to have a proof by induction on the structure of the terms. In this inductive proof we shall use the following preservation properties.

Lemma 2.3.35 (Preservation properties).

(i) \( M \in C \land M \rightarrow M' \Rightarrow M' \in C \),

(ii) \( M \in C_\sigma^* \land M \rightarrow M' \Rightarrow M' \in C_\sigma^* \),

(iii) \( M \in C_\sigma^* \land \vec{F} \subseteq C \Rightarrow M[\vec{\bar{x}} := \vec{F}] \in C_\sigma^* \),

(iv) \( M \in C_\sigma^* \Rightarrow \text{SN}(M) \).

Proof. (i) The proof is by induction on the type of \( M \). If \( M \in C_0 \), i.e. \( \text{SN}(M) \) then \( \text{SN}(M') \) for if not, then \( M \rightarrow M' \rightarrow \ldots \), contradiction. If \( M \in C_\sigma \rightarrow \tau \), then for arbitrary \( N \in C_\sigma \) we have \( MN \in C_\tau \) so by i.h. \( M'N \in C_\tau \). This shows \( M' \in C \).

(ii) Assume \( M \in C_\sigma^* \) and \( M \rightarrow M' \). It follows that \( M[\vec{\bar{x}} := \vec{F}] \rightarrow M'[\vec{\bar{x}} := \vec{F}] \). This can be proved by induction on the derivation of \( \rightarrow \), cf. [S] 3.1.14. Let \( \vec{F} \subseteq C \), then we know \( M[\vec{\bar{x}} := \vec{F}] \in C \) and by (i), \( M'[\vec{\bar{x}} := \vec{F}] \in C \). This shows \( M' \in C_\sigma^* \).

(iii) Let \( \vec{y} \) contain the free variables of \( M' \). We must show that for \( \vec{Q} \subseteq C \) we have \( M'[\vec{\bar{y}} := \vec{Q}] \in C \), i.e. \( M[\vec{\bar{x}} := \vec{F}]|_{\vec{y} := \vec{Q}} \in C \), i.e. \( M[\vec{\bar{x}}, \vec{\bar{y}} := \vec{\bar{F}}, \vec{\bar{Q}}] \in C \), which holds because \( M \in C_\sigma^* \).

(iv) Assume \( M \in C_\sigma^* \) and let \( \vec{F} \subseteq C \), then for \( \vec{\bar{x}} \supseteq \text{FV}(M) \) we have \( M[\vec{\bar{x}} := \vec{F}] \in C \) and hence by 2.3.33 (i) \( \text{SN}(M[\vec{\bar{x}} := \vec{F}]) \). Therefore certainly \( \text{SN} \) holds for \( M \) itself. \( \square \)

Lemma 2.3.36 \( M \in \Lambda_\sigma \Rightarrow M \in C_\sigma^* \).

Proof. The proof is by induction on the structure of \( M \).
2.3. LAMBDA CALCULUS

The first three cases are easy: Case 1 is \( M \equiv x \). Now for all \( P \in C \) one has \( x \langle x \rangle := P \langle \rangle \in C \). Case 2 is \( M \equiv c \), which is trivial. Case 3 is \( M \equiv f_1(M_1, \ldots, M_n) \). Use i.h. and the fact that each \( M_i \) is of type 0.

Case 4. \( M \equiv AB \) with \( A \in \Lambda_\nu, B \in \Lambda_\sigma \). Let \( \overline{A} \subseteq C \). By i.h. \( A \in C_{\nu}^* \) and \( B \in C_{\sigma}^* \), i.e. \( A[\overline{x} := \overline{A}] \in C_{\nu} \), and \( B[\overline{x} := \overline{B}] \in C_{\sigma} \). Now \( M[\overline{x} := \overline{A}] \equiv A[\overline{x} := \overline{A}]B[\overline{x} := \overline{B}] \) which is in \( C \) and hence is in \( C \). Hence \( M \in C_{\nu}^* \).

Case 5. \( M \equiv \lambda y \subseteq A.B \) with \( A \in \Lambda_\nu, B \in \Lambda_\sigma \). Let \( \overline{A} \subseteq C \). By i.h. \( A \in C_{\nu}^* \), \( B \in C_{\sigma}^* \). We may assume \( y \not\subseteq \overline{A} \). Abbreviate \( A^* \equiv A[\overline{x} := \overline{A}], B^* \equiv B[\overline{x} := \overline{B}] \) so \( A^* \in C_{\nu} \) and by 2.3.35 (iii) \( B^* \in C_{\sigma}^* \). Note that \( y \) may still occur free in \( B^* \). Let \( r = r_1 \rightarrow \ldots \rightarrow r_n \rightarrow 0 \).

\[
M \in C_{\nu}^* \quad \text{if}
\]
\[
M^* \equiv \lambda y \subseteq A^*.B^* \in C_{\nu}^* \quad \text{if}
\]
for \( S \in C_\nu \), \( (\lambda y \subseteq A^*.B^*)S \in C_\nu \) if
for \( T_1 \in C_\nu, \ldots, T_n \in C_\nu \), \( SN((\lambda y \subseteq A^*.B^*)ST_1 \ldots T_n) \) which holds,
for if not, then three possibilities arise, each leading to a contradiction.

Either (a) \( A^* \) or \( B^* \) have an infinite reduction. But \( B^* \in C_{\sigma}^* \) so by lemma 2.3.35 (iv) \( SN(B^*) \) and by lemma 2.3.33 (i) also \( SN(A^*) \). Contradiction.

or (b) \( S \) or \( T_1 \ldots T_n \) have an infinite reduction which contradict \( S \in C \), \( T_1 \in C \), \ldots, \( T_n \in C \) and lemma 2.3.33 (i).

or (c) \( (\lambda y \subseteq A^*.B^*)ST_1 \ldots T_n \rightarrow (\lambda y \subseteq A^*.B^*)ST_1' \ldots T_n' \rightarrow B^*[y := S'[T_1' \ldots T_n'] \rightarrow \ldots \). But by lemma 2.3.35 (i) \( S' \in C \), \( T_1' \ldots T_n' \in C \) and by lemma 2.3.35 (ii) \( B^* \in C_{\sigma}^* \). Therefore by the definition of \( C_{\nu}^* \) we have \( B'^* := B^*[y := S'[T_1' \ldots T_n']] \in C \). From the definition of \( C \) we have \( B'^*T_1' \ldots T_n' \in C_\nu \), i.e. \( SN(B'^*T_1' \ldots T_n') \). Contradiction.

\[ \square \]

Theorem 2.3.37 (SN). In \( \lambda \pi \) every term strongly normalises.

Proof. By lemma 2.3.36 and lemma 2.3.35 (iv) we have that in \( \lambda \beta \) every term strongly normalises. It follows that in \( \lambda \pi \) every term strongly normalises, for suppose that \( M \) has an infinite reduction in \( \lambda \pi \), then so it has in \( \lambda \beta \) which cannot be the case.

\[ \square \]

Remark 2.3.38 An alternative proof of SN for \( \lambda \pi \)-calculus can be given by using a technique due to Plotkin (who actually considered \( \lambda \)-typed \( \lambda \)-calculus), suggested to us by L.S. van Benthem Jutting. Let \( \Lambda_\pi \) denote the set of terms of \( \lambda \pi \)-calculus and let \( \Lambda \) denote the set of terms of classical simple-typed \( \lambda \)-calculus. Define a mapping \( \gamma : \Lambda_\pi \rightarrow \Lambda \) as follows:
\( \overline{x} \equiv x' \),
\( \overline{c_n} \equiv c_n \),
\( f(p_1, \ldots, p_n) \equiv f_1(p_1, \ldots, p_n) \),
\( [M N] \equiv (M \overline{N}) \),
\( (\lambda x^a \overline{M}. N) \equiv ((\lambda y^a \rightarrow x^a \overline{N}) \overline{M}) \) (\( y^a \) fresh).

By induction on the structure of \( M \) it can be shown that \( \overline{M}[x \equiv N] \equiv \overline{M}[x \equiv \overline{N}] \). Using this, one shows by induction on the derivation of \( \Gamma \vdash M \rightarrow N \) that

\[ \Gamma \vdash M \rightarrow N \Rightarrow \overline{M} \not\rightarrow \overline{N}. \]

Now if a term \( M \in \mathcal{L} \) has an infinite reduction path, then so does \( \overline{M} \), which cannot be the case by SN for classical simple-typed \( \lambda \)-calculus. \( \square \)

### 2.3.7 Confluence

The diamond property (confluence = Church-Rosser property) holds for \( \lambda \pi \). This means that when a term \( M \) allows for two different reductions, \( \Gamma \vdash M \rightarrow M_1, M \rightarrow M_2 \) say, then these can always be "brought together" by further reductions, i.e. there is a \( M_3 \) with \( \Gamma \vdash M_1 \rightarrow M_3, M_2 \rightarrow M_3 \). The remainder of this section is about proving this property. The reader may want to skip this proof and in that case he can proceed with Section 2.4. By way of preparation for this proof, we introduce the so-called weak diamond property and we shall show that the latter holds for \( \lambda \pi \) first. This weak diamond property is relatively easy to prove. In combination with the SN property it will give us a cheap route towards proving the full diamond property.

**Definition 2.3.89** A one-step reduction relation \( \rightarrow \) satisfies the weak diamond property if (\( \rightarrow \) denoting the reflexive transitive closure)

\[ \Gamma \vdash M \rightarrow M_1, M \rightarrow M_2 \Rightarrow \exists M_3 (\Gamma \vdash M_1 \rightarrow M_3, M_2 \rightarrow M_3). \]  

**Lemma 2.3.40** (Substitution)

(i) \( \Gamma \vdash B \rightarrow B' \Rightarrow \Gamma[x \equiv R] \vdash B[x \equiv R] \rightarrow B'[x \equiv R] \),
(ii) \( \Gamma \vdash R \rightarrow R' \Rightarrow \Gamma \vdash B[x \equiv R] \rightarrow B[x \equiv R'] \).

**Proof.** (i) By induction on the derivation of \( \Gamma \vdash B \rightarrow B' \). (ii) By induction on the structure of \( B \). \( \square \)
Lemma 2.3.41 The reduction relation → satisfies the weak diamond property.

Proof. By induction on the derivation of \( \Gamma \vdash M \rightarrow M_1 \) it will be shown that for all \( M_2 \) such that \( \Gamma \vdash M \rightarrow M_2 \) there is an \( M_3 \) such that \( \Gamma \vdash M_1 \rightarrow M_3 \) and \( \Gamma \vdash M_3 \rightarrow M_2 \). For the trivial \( M_1 \equiv M_3 \) we can always take \( M_3 \equiv M_1 \) so in some cases we shall not mention this trivial \( M_3 \). The numbering of the cases below corresponds to the numbering of the cases in 2.3.28.

1. \( \Gamma \vdash M \rightarrow M_1 \) is \( \Gamma \vdash (\lambda x \subseteq A.B)R \rightarrow B[x := R] \) because \( \Gamma \vdash R \subseteq A \).
   (i) if \( M \rightarrow M_2 \) is \( M \rightarrow B[x := R] \) we are done immediately: \( M_3 \equiv M_1 \equiv M_2 \).
   (ii) if \( M \rightarrow M_2 \) is \( M \rightarrow (\lambda x \subseteq A'.B')R \), take \( M_3 \equiv M_1 \) (noting \( \Gamma \vdash R \subseteq A' \)).
   (iii) if \( M \rightarrow M_3 \) is \( M \rightarrow (\lambda x \subseteq A.B')R \) then take \( M_3 \equiv B'[x := R] \) and apply lemma 2.3.40 (i).
   (iv) if \( M \rightarrow M_2 \) is \( M \rightarrow (\lambda x \subseteq A.B)R \) then take \( M_3 \equiv B'[x := R] \) and apply lemma 2.3.40 (ii).

2. \( \Gamma \vdash M \rightarrow M_1 \) is \( \Gamma \vdash f_j(\ldots,X_i,\ldots) \rightarrow f_j(\ldots,X_1,\ldots) \) because \( \Gamma \vdash X \rightarrow X_1 \). Two cases arise:
   (i) \( M \rightarrow M_2 \) is \( M \rightarrow f_j(\ldots,X_1,\ldots) \). By i.h. there exists an \( X_3 \) such that \( X_1 \rightarrow X_3 \) and \( X_2 \rightarrow X_3 \). Take \( M_3 \equiv f_j(\ldots,X_1,\ldots) \).
   (ii) \( M \rightarrow M_2 \) is \( f_j(\ldots,X_i,\ldots,Y_1,\ldots) \rightarrow f_j(\ldots,X_i,\ldots,Y_1,\ldots) \). Take \( M_3 \equiv f_j(\ldots,X_1,\ldots,Y_1,\ldots) \).

3. \( \Gamma \vdash M \rightarrow M_1 \) is \( \Gamma \vdash ZX \rightarrow ZX_1 \) because \( \Gamma \vdash X \rightarrow X_1 \).
   (i) if \( M \rightarrow M_2 \) is \( ZX \rightarrow Z'X_1 \), then take \( M_3 \equiv Z'X_1 \).
   (ii) if \( M \rightarrow M_2 \) is \( ZX \rightarrow ZX_1 \), then take \( M_3 \equiv ZX_2 \) where \( X_2 \) is given by i.h.
   (iii) if \( M \rightarrow M_2 \) is \( (\lambda x \subseteq A.B)X \rightarrow B[x := X] \), then take \( M_3 \equiv B[x := X_1 \] and apply lemma 2.3.40 (ii).

4. \( \Gamma \vdash M_1 \rightarrow M_1 \) is \( \Gamma \vdash XZ \rightarrow X_1Z \) because \( \Gamma \vdash X \rightarrow X_1 \).
   (i) if \( M \rightarrow M_2 \) is \( XZ \rightarrow X_1Z \), then take \( M_3 \equiv X_1Z \) where \( X_3 \) is given by i.h.
   (ii) if \( M \rightarrow M_3 \) is \( XZ \rightarrow XZ' \), then take \( M_3 \equiv X_1Z' \).
(iii) if \( M \rightarrow M_2 \) is \((\lambda x \subseteq A.B)Z \rightarrow B[x := Z]_1\) then either \( X \rightarrow X_1 \) by \( A \rightarrow A_1 \) or \( X \rightarrow X_1 \) by \( B \rightarrow B_1 \). If by \( A \rightarrow A_1 \) then take \( M_1 \equiv B[x := Z]_1 \). If by \( B \rightarrow B_1 \) then take \( M_1 \equiv B_1[x := Z]_1 \) and apply lemma 2.3.40 (i).

5. \( \Gamma \vdash M \rightarrow M_1 \) is \( \Gamma \vdash (\lambda x \subseteq P.X) \rightarrow (\lambda x \subseteq P_1.X) \) because \( \Gamma \vdash P \rightarrow P_1 \).

(i) if \( M \rightarrow M_2 \) by \( P \rightarrow P_2 \) then take \( M_3 \equiv \lambda x \subseteq P_2.X \) where \( P_3 \) is given by i.h.

(ii) if \( M \rightarrow M_2 \) by \( X \rightarrow X_1 \) then take \( M_3 \equiv \lambda x \subseteq P_1.X \).

6. \( \Gamma \vdash M \rightarrow M_1 \) is \( \Gamma \vdash (\lambda x \subseteq P.X) \rightarrow (\lambda x \subseteq P.X_1) \) because \( \Gamma \vdash [x \subseteq P] \vdash X \rightarrow X_1 \) and \( x \notin \Gamma \).

(i) if \( M \rightarrow M_2 \) by \( P \rightarrow P_1 \) then take \( M_3 \equiv \lambda x \subseteq P_1.X \) and use 2.3.30 (i), rule (subst.) and the cut-rule 2.3.20 to show that \( \Gamma \vdash [x \subseteq P] \vdash X \rightarrow X_1 \).

(ii) if \( M \rightarrow M_2 \) by \( X \rightarrow X_1 \) then take \( M_3 \equiv \lambda x \subseteq P.X_1 \) where \( X_3 \) is given by i.h. (in context \( \Gamma \vdash [x \subseteq P] \)).

The following is known as Newman’s lemma and it provides us with an easy way of proving the diamond property.

Lemma 2.3.42 If a one-step reduction relation \( \rightarrow \) satisfies the weak diamond property and has the strong normalisation property \((SN)\), then its reflexive transitive closure \( \rightarrow \) has the diamond property \((\Diamond)\).


Theorem 2.3.43 \((\Diamond)\). \( \rightarrow \) has the diamond property, i.e. if \( \Gamma \vdash M \rightarrow M_1, M \rightarrow M_2 \) then there is an \( M_3 \) with \( \Gamma \vdash M_1 \rightarrow M_3, M_2 \rightarrow M_3 \).

Proof. By 2.3.42, 2.3.41 and 2.3.37.

The diamond property for \( \lambda x \) does not essentially depend on SN. As G.R. Renardel de Lavalette has shown, it is possible to define an untyped version of \( \lambda x \) which satisfies the diamond property. Tait’s proof method (see [5] pp. 59-62) is applicable, both for the untyped and typed versions of the calculus.
2.4 Components

2.4.1 Introduction

Let us assume that an algebraic system $R$ is given. Roughly speaking, $R$ provides us with a notion of a construct 'being constructed from' a number of primitive constructs and with ways of fitting constructs together. Furthermore the possibility that one construct is a specification of another construct exists. On top of this we have the $\Lambda R$-calculus and we assume that the developer is allowed to make use of the possibilities of this lambda calculus whenever possible.

$\Lambda R$ is a starting point from which a software developer can start his constructive activity. The developer could try to write one (probably large) term such that his customer is satisfied with this term. Let us call such a term an end-product. Clearly this is not a feasible approach, so there must be a top-level specification which serves as a kind of contract between developer and customer. But even with a top-level specification it is difficult to have a manageable development process. Therefore we shall choose another representation for the end-product, viz. a representation in which the term consists of sub-terms which can be referred to and such that some redundant terms have been added. The latter terms serve as specifications and we shall call them black-box descriptions.

For the customer, the references and the blackbox descriptions are probably not interesting; in any case they do no harm, since it is always possible to look-up the references and to throw away the black-box descriptions. For the developer, the references and the black-box descriptions are extremely helpful during the development since they make it easier to manipulate the product.

We adopt the viewpoint that introducing references and adding black-box descriptions should be combined, in the sense that whenever a name (a reference) is given to some term $P$, there should also be a black-box description associated with $P$. This viewpoint immediately leads us to a formal notion of a component. We formalise components as triples, consisting of a name and two terms, where the second term is the black-box description.

We must add one remark here. Strictly speaking, it is possible to have black-box descriptions without using references. This approach is often used in combination with Hoare's logic. The black-box descriptions correspond to pairs of pre- and postconditions. The black-box descriptions are textually included (as a comment) at the appropriate places in the program text. However, the use of references is more general.
CHAPTER 2. A FORMALISATION OF DESIGN STRUCTURES

We shall say that a component is correct if the two terms of the component are in the implementation relation. In order to benefit from the introduction of components, one should require that all components are correct. This requirement can serve as an invariant of the development process. We shall formalise what it means that a component is correct – in a given context.

We shall also have the possibility to have references to constructs which are outside the scope of the developer and which are given by a name and a black-box description only. We introduce a symbol prim which in these cases will serve as a placeholder.

2.4.2 Formal definitions

For given $\mathcal{R}$ we could, as a first attempt, define the set of components as $\{x_i \mid i \in \mathbb{N}\} \times (\Lambda_{\mathcal{R}} \cup \{\text{prim}\}) \times \Lambda_{\mathcal{R}}$. However, we must take care to respect the typing, so we adopt the following definition:

**Definition 2.4.1** Let $\mathcal{R}$ be an algebraic system with preorder.

(i) Define $C_{\mathcal{R}} =\{x_i \mid i \in \mathbb{N}\} \times (\Lambda_{\mathcal{R}} \cup \{\text{prim}\}) \times \Lambda_{\mathcal{R}}$.

(ii) The set $C_{\mathcal{R}}$ of components is defined by $C_{\mathcal{R}} = \{(x_i, P, Q) \in C_{\mathcal{R}} \mid Q$ has type $\sigma$ and if $P \neq \text{prim}$ then $P$ has type $\sigma\}$.

(iii) For a component $c = (x, P, Q)$ we say that $x$ is the name of $c$, $P$ is the glass-box description of $c$ and $Q$ is the black-box description of $c$. We shall often just write $C$ for $C_{\mathcal{R}}$.

Recall that we have the implementation relation denoted by $\sqsubseteq$.

**Definition 2.4.2** A component $(x, P, Q)$ where $P \neq \text{prim}$ is correct in context $\Gamma$ if

$\Gamma \vdash P \sqsubseteq Q$.

A component $(x, \text{prim}, Q)$ is always correct in any context.

**Notation 2.4.3** (Concrete syntax). We shall write

$z := P \sqsubseteq Q$

to denote the component $(x, P, Q)$.

The concrete syntax is intended to be used in very small examples. In the practical applications we have in mind, we have at least two possibilities.
2.5. **DESIGNS**

The first possibility is to have a so-called design-engineering database with operations among which an operation that yields the glass-box description of a component and an operation that yields the black-box description of a component. The second possibility is to include the notion of a component in a design language (as done for COLD-K) and we refer to [11] for a concrete syntax used to represent components.

**Example 2.4.4** Consider $R_1$ as before. Now the component

\[ y := x + x \sqsubseteq 3 \]

is correct in the context $[x \sqsubseteq 1]$ since in the $\lambda\pi$-calculus we have

\[ [x \sqsubseteq 1] \vdash x + x \sqsubseteq 3. \]

Components are somewhat similar to 'lines' in Automath [13], [14]. The similarity exists only on the syntactical level because the implementation relation of Automath (a typing relation) is completely different from $\sqsubseteq$.

### 2.5 Designs

#### 2.5.1 Introduction

In the previous section we discussed the notion of a component. Based on this notion we shall define the notion of a design\(^5\). First of all, it is clear that a design should contain a collection of components. In fact we shall define a design such that it contains a sequence of components. In a design it is possible that a component contains references to other components, but we want to exclude the possibility of circular referencing. Because the components are elements of a sequence, this can be achieved simply by requiring that no name is used before it has been introduced. We shall formulate a well-formedness predicate on designs.

Besides this sequence of components, each design contains one additional term, which we shall call the **system** of the design. The system is to be viewed as an indication of the actual product to be delivered to the customer. In most cases this system will be a very simple term which contains the names of one or more components of the design.

We shall define two notions of correctness for designs, which we shall call **glass-box correctness** and **black-box correctness**. Both notions are interesting

\(^5\)We use 'design' in the sense of 'design object' rather than 'design operation'.
from a methodological point of view. The syntax and the correctness of
designs will be defined in Section 2.5.2.

It is possible to translate each well-formed design into a lambda term. For
an arbitrary design \( d \) the lambda term resulting from this translation can
be viewed as the meaning of \( d \). In this translation, there is an abstraction-
application pair corresponding to each non-prim component. This translation
will be defined in subsection 2.5.3. Also in subsection 2.5.3, we shall see
that several interesting properties of designs have simple counterparts in the
\( \lambda \pi \)-calculus. In subsection 2.5.4 we shall investigate correctness-preserving
modifications of designs.

2.5.2 Syntax and Correctness

Definition 2.5.1 The set \( D_\mathcal{R} \) of designs is defined by
\[
D_\mathcal{R} = (C_\mathcal{R})^* \times \lambda_\mathcal{R}.
\]

For a design \( d = (\{c_1, \ldots, c_n\}, S) \) we say that \( \{c_1, \ldots, c_n\} \) is the set of com-
ponents of \( d \) and we denote it by \( \text{set}(d) \) and we call \( S \) the system of \( d \). We
shall often just write \( D \) for \( D_\mathcal{R} \).

Notation 2.5.2 We shall employ the following concrete syntax

\[
\begin{align*}
x_1 &:= P_1 \sqsubseteq Q_1 \\
x_2 &:= P_2 \sqsubseteq Q_2 \\
\cdots \\
x_n &:= P_n \sqsubseteq Q_n \\
\text{system } S
\end{align*}
\]

to denote the design \(((x_1, P_1, Q_1), (x_2, P_2, Q_2), \ldots, (x_n, P_n, Q_n)), S)\).

Again the concrete syntax is intended to be used for very small examples.
Designs are somewhat similar to Automath 'books' [13], [14] where the com-
ponents correspond to Automath 'lines'. However, there is no such thing as
a 'system' in Automath books.

Definition 2.5.3 Let \( d \) be given as \( d = (\{c_1, \ldots, c_j, \ldots, c_n\}, S) \), then we call
the design \(((c_j, \ldots, c_n), S)\) the \( j \)-th sub-design of \( d \) and we denote it as \( d_{(j)} \).
2.5. **DESIGNS**

**Definition 2.5.4** A design \( d \) is *well-formed* (abbreviated as *wf*) if the following conditions hold:

(i) All components of \( d \) have distinct names.

(ii) No variable occurring in \( d \) is used before it has been introduced as the name of a component. Formally, if \( d \) is given as in 2.5.2 (concrete syntax), then for \( 1 \leq j \leq n \), \( FV(P_j) \cup FV(Q_j) \subseteq \{x_1, \ldots, x_{j-1}\} \), taking \( FV(\text{prim}) := \emptyset \).

(iii) Let \( S \) be the system of \( d \), then each variable in \( FV(S) \) must be the name of one of the components of \( d \). \( \Box \)

We defined already what it means that a component is correct in a given context. Now we want to define correctness for *wf* designs. Roughly speaking, we shall define this in such a way that a design is correct if each of its components is correct. In order to make this idea precise we must be explicit about the contexts in which the correctness of the components is to be derived. There are two reasonable possibilities for defining these contexts. Therefore we shall have two notions of correctness for designs.

The first notion of correctness corresponds to the possibility that there is no information hiding: if the developer reasons about a name \( x_k \) for which there is a component \( (x_k, P_k, Q_k) \) in the design, then the developer may use the fact that \( x_k \) stands for \( P_k \). If \( P_k \equiv \text{prim} \), then the best assumption he can make about \( x_k \) is \( [x_k \equiv Q_k] \).

**Definition 2.5.5** (gbc) Let the design \( d \) be given as

\[
\begin{align*}
x_1 & := P_1 \sqsubseteq Q_1 \\
\vdots \quad & \\
x_n & := P_n \sqsubseteq Q_n \\
\text{system } S.
\end{align*}
\]

Assume that \( d \) is *wf*. We say that \( d \) is *glass-box correct* (abbreviated *gbc*) if for each component \( (x_j, P_j, Q_j) \in \text{co} \text{set}(d) \) where \( P_j \not\equiv \text{prim} \) we have

\[
\Gamma _j \vdash P_j \sqsubseteq Q_j
\]

where \( \Gamma _j = \varphi _1, \ldots, \varphi _{j-1} \) and where for \( 1 \leq k \leq j - 1 \) the \( \varphi _k \) are defined by

(i) \( \varphi _k = [x_k = P_k] \quad (P_k \not\equiv \text{prim}), \)

(ii) \( \varphi _k = [x_k \equiv Q_k] \quad (P_k \equiv \text{prim}). \) \( \Box \)
The second notion of correctness corresponds to the possibility that the "black-box" descriptions are hidden: if the developer reasons about a name \( x_k \) for which there is a component \((x_k, P_k, Q_k)\) in the design, then he may only use the fact that the term for which \( x_k \) stands, is specified by \( Q_k \), i.e. he may use the assumption \([x_k \sqsubseteq Q_k]\).

**Definition 2.5.6 (bbc)** Let the design \( d \) be given as

\[
\begin{align*}
x_1 & := P_1 \sqsubseteq Q_1 \\
\vdots \\
x_n & := P_n \sqsubseteq Q_n \\
\text{system } S.
\end{align*}
\]

Assume that \( d \) is wf. We say that \( d \) is black-box correct (abbreviated bbc) if for each component \((x_j, P_j, Q_j) \in \text{cset}(d)\) where \( P_j \not\in \text{prim} \) we have

\[
\Gamma_j \vdash P_j \sqsubseteq Q_j,
\]

where \( \Gamma_j = \varphi_1, \ldots, \varphi_{j-1} \) and where for \( 1 \leq k \leq j - 1 \) the \( \varphi_k \) are defined by

\[
\varphi_k = [x_k \sqsubseteq Q_k].
\]

\[\Box\]

**Example 2.5.7** Consider \( R_1 \) as before. The following design is gbc but not bbc.

\[
\begin{align*}
x_1 & := 5 \sqsubseteq 10 \\
x_2 & := x_1 \sqsubseteq 7 \\
\text{system } & x_2.
\end{align*}
\]

\[\Box\]

We defined two notions of correctness, viz. gbc and bbc. Intuitively it is clear that bbc is stronger than gbc, since the facts \( P_j \sqsubseteq Q_j \) to be derived are the same for gbc and bbc, but for bbc these facts should be derived with less knowledge than for gbc. This intuition is made more precise below.

**Theorem 2.5.8** Let \( d \) be a wf design. Then we have

\[
d \text{ is bbc } \Rightarrow \ d \text{ is gbc}.
\]

**Proof.** We give the details for a design \( d \) with two components. After that we give a proof-sketch for the general case. Let \( d \) be given as in 2.5.6 (definition bbc). Assume that \( d \) is bbc. We distinguish two cases, where each case falls apart into two subcases.
2.5. DESIGNS

1. \( P_1 \equiv \text{prim} \). The first component is trivially correct. For the second component we distinguish two cases.

   (i) \( P_1 \equiv \text{prim} \). The second component is also trivially correct. Therefore \( d \) is gbc.

   (ii) \( P_1 \not\equiv \text{prim} \). For the second component we must show \([x_1 \subseteq Q_1] \vdash P_2 \subseteq Q_2\), which follows from the fact that \( d \) is bbc. Therefore \( d \) is gbc.

2. \( P_1 \not\equiv \text{prim} \). The bbc-correctness condition for the first component is the same as its gbc-correctness condition, viz. \( \vdash P_1 \subseteq Q_1 \). So the first component is correct. For the second component we distinguish two cases.

   (i) \( P_2 \equiv \text{prim} \). The second component is now trivially correct. Therefore \( d \) is gbc.

   (ii) \( P_2 \not\equiv \text{prim} \). From the fact that \( d \) is bbc we have \( \vdash P_1 \subseteq Q_1 \)

   and \([x_2 \subseteq Q_1] \vdash P_2 \subseteq Q_2\). From \( \vdash P_1 \subseteq Q_1 \) by (subst.) we get

   \([x_1 = P_1] \vdash P_2 \subseteq Q_2\) and now we can use the cut-rule to get

   \([x_2 = P_2] \vdash P_2 \subseteq Q_2\). This shows that the second component is correct in the gbc sense. Therefore \( d \) is gbc.

This concludes the proof for a design \( d \) with two components. For the general case, the following assertion is proved by induction on the number of components \( n \).

\[
\bigwedge_{j=1}^{n} ( P_j \not\equiv \text{prim} \Rightarrow \Gamma_j \vdash P_j \subseteq Q_j )
\]

where \( \Gamma_j = \varphi_1, \ldots, \varphi_{j-1} \) and where for \( 1 \leq k \leq n - 1 \) the \( \varphi_k \) are defined by

(i) \( \varphi_k = [x_k = P_k] \) \( (P_k \not\equiv \text{prim}) \),

(ii) \( \varphi_k = [x_k \subseteq Q_k] \) \( (P_k \equiv \text{prim}) \).

Again use rule (subst.) and the (generalised) cut-rule. \( \square \)
2.5.3 Semantics

It is possible to translate each design \( d \) into a lambda term\(^3\). The lambda term resulting from this translation can be viewed as the meaning of \( d \) and we shall denote it as \([d]\). For each component of \( d \) for which the glass-box description is not prim, there is an abstraction-application pair in \([d]\). For each component of \( d \) for which the glass-box description is prim, there is an abstraction in \([d]\). A similar technique of using abstraction-application pairs and abstractions to describe the role of names has been proposed by N.G. de Bruijn in the context of Automath. Readers who are not familiar with Automath could skip this explanation and proceed with the next paragraph (example 2.5.9). In particular each so-called 'book', which is a structure as for example

\[
\begin{align*}
\text{nat} & := \text{prim} : \text{type} \\
\text{zero} & := \text{prim} : \text{nat} \\
\text{suc} & := \text{prim} : (\text{nat} \to \text{nat}) \\
\text{one} & := \text{suc(zero)} : \text{nat}
\end{align*}
\]

can be viewed as a \( \lambda \)-term, viz. \( \lambda \text{nat} : \text{type}. \lambda \text{zero} : \text{nat}. \lambda \text{suc} : (\text{nat} \to \text{nat}). (\lambda \text{one} : \text{nat}. \text{type})(\text{suc(zero)}) \). De Bruijn noted this 'books-as-lambda-terms' analogy where the correctness of a book follows from the well-typedness of its \( \lambda \)-term. In fact the analogy is somewhat complicated when considering \( \delta \)-reductions. Of course, de Bruijn's typing relation \( '!' \) is a kind of implementation relation which has completely different properties as our \( '\equiv' \). Yet the idea works for designs in the sense that the author noted a 'designs-as-lambda-terms' analogy.

We explain the translation procedure for designs in a step by step manner for a simple case. After that we give a formal definition which deals with the general case.

Example 2.5.9 Consider the design \( d \) of example 2.5.7. To translate this \( d \) we begin with its first component \((x_1, 5, 10)\) which we translate into the following 'term with hole' \((\lambda x_1 \subseteq 10. \cdots)5\) where the dots \( \cdots \) indicate the hole. Later the hole gets filled with the translation of the rest of the design. Next, we proceed with the second component \((x_2, x_1, 7)\) which we translate into \((\lambda x_2 \subseteq 7. \cdots)x_1\). The latter 'term with hole' is put in the first hole and this yields \((\lambda x_1 \subseteq 10. (\lambda x_2 \subseteq 7. \cdots)x_1)5\). Now we are left with one hole again and the last step is to put the system in that hole. So \([d] = (\lambda x_1 \subseteq 10. (\lambda x_2 \subseteq 7. \cdots)x_1)5\).  

\(^3\)The idea of using lambda terms as a semantic domain is not new and has e.g. been used by Landin [12] who mapped ALGOL60 to an enriched lambda calculus; in particular, it is interesting to note that let \( x = Z \); \( X \) is mapped to \((\lambda x.X)Z\).
2.5. DESIGNS

10. \((\lambda x \in \{7, x\} x_1)\delta\).

For components with \texttt{prim}, a slightly different treatment is needed, which
is not shown yet by this example. By way of preparation for the precise
definition of the translation procedure, we need an auxiliary definition.

\textbf{Definition 2.5.10} For a design \(d = ((c_1, \ldots, c_n), S)\) with \(n \geq 1\) we define
\[\text{first}(d) := c_1 \quad \text{and} \quad \text{rest}(d) := ((c_2, \ldots, c_n), S).\]

So \(\text{first}(d)\) yields the first component of \(d\) and \(\text{rest}(d)\) is \(d\) with its first
component removed. Note that 'rest' does not preserve \texttt{wf}. We give the
formal definition, which is defined by recursion.

\textbf{Definition 2.5.11 (Meaning function).} Define \(M : D \rightarrow A_S\). Write \([d] \) for
\(M(d)\).

1. \([\text{system } S] \equiv S,
2. \([d] \equiv (\lambda z \subseteq Q. \text{rest}(d)) P \quad \text{if } \text{first}(d) = (x, P, Q) \text{ and } P \neq \text{prim},\]
   \[\equiv (\lambda z \subseteq Q. \text{rest}(d)) \quad \text{if } \text{first}(d) = (x, \text{prim}, Q).\]

The recursive formulation of \(M\) was originally defined by the author. A
slightly different formulation has been employed later by H.B.M. Jonkers. It
is a continuation semantics where a translation is given first for each compo-
nent.

\textbf{Definition 2.5.12 (Alternative formulation) First define } M : C \rightarrow A_S \rightarrow A_S, \text{ where } C \text{ is the set of components. Write } [c] \text{ for } M(c).

1. \([x, P, Q] [R] := (\lambda x \subseteq Q. R) P \quad \text{if } P \neq \text{prim},\]
2. \([x, \text{prim}, Q] [R] := (\lambda x \subseteq Q. R).\]

Define \(M : D \rightarrow A_S\). Write \([d] \) for \(M(d)\).

1. \([((c_1, \ldots, c_n), S)] := (\langle c_1 \circ \cdots \circ c_n \rangle)(S)\]

where \(\circ\) denotes function composition.

We can make several observations which support the idea that \([d]\) can be
viewed as the meaning of \(d\). First of all, the components \((x, \text{prim}, Q)\) be-
tone lambda abstractions where the black-box description of the component
becomes the restriction associated with this abstraction \((\lambda x \subseteq Q, \ldots)\). This
corresponds precisely to the idea that when the design is applied in the real
world, something satisfying the black-box description has to be filled in for a
prin. We see that the components with a prin glass-box description can be viewed as the formal parameters of the design in which they occur. Secondly, we can observe that reducing $[d]$ corresponds to replacing the names by the things they stand for (i.e., by the corresponding glass-box descriptions). If a component $(z_j, P_j, Q_j)$ is not correct, then we may consider replacing $z_j$ by $P_j$ as illegal. This is precisely reflected in $[d]$, where a candidate-redex $(\lambda x_j \subseteq Q_j, \ldots)P_j$ cannot be contracted unless $P_j \subseteq Q_j$.

Example 2.5.13 Assume variables $w, x, y, z$ and let $Q_1, Q_2, Q_3, Q_4, P_2, P_4, S \in \Lambda_w$. Let the design $d$ be given by

\[
\begin{align*}
  w &:= \text{prin} \subseteq Q_1, \\
  x &:= P_2 \subseteq Q_2, \\
  y &:= \text{prin} \subseteq Q_3, \\
  z &:= P_4 \subseteq Q_4, \\
  \text{system } S.
\end{align*}
\]

The meaning of $d$, denoted as $[d]$ is given by

\[
[d] \equiv \lambda w \subseteq Q_1. (\lambda x \subseteq Q_2.(\lambda y \subseteq Q_3.(\lambda z \subseteq Q_4.S)P_4))P_2.
\]

$\Box$

Theorem 2.5.14 $d$ is wff $\Rightarrow [d]$ is closed. $\Box$

Proof. Obvious from $M$.

We want to have a look at reduction strategies for terms $[d]$ and in order to formulate these strategies precisely, we introduce a bit of machinery for marking candidate-redexes and keeping track of what happens with these marked candidate-redexes during reduction. This technique is known as lifting and we adopt the notations and conventions from [5] 11.1.2, 11.2.1 – adapted to $\lambda$-calculus. We mark a candidate-redex by giving an index to its first lambda. For this purpose we introduce an auxiliary extension of the set of terms $\Lambda_w$. The following should be compared with definition 2.3.3.

Definition 2.5.15 (Terms with indices)

$\Lambda'_w$ is a set of terms constructed over the alphabet of $\Lambda_w$ extended with symbols $\lambda_0, \lambda_1, \lambda_2, \ldots$. This set $\Lambda'_w$ is inductively defined by the clauses below, where it is understood that furthermore the same rules and restrictions with respect to the types as for $\Lambda_w$, apply in the obvious way.

1. $x_i \in \Lambda'_w$ ($i \in \mathbb{N}$),
2. $c_i \in \Lambda'_w$ ($i \in I$),
3. if $P_1, \ldots, P_n \in \Lambda'_w$ then $f_i(P_1, \ldots, P_n) \in \Lambda'_w$,
4. if $P, Q \in \Lambda'_w$ then $(PQ) \in \Lambda'_w$. 

5. if \( P, Q \in \Lambda' \) and \( x_i \not\in FV(P) \), then \( (\lambda x_i \equiv P.Q) \in \Lambda' \).

6. if \( P, Q, R \in \Lambda' \), \( j \in \mathbb{N} \) and \( x_i \not\in FV(P) \), then \( ((\lambda_j x_i \equiv P.Q)R) \in \Lambda' \).
   In this case we say that the candidate-redex \( ((\lambda_j x_i \equiv P.Q)R) \) has index \( j \).

If \( P \in \Lambda' \), then \( |P| \in \Lambda' \) is \( P \) with all indices removed.

This provides us with an extended set of terms and the obvious next step is to say how the indices are treated during reduction.

**Definition 2.5.16 (Reduction with indices)**

(i) Substitution on \( \Lambda' \) is defined in the usual way. See [5] 11.1.3.

(ii) In addition to normal \( \pi \)-reduction \( \rightarrow \), we define \( \pi_j \)-reduction (also denoted by \( \rightarrow_j \)) which means to contract a candidate-redex having index \( j \) (one notion of reduction for each index \( j \in \mathbb{N} \)). In particular, \( \Gamma \vdash |R| \equiv |A| \vdash \Gamma \vdash (\lambda_j x \equiv A.B)R \rightarrow B[x := R] \) and we refer to the latter \( \rightarrow \) step as a \( \pi_j \)-reduction step.

(iii) The notion of reduction \( \pi' \) is \( \pi \cup \bigcup_{j \in \mathbb{N}} \pi_j \).

So now we have the features that candidate-redexes can be indexed optionally and that we can keep track of these candidate-redexes during reduction. The following should be compared with [5] 11.1.6 (i).

**Lemma 2.5.17 (Lifting)** Let \( P, Q \in \Lambda' \) such that \( \Gamma \vdash P \rightarrow Q \) where \( \rightarrow \) is based on \( \pi \)-reduction, then for all \( P' \in \Lambda' \) with \( |P'| \equiv P \) there is a \( Q' \) with \( |Q'| \equiv Q \) such that \( \Gamma \vdash P' \rightarrow Q' \), where the latter \( \rightarrow \) is based on \( \pi' \)-reduction.

**Proof.** If \( \Gamma \vdash P \rightarrow Q \) is a one-step \( \pi \)-reduction, where a candidate-redex is contracted, \( \Delta \) say, then it is possible to identify a corresponding candidate-redex \( \Delta' \), say in \( P' \). Take the \( Q' \) obtained by contracting this \( \Delta' \) in \( P' \). If \( Q \equiv P \) then take \( Q' \equiv P' \). If \( P \rightarrow Q \) involves more than one step, then use transitivity.

The following should be compared with [5] 11.1.6 (ii).

**Lemma 2.5.18 (Projecting)** Let \( P', Q' \in \Lambda' \) such that \( \Gamma \vdash P' \rightarrow Q' \) where \( \rightarrow \) is based on \( \pi \)-reduction, then \( \Gamma \vdash |P'| \rightarrow |Q'| \) where the latter \( \rightarrow \) is based on \( \pi \)-reduction.

**Proof.** Leave out all indices from a reduction path from \( P' \) to \( Q' \).
When translating designs to λ∀-terms, each non-prim component gives rise to a candidate-redex and we shall associate indices with these candidate-redexes in a systematic way: index 1 for a candidate-redex derived from the first component, index 2 for a candidate-redex derived from the second component etc. In order to formulate this precisely we give a re-formulation of the meaning function $M$ introduced before. Therefore we introduce an extended meaning function $M'$ using an auxiliary which takes two arguments and which is also denoted by $M'$.

**Definition 2.5.19** (Meaning function with indices). First define $M' : D \times \mathbb{N} \to \Lambda_\mathbb{N}$. Write $[d]_j'$ for $M'(d, j)$. Intuitively $[d]_j'$ is the meaning of $d$ where $j$ is the next number to be used for indexing.

1. $[\text{system } S]_j' \equiv S$,
2. $[d]_j' \equiv (\lambda x \subseteq Q. [\text{rest}(d)]_{j+1}')$ if first($d$) = $(x, P, Q)$, $P \neq \text{prim}$,

$$
\equiv (\lambda x \subseteq Q. [\text{rest}(d)]_{j+1}') \quad \text{if first($d$) = $(x, \text{prim}, Q)$}.
$$

Using this we define $M' : D \to \Lambda_\mathbb{N}$, writing $[d]'$ for $M'(d)$, as follows: $[d]' := [d]_1'$.

This has the effect that we can explicitly refer to a candidate-redex derived from the $j$-th component by means of the index $j$. Note that each $j$ refers to at most one candidate-redex occurrence in $[d]'$ and that $\pi^\prime$-reduction preserves this property -- which is because of the special structure of the terms resulting from the translation of designs by $M'$. The following lemma relates both formulations of the translation from designs to lambda terms, $M$ and $M'$.

**Lemma 2.5.20** Let $d$ be a design, then $[[d]'] \equiv [d]$.

**Proof.** The following stronger statement is proved by induction on the number of components of the design: $\forall j \in \mathbb{N} \left( [d]_j' \equiv [d] \right)$. $\square$

We illustrate $M'$ with a small example.

**Example 2.5.21** Assume variables $x, y, z$ and let $Q_1, Q_2, Q_3, P_1, P_2, P_3, S \in \Lambda_\mathbb{S}$.

Let the design $d$ be given by

\[
\begin{align*}
x & := \text{prim} \subseteq Q_1 \\
y & := P_2 \subseteq Q_2 \\
z & := P_3 \subseteq Q_3 \\
\text{system } S.
\end{align*}
\]
2.5. DESIGNS

The meaning with indices of \( d \) is given by

\[
[d]'' \equiv \lambda x \, Q_1 \cdot (\lambda y \, Q_2 \cdot (\lambda z \, Q_3 \cdot S \cdot y := P_3 \cdot S \cdot y := P_3) \cdot P_3) \cdot P_3.
\]

In this example the first lambda can not have an index because this lambda does not belong to a candidate-redex. The second component \((y, P_3, Q_3)\) gives rise to a candidate-redex having index 2 and the third component gives rise to a candidate-redex having index 3. Note that also after a reduction step, the residuals of a certain candidate-redex can still be identified by their index.

E.g. suppose that for \([d]'\) as above, \([d]'\) reduces to \(\lambda x \, Q_1 \cdot (\lambda y \, Q_2 \cdot y := P_3 \cdot S \cdot y := P_3) \cdot P_3[y := P_3]\), then the original candidate-redex indexed 3 need not be present any more, but \(\lambda_3\) still marks the beginning of a candidate-redex which is reminiscent of the third component.

There is a very remarkable relation between our notions of correctness and certain reduction strategies for lambda terms. If \( d \) is wf and gbc, then \([d]'\) is outermost reducible, by which we mean that the candidate-redexes introduced by the translation of \( d \) by \( M \), can be contracted in an outside-in (i.e. left-to-right) order. If \( d \) is wf and gbc, then \([d]''\) is innermost reducible, by which we mean that the candidate-redexes introduced by the translation of \( d \) by \( M \), can be contracted in inside-out (i.e. right-to-left) order. The formulation of these notions of outermost reducible and innermost reducible will be relative to a given design \( d \) because we want to restrict our considerations to those candidate-redexes which are introduced by the translation of \( d \) by \( M \).

In order to formulate this precisely we use the technique of lifting introduced above. We observe that in a term \([d]'\) the indices occur in increasing order, i.e. when this term contains indexed lambdas \( \lambda_j \), \( \lambda_k \) with \( j < k \) then the \( \lambda_j \) occurs to the left of the \( \lambda_k \). This observation justifies the definitions below where 'outermost' will be defined as 'in increasing order of indices' and where 'innermost' will be defined as 'in decreasing order of indices'.

**Definition 2.5.22 (omr)** For wf design \( d \), we say that \([d]'\) is outermost reducible if there exist terms \( R_1, \ldots, R_m \) such that there is a reduction path \((\rightarrow \text{ denoting } \pi'-\text{reduction})\)

\[
\vdash [d]' \rightarrow R_1 \rightarrow \ldots \rightarrow R_m
\]

where indexed candidate-redexes are contracted in increasing order of indices and such that the last term \( R_m \) contains no more indices. We say that \([d]''\) is outermost reducible with respect to the candidate-redexes introduced by the translation of \( d \) by \( M \) if \([d]'\) is outermost reducible. When \( d \) is clear from the context, we shall just say that \([d]\) is outermost reducible (abbreviated omr) and then it is to be understood that we mean that it is outermost.
reducible with respect to the candidate-redexes which are introduced by the translation of \( d \) by \( \mathcal{M} \).

Very much in the same way we define 'innermost reducible' below.

**Definition 2.5.28** (imr) For \( \text{w}f \) design \( d \), we say that \( [d'] \) is innermost reducible if there is a reduction path starting with \( [d'] \) where indexed candidate-redexes are contracted in decreasing order of indices and such that the last term contains no more indices. We adopt an obvious terminology analogous to that of 2.5.22.

Before we prove the relation between the notions of correctness and the reduction strategies, we first give an example. The following example shows three very simple designs and we verify for these designs that \( \text{gbc} \) corresponds to \( \text{omr} \) and that \( \text{bbc} \) corresponds to \( \text{imr} \).

**Example 2.5.24** Consider \( \mathcal{R}_1 \) as before.

(i) The following design \( d \) is neither \( \text{gbc} \) nor \( \text{bbc} \).

\[
x_1 := 5 \quad \mathcal{L} \quad 4 \\
x_2 := x_1 \quad \mathcal{L} \quad 7 \\
\text{system } x_1.
\]

\([d'] \equiv (\lambda_1 x_1 \mathcal{L} 4 (\lambda_2 x_2 \mathcal{L} 7 x_2) x_1) 5\) which is not \( \text{omr} \) because \( \not\exists \ 5 \mathcal{L} 4 \). It is not \( \text{imr} \) because although \([d'] \rightarrow (\lambda_1 x_1 \mathcal{L} 4 x_1) 5\), we cannot contract the candidate-redex \((\lambda_1 x_1 \mathcal{L} 4 x_1) 5\).

(ii) The following design \( d' \) is \( \text{gbc} \) but not \( \text{bbc} \).

\[
x_1 := 5 \quad \mathcal{L} \quad 10 \\
x_2 := x_1 \quad \mathcal{L} \quad 7 \\
\text{system } x_2.
\]

\([d'] \equiv (\lambda_1 x_1 \mathcal{L} 10 (\lambda_2 x_2 \mathcal{L} 7 x_2) x_1) 5\) which is \( \text{omr} \) because \([d'] \rightarrow (\lambda_2 x_2 \mathcal{L} 7 x_2) x_1 [x_1 := 5] \equiv (\lambda_2 x_2 \mathcal{L} 7 x_2) 5 \rightarrow x_2 [x_2 := 5] \equiv 5\). It is not \( \text{imr} \) because we cannot make the reduction step \( \vdash (\lambda_1 x_1 \mathcal{L} 10 (\lambda_2 x_2 \mathcal{L} 7 x_2) x_1) 5 \rightarrow (\lambda_1 x_1 \mathcal{L} 10 x_2 [x_2 := x_1]) 5\). To make the latter reduction step requires (cf. 2.3.28 clauses 4, 6 & 1) that \([x_1 \mathcal{L} 10] \vdash x_1 \mathcal{L} 7\) which is not the case.

(iii) The following design \( d'' \) is both \( \text{gbc} \) and \( \text{bbc} \).

\[
x_1 := 5 \quad \mathcal{L} \quad 5 \\
x_2 := x_1 + 1 \mathcal{L} \quad x_1 + 5 \\
\text{system } x_1 + x_2.
\]
\[ [d'''] \equiv (\lambda x_1 \equiv 5(\lambda z_2 \equiv x_1 + 5.x_1 + z_2)(x_1 + 1))5 \text{ which is omr because} \\
\] 
\[ [d'''] \rightarrow (\lambda z_2 \equiv 5 + 5.5 + z_2)(5 + 1) \rightarrow 5 + (5 + 1). \text{ It is imr because} \\
\] 
\[ [d'''] \rightarrow (\lambda x_1 \equiv 5.x_1 + (x_1 + 1))5 \rightarrow 5 + (5 + 1). \]

It is easy to see why omr corresponds with gbc. When performing an outermost \( \pi \)-reduction, the resulting substitutions will affect the parameter restrictions occurring deeper inside the term. This shows that the glass-box descriptions \( P_i \) are relevant in the sense that substitutions \( [x_i := P_i] \) take place, influencing preconditions of subsequent steps. Indeed, for gbc the \( P_i \) are relevant for the correctness conditions of subsequent components. On the other hand, when performing an innermost reduction, the early reduction steps do not influence the preconditions of the subsequent steps. Similarly for bbc, the \( P_i \) are not relevant for the contexts of the correctness conditions.

By way of preparation for the theorem "\( d \text{ is gbc} \Leftrightarrow [d] \text{ is omr} \)" we need a definition.

**Definition 2.5.25** Let the wf design \( d \) be given as in 2.5.2 (concrete syntax). The unfolding of \( d \), denoted as \( \text{unf}(d) \) is defined as the design

\[
x_1 := P_1 S_1 \equiv Q_1 S_1 \\
... \\
x_n := P_n S_n \equiv Q_n S_n \\
\text{system } S
\]

where each \( S_j (1 \leq j \leq n) \) is a sequence of substitutions \( S_j \equiv \sigma_1, \ldots, \sigma_{j-1} \) where for \( (1 \leq k \leq n) \) the \( \sigma_k \) are defined as follows:

\[
\sigma_k \equiv \begin{cases} \text{(empty)} & \text{(if } P_k \equiv \text{prim}) \\
\sigma_k \equiv [x_i := P_k S_k] & \text{(if } P_k \not\equiv \text{prim}) 
\end{cases}
\]

where for \( S_k \equiv \sigma_1, \ldots, \sigma_{k-1} \) we write \( P_k S_k \) to denote \( P_k \sigma_1, \ldots, \sigma_{k-1} \).

The intuition behind \( \text{unf}(d) \) is the design obtained from \( d \) by replacing all names by the things they stand for (if possible) while preserving the structure of \( d \).

**Theorem 2.5.26** Let \( d \) be a wf design. Now we have

\( d \text{ is gbc} \Leftrightarrow [d] \text{ is omr}. \)

**Proof.** We give the details for a design \( d \) with two components. After that we give a proof-sketch for the general case. Let \( d \) be given as in 2.5.6 (definition bbc). We distinguish four cases.
1. $P_1 \equiv \text{prim} \wedge P_2 \equiv \text{prim}$. In this case $[d]' \equiv \lambda x_1 \subseteq Q_1, \lambda x_2 \subseteq Q_2, S$. Now $d$ is trivially gbc and also $[d]$ is trivially omr.

2. $P_1 \equiv \text{prim} \wedge P_2 \not\equiv \text{prim}$. In this case $[d]' \equiv \lambda x_1 \subseteq Q_1, (\lambda x_2 \subseteq Q_2, S)P_2$. Now $d$ is gbc iff $[x_1 \subseteq Q_1] \vdash P_2 \subseteq Q_2$ which is precisely the case iff $\vdash \lambda x_1 \subseteq Q_1, (\lambda x_2 \subseteq Q_2, S)P_2 \rightarrow \lambda x_1 \subseteq Q_1, S[x_2 := P_2]$ which means that $[d]$ is omr.

3. $P_1 \not\equiv \text{prim} \wedge P_2 \equiv \text{prim}$. In this case $[d]' \equiv (\lambda x_1 \subseteq Q_1, \lambda x_2 \subseteq Q_2, S)P_1$. Now $d$ is gbc iff $\vdash P_1 \subseteq Q_1$ which is precisely the case iff $\vdash (\lambda x_1 \subseteq Q_1, \lambda x_2 \subseteq Q_2, S)P_1 \rightarrow (\lambda x_2 \subseteq Q_2, S)[x_1 := P_1]$ which means that $[d]$ is omr.

4. $P_1 \not\equiv \text{prim} \wedge P_2 \not\equiv \text{prim}$. In this case $[d]' \equiv (\lambda x_1 \subseteq Q_1, (\lambda x_2 \subseteq Q_2, S)P_2)P_1$. Now $d$ is gbc iff $\vdash P_1 \subseteq Q_1$ and $[x_1 = P_1] \vdash P_2 \subseteq Q_2$.

First note that $\vdash P_1 \subseteq Q_1$ is precisely the case iff the candidate redex of $\lambda_1$ can be reduced. The result of this first reduction step is $(\lambda x_2 \subseteq Q_2[x_1 := P_1], S[x_1 := P_1])P_2[x_1 := P_1]$.

Secondly, we shall show that $[x_1 = P_1] \vdash P_2 \subseteq Q_2$ is precisely the case iff in the result of the first reduction we can perform the $\lambda_2$ reduction step. Therefore we show $[x_1 = P_1] \vdash P_2 \subseteq Q_2 \iff \vdash P_2[x_1 := P_1] \subseteq Q_2[x_1 := P_1]$.

To show $(\Rightarrow)$, assume $[x_1 = P_1] \vdash P_2 \subseteq Q_2$. By (subst.) $[x_1 = P_1] \vdash P_2[x_1 := P_1] \subseteq Q_2[x_1 := P_1]$. Now $x_1$ does not occur any more at the right-hand side of the $'\vdash'$ and therefore the assumption $[x_1 = P_1]$ is non-essential. To see this, apply lemma 2.3.19, performing the substitution $[x_1 := P_1]$; this yields $[P_1 = P_1] \vdash P_2[x_1 := P_1] \subseteq Q_2[x_1 := P_1]$ whence by (refl.) and the cut-rule $\vdash P_2[x_1 := P_1] \subseteq Q_2[x_1 := P_1]$.

To show $(\Leftarrow)$ assume $\vdash P_2[x_1 := P_1] \subseteq Q_2[x_1 := P_1]$. By weakening and rule (subst.) we get $[x_1 = P_1] \vdash P_2 \subseteq Q_2$. This concludes $(\Leftarrow)$. So $d$ is gbc iff $[d]$ is omr.

For the general case, first prove

(i) $d$ is gbc $\iff$ unif$(d)$ is gbc,

(ii) unif$(d)$ is gbc $\iff$ $[d]$ is omr.

The theorem is a direct consequence of (i) $\wedge$ (ii).

\[ \Box \]

Theorem 2.5.27 Let $d$ be a wf design. Now we have

$d$ is bbc $\iff$ $[d]$ is imr.
2.5. DESIGNS

Proof. Again we give the details for a design \( d \) with two components and after that a proof-sketcho for the general case. Let \( d \) be given as in 2.5.6 (definition bbc). We distinguish four cases.

1. \( P_1 \equiv \text{prim} \land P_2 \equiv \text{prim} \). In this case \([d]|\equiv \lambda x_1 \in Q_1.\lambda x_2 \in Q_2.S\).

Now \( d \) is trivially bbc and also \([d] \) is trivially imr.

2. \( P_1 \equiv \text{prim} \land P_2 \not\equiv \text{prim} \). In this case \([d]|\equiv \lambda x_1 \in Q_1.(\lambda x_2 \in Q_2.S)P_2 \).

Now \( d \) is bbc iff \( x_1 \in Q_1 \models P_2 \subseteq Q_2 \) which is precisely the case iff \( \models \lambda x_1 \in Q_1.\lambda x_2 \in Q_2.S)P_2 \rightarrow \lambda x_1 \in Q_1.S[x_2 := P_2] \) which means that \([d] \) is imr.

3. \( P_1 \not\equiv \text{prim} \land P_2 \equiv \text{prim} \). In this case \([d]|\equiv (\lambda x_1 \in Q_1.\lambda x_2 \in Q_2.S)P_1 \).

Now \( d \) is bbc iff \( P_1 \subseteq Q_1 \) which is precisely the case iff \( \models (\lambda x_1 \in Q_1.\lambda x_2 \in Q_2.S)P_1 \rightarrow (\lambda x_2 \in Q_2.S)[x_1 := P_1] \) which means that \([d] \) is imr.

4. \( P_1 \not\equiv \text{prim} \land P_2 \not\equiv \text{prim} \). In this case \([d]|\equiv (\lambda x_1 \in Q_1.\lambda x_2 \in Q_2.S)P_2 \).

First note that \( x_1 \subseteq Q_1 \models P_2 \subseteq Q_2 \) is precisely the case iff the candidate-redex of \( \lambda x \) can be reduced. The result of this first reduction step is \((\lambda x_1 \in Q_1.S[x_2 := P_2])P_1 \).

Secondly we see that \( \models P_2 \subseteq Q_2 \) precisely the case iff in the result of the first reduction, we can perform the \( \lambda x \) reduction step. So \( d \) is bbc iff \([d] \) is imr.

The general case goes in essentially the same way, noting that for performing a contraction internally within a term \((\lambda x \subseteq Q. \ldots)\) or within a term \((\lambda x \subseteq Q. \ldots)\) we may work in a context with \([x \subseteq Q]\) (cf. 2.3.28 clauses 4, 6 & 1). Therefore the fact that \( \Gamma_j \vdash P_2 \subseteq Q_j \) is precisely the condition that is needed in order to perform the contraction \((\Gamma_j \text{ as in 2.5.6, definition bbc)}\).

\( \square \)

2.5.4 Correctness-preserving Modifications

One of the ideas behind the notion of a design is that there is a locality-principle which, roughly speaking, can be stated as follows: "It should be possible to implement each component in a design without worrying about the implementation of the other components in the design". In order to make this idea more precise we shall define several kinds of so-called correctness-preserving modifications of designs and we shall investigate their properties.

There are two (binary) criteria to classify the modifications and therefore we shall have four kinds of modifications. The first criterion deals with "what
We consider two cases: either some glass-box description is modified, or some black-box description is modified. The second criterion deals with "which notion of correctness is adopted?". Again we consider two cases: glass-box correctness and black-box correctness. The modifications are defined to establish the adopted notion of correctness – at least locally for the modified component. The scheme is that we have $X$-preserving $Y$-modifications for $X \in \{\text{gbc, bbc}\}$ and $Y \in \{\text{glass-box, black-box}\}$. We study what happens if we take a correct design (in the $X$ sense) and modify a $Y$-description. The interesting question is "is the result correct again in the $X$ sense?", or alternatively "has $X$ been preserved?".

**Definition 2.5.28** Let the $\text{wf}$ design $d$ be given as

$$
\begin{align*}
  x_1 & := P_1 \sqsubseteq Q_1 \\
  \vdots & \ \\
  x_n & := P_n \sqsubseteq Q_n \\
  \text{system } S.
\end{align*}
$$

We say that $d'$ is a *gbc-preserving glass-box modification* of $d$, (abbreviated as $d'$ is *gbc-gb-mod* of $d$) if $d'$ is obtained from $d$ via the replacement:

$$(x_j, P_j, Q_j) := (x_j, P'_j, Q_j)$$

for some $j \in \{1, \ldots, n\}$ such that $P_j \not\equiv \text{prim}$, provided $d'$ is $\text{wf}$ and for $\Gamma_j$ as in definition 2.5.5 (definition gbc)

$$
\Gamma_j \vdash P'_j \sqsubseteq Q_j.
$$

In the same way we define what it means that $d'$ is a *bbc-preserving glass-box modification* of $d$, by taking $\Gamma_j$ as in 2.5.6.

The intuition behind a gbc-preserving glass-box modification is that a modification takes place within one component while preserving the local correctness (in the gbc sense) of that component. The question if preserving local correctness implies preserving correctness of the whole design, is investigated below.

**Remark 2.5.29** The proposition that for $d'$ is gbc-gb-mod of $d$ we have

$$d \text{ is gbc } \Rightarrow \text{ } d' \text{ is gbc.}$$

just does not hold in general.

The counter-example is as follows. Consider $R_1$ as before and let the design $d$ given as
2.5. DESIGNS

\[ z_1 := 5 \quad \subseteq \quad 10 \]
\[ z_2 := z_1 \quad \subseteq \quad 5 \]

system \( z_2 \)

and consider a replacement as follows:

\( (x_1, 5, 10) := (x_1, 6, 10) \).

Clearly \( d \) is gbc but \( d' \) is not, because \( [x_1 = 6] \forall x_1 \subseteq 5 \). \( \Box \)

Theorem 2.5.30 If \( d' \) is bbc-gb-mod of \( d \), we have

\( d \) is bbc \( \Rightarrow \) \( d' \) is bbc.

Proof. Assume \( d \) is as in 2.5.28 (definition bbc-gb-mod) and consider the replacement \( (x_j, P_j, Q_j) := (x_j, P'_j, Q_j) \) for some \( j \in \{1, \ldots, n\} \) such that \( P_j \not\equiv \text{prim} \), provided \( d' \) is wf and for \( \Gamma_j \) as in definition 2.5.6 (definition bbc) \( \Gamma_j \vdash P'_j \subseteq Q_j \). For all \( i \in \{1, \ldots, n\} \) where \( P_i \not\equiv \text{prim} \) we must show

\[ \Gamma_i \vdash P_i \subseteq Q_i \quad (P'_i \text{ if } i = j) \]

where \( \Gamma_i = [x_1 \subseteq Q_1], \ldots, [x_{i-1} \subseteq Q_{i-1}] \). For \( i \neq j \) we are done since \( d \) is bbc. For \( i = j \) we are also done since for \( \Gamma_j \) as in 2.5.6 (definition bbc) we have by the definition of bbc-gb-mod \( \Gamma_j \vdash P'_j \subseteq Q_j \). \( \Box \)

The intuition behind the remark and the theorem given above is that gbc designs do not offer implementation freedom but bbc designs do.

Instead of considering correctness-preserving glass-box modifications one can also consider correctness-preserving black-box modifications. The intuition behind the latter kind of modification is a change of specification for a component which has already been implemented. It does not come as a surprise that when one adopts glass-box correctness, these modifications preserve correctness for the whole design — since the black-box descriptions are in fact not used. It is also easy to see that when one adopts black-box correctness, these modifications may disturb the correctness of the whole design.

Definition 2.5.31 Let the wf design \( d \) be given as

\[ z_1 := P_1 \quad \subseteq \quad Q_1 \]
\[ \ldots \]
\[ z_n := P_n \quad \subseteq \quad Q_n \]

system \( S \).
We say that $d'$ is a gbc-preserving black-box modification of $d$, if $d'$ is obtained from $d$ via the replacement:

$$(x_j, P_j, Q_j) := (x_j, P_j, Q'_j)$$

for some $j \in \{1, \ldots, n\}$ such that $P_j \neq \text{prim}$, provided $d'$ is wf and for $\Gamma_j$ as in 2.5.5 (definition gbc)

$$\Gamma_j \vdash P_j \subseteq Q'_j$$

In the same way we define what it means that $d'$ is a bbc-preserving black-box modification of $d$, by taking $\Gamma_j$ as in 2.5.6.

Remark 2.5.32 The proposition that for $d'$ is bbc-bb-mod of $d$ we have

$$d \text{ is bbc } \Rightarrow \ d' \text{ is bbc.}$$

just does not hold in general.

The counter-example is as follows. Consider $\mathbb{R}_1$ as before and let the design $d$ be given as

- $x_1 := 1 \subseteq 2$
- $x_2 := x_1 \subseteq 2$
- system $x_3$

and consider a replacement as follows:

$$(x_1, 1, 2) := (x_1, 1, 3).$$

Now $d$ is bbc but $d'$ is not.

Theorem 2.5.33 If $d'$ is gbc-bb-mod of $d$, we have

$$d \text{ is gbc } \Rightarrow \ d' \text{ is gbc.}$$

Proof. Assume $d$ is as in 2.5.31 (definition gbc-bb-mod) and consider the replacement $(x_j, P_j, Q_j) := (x_j, P_j, Q'_j)$ for some $j \in \{1, \ldots, n\}$ such that $P_j \neq \text{prim}$, provided $d'$ is wf and for $\Gamma_j$ as in definition 2.5.5 (definition gbc) $\Gamma_j \vdash P_j \subseteq Q'_j$. For all $i \in \{1, \ldots, n\}$ where $P_i \neq \text{prim}$ we must show

$$\Gamma_i \vdash P_i \subseteq Q'_i \quad (Q'_i \text{ if } i = j)$$

where $\Gamma_i = \varphi_1, \ldots, \varphi_{i-1}$ and where $\varphi_k = [x_k = P_k] (P_k \neq \text{prim}), \varphi_k = [x_k \subseteq Q_k] (P_k = \text{prim})$. 
2.5. DESIGNS

Now notice that neither $Q_i$ nor $Q'_j$ occurs in $\Gamma$, so we are done for $i \neq j$ because $d$ is gbc. For $i = j$ we know $\Gamma_j \vdash P_j \subseteq Q'_j$ by the definition of gbc-bb-mod.

The following table summarises the last two theorems and remarks. For the modifications which lead to a correct design, the corresponding entry in the table contains a ‘+’. For the modifications which may lead to an incorrect design, the corresponding entry in the table contains a ‘−’.

<table>
<thead>
<tr>
<th>preserving</th>
<th>modifying</th>
</tr>
</thead>
<tbody>
<tr>
<td>glass-box correctness</td>
<td>black-box description</td>
</tr>
<tr>
<td>black-box correctness</td>
<td>black-box description</td>
</tr>
</tbody>
</table>

In the above table we see two entries where the modification may lead to an incorrect design, although locally (i.e. for the replaced component) correctness is preserved. One is tempted to think that by restricting the modifications to those modifications where a term is replaced by a term which is less than or equal to that term, the resulting designs are still correct. This turns out to be the case indeed for bbc-preserving black-box modifications, but it does not hold for gbc-preserving glass-box modifications.

Theorem 2.5.34 Let the wf and bbc design $d$ be given as

$$
\begin{align*}
x_1 & := P_1 \subseteq Q_1 \\
\vdots & \\
x_n & := P_n \subseteq Q_n \\
\text{system } S
\end{align*}
$$

and let $d'$ be obtained from $d$ via the replacement

$$(x_j, P_j, Q_j) := (x_j, P_j, Q'_j)$$

for some $j \in \{1, \ldots, n\}$ such that $P_j \neq \text{prim}$, provided $d'$ is wf and

$$[x_1 \subseteq Q_1, \ldots, [x_{j-1} \subseteq Q_{j-1}] \vdash P_j \subseteq Q'_j \subseteq Q_j.$$ 

Then $d'$ is bbc.

**Proof.** Clearly the first $j-1$ components are still correct. The condition $[x_1 \subseteq Q_1, \ldots, [x_{j-1} \subseteq Q_{j-1}] \vdash P_j \subseteq Q'_j$ tells us that the $j$-th component
is correct. Now consider the $j$-th component for $j < i \leq n$. Define $\Gamma' = [x_1 \subseteq Q_1, \ldots, x_{j-1} \subseteq Q_{j-1}, x_j \subseteq Q_j, \ldots, x_{i-1} \subseteq Q_{i-1}]$. Define $\Gamma'' = [x_1 \subseteq Q_1, \ldots, x_j \subseteq Q_j, \ldots, x_{i-1} \subseteq Q_{i-1}]$. Since $d$ is bbc we have
\[ \Gamma'' \vdash P_i \subseteq Q_i. \]

Now $[x_1 \subseteq Q_1, \ldots, x_{j-1} \subseteq Q_{j-1}] \subseteq \Gamma'$ so $\Gamma'' \vdash Q_j \subseteq Q_j$, and since $[x_j \subseteq Q_j] \in \Gamma'$ we can use the rule (trans.) and we obtain $\Gamma'' \vdash x_j \subseteq Q_j$. By the generalised cut-rule we get
\[ \Gamma'' \vdash P_i \subseteq Q_i. \]

\[ \square \]

Remark 2.5.35 The proposition that for the wf and gbc design $d$ given as
\[
\begin{align*}
x_1 &:= P_1 \subseteq Q_1 \\
\vdots &\quad \vdots \\
x_n &:= P_n \subseteq Q_n \\
\text{system} &\Rightarrow S
\end{align*}
\]

and for $d'$ obtained from $d$ via the replacement
\[(x_j, P_j, Q_j) := (x_j, P'_j, Q_j)\]

for some $j \in \{1, \ldots, n\}$ such that $P_j \not\in \text{prim}$, provided $d'$ is wf and for $\Gamma$, as in 2.5.5 (definition gbc)
\[ \Gamma_j \vdash P'_j \subseteq P_j \subseteq Q_j \]

we have $d'$ is gbc, just does not hold in general.

The counter-example is as follows. Consider $\mathcal{R}_1$ as before and let the design $d$ be given as
\[
\begin{align*}
x_1 &:= 5 \subseteq 10 \\
x_2 &:= 5 \subseteq x_1 \\
\text{system} &\Rightarrow x_2
\end{align*}
\]

and consider a replacement as follows:
\[(x_1, 5, 10) := (x_1, 4, 10).\]

Clearly $d$ is gbc but $d'$ is not, because $[x_1 = 4] \not\vdash 5 \subseteq x_1$.  \[ \square \]
2.6 Looking back

Recall the aim of this chapter, as explained in the introduction, viz. to obtain a theory about the component-wise construction and specification of complex systems, with a focus on issues of modularisation, parameterisation, abstraction and information hiding. Our main tool when constructing such a theory is formalisation. In order to obtain the necessary formalisations, in Section 2.2 we abstract from the details of the underlying design language, by viewing a design language (without parameterisation) as the set of terms $T_S$ of an algebraic system with preorder $\mathfrak{R}$. After that a version of lambda calculus, called $\lambda\pi$ is introduced, which serves two purposes. First of all it adds parameterisation to the algebraic system with preorder $\mathfrak{R}$ and secondly it allows us to give a meaning to the notion of a design by means of a 'design-as-lambda-terms' analogy. The set of terms in this calculus is denoted as $A_\mathfrak{R}$. In $\lambda\pi$-calculus one can derive facts of the form $\Gamma \vdash P \sqsubseteq Q$ with intuition 'P implements Q in context $\Gamma$'. The most significant differences between $\lambda\pi$-calculus and classical $\lambda$-calculus are that there is a restriction $x \subseteq P$ associated with each abstraction and that a term $(\lambda x : P.Q)A$ can only be contracted if the argument $A$ meets its restriction in the sense that $A \subseteq P$. This calculus is shown to have reasonable properties such as cut-elimination, strong normalisation and confluence. This $\lambda\pi$ is not the same as typed $\lambda$-calculus with subtypes (e.g. [15]). In $\lambda\pi$, the ordering relation $\sqsubseteq$ is on the $\lambda$-terms whereas in [15] there is a relation $\leq$ on their types. Intuitively, in $\lambda\pi$ the $x$ is 'comparable with' $P$ rather than 'in' $P$. A connection with $\lambda$-typed $\lambda$-calculus can be made by viewing a term $\lambda x : P.Q$ as the image of $\lambda x : * \rightarrow y : (x \subseteq P), Q$ under a forgetful mapping, since in $\lambda\pi$ the proof ($y$) of $(x \subseteq P)$ remains implicit. This connection is not worked out further here.

The formal definition of the set $C$ of components and the set $D$ of designs is straightforward: a component is a triple $(x, P, Q)$ (where $x$ is a new name and where $P$ is either a term or prim) and it is correct in context $\Gamma$ if $\Gamma \vdash P \sqsubseteq Q$. We call $P$ the glass-box description and $Q$ the black-box description. A design consists of a list of components and one additional term. Designs can be represented as objects of the form:

$$
\begin{align*}
    x_1 & : P_1 \sqsubseteq Q_1 \\
    x_2 & : P_2 \sqsubseteq Q_2 \\
    x_3 & : P_3 \sqsubseteq Q_3 \\
    \text{system } S &
\end{align*}
$$

where $P_1, P_2, P_3$ can be prim's or terms and where $Q_1, Q_2, Q_3$ are terms.

The definition of correctness for designs is non-trivial because of the contexts
in which components must be shown correct. Two possibilities arise, leading us to the definitions of glass-box correctness and black-box correctness. A translation from designs into lambda terms is defined, thereby obtaining a 'meaning' for our notion of a design. The meaning of a design \( d \) is a term \([d] \in A_\Sigma\). For each component of \( d \) for which the glass-box description is not \( \text{prim} \) there is an abstraction-application pair in \([d]\). For each component of \( d \) for which the glass-box description is \( \text{prim} \) there is an abstraction in \([d]\). This translation provides us with a better insight in the nature of components with a \( \text{prim} \) glass-box description: these components are the formal parameters of the design in which they occur. Furthermore there is a very remarkable relation between our notions of correctness and certain reduction strategies for lambda terms: glass-box correctness corresponds to the possibility to perform a outermost reduction and black-box correctness corresponds to the possibility to perform an innermost reduction.

There exist various kinds of modifications of designs, with the intuition that a modification takes place within one component while preserving the local correctness of that component. The question if preserving local correctness implies preserving correctness of the whole design, is investigated for various kinds of modifications. There are a number of positive and negative results. For example, that glass-box correct designs do not offer implementation freedom but black-box correct designs do. These results are summarised by the following table:

<table>
<thead>
<tr>
<th>preserving</th>
<th>modifying</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>glass-box description</td>
</tr>
<tr>
<td>black-box correctness</td>
<td>+</td>
</tr>
<tr>
<td>glass-box correctness</td>
<td>-</td>
</tr>
</tbody>
</table>

It is interesting to compare the results achieved in this chapter with the approach sketched by Girard et al. in [16] (p. 17). The principle of black-box correctness is implicitly already present in this citation.

At a more general level, abstracting away from any peculiar syntactic choice, one should see a type as an instruction for plugging things together. Let us imagine that we program with modules, i.e. closed units, which we can plug together. A module is absolutely closed, we have no right to open it. We just have the ability to use it or not, and to choose the manner of use (plugging). The type of a module is of course completely determined by all the possible pluggings it allows without crashing. In particular, one can always substitute a module with another of the same type, in the event of a
breakdown, or for the purpose of optimisation. This idea of arbitrary pluggings seems mathematisable, but to attempt this would lead us too far astray.

In a certain sense, our research can be viewed a 'mathematisation' (= formalisation) of these ideas. A precise understanding of the notions of component, black-box description and design is of great methodological importance and our formalisation and results should be of help in the systematic development of designs in realistic applications.

2.7 Acknowledgements

The formalisation of algebraic systems with preorder and the definition of the \( \lambda \pi \)-calculus and its properties have been done in cooperation with H.B.M. Jonkers, C.P.J. Roymans and G.R. Renardel de Lavalette.
Bibliography


Appendix A

Model Construction

There are several reasons why one wants to have a model for the $\lambda\pi$-calculus. First of all, if we have a non-trivial model, then we know that the calculus is consistent in the sense that not every equality (or every inequality) is derivable. Secondly, by constructing a model we can make our intuition that a lambda term $\lambda z \in P.Q$ denotes a function more precise. In this section we shall construct such a model, denoted by $\mathcal{R}^+$, provided $\mathcal{R}$ satisfies certain restrictions. The calculus is not complete with respect to this $\mathcal{R}^+$, i.e. $(\mathcal{R}^+ \models \varphi) \Rightarrow (\vdash \varphi)$ does not hold. We do not consider this as a disadvantage of our approach, because the calculus has a certain value in its own right, which does not depend on completeness. The calculus is sound with respect to $\mathcal{R}^+$, i.e. $(\vdash \varphi) \Rightarrow (\mathcal{R}^+ \models \varphi)$ – otherwise we should not call it a model.

Under certain assumptions, the model $\mathcal{R}^+$ is obtained as an extension of the underlying algebraic system with preorder $\mathcal{R}$. Lambda terms are interpreted as normal set-theoretic functions. The model contains elements $*$, which correspond to the 'undefined' terms, by which we mean the terms which cannot be contracted because of the condition in the rule ($\pi$).

In order to keep things simple, we shall adopt two additional assumptions about the underlying model $\mathcal{R}$ and we shall discuss the various possibilities arising if these additional assumptions do not hold. The first assumption is that in $\mathcal{R}$ the preorder $R$ satisfies the additional property that $\forall x, y \ (x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y)$ which means that $R$ is a partial order. The second assumption is that there is a maximum element, i.e. $\exists x \forall y \ (y \sqsubseteq x)$. Note that the example CA of Section 2.2.3 satisfies this property with 'class none end' as maximum element.

If $R$ is not a partial order, then several possibilities arise. The first possibility is that $\mathcal{R}$ has at least a substitution property for the relation $=_R$ defined as $R \cap R^{-1}$, i.e. $x =_R y :\Rightarrow x \sqsubseteq y \land y \sqsubseteq x$. The required substitution property
is that \( x =_R y \Rightarrow f_j(\ldots, x, \ldots) =_R f_j(\ldots, y, \ldots) \) for all \( j \in J \), which means that \( =_R \) is a congruence. For this first possibility we have the option that \( = \) is interpreted by \( =_R \). With this option, the model construction can still be done, but things become a bit more complicated and also the fact that \( = \) can not be interpreted by real equality is not elegant.

When \( R \) is not a partial order, the second possibility is that \( \mathbb{R} \) does not even have the substitution property for \( =_R \) mentioned above. In that case the rules \((=I)\) and \((\text{subst.})\) are essentially non-conservative. The calculus with these rules is still usable and it deals just with implementation (denoted by \( \sqsubseteq \) ) and bi-implementation (denoted by \( = \) ) where the rules \((=I)\) and \((\text{subst.})\) have an axiomatic status. Another option is to remove the rule \((=I)\) and to adapt the calculus correspondingly - as has been done in [4].

If there is no maximum element, then the model construction must be done somewhat differently. We shall discuss this below. So from now on we restrict ourselves to an algebraic system \( \mathbb{R} \) where \( R \) is a partial order and we require the existence of the maximum element.

**Notation A.1.1** If \( x \) is a function \( x : X \to A \) then \( s[x \to a] \) denotes the function which everywhere equals \( x \) but for the argument \( x \), for which it results \( a \); formally \( s[x \to a](y) = s(y) \) for \( y \neq x \) and \( s(x) = a \). □

First we define the function domains and we define the ordering of functions.

**Definition A.1.2** \( (A, \sqsubseteq, r) \). Consider \( \mathbb{R} \) as before. We define \( A_r \) and \( \sqsubseteq_r \) by induction on the structure of the type symbol \( r \). Let \( *_0 \) denote the maximum element, which is already present.

1. \( A_0 := A \),
   \( a \sqsubseteq_0 b : \Leftrightarrow a R b \).

2. \( A_{r+} := A_r \to A_r \),
   \( f \sqsubseteq_{r+} g : \Leftrightarrow \forall x \in A_r (f(x) \sqsubseteq_r g(x)) \).

When there is no maximum element, we add \( *_0 \) as a fresh 'junk object' and we define \( A_0 := A \cup \{*_0\} \) and \( a \sqsubseteq_0 b : \Leftrightarrow a R b \lor b =*_0 \). The fact that this works depends on the restriction to closed \( \varphi \) in rule \((=I)\); to see this consider a 'one-object' \( \mathbb{R} \), by which we mean a \( \mathbb{R} \) such that \( \mathbb{R} \models x = y \), where adding an object makes the formula \( \forall x \forall y ((x = y)) \) false.

We shall write \( *_{r+} \), for \( \lambda a \in A_{r+} \), where \( \lambda \) denotes functional abstraction. If no confusion arises we omit the subscripts for \( * \) and \( \sqsubseteq \). Note that \( *_{r+} \) is the maximum element of \( A_{r+} \). Note also that \( \sqsubseteq_r \) is a partial order again for all type symbols \( r \), as is easily shown by induction on the structure of \( r \). □
The model $R^+$ is a structure somewhat similar to $R$, where the most important difference is that $R^+$ has a collection of domains rather than just one. There is one domain $A_\tau$ for each type $\tau$ and consequently there is also a collection of relations $\subseteq_\tau$, rather than just one. Again we have functions $F_j$ and constants $C_i$, but these are related to $A_0$ only.

Definition A.1.3 ($R^+$). Consider $R$ as before and let Typ be the set of type symbols.

(i) The model $R^+$ is defined as
\[\{\{A_\tau \mid \tau \in \text{Typ}\}, \{\subseteq_\tau \mid \tau \in \text{Typ}\}, \{F_j \mid j \in J\}, \{C_i \mid i \in I\}\}.\]

(ii) An assignment $z$ is a map variables $\rightarrow \bigcup \{A_\tau \mid \tau \in \text{Typ}\}$ such that $z(x') \in A_\tau$, i.e. such that $z$ respects the typing.

Definition A.1.4 The interpretation of terms in $R^+$ under an assignment $z$ is defined by induction over the structure of terms. We write $P(z)$ to denote the interpretation of term $P$ under assignment $z$.

1. $z_i(z) = z(x_i)$,
2. $c_i(z) = C_i$,
3. $f_j(P_1, \ldots, P_n)(z) = F_j(P_1(z), \ldots, P_n(z))$,
4. $(PQ)(z) = P(z)Q(z)$
5. $(\lambda x : \tau. P)(z) = \mathcal{M} \ a \in A_\tau$, if $a \subseteq P(z)$ then $Q(z[x \rightarrow a])$ else $\star$. □

Note that for a term $M$ the relevant part of an assignment is the restriction of the assignment to the free variables $\bar{x} = \text{FV}(M)$; in particular, if $\forall y \notin \bar{x} \exists \bar{z}(x_1(y) = z_2(y))$ then $M(z_1) = M(z_2)$.

Definition A.1.5 We define $R^+ \models \varphi$ for $A_R$-formulae $\varphi$ built from atoms, conjunction and implication as follows.

(i) $R^+ \models (P = Q)(z) :\iff P(z) = Q(z)$.
(ii) $R^+ \models (P \subseteq Q)(z) :\iff P(z) \subseteq Q(z)$.
(iii) $R^+ \models (\varphi \land \psi)(z) :\iff R^+ \models \varphi(z)$ and $R^+ \models \psi(z)$.
(iv) $R^+ \models (\varphi \rightarrow \psi)(z) :\iff R^+ \models \varphi(z) \Rightarrow R^+ \models \psi(z)$.
(v) $R^+ \models \varphi :\iff \text{for all } z \ R^+ \models \varphi(z)$. □

Lemma A.1.6 For atomic $T_R$-formulae we have

$R \models \varphi \iff R^+ \models \varphi.$
Proof. Immediate from the definitions of $\mathbb{R} \models \varphi$ and $\mathbb{R}^+ \models \varphi$.

As a next step we want to show the soundness of $\mathbb{R}^+$ and for this purpose we introduce several lemmas about assignments and substitutions.

**Lemma A.1.7** $R(z := P)(z) = R(z[x \rightarrow P(z)])$.

**Proof.** By induction over the structure of $R$.

- **Case 1:** $R \equiv z$. Then $R(z := P)(z) = P(z)$ and $R(z[x \rightarrow P(z)]) = z[x \rightarrow P(z)] = P(z)$. Otherwise $R \equiv y \neq z$. Then $R(z := P)(z) = z(y)$ and $R(z[x \rightarrow P(z)]) = y(x \rightarrow z) = z(y)$.
- **Case 2:** $R \equiv e$. Trivial.
- **Case 3:** $R \equiv f(P_1, \ldots, P_n)$. Then $R(z := P)(z) = f(P_1(z := P), \ldots, P_n(z := P))(z) = f(P_1(z \rightarrow P(z)), \ldots, P_n(z \rightarrow P(z)))(z) = f(P_1, \ldots, P_n)(z \rightarrow P(z))$.
- **Case 4:** $R \equiv (P_1P_2)$. Then $R(z := P)(z) = P_1(z := P)(z)(P_2(z := P)(z))$ which we write as $f(a)$. Similarly we write $(P_1P_2)(z[x \rightarrow P(z)])$ as $g(b)$. By i.h. we have $f = g$ and $a = b$. Therefore $f(a) = g(b)$.
- **Case 5:** $R \equiv \lambda y \subseteq Q_1Q_2$. By the variable convention $x \neq y$. Define $f := R(z := P)(z)$ which equals
  $\lambda a$. if $a \subseteq Q_1[z := P](z)$ then $Q_1[z := P](z)$ \[a \rightarrow a\] else \(*\), and define
  $g := R(z[x \rightarrow P(z)])$ which equals
  $\lambda a$. if $a \subseteq Q_2(z[x \rightarrow P(z)])$ then $Q_2(z[x \rightarrow P(z)])(z \rightarrow a)$ else \(*\).
  Now consider an arbitrary argument, $c$ say and prove (using i.h.) that
  $f(c) = g(c)$. In particular, the induction hypothesis yields $Q_1[z := P](z)(z \rightarrow a) = Q_1(z[x \rightarrow P(z)])(z \rightarrow a)$. We must use that
  by the variable convention $P$ does not contain $y$ whence $P(z[y \rightarrow a]) = P(z)$.

**Lemma A.1.8** (Substitution) Let $z$ be an assignment. Assume $P(z) = Q(z)$.

(i) $R(z := P)(z) = R(z := Q)(z)$.

(ii) If $R_1(z := P)(z) = R_2(z := P)(z)$,
then $R_1(z := Q)(z) = R_2(z := Q)(z)$.

(iii) If $R_1(z := P)(z) \subseteq R_2(z := P)(z)$,
then $R_1(z := Q)(z) \subseteq R_2(z := Q)(z)$.

**Proof.** (i) We have $R(z[x \rightarrow P(z)]) = R(z[x \rightarrow Q(z)])$ and hence by
lemma A.1.7 we obtain \( R[x := P](a) = R[x := Q](a) \).
(ii) Write \( R[x := P] \) as \( R(P) \) etc. By (i) we have \( R_1(Q)(a) = R_1(P)(a) \) and by assumption \( R_2(P)(a) = R_2(Q)(a) \) and by (i) again \( R_3(P)(a) = R_3(Q)(a) \).
Use transitivity of =.
(iii) By (i) we have \( R_1(Q)(a) = R_1(P)(a) \), and by assumption \( R_2(P)(a) \subseteq R_2(Q)(a) \) and by (i) again \( R_3(P)(a) = R_3(Q)(a) \). Use reflexivity and transitivity of \( \subseteq \).
\( \square \)

**Theorem A.1.9 (Soundness).**

\[ \Gamma \vdash \varphi \implies \mathcal{R}^+ \models \bigwedge \Gamma \rightarrow \varphi. \]

**Proof.** The proof is by induction on the length of the derivation of \( \Gamma \vdash \varphi \).

- \((\vdash_1)\) If \( \Gamma \vdash \varphi \) is \( \Gamma \vdash f(A) \subseteq f(B) \) because then \( \Gamma \vdash A \subseteq B \) for \( f \) corresponding with a monotonic function. Just use i.h. and the monotonicity of the corresponding \( F \).
- \((\vdash_2)\) If \( \Gamma \vdash \varphi \) is a direct consequence of \( \mathcal{R} \vdash \varphi \). This case is obvious.
- \((\text{context})\). \( \Gamma, \varphi \vdash \varphi \). We must show that \( \mathcal{R}^+ \models (\bigwedge \Gamma \land \varphi)(a) \) implies \( \mathcal{R}^+ \models \varphi(a) \) which is obvious.
- \((\text{refl.})\). \( \Gamma \vdash P \subseteq P \). We must show that \( \mathcal{R}^+ \models \bigwedge \Gamma \) implies \( \mathcal{R}^+ \models P \subseteq P \) which follows from the fact that the relations \( \subseteq_r \) of \( \mathcal{R}^+ \) are partial orders.
- \((\text{trans.})\). As refl.
- \((\lambda x_1)\). \( \Gamma \vdash (\lambda x \subseteq P. Q_1) \subseteq (\lambda x \subseteq P. Q_2) \). We have \( \mathcal{R}^+ \models \bigwedge \Gamma \land x \subseteq P \rightarrow Q_1 \subseteq Q_2 \) (i.h.). Take an arbitrary \( x \) and assume \( \mathcal{R}^+ \models \bigwedge \Gamma(x) \).
Define \( f := (\lambda x \subseteq P. Q_1)(x) \) and \( g := (\lambda x \subseteq P. Q_2)(x) \). For \( a \subseteq P(x) \) we shall show \( f(a) \subseteq g(a) \). Since \( x \not\in \Gamma \) we have \( \mathcal{R}^+ \models \bigwedge \Gamma(x \rightarrow a) \).
From i.h. \( Q_1(x \rightarrow a) \subseteq Q_2(x \rightarrow a) \), i.e. \( f(a) \subseteq g(a) \). This shows \( \mathcal{R}^+ \models \bigwedge \Gamma(x) \Rightarrow f \subseteq g \).
- \((\lambda x_2)\). \( \Gamma \vdash (\lambda x \subseteq P_2. Q) \subseteq (\lambda x \subseteq P_2. Q) \). We have \( \mathcal{R}^+ \models \bigwedge \Gamma \rightarrow P_1 \subseteq P_2 \) (i.h.). Take an arbitrary \( x \) and assume \( \mathcal{R}^+ \models \bigwedge \Gamma(x) \).
Define \( f := (\lambda x \subseteq P_2.Q)(x) \) and \( g := (\lambda x \subseteq P_1.Q)(x) \). We show for arbitrary \( a \) that \( f(a) \subseteq g(a) \).
If \( a \not\subseteq P_2(x) \) and \( a \not\subseteq P_1(x) \) then \( f(a) = g(a) = \ast \).
If \( a \subseteq P_2(x) \) but \( a \not\subseteq P_1(x) \) then \( f(a) = \ast \).
If \( a \subseteq P_1(x) \) and \( a \not\subseteq P_2(x) \) then \( f(a) = g(a) \).
This shows \( \mathcal{R}^+ \models \bigwedge \Gamma \Rightarrow f \subseteq g \) so \( \mathcal{R}^+ \models \bigwedge \Gamma \rightarrow (\lambda x \subseteq P_2.Q) \subseteq (\lambda x \subseteq P_1.Q) \).
- \((\text{ap.})\). \( \Gamma \vdash (P_2.Q) \subseteq (P_2.Q) \). Take an arbitrary \( x \). Assume \( \mathcal{R}^+ \models \bigwedge \Gamma(x) \).
A. MODEL CONSTRUCTION

We have $$\mathcal{R}^+ \models \Lambda \Gamma \rightarrow P_1 \sqsubseteq P_3$$ (i.h.). Define $$a := Q(x)$$, $$f := P_3(x)$$ and $$g := P_3(x)$$. We have $$f \sqsubseteq g$$ (by i.h.). We must show $$f(a) \sqsubseteq g(a)$$. This follows directly from the definition of $$\sqsubseteq$$.

- (π). $$\Gamma \vdash (\lambda x \sqsubseteq P_1.Q)P_2 = Q[x := P_2]$$. We have $$\mathcal{R}^+ \models \Lambda \Gamma \rightarrow P_1 \sqsubseteq P_3$$ (i.h.). Take an arbitrary $$x$$. Assume $$\mathcal{R}^+ \models \Lambda \Gamma(x)$$. 
  $$(\lambda x \sqsubseteq P_1.Q)P_2(x) = \{ a \text{ if } a \sqsubseteq P_1(x) \text{ then } Q(x[a \rightarrow a]) \text{ else } \varnothing | P_3(x) \}$$
  = (i.h.) $$Q[x \rightarrow P_3(x)]$$ = $$Q[x := P_3(x)]$$ where we used lemma A.1.7.

- (≡) $$\Gamma \vdash P_1 = P_3$$ because $$\Gamma \vdash P_1 \sqsubseteq P_3$$ and $$\Gamma \vdash P_3 \sqsubseteq P_1$$. We have $$\mathcal{R}^+ \models \Lambda \Gamma \rightarrow P_1 \sqsubseteq P_3$$ and $$\mathcal{R}^+ \models \Lambda \Gamma \rightarrow P_3 \sqsubseteq P_1$$ (i.h.). Use the fact that $$\sqsubseteq$$ is a partial order.

- (subst.). $$\Gamma \vdash R_3(Q) = R_2(Q)$$ where we assume that $$\varphi$$ is an equality. We have $$\mathcal{R}^+ \models \Lambda \Gamma \rightarrow R_3(P) = R_2(P)$$ and $$\mathcal{R}^+ \models \Lambda \Gamma \rightarrow P = Q$$ (i.h.). Take an arbitrary $$x$$ and assume $$\mathcal{R}^+ \models \Lambda \Gamma(x)$$. From (i.h.)
  $$R_3(P)(x) = R_2(P)(x)$$ and $$P(x) = Q(x)$$ so by the substitution lemma (ii) we get $$R_1(Q)(x) = R_2(Q)(x)$$, i.e. $$\mathcal{R}^+ \models (R_1(Q) = R_2(Q))(x)$$. If $$\varphi$$ contains the symbol $$\sqsubseteq$$ instead of $$\varnothing$$, we proceed in a similar way, using the substitution lemma (iii).
Appendix B

List of symbols

In this appendix we give a list of the symbols used. For each symbol the list contains a very short informal description. The list has been subdivided into a number of sub-lists. The first sub-list contains general mathematical symbols. The second sub-list contains the symbols which are introduced and/or used first in Section 2. In a similar way the third sub-list contains the symbols which are introduced and/or used first in Section 3, and so on. The list does not include symbols which denote in some sense the negation of the meaning of another symbol e.g. \( \notin \) negates \( \in \) and therefore \( \notin \) is not in the list. For some symbols the list contains a relevant page number — usually the defining occurrence of the symbol.

General mathematical symbols

\( \Rightarrow, \iff \) Logical implication
\( \equiv \) Logical equivalence
\( \land, \lor \) Conjunction, disjunction
\( \forall \) Universal quantification
\( \Lambda \) Generalised conjunction
\( \equiv \) Syntactical equality
\( = \) Equality
\( \{ \} \) Set construction
\( \{ | \} \) Set comprehension
\( \in \) Set membership
\( \emptyset \) Empty set
\( \cup \) Set union
\( \cap \) Set intersection
\( \setminus \) Set difference
\( \subseteq \) Set inclusion
\( \cup \) Generalised union
B. LIST OF SYMBOLS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>|</td>
<td>Cardinality</td>
</tr>
<tr>
<td>( \mathcal{P} )</td>
<td>Power set</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>Function space</td>
</tr>
<tr>
<td>( (C)^\ast )</td>
<td>The set of sequences with elements from ( C )</td>
</tr>
<tr>
<td>( \times )</td>
<td>Cartesian product</td>
</tr>
<tr>
<td>( (, , ) )</td>
<td>Pair</td>
</tr>
<tr>
<td>( (, , \ldots , ) )</td>
<td>Tuple (= sequence)</td>
</tr>
<tr>
<td>( \mathbb{N} )</td>
<td>The set of natural numbers</td>
</tr>
<tr>
<td>( \leq )</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>0, 1, 2, \ldots</td>
<td>Natural numbers</td>
</tr>
<tr>
<td>( f : A \to B )</td>
<td>( f ) is a function from ( A ) to ( B )</td>
</tr>
<tr>
<td>:=</td>
<td>Substitution</td>
</tr>
<tr>
<td>( R^{-1} )</td>
<td>Inverse of relation ( R )</td>
</tr>
<tr>
<td>( \text{i.h.} )</td>
<td>Induction hypothesis</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>Functional abstraction</td>
</tr>
<tr>
<td>( \circ )</td>
<td>Function composition</td>
</tr>
</tbody>
</table>

Symbols concerning constructs

- \( m_1, m_2, \ldots \): Typical modules
- \( \mathbb{R} \): Typical algebraic system with preorder
- \( A \): Typical domain
- \( \mathcal{R} \): Typical preorder
- \( F_i \): Typical function
- \( J \): Typical index set of functions
- \( a_i \): Arity of function \( F_i \)
- \( C_i \): Typical constant
- \( I \): Typical index set of constants
- \( f_i \): Typical function symbol
- \( c_i \): Typical constant symbol
- \( x_0, x_1, x_2 \): Variables
- \( T_R \): Set of terms for \( \mathbb{R} \)
- \( \text{Sig} \): Signature of ...
- \( = \): Equality symbol
- \( \sqsubseteq \): Preorder (symbol)
- \( \models \): Truth (in a model)
- \( \varphi, \psi \): Typical formulae
- \( \mathbb{R}_i \): \( \mathbb{R} \) as algebraic system with preorder
- \( S \): Arbitrary set
- \( \mathbb{R}_2 \): \( \mathcal{P}(S) \) as algebraic system with preorder
- \( \text{CA} \): The Class Algebra of COLD
Symbols concerning lambda calculus

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta$</td>
<td>Classical Contraction rule</td>
<td>60</td>
</tr>
<tr>
<td>$\lambda \pi$</td>
<td>Lambda calculus based on rule ($\pi$)</td>
<td>60</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Abstractor (symbol)</td>
<td>61</td>
</tr>
<tr>
<td>$\Lambda_\pi$</td>
<td>Set of terms for $\lambda \pi$</td>
<td>62</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Basic type symbol</td>
<td>61</td>
</tr>
<tr>
<td>$\sigma, \tau$</td>
<td>Typical type symbols</td>
<td>61</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>Constructor for type symbols</td>
<td>61</td>
</tr>
<tr>
<td>$x_0, x_1, \ldots$</td>
<td>Variables</td>
<td>62</td>
</tr>
<tr>
<td>$x^i$</td>
<td>Typical variable of type $\tau$</td>
<td>62</td>
</tr>
<tr>
<td>$\sqsubseteq$</td>
<td>Partial order (symbol)</td>
<td>62</td>
</tr>
<tr>
<td>$A, B, P, Q, R, S, Z$</td>
<td>Typical elements of $\Lambda_\pi$</td>
<td>63</td>
</tr>
<tr>
<td>$x, y, z$</td>
<td>Typical variables</td>
<td>63</td>
</tr>
<tr>
<td>$FV(\ )$</td>
<td>Set of free variables</td>
<td>62</td>
</tr>
<tr>
<td>$[\ :=\ ]$</td>
<td>Substitution in a lambda term</td>
<td>67</td>
</tr>
<tr>
<td>$\top$</td>
<td>Constant true</td>
<td>63</td>
</tr>
<tr>
<td>$\wedge$</td>
<td>Conjunction</td>
<td>63</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>Implication</td>
<td>63</td>
</tr>
<tr>
<td>$\Lambda$</td>
<td>Conjunction (etc.)</td>
<td>64</td>
</tr>
<tr>
<td>$[\phi]$</td>
<td>Formula $\phi$ as an assumption</td>
<td>64</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>Typical context</td>
<td>64</td>
</tr>
<tr>
<td>$\vdash$</td>
<td>Derivation symbol</td>
<td>64</td>
</tr>
<tr>
<td>($\vdash_1$)</td>
<td>Rule of $\lambda \pi$ (algebraic system oracle)</td>
<td>65</td>
</tr>
<tr>
<td>($\vdash_2$)</td>
<td>Rule of $\lambda \pi$ (algebraic system oracle)</td>
<td>65</td>
</tr>
<tr>
<td>(context.)</td>
<td>Rule of $\lambda \pi$ (context rule)</td>
<td>65</td>
</tr>
<tr>
<td>(refl.)</td>
<td>Rule of $\lambda \pi$ (reflexivity)</td>
<td>66</td>
</tr>
<tr>
<td>(trans.)</td>
<td>Rule of $\lambda \pi$ (transitivity)</td>
<td>66</td>
</tr>
<tr>
<td>($\lambda f_1$)</td>
<td>Rule of $\lambda \pi$ (lambda introduction)</td>
<td>66</td>
</tr>
<tr>
<td>($\lambda f_2$)</td>
<td>Rule of $\lambda \pi$ (lambda introduction)</td>
<td>66</td>
</tr>
<tr>
<td>(ap.)</td>
<td>Rule of $\lambda \pi$ (application)</td>
<td>66</td>
</tr>
<tr>
<td>($\pi$)</td>
<td>Rule of $\lambda \pi$ (partial contraction rule)</td>
<td>67</td>
</tr>
</tbody>
</table>
B. LIST OF SYMBOLS

\( (=I) \)
Rule of \( \lambda \pi \) (= introduction) 68

\( (\text{subst.}) \)
Rule of \( \lambda \pi \) (substitution) 68

\( R^+ \)
Model for \( \lambda \pi \) 70

\( (\text{mon.}) \)
Rule for monotonicity 73

\( \lambda \pi_{\text{mon}} \)
Version of \( \lambda \pi \) with built-in monotonicity 73

\( R^+_{\text{mon}} \)
Model for \( \lambda \pi_{\text{mon}} \) 74

\( \rightarrow \)
One step \( \pi \) Reduction 74

\( \rightarrow^* \)
\( \pi \) Reduction 74

\( \equiv \)
\( \pi \) Equality 75

\( \text{SN} \)
Strong normalization property 76

\( \lambda \beta \)
Calculation with \( \beta \) instead of \( \pi \) 76

\( C_r, C_r^{*} \)
'Computable' terms 77

\( C_r^{*} \)
'Substitutive-computable' terms 78

\( \diamond \)
Diamond property (confluence, Church-Rosser) 80

\( \Lambda \)
Terms of classical simple-typed \( \lambda \)-calculus 79

\( \lambda \)
Coding: \( \Lambda_R \implies \lambda \) 79

\( \neq \)
Reduction with at least one step 80

Symbols concerning components

\text{prim}
Symbol serving as a placeholder for constructs 84

\( C_r' \)
The set of triples \((x, P, Q)\) 84

\( C_R, C \)
The set of components 84

\( P \)
Typical element of \( \Lambda_R \cup \{\text{prim}\} \) 84

\( c \)
Typical component 84

Symbols concerning designs

\( D_R, D \)
The set of designs 86

\( d, d', d'' \)
Typical designs 86

\( \text{csed( )} \)
Set of components of a design 86

\( d(j) \)
The \( j \)-th sub-design of \( d \) 86

\( \vdash \)
Symbol used in concrete syntax of designs 86

\( \models \)
Symbol used in concrete syntax of designs 86

\text{system}
Symbol used in concrete syntax of designs 86

\( \text{wf} \)
Well-formed 87

\( i, j, k, l, n, m \)
Typical natural numbers 87

\( \text{gbc} \)
Glass-box correct 87

\( \text{bbc} \)
Black-box correct 88

\( \mathcal{M}, \parallel \parallel \)
Meaning function for designs 91

\( \lambda_i \)
Lambda with index 92

\( \Delta_R \)
Set of terms with indices 92

\( \pi_j \)
reduction of index \( j \) 93
\[ \pi' \] reduction with indices 93
\[ \mid \mid \] mapping forgetting indices 93
\[ M', \{ \} ' \] Meaning function with indices 94
omr Outermost reducible 95
imr Innermost reducible 96
unf(d) Unfolding of d 97
\[ \sigma_{\beta}, \rho_{\beta} \] Typical substitutions 97
\[ S \] Typical sequence of substitutions 97
\[ \text{gbc-gb-mod} \] gbc-preserving glass-box modification 100
\[ \text{bbc-gb-mod} \] bbc-preserving glass-box modification 100
\[ \text{gbc-bb-mod} \] gbc-preserving black-box modification 101
\[ \text{bbc-bb-mod} \] bbc-preserving black-box modification 102

Symbols concerning Model Construction

\[ \varphi \{ \xi \} \] Interpretation under assignment \( \xi \) 112
\[ *_{r} \] Interpretation of undefined terms 110
\[ \lambda_{0} \] Basic domain 111
\[ A_{\rightarrow} \] Function domain 111
\[ \xi \mid x \rightarrow a \mid \] Modification of an assignment 112
\[ Typ \] Set of type symbols 112
\[ *_{r} \] Maximal element 111
Chapter 3

Correctness-Preserving Transformations of Designs

3.1 Introduction

In this chapter we want to study correctness preserving transformations of designs. We refer to Chapter 2 for the definition of the syntax and the semantics of designs. In Chapter 2 we investigated already a number of simple transformations such as the black-box correctness preserving glass-box modifications (bbc-gb-mod). These modifications take one design as an input and yield one design as a result. Now we shall also consider binary operations (\(\ast, \circ\)) on designs. This implies that we consider the construction of a design by fitting together two designs. Conversely, these operations can be used for describing the process of splitting a given design into smaller designs. The interesting point, of course, is that under certain conditions the result of applying a binary operation to two bbc designs is a bbc design again. We must investigate these conditions and the algebraic laws that hold for the binary operations (both at the level of designs and of their semantics in terms of \(\lambda\pi\), via \(\beta\)). It turns out that in order to define these operations on designs it is convenient to use sequences within \(\lambda\pi\). Therefore we need an extension of the \(\lambda\pi\)-calculus. Such an extension is given in appendix A.

In Section 2 we give a new definition of the set of designs. This definition is somewhat more general than the definition given in Chapter 2. Furthermore we discuss two conditions on designs which enable us to restrict ourselves to a convenient kind of designs.

In Section 3 we shall investigate several operations on designs. We shall have binary operations \(\ast\) and \(\circ\). We briefly investigate a binary operation \(\|$ which however turns out to be somewhat disappointing. We also have two unary
operations called bot and top. Furthermore there are two binary predicates called gbv and bbv. In this way we obtain an algebra of designs. At the end of Section 3 we shall include a summary of this algebra of designs.

In Section 4 we shall investigate models of the development process which describe design creation. Therefore we shall start with a discussion of the validation of a given design with respect to a given machine/user-context. We shall describe models of the development process as highly non-deterministic design-programs. These design-programs are of an imperative nature, by which we mean that they may contain assignment statements. We need a simple language which serves as a design-development language in which we express these design-programs. Such a language, inspired by COLD, but precisely tailored to our needs is given in appendix B. In Section 3.4 we include a brief sketch of the main ingredients of this design-development language – see subsection 3.4.3. The binary operation $\circ$ will play a role in our discussion of validation. The binary operation $\ast$ will be used in our description of design creation.

In Section 5 we shall discuss the evolution of a given design with respect to its context in connection with issues of design validation. We shall present some simple models of the development process which deal with design evolution. The binary operation $\circ$ will be used in our description of design evolution.

The fact that we can partition designs and reassemble them again makes it possible to discuss models of the development process where two (or more) developers each operate on a part of a design such that when each of them has finished his part, their results are fitted together to yield a new design which is both bdc and valid. We shall refer to this type of development process as parallel development. Both binary operations $\ast$ and $\circ$ will be used in our description of parallel development. Section 6 is devoted to design partition and to parallel development.

In this chapter we shall focus on those aspects of the software development process which are connected with the idea of structuring the software product as a design. In view of the fact that the notion of a design is a formalisation of the approach of having 'black-box descriptions', this focusing leads us to an investigation of concepts which deal with the manageability of the software development process: top-down development, layered designs, validation, design evolution. It will be the 'leitmotiv' of this chapter that whenever possible we shall cast the relevant notions in the form of designs and algebraic operations on designs.

The models of the development process described in this chapter are most certainly not meant to be exclusive: any technique for obtaining valid and correct designs is a good technique as such. Rather than being prescriptive,
we aim at the formulation of some general principles which may be of help
for realistic development processes.

The proofs of the theorems and lemmas in this chapter are given with a level
of detail which is meant for convincing the reader rather than for enabling
mechanical proof-checking.

The Sections 1, 2, 3, 4, 5 and 6 constitute the main line of development of this
chapter whereas appendix A and appendix B are rather technical digressions.
Definitions from appendix A are used in Sections 2 and 3. Definitions from
appendix B are used in Sections 4, 5 and 6. It has been tried to make both
appendix A and appendix B independent of the Sections 1-6. We suggest the
following order to the reader: 1-2-3-4-5-6 (possibly reading parts of A and
B on a need-to-know basis), followed by appendices A and B. Appendix C
contains a list of symbols.

3.2 Designs

3.2.1 Designs where the system is a sequence

In this section we give a new definition of the set of designs. This definition is
somewhat more general than the definition given in Chapter 2. Furthermore
we introduce some notation.

In Chapter 2 the set $D_R$ of designs has been defined for a given algebraic
system with preorder $R$. This notion of 'design' is not symmetric in the
sense that a design can have several prim components whereas its system
consists of just one term. This fact seems to be an obstacle when we want to
define binary operations on designs. We view it as an obstacle because we
want to define an operation $\circ$ on designs where we replace the prims of one
design by elements from the system of another design. Therefore we extend
our definition of 'design' and we shall use the $\lambda\pi$-calculus with sequences
as given in appendix A. We consider an algebraic system with preorder $R$.
The set of terms of $\lambda\pi$-calculus is denoted as $\Lambda_R$. Among the terms we have
so-called sequences which are of the form $[P_1, \ldots, P_m]$ and which have a type
of the form $\langle r_1, \ldots, r_m \rangle$; the introduction of sequences does not disturb the
good properties of $\lambda\pi$ such as confluence and strong normalization. The set
$C_R$ of components is defined as in Chapter 2, except for the fact that we use
the $\lambda\pi$-calculus with sequences.

**Definition 3.2.1** Consider the $\lambda\pi$-calculus with sequences and whose set of
terms is $\Lambda_R$. Consider also $C_R$ which is the set of components constructed
over this $\Lambda_R$. Each component is either a triple $(x, P, Q)$ or $(x, \text{prim}, Q)$ for
$P, Q \in \Lambda_{\mathbb{R}}$. Then the set $D'_0$ of designs is defined as the subset of $(C_{\mathbb{R}})^* \times \Lambda_{\mathbb{R}}$ given by

\[
\{(c_1, \ldots, c_k), S) \mid S \text{ has a type } (r_1, \ldots, r_m)\}
\]

for $k \geq 0, m \geq 0$, (k and m not fixed).

So in fact we have very much the same kind of designs as in Chapter 2. The only differences are that now we adopted a richer set of lambda terms and that now the system of each design must be a sequence. We adopt an obvious notation for components and designs with symbols $:=$ and $\sqsubseteq$ and system. Note that we can embed $D_0$ in $D'_0$ by mapping `$\ldots$ system $S$' to `$\ldots$ system $[S]$'. We adopt the same definition of well-formedness (wf), glass-box correctness (gbc), black-box correctness (bbc) and semantics ($\models$) etc. for designs as in Chapter 2. For the remainder of this chapter we restrict ourselves to designs from $D'_0$.

**Example 3.2.2** Consider $\mathbb{R}_1 = (\mathbb{N}, \leq, \{+\}, \{0, 1, 2, \ldots\})$. The following design is an element of $D'_0$ because its system is a sequence.

\[
x := \text{prim} \sqsubseteq 7
\]

\[
y := x + 2 \sqsubseteq 9
\]

\[
\text{system } [x, y, x + y + 1].
\]

**Remark 3.2.3** It is important to realize that the mechanism of introducing components, `$x := P \sqsubseteq Q$' say, is much more complex than an abbreviation mechanism as in `$x := P$'. The essential difference is that with components non-trivial issues of information-hiding arise. Yet we think that it is a good idea to employ an abbreviation mechanism as well: it should be possible to write `$x := P$' and from then on use $x$, knowing that $x$ can be replaced by $P$. Abbreviations which have a scope consisting of one design are called local. It is possible to view a local abbreviation `$x := P$' as a special kind of component. Abbreviations which have a scope consisting of a collection of designs are called global. We think that both a facility for local abbreviations and a facility for global abbreviations are useful. In this chapter we shall not investigate such abbreviation facilities any further.

We shall need some notation.

**Definition 3.2.4** Let $d$ be a wf design.

(i) $\text{cs}(d) :=$ the set of component names of $d$,

(ii) $\text{sys}(d) :=$ the set of comp. names that occur freely in the system of $d$. 

3.2. DESIGNS

Note that \( \text{sys}(d) \subseteq \text{cset}(d) \).

Definition 3.2.5 (arity). (i) For a design \( d \), the **arity of \( d \)**, notation \( \text{arity}(d) \) is the pair \( (\sigma_1, \ldots, \sigma_n, (r_1, \ldots, r_m)) \) where the \( \sigma_1, \ldots, \sigma_n \) are the types of the names of the \( \text{prim} \) components of \( d \) and where \( (r_1, \ldots, r_m) \) is the type of the system of \( d \).

(ii) For a design \( d \), the **reduced arity of \( d \)**, also denoted as \( \text{arity}(d) \) is the pair \( (n, m) \) where \( n \) is the number of \( \text{prim} \) components of \( d \) and where \( m \) is the length of the system of \( d \).

Throughout this chapter we shall restrict ourselves to designs where all glass-box descriptions have type 0 and where the system has type \( (0, \ldots, 0) \), i.e. all system elements have type 0. This restriction gives us the advantage that we need not worry about the types of terms and the arities of designs. Under this restriction it will be sufficient to employ the reduced arity only. The restriction is by no means essential and the extension of our definitions and results towards arbitrary designs is straightforward. In view of this restriction we shall simply use the term ‘arity’ for ‘reduced arity’.

We must be precise about **equality** of designs.

Definition 3.2.6 ( = ). Let \( d_1, d_2 \) be \( \text{wf} \) designs. \( d_1 = d_2 \) means that \( d_2 \) can be obtained from \( d_1 \), by means of a systematic renaming of bound variables. In this definition it is understood that both the variables bound by lambdas and the names of components are considered as bound variables. It is also understood that clashes of free and bound variables must be avoided. Sometimes we shall state explicitly that two designs \( d_1 \) and \( d_2 \) must have disjoint sets of component names \( \text{cset}(d_1) \cap \text{cset}(d_2) = \emptyset \) in order that some operation is defined.

Recall that in Chapter 2 we defined a mapping \( \llbracket \cdot \rrbracket \) from designs to lambda terms, viewing \( \llbracket d \rrbracket \) as the semantics of \( d \). (Of course it is possible to push the process of semantical interpretation one step further by interpreting \( \llbracket d \rrbracket \) in some model of \( \lambda \xi \), but throughout this chapter we shall not do that.) We shall say that two \( \text{wf} \) designs \( d_1 \) and \( d_2 \) are **semantically equivalent** if \( \llbracket d_1 \rrbracket \) and \( \llbracket d_2 \rrbracket \) are equal in the sense of \( \lambda \xi \)-calculus, i.e. if \( \vdash \llbracket d_1 \rrbracket = \llbracket d_2 \rrbracket \) is derivable. Sometimes we write \( d_1 =_{\llbracket \cdot \rrbracket} d_2 \) if \( \vdash \llbracket d_1 \rrbracket = \llbracket d_2 \rrbracket \). Note that \( d_1 = d_2 \) implies \( d_1 =_{\llbracket \cdot \rrbracket} d_2 \) by rule (ref.).

Even without referring to properties of the algebraic system with preorder \( \mathcal{R} \), each design can be shown to have a lot of semantical equivalents. Under certain conditions, this fact will give us some freedom in choosing a convenient kind of representatives from the classes of semantically equivalent designs. We shall have two such conditions, viz. a condition called **directly specified**
(abbreviated ds) to be introduced in Section 3.2.3 and glass-box correctness (gbc).

3.2.2 Permuting Components

This section contains preparations which we shall need in Section 3.2.3 when we shall introduce a condition called 'prims-first' (abbreviated pf). These preparations include a lemma which we shall call the 'component swap lemma'.

From an intuitive point of view, it is clear that the order of the non-prim components in a design to a certain extent is irrelevant. First note that the order of the prim components is relevant however, as is illustrated by the following example.

Example 3.2.7 Consider the algebraic system with preorder $\mathcal{R}_1$ as before. Let $d$ and $d'$ respectively be given by

\[
\begin{align*}
y & := \text{prim} \sqsubseteq 1 & x & := \text{prim} \sqsubseteq 2 \\
z & := \text{prim} \sqsubseteq 2 & y & := \text{prim} \sqsubseteq 1 \\
\text{system} & \lfloor x \rfloor, & \text{system} & \lfloor x \rfloor.
\end{align*}
\]

Now $[d] = \lambda y \sqsubseteq 1.\lambda x \sqsubseteq 2.\lfloor x \rfloor$ whereas $[d'] = \lambda x \sqsubseteq 2.\lambda y \sqsubseteq 1.\lfloor x \rfloor$ whence $\forall [d] = [d']$.

The next example shows a case where two components can be swapped.

Example 3.2.8 Consider the algebraic system with preorder $\mathcal{R}_1$ as before. Let $d$ and $d'$ respectively be given by

\[
\begin{align*}
y & := 1 \sqsubseteq 2 & x & := \text{prim} \sqsubseteq 2 \\
z & := \text{prim} \sqsubseteq 2 & y & := 1 \sqsubseteq 2 \\
x & := x + y \sqsubseteq 4 & z & := x + y \sqsubseteq 4 \\
\text{system} & \lfloor z \rfloor, & \text{system} & \lfloor z \rfloor.
\end{align*}
\]

Now $[d] = (\lambda y \sqsubseteq 2.\lambda x \sqsubseteq 2.(\lambda x \sqsubseteq 4.\lfloor x \rfloor)(x + y))1$ and $[d] = \lambda x \sqsubseteq 2.\lfloor x + 1 \rfloor$. Also $[d'] = (\lambda y \sqsubseteq 2.(\lambda x \sqsubseteq 4.\lfloor x \rfloor)(x + y))1$ and $[d'] = \lambda x \sqsubseteq 2.\lfloor x + 1 \rfloor$.

We formalise the idea behind the latter example below.

Lemma 3.2.9 (Component swap lemma). Let $d$ and $d'$ be wf designs. Let $d$ be gbc. Let $d'$ be obtained from $d$ by swapping two adjacent components, where at least one of these components is non-prim. Then we have:
3.2. DESIGNS

(i) $d'$ is gbc,

(ii) $d$ is bbc $\iff d'$ is bbc,

(iii) $\vdash [d] = [d']$.

Proof. Let $d$ and $d'$ respectively be given by

$$
\begin{align*}
x_1 & := P_1 \sqsubseteq Q_1 & x_1 & := P_1 \sqsubseteq Q_1 \\
\vdots & & \vdots & \\
x_i & := P_i \sqsubseteq Q_i & x_{i+1} & := P_{i+1} \sqsubseteq Q_{i+1} \\
x_{i+1} & := P_{i+1} \sqsubseteq Q_{i+1} & x_i & := P_i \sqsubseteq Q_i \\
\vdots & & \vdots & \\
x_n & := P_n \sqsubseteq Q_n & x_n & := P_n \sqsubseteq Q_n \\
\text{system } [M] & & \text{system } [M].
\end{align*}
$$

First of all we note that since $d$ is wf, $x_{i+1}$ does not occur in $P_i, Q_i, P_{i+1}$ and $Q_{i+1}$. Similarly, since $d'$ is wf, $x_i$ does not occur in $P_i, Q_i, P_{i+1}$ and $Q_{i+1}$.

(i) We know that $d$ is gbc and we must prove that $d'$ is gbc. Throughout this part (i) we shall employ several of the derived rules of $\lambda \alpha$ as worked out in Chapter 2 (weakening, cut-rule etc.). Clearly, for all components but for those named $x_i$ and $x_{i+1}$, the correctness condition is not affected by the swapping. The correctness of the component named $x_i$ in $d'$ follows from its correctness in $d$ by weakening. For the component named $x_{i+1}$ in $d'$ we reason as follows. If $P_{i+1} \equiv \text{prim}$ then the correctness condition is trivially true, so we assume that $P_{i+1} \not\equiv \text{prim}$. Let us consider the case where also $P_i \not\equiv \text{prim}$. We use the fact that $d$ is gbc and hence $\Gamma, [x_i = P_i] \vdash P_{i+1} \sqsubseteq Q_{i+1}$ where $\Gamma$ is the context with $i - 1$ assumptions as given by the definition of gbc. Now note that $x_i$ occurs neither in $\Gamma$ nor in $P_i, P_{i+1}, Q_{i+1}$. By performing the substitution $[x_i := P_i]$ [both left and right of the ""] we get $\Gamma', [P_i = P_i] \vdash P_{i+1} \sqsubseteq Q_{i+1}$. Now we can use the reflectivity of $\equiv$ in $\lambda \alpha$ and the cut-rule to get $\Gamma \vdash P_{i+1} \sqsubseteq Q_{i+1}$. We might summarise this by saying that the assumption $[x_i = P_i]$ does not play an essential role in the derivation of $P_{i+1} \sqsubseteq Q_{i+1}$. We have shown that the component named $x_{i+1}$ is correct in $d'$. In the case where $P_i \equiv \text{prim}$ we can have a similar reasoning. This shows that $d'$ is gbc.

(ii) We can show (\implies) in a similar way as (i), where instead of an assumption $[x_i = P_i]$ we have an assumption $[x_i \subseteq Q_i]$. For the converse (\iff) we note that if $d'$ is obtained from $d$ by swapping, then also $d$ can be obtained from $d'$ by swapping.

(iii) Since $d$ is gbc, we can contract all abstraction-application pairs of $[d]$ which correspond to a non-prim component and we get $\vdash [d] = (\mathcal{L}[M])$ where
- \( \mathcal{L} \) is the sequence of abstractions \( \lambda x_j \subset Q_j \),
  for \( P_j \equiv \text{prim} \) with increasing \( j \).
- \( S \) is the sequence of substitutions \( [x_j := P_j] \)
  for \( P_j \not\equiv \text{prim} \) with decreasing \( j \).

Similarly let \( \vdash [d'] = (\mathcal{L}[M])S' \). If \( P_i \equiv \text{prim} \) or \( P_{i+1} \equiv \text{prim} \), then \( x_i \)
is bound in \( \mathcal{L} \) whereas the component named \( x_{i+1} \) plays a role in \( S \) (or conversely). In both cases the order of these components in the design does not matter, by which we mean that \( (\mathcal{L}[M])S \equiv (\mathcal{L}[M])S' \). The remaining case is \( P_i \not\equiv \text{prim} \) and \( P_{i+1} \not\equiv \text{prim} \). Let \( S \) be given as \( S_1[x_{i+1} := P_{i+1}][x_i := P_i]S_2 \), then \( S' \) must be \( S_1[x_i := P_i][x_{i+1} := P_{i+1}]S_2 \). Note that \( S \) and \( S' \) are effectively the same substitutions because \( x_i \not\in FV(P_{i+1}) \) and \( x_{i+1} \not\in FV(P_i) \). This shows \( \vdash [d] = [d'] \).

By repeated application of the component swap lemma it is also possible to swap components that are not adjacent. Some care is needed in order to preserve the applicability conditions of the component swap lemma during the intermediate swap steps. Several cases arise and we show one of them as a lemma below.

**Lemma 3.2.10** Let \( d \) and \( d' \) be wf designs. Let \( d \) be gbc. Let \( d' \) be obtained from \( d \) by swapping two non-prim components. Then the same conclusions (i), (ii) and (iii) as in lemma 3.2.9 hold.

**Proof.** Let \( d \) and \( d' \) respectively be given by

\[
\begin{align*}
  x_1 &:= P_1 \subset Q_1 & x_1 &:= P_1 \subset Q_1 \\
  \vdots & & \vdots \\
  x_i &:= P_i \subset Q_i & x_{i+k} &:= P_{i+k} \subset Q_{i+k} \\
  \vdots & & \vdots \\
  x_{i+k} &:= P_{i+k} \subset Q_{i+k} & x_i &:= P_i \subset Q_i \\
  \vdots & & \vdots \\
  x_n &:= P_n \subset Q_n & x_n &:= P_n \subset Q_n \\
  \text{system } [M] & & \text{system } [M],
\end{align*}
\]

which means that in \( d \) the two components to be swapped are separated by \( k-1 \) components in between. Our strategy is to move component \( i \) downwards past component \( i+k \) by \( k \) swaps first. The essential observation is that since \( d' \) is wf, \( x_i \) does not occur in \( P_i, Q_i, P_{i+1}, Q_{i+1}, \ldots, P_{i+k}, Q_{i+k} \). Therefore the intermediate designs resulting after \( 1, 2, \ldots, k \) steps are all wf. So for each swap the conclusions of the component swap lemma carry over to the resulting intermediate design. In particular, the intermediates are gbc.

As the second part of our strategy, we make component \( i+k \) (by which
we mean $x_{i+k} := P_{i+k} \sqsubseteq Q_{i+k}$) bubble upwards by $k-1$ swaps. We can apply the component swap lemma $k-1$ times. Note also that the $k-1$ 'in-between' components are allowed to contain prim's because the components $i$ and $i+k$ are both non-prim. Hence in each swap, at least one of the components involved is non-prim.

Very much in the same way we have that the conclusions of the component swap lemma also hold if the components are not adjacent, provided the components to be swapped have no prim component between them.

### 3.2.3 The conditions 'pf' and 'ds'

We think that it is convenient to restrict ourselves to designs organised as indicated by the following condition:

**Definition 3.2.11 (pf)** Let $d$ be a design. We say that $d$ is prim-first (abbreviated pf) if its prim components occur before its non-prim components.

We give an example.

**Example 3.2.12** Suppose that in the design $d$ given below $P_1, \ldots, P_m$ are not prim. Then $d$ is pf.

\[
\begin{align*}
x_1 & := \text{prim} \sqsubseteq M_1 \\
& \quad \ldots \\
x_n & := \text{prim} \sqsubseteq M_n \\
y_1 & := P_1 \sqsubseteq Q_1 \\
& \quad \ldots \\
y_m & := P_m \sqsubseteq Q_m \\
\text{system} & \mid [S_1, \ldots, S_t].
\end{align*}
\]

Under a certain condition (called 'directly specified') to be worked out below, a design which is wf and gbc can be transformed into a semantically equivalent pf design.

**Definition 3.2.13 (ds).** We say that a design $d$ is directly specified (abbreviated ds) if it is wf and no black-box description contains the name of a component.

Let us have a look at an example of a design which is not ds. Consider the algebraic system with preorder $R_1 = (\mathbb{N}, \leq, \{+\}, \{0, 1, 2, \ldots\})$. Now consider the following design:
Example 3.2.14

\[ x := \text{prim} \subseteq 7 \]
\[ y := P \subseteq x \]
\[ z := \text{prim} \subseteq y \]

system \([S_1, \ldots, S_i]\). □

In order to see that there is something strange with this design, assume that we are interested in choosing \(P\) such that we can show the correctness of the second component, while adopting black-box correctness. We must prove \(P \subseteq x\) under the assumption \(x \subseteq 7\). Since we can think of \(\subseteq\) as the ordering \(\leq\) on natural numbers, we are very limited in finding some \(P \leq x\) if all we know about \(x\) is \(x \leq 7\) (of course we could take \(x\) itself). Furthermore, as another problem, it should be noted that if we swap the second and the third component, we get a design which is not even wf. The condition \(ds\) serves to avoid this and similar situations.

Lemma 3.2.15 Let \(d\) be a ds design. Then there is a wf and pf design \(d'\) such that

(i) \(d\) is gbc \(\Leftrightarrow\) \(d'\) is gbc,
(ii) \(d\) is bbc \(\Leftrightarrow\) \(d'\) is bbc,
(iii) \(d\) is gbc \(\Rightarrow\) \(\vdash [d] = [d']\).

Proof. The fact that \(d\) is ds enables us to put the prim components before the non-prim components without violating the condition wf. The fact that this transformation preserves glass-box correctness, black-box correctness and semantics follows by repeated application of the component swap lemma 3.2.9. The possibility to apply lemma 3.2.9 repeatedly depends on the fact that after each swap we get a wf and ds design again; this is the case since the design is ds and one of the components involved in the swap is always a prim component moving upwards. □

We are often interested in designs which are ds and gbc. In fact we shall prefer designs which are even bbc but this condition seems not needed here yet. If all designs are ds and gbc then by lemma 3.2.15 we can transform each design into an equivalent pf one. Therefore we shall from now on focus on designs which are pf and also ds. It follows that for each algebraic operation which we shall define, we shall have to verify that the result of applying the operation to pf and ds arguments yields a pf and ds design again.

We end this section with a lemma which sometimes is useful for simplifying designs.
Lemma 3.2.16 (Component cancellation lemma). Let $d$ and $d'$ be pf ds designs. Assume that in $d$ there are components $(x_i := P_i \sqsubseteq Q_i)$ and $(x_j := x_i \sqsubseteq Q_i)$ where $i \neq j$. Let $d'$ be obtained from $d$ by removing the component $(x_j := x_i \sqsubseteq Q_i)$ and replacing all occurrences of $x_j$ by $x_i$. Then the following hold:

(i) $d$ is bbc $\Rightarrow$ $d'$ is bbc,
(ii) $\Gamma \vdash [d] = [d']$.

Proof. Let $d$ and $d'$ respectively be given by

\[
\begin{align*}
  x_1 & := P_1 \sqsubseteq Q_1 \\
  \ldots \\
  x_i & := P_i \sqsubseteq Q_i \quad x_i := P_i \sqsubseteq Q_i \\
  \ldots \\
  x_{j-1} & := P_{j-1} \sqsubseteq Q_{j-1} \\
  x_{j-1} & := P_{j-1} \sqsubseteq Q_{j-1} \\
  x_j & := x_i \sqsubseteq Q_i \\
  x_{j+1} & := P_{j+1}[x_j := x_i] \sqsubseteq Q_{j+1} \\
  \ldots \\
  x_n & := P_n[x_j := x_i] \sqsubseteq Q_n \\
\end{align*}
\]

system $S$.

(i) Components named $x_1, \ldots, x_{j-1}$ are correct in $d$ iff they are correct in $d'$. For a component $(x_k := P_k \sqsubseteq Q_k)$ with $j < k \leq n$ the correctness condition in $d$ is $\Gamma, [x_j \sqsubseteq Q_j], [x_{j+1} \sqsubseteq Q_{j+1}], \ldots, [x_{k-1} \sqsubseteq Q_{k-1}] \vdash P_k \sqsubseteq Q_k \quad \ldots(*)$

and in $d'$ it is $\Gamma, [x_{j+1} \sqsubseteq Q_{j+1}], \ldots, [x_{k-1} \sqsubseteq Q_{k-1}] \vdash P_k[x_j := x_i] \sqsubseteq Q_k \quad \ldots(\ast\ast)$

where $\Gamma$ abbreviates $[x_1 \sqsubseteq Q_1], \ldots, [x_{j-1} \sqsubseteq Q_{j-1}]$. Assume $(\ast)$.

By performing the substitution $[x_j := x_i]$ (both to the left and right of the '$\vdash$') we get $\Gamma, [x_i \sqsubseteq Q_i], [x_{j+1} \sqsubseteq Q_{j+1}], \ldots, [x_{k-1} \sqsubseteq Q_{k-1}] \vdash P_k[x_j := x_i] \sqsubseteq Q_k$. Here we used the fact that $x_j$ does not occur in $\Gamma$ and hence $\Gamma[x_j := x_i] = \Gamma$. Now note that the assumption $[x_i \sqsubseteq Q_i]$ is already in $\Gamma$. This shows $(\ast\ast)$.

(ii) Consider the term $[d]$. This term contains the abstraction-application pair $(\lambda x_j \in Q_j \cdots . x_i)$ which corresponds to the component $(x_j := x_i \sqsubseteq Q_i)$. This abstraction-application pair occurs in the scope of an abstraction $\lambda x_j \in Q_j$, and therefore it can be contracted via $\pi$-reduction. This $\pi$-reduction yields precisely $[d']$.

□

Remark 3.2.17 (Strengthening of the component cancellation lemma). The proposition that in the component cancellation lemma 3.2.16 we have $d'$ is bbc $\Rightarrow$ $d$ is bbc, (i.e. the converse of (i)) is false. This can be shown by the
following counterexample. Let the \( \text{pf} \land \text{ds} \) designs \( d \) and \( d' \) respectively be given as

\[
\begin{align*}
x_1 & := \text{prim} \sqsubseteq Q_1 & x_1 & := \text{prim} \sqsubseteq Q_1 \\
x_2 & := x_1 \sqsubseteq Q_1 & x_3 & := (\lambda z \sqsubseteq x_2, Q_2)x_2 \sqsubseteq Q_2 \\
x_3 & := (\lambda z \sqsubseteq x_3, Q_3)x_3 \sqsubseteq Q_3 & \quad \text{system } S.
\end{align*}
\]

Now \( d' \) is bbc because \( \vdash (\lambda z \sqsubseteq x_3, Q_3)x_3 = (\text{rule } \pi, \text{refl.}) \quad Q_2[x := x_1] \equiv Q_2 \sqsubseteq Q_2 \) whereas in \( d \) we can not prove the black-box correctness of the third component.

If for \( d \) and \( d' \) as in the component cancellation lemma 3.2.16 we have additionally that there is no glass-box description \( P_k \) \((j + 1 \leq k \leq n)\) in which both \( x_i \) and \( y_j \) occur then the converse of (i) holds: \( d' \) is bbc \( \Rightarrow \) \( d \) is bbc. \( \square \)

Sometimes it will be convenient to represent a context (i.e. a set of assumptions) by a design and this is formalised in definition 3.2.18 below.

**Definition 3.2.18** Let the \( \text{pf} \land \text{ds} \) design \( d \) be given as

\[
\begin{align*}
x_1 & := \text{prim} \sqsubseteq M_1 \\
\quad \ldots \\
x_n & := \text{prim} \sqsubseteq M_n \\
y_1 & := P_1 \sqsubseteq Q_1 \\
\quad \ldots \\
y_m & := P_m \sqsubseteq Q_m \\
\quad \text{system } [S_1, \ldots, S_l].
\end{align*}
\]

where \( P_1, \ldots, P_m \) are not equal to \text{prim}. Then we define the black-box context of \( d \) and the glass-box context of \( d \) respectively as follows.

(i) \( \Gamma_{\text{bb}}(d) \) is \( [x_1 \sqsubseteq M_1], \ldots, [x_n \sqsubseteq M_n], [y_1 \sqsubseteq Q_1], \ldots, [y_m \sqsubseteq Q_m] \),

(ii) \( \Gamma_{\text{gb}}(d) \) is \( [x_1 \sqsubseteq M_1], \ldots, [x_n \sqsubseteq M_n], [y_1 = P_1], \ldots, [y_m = P_m] \). \( \square \)

### 3.3 Algebraic Operations on Designs

#### 3.3.1 The operation \( \ast \)

The first operation on designs is called concatenation and is denoted by \( \ast \). The concatenation of \( d_1 \) and \( d_2 \) yields a design containing the components of \( d_1 \) and the components of \( d_2 \) and having as its system simply the concatenation of the systems of \( d_1 \) and \( d_2 \).
3.3. ALGEBRAIC OPERATIONS ON DESIGNS

We have the following intuition. The designs \( d_1 \) and \( d_2 \) are considered as ‘disjoint’ designs and by constructing \( d_1 \ast d_2 \), we simply take a kind of union of their component sets. It is important to realise that designs \( d_1 \) and \( d_2 \) have no such thing as “free parameters” and that each design constitutes very much a closed unit. Conversely, if we can split a design \( d \) into \( d_1 \) and \( d_2 \), such that \( d = d_1 \ast d_2 \) then this means that \( d \) consisted in fact already of two unrelated parts.

This binary operation \( \ast \) will be used in our description of design creation (Section 3.4); there \( \ast \) turns out to be useful for describing the addition of one or more \( \text{prim} \) components to a given design. Furthermore, since \( \ast \) can be used for describing the splitting of a design into two smaller designs, it will be used in our description of design partition and parallel development (Section 3.6).

**Definition 3.3.1 (\( \ast \)).** Assume \( \text{pf} \wedge \text{ds} \) designs \( d_1 \) and \( d_2 \) where \( \text{cset}(d_1) \cap \text{cset}(d_2) = \emptyset \). Let \( d_1 \) and \( d_2 \) respectively be given by

\[
\begin{align*}
\mathbf{z}_1 & := \text{prim} \subseteq M_1 & \mathbf{u}_1 & := \text{prim} \subseteq N_1 \\
\vdots & & \vdots \\
\mathbf{z}_n & := \text{prim} \subseteq M_n & \mathbf{u}_n & := \text{prim} \subseteq N_n \\
\mathbf{y}_1 & := P_1 \subseteq Q_1 & \mathbf{v}_1 & := A_1 \subseteq B_1 \\
\vdots & & \vdots \\
\mathbf{y}_j & := P_j \subseteq Q_j & \mathbf{v}_j & := A_j \subseteq B_j \\
\text{system} [S_1, \ldots, S_m], & & \text{system} [T_1, \ldots, T_m],
\end{align*}
\]

where \( P_1, \ldots, P_j \) are not equal to \( \text{prim} \) and where \( A_1, \ldots, A_j \) are not equal to \( \text{prim} \). Then we define \( d_1 \ast d_2 \) as

\[
\begin{align*}
\mathbf{z}_1 & := \text{prim} \subseteq M_1 \\
\vdots & & \vdots \\
\mathbf{z}_n & := \text{prim} \subseteq M_n \\
\mathbf{u}_1 & := \text{prim} \subseteq N_1 \\
\vdots & & \vdots \\
\mathbf{u}_n & := \text{prim} \subseteq N_n \\
\mathbf{y}_1 & := P_1 \subseteq Q_1 \\
\vdots & & \vdots \\
\mathbf{y}_j & := P_j \subseteq Q_j \\
\mathbf{v}_1 & := A_1 \subseteq B_1 \\
\vdots & & \vdots \\
\mathbf{v}_j & := A_j \subseteq B_j \\
\text{system} [S_1, \ldots, S_m, T_1, \ldots, T_m].
\end{align*}
\]

Note that \( d_1 \ast d_2 \) is a \( \text{pf} \wedge \text{ds} \) design again. Note also that if \( \text{arity}(d_1) = \)
(n₁, m₁) and arity(d₂) = (n₂, m₂), then arity(d₁ * d₂) = (n₁ + n₂, m₁ + m₂).
Finally note that cset(d₁ * d₂) = cset(d₁) ∪ cset(d₂).

In Section 3.1 we already announced an algebra of designs. We shall now show two simple algebraic laws which state that the set of pf ∧ ds designs together with the operation * constitutes a monoid. Strictly speaking, it is a partial monoid, but when we take into account that we always can perform a systematic renaming, we can also view it as a non-partial monoid.

Lemma 3.3.2 (Algebraic properties of *). Let d₁, d₂, and d₃ be pf ∧ ds designs and let e = system[]. Then d * e and e * d are defined and we have

(i) d * e = e * d = d,
(ii) (d₁ * d₂) * d₃ = d₁ * (d₂ * d₃)

provided in (ii) everything is defined, i.e. cset(d₁) ∩ cset(d₂) = ∅ for i ≠ j.

Proof. Directly from definition 3.3.1.

Of course we are also interested in the behaviour of * with respect to glass-box correctness (gbc) and black-box correctness (bbc). Since the intuition behind * is that of taking the union of the component sets of two 'disjoint' designs the following should not come as a surprise.

Lemma 3.3.3 (Correctness-preserving properties of *). Let d₁ and d₂ be pf ∧ ds designs with disjoint sets of component names.

(i) d₁ and d₂ are gbc if and only if d₁ * d₂ is gbc,
(ii) d₁ and d₂ are bbc if and only if d₁ * d₂ is bbc.

Proof. (i) Let d₁ and d₂ be as in definition 3.3.1. Throughout this part (i) we shall employ several of the derived rules of λπ as worked out in Chapter 2 again (weakening, cut-rule etc.). We show (⇒) first. Assume that both d₁ and d₂ are gbc. We must show that d₁ * d₂ is gbc. If in showing the correctness of some component in d₁ we have the condition Γ ⊢ P_j ⊆ Q_j, where Γ is the context as given by the definition of gbc, then the same component appears also in d₁ * d₂ with correctness condition Γ' ⊢ P_j ⊆ Q_j for some Γ' ⊇ Γ. By weakening it can be shown that Γ' ⊢ P_j ⊆ Q_j implies Γ ⊢ P_j ⊆ Q_j. Therefore this component is also correct as a component in d₁ * d₂. In a similar way can see that each component from d₂ is also correct in d₁ * d₂.

Next we shall show (⇐). Assume d₁ * d₂ is gbc. First we must show that d₁ is gbc. Consider a non-prim component in d₁ * d₂ which comes from d₁. Suppose that we have the correctness condition Γ' ⊢ P_j ⊆ Q_j for this
component in $d_1 * d_2$. We must prove $\Gamma \vdash P_i \subseteq Q_j$ for the same component in $d_1$, and we have $\Gamma' = \Gamma \cup \Delta$ for some set of assumptions $\Delta$.

$\Delta$ contains assumptions of the form $[u_i \subseteq N_i]$ for $1 \leq i \leq n_2$ where each $u_i$ occurs at most once in the left-hand side of an assumption. Since $\Delta$ stems from $d_2$, we have $u_i \not\in \text{cset}(d_1)$ and hence the $u_i$ do not occur freely in $\Gamma$ or in $P_i, Q_j$. Therefore the assumptions in $\Delta$ do not play an essential role in the derivation of $P_j \subseteq Q_j$ (show this by again using the technique of substituting both left and right of the `\vdash`). Hence $\Gamma \vdash P_j \subseteq Q_j$. This shows that $d_1$ is gbc. The fact that $d_2$ is gbc can be shown in a similar way where we deal with non-essential assumptions $[x_i \subseteq M_i]$ for $1 \leq i \leq n_1$ and $[y_i = P_i]$ for $1 \leq i \leq l_1$.

(ii) Similarly as (i).

Note that neither $d_1 * d_2 = d_2 * d_1$ (obvious) nor $\vdash [d_1 * d_2] = [d_2 * d_1]$ - essentially for the same reasons that forbid us to swap prim components.

The following lemma states that the operation $*$ on designs can be interpreted (via $\parallel$) as the operation $*$ of the $\lambda \pi$-calculus with sequences (see Appendix A).

**Lemma 3.3.4 (Semantics of $*$).** Let $d_1$ and $d_2$ be pf $\land$ ds and gbc designs with disjoint sets of component names, then we have

$$\vdash [d_1 * d_2] = [d_1] \parallel [d_2].$$

**Proof.** Let $d_1, d_2$ be given as in definition 3.3.1. Since $d_1 * d_2$ is gbc, we can contract all abstraction-application pairs of $[d_1 * d_2]$ which correspond to non-prim components and we get $\vdash [d_1 * d_2] = L_1 L_2 ([\tilde{S} T] S_2 S_1)$ where

- $L_1$ is the sequence of abstractions $\lambda x_j \subseteq M_j$, for $1 \leq j \leq n_1$ with increasing $j$,
- $L_2$ is the sequence of abstractions $\lambda u_j \subseteq N_j$, for $1 \leq j \leq n_2$ with increasing $j$,
- $S_2$ is the sequence of substitutions $[u_j := A_j]$ for $1 \leq j \leq l_2$ with decreasing $j$,
- $S_1$ is the sequence of substitutions $[y_j := P_j]$ for $1 \leq j \leq l_1$ with decreasing $j$.

Now we have the following calculation in $\lambda \pi$-calculus:

$$\vdash [d_1 * d_2] = L_1 L_2 ([\tilde{S} T] S_2 S_1)$$

$$= L_1 L_2 [\tilde{S} S_1 T S_2].$$
\[ \equiv L_1 L_2 ([S_1] * [T S_1]) \]
\[ \equiv L_1 ([\overline{S}] S_1) * (L_1 ([\overline{T}] S_1)) \]
\[ = [d_1] * [d_2]. \] □

3.3.2 The operation \( \circ \)

The second (partial) operation is called \textit{composition} and is denoted by \( \circ \). It is related to functional composition, usually also denoted by \( \circ \). Roughly speaking, \( d_1 \circ d_2 \) is the design which is obtained by appending \( d_1 \) to \( d_2 \) while replacing the \textit{prims} of \( d_1 \) by the elements of the system of \( d_1 \). Furthermore there is a notion of \textit{validation}, by which we mean that we shall define when the composition of \( d_1 \) with \( d_2 \) is \textit{valid}. In Section 3.4.3 we shall argue that this use of the terms 'valid' and 'validation' is consistent with the usual terminology (as used in e.g. [1]).

This binary operation \( \circ \) will play a key role in the discussion of validation (Section 3.4.3) and in the description of design evolution (Section 3.5). Furthermore \( \circ \) will be used in the description of parallel development (Section 3.6).

\textbf{Definition 3.3.5 (\( \circ \)).}

Assume \( p \wedge d \) designs \( d_1 \) and \( d_2 \) where \( cset(d_1) \cap cset(d_2) = \emptyset \). Let \( d_1 \) and \( d_2 \) respectively be given by

\[
\begin{align*}
  z_1 & := \text{prim} \subseteq M_1 & z_1 & := A_1 \subseteq B_1 \\
  \vdots & & \vdots \\
  z_n & := \text{prim} \subseteq M_n & z_i & := A_i \subseteq B_i \\
  y_1 & := P_1 \subseteq Q_1 & \text{system} \{ S_1, \ldots, S_m \}. & \text{system} \{ S_1, \ldots, S_m \}. \\
  \vdots & & \vdots \\
  y_m & := P_m \subseteq Q_m \\
  \text{system} \{ L_i \}, & \text{system} \{ L_i \},
\end{align*}
\]

We assume that \( P_1, \ldots, P_m \) are not \textit{prim}, whereas some of the \( A_i \) may be \textit{prim} \( (1 \leq i \leq l) \). We define \( d_1 \circ d_2 \) as the design given by

\[
\begin{align*}
  z_1 & := A_1 \subseteq B_1 \\
  \vdots & \vdots \\
  z_i & := A_i \subseteq B_i \\
  z_1 & := S_1 \subseteq M_1 \\
  \vdots & \vdots
\end{align*}
\]
3.3. ALGEBRAIC OPERATIONS ON DESIGNS

\[ x_n := S_n \subseteq M_n \]
\[ y_1 := P_1 \subseteq Q_1 \]
\[ \ldots \]
\[ y_m := P_m \subseteq Q_m \]

system \( L \).

Note that \( d_1 \circ d_2 \) is a \( pf \land ds \) design again. Note also that if \( \text{arity}(d_1) = (n, k) \)
and \( \text{arity}(d_2) = (h, n) \), then \( \text{arity}(d_1 \circ d_2) = (h, k) \). \( \square \)

The design \( d_1 \circ d_2 \) can be viewed as a ‘layered’ design with layers \( d_1 \) and \( d_2 \).

**Remark 3.3.6** We get a highly intuitive view of the construction of \( d_1 \circ d_2 \)
if we omit the keyword system in \( d_2 \) and the keywords prim in \( d_1 \) and write
the system of \( d_2 \) as a column vector. We show this below. Write \( d_2 \) as

\[
\begin{bmatrix}
  z_1 := A_1 \\ \vdots \\
  z_i := A_i \\ \vdots \\
  z_n := A_n \\
\end{bmatrix} \subseteq B_i
\]

and write \( d_1 \) as

\[
\begin{bmatrix}
  x_1 := S_1 \\
  \vdots \\
  x_n := S_n \\
\end{bmatrix} \subseteq M_1
\]
\[
\begin{bmatrix}
  y_1 := P_1 \\
  \vdots \\
  y_m := P_m \\
\end{bmatrix} \subseteq Q_1
\]

system \( L \).

Now one can view the construction of \( d_1 \circ d_2 \) as a matter of plugging the
system of \( d_1 \) into the hole corresponding to the prims of \( d_1 \). \( \square \)

We have one simple algebraic law for \( \circ \).

**Lemma 3.3.7** (Algebraic properties of \( \circ \)). Let \( d_1, d_2 \) and \( d_3 \) be \( pf \land ds \)
designs.

\((d_1 \circ d_2) \circ d_3 = d_1 \circ (d_2 \circ d_3)\).

provided everything is defined.
Proof. Directly from definition 3.3.5.

Remark 3.3.8 (i) There is no neutral element $e$, such that for all designs $d$ we have $d \circ e = e \circ d = d$. Note that $e = \text{system } []$ certainly does not do the job, since in general the arity of $e$ need not match the arity of an arbitrary $d$.

(ii) If we would adopt a 'top' element $\top$ in $\lambda w$-calculus, then for any $n$, the design $e_n$ given by

\[
\begin{align*}
    z_1 &:= \text{prin} \sqsubseteq \top \\
    \vdots \\
    z_n &:= \text{prin} \sqsubseteq \top \\
    \text{system } [x_1, \ldots, x_n]
\end{align*}
\]

would have the property that for every design $d$ with arity $(m, n)$ the following holds:

\[ \vdash [e_n \circ d] = [d]. \]

Therefore it would be possible to view such an $e_n$ as a left neutral element. There is no such right neutral element. This is because when $\vdash [d \circ e] = [d]$, then $[d \circ e]$ must have the same parameter restrictions for its un-applied lambdas as $[d]$. But the parameter restrictions $[d \circ e]$ come from the black-box descriptions of its prin-components, which can not be chosen to fit all $d$.

(iii) We could have defined $d_1 \circ d_2$ differently, e.g. by having no restrictions on the arities of $d_1$ and $d_2$. In that case the elements of the system of $d_2$ for which there is no prin in $d_1$ could appear again in the system of the resulting $d_1 \circ d_2$. In a similar way we could deal with the possibility that there are prims for which there is no system element. If we would have defined $d_1 \circ d_2$ in such a way, we would have $e = \text{system } []$ as a neutral element. □

The following example shows one of the applications for the operation $\circ$.

**Example 3.3.9** Let us assume that we have a library consisting of two implemented components and that we have a design $d$ which must use this library. Let the library be given as a design $d_{lib}$. Let $d_{lib}$ and $d$ respectively be given as

\[
\begin{align*}
    z_1 &:= P_1 \sqsubseteq Q_1 & z'_1 &:= \text{prin} \sqsubseteq Q_1 \\
    z_2 &:= P_2 \sqsubseteq Q_2 & z'_2 &:= \text{prin} \sqsubseteq Q_2 \\
    \text{system } [z_1, z_2], & & x_3 &:= P_3(z'_1, z'_2) \sqsubseteq Q_3 \\
    & & x_4 &:= P_4(z'_3, z'_2) \sqsubseteq Q_4 \\
    & & \text{system } S (z'_1, z'_2, x_3, x_4).
\end{align*}
\]

The instantiation of $d$ with $d_{lib}$ can be described by the composition $d \circ d_{lib}$. 
3.3. ALGEBRAIC OPERATIONS ON DESIGNS

We can simplify \( d \circ d_{10} \) to a design \( d' \). The designs \( d \circ d_{10} \) and \( d' \) respectively are given as

\[
\begin{align*}
x_1 & := P_1 \subseteq Q_1 \\
x_2 & := P_2 \subseteq Q_2 \\
x_1' & := x_1 \subseteq Q_1 \\
x_2' & := x_2 \subseteq Q_2 \\
x_3 & := P_3(x_1, x_2) \subseteq Q_3 \\
x_4 & := P_4(x_1, x_2, x_3) \subseteq Q_4 \\
\text{system} & \quad S(x_1, x_2, x_3, x_4).
\end{align*}
\]

The fact that \( \Gamma[d \circ d_{10}] = \Gamma[d'] \) follows from the component cancellation lemma 3.2.16.

In the above example we see that the fact that the library is modeled as a design and that the components using it are modeled as a design introduce some overhead. In the example this overhead consists of the components \((x_1 := x_1 \subseteq Q_1)\) and \((x_2 := x_2 \subseteq Q_2)\). However the additional components can easily be removed, as shown by the example. The feasibility of this approach in practice may depend on the existence of an abbreviation facility which should provide for 'global' abbreviations (as mentioned in remark 3.2.3). If there is no such abbreviation facility, then the overhead due to the duplication of block-box descriptions (the \( Q_1 \) and \( Q_2 \) in the example) may become too large.

We have one more algebraic law, in which both \(*\) and \(\circ\) occur. By way of introduction to this law we first give an example.

**Example 3.3.10** Let the \( \text{pf} \land \text{ds} \) designs \( d_1 \) and \( d_2 \) respectively be given by

\[
\begin{align*}
x_1 & := \text{prim} \subseteq A_1 \\
x_2 & := A_2 \subseteq A_3 \\
\text{system} & \quad [A_4],
\end{align*}
\]

and let the \( \text{pf} \land \text{ds} \) designs \( d_3 \) and \( d_4 \) respectively be given by

\[
\begin{align*}
x_1 & := \text{prim} \subseteq C_1 \\
x_2 & := C_2 \subseteq C_3 \\
\text{system} & \quad [C_4],
\end{align*}
\]

then the designs \((d_1 \circ d_2) \circ (d_3 \circ d_4)\) and \((d_1 \circ d_2) \circ (d_3 \circ d_4)\) respectively are given by

\[
\begin{align*}
x_1 & := \text{prim} \subseteq C_1 \\
x_2 & := \text{prim} \subseteq C_1 \\
\text{system} & \quad [C_4],
\end{align*}
\]
The latter two designs are semantically equivalent by the component swap
lemma 3.2.9 provided they are gbc.

Lemma 3.3.11 (Interchange law for * and o). Consider prf \wedge ds designs \( d_1, d_2, d_3 \) and \( d_4 \). Assume that \( (d_1 \circ d_2) \circ (d_3 \circ d_4) \) is gbc, then

\[
\vdash ((d_1 \circ d_2) \circ (d_3 \circ d_4)) = [((d_1 \circ d_3) \circ (d_2 \circ d_4))].
\]

provided \((d_1 \circ d_3) \circ (d_2 \circ d_4)\) is defined.

Proof. First note that the left-hand side of the equation must be defined – as is easily seen by analysing the arities of \( d_1 \circ d_2 \) and \( d_3 \circ d_4 \). The design \((d_1 \circ d_2) \circ (d_3 \circ d_4)\) has precisely the same components as the design \((d_1 \circ d_3) \circ (d_2 \circ d_4)\). Furthermore these designs have the same sequence of prim components: first the prim components of \( d_3 \), followed by the prim components of \( d_4 \). These designs also have the same system: the concatenation of the systems of \( d_1 \) and \( d_2 \). Finally the fact that these designs are semantically equivalent follows by repeated application of the component swap lemma 3.2.9.

Remark 3.3.12 The term ‘interchange law’ is taken from [2] (page 44). In order to have an intuitive understanding of the interchange law we shall consider a very simple algebraic system in which this law also holds. Consider an algebraic system with walls as objects and with two operations: putting one wall next to another and putting one wall on top of another. Of course we are given a number of bricks. We can put one wall beside another wall:

\[
\begin{array}{c|c}
A & B \\
\hline
A & B
\end{array}
\]

We can put one wall on top of another wall:

\[
\begin{array}{c|c}
A & B \\
\hline
A & B
\end{array}
\]

Now we have

\[
( \begin{array}{c}
A \\
\end{array} \circ \begin{array}{c}
B \\
\end{array}) \circ (\begin{array}{c}
C \\
\end{array} \circ \begin{array}{c}
D \\
\end{array}) = \begin{array}{c}
A \circ B \\
\begin{array}{c}
C \\
\end{array} \circ \begin{array}{c}
D \\
\end{array}
\end{array}
\]
3.3. ALGEBRAIC OPERATIONS ON DESIGNS

But we also have

\[
\begin{bmatrix}
A & C
\end{bmatrix} \circ
\begin{bmatrix}
B & D
\end{bmatrix} =
\begin{bmatrix}
A \\
C
\end{bmatrix} \cdot
\begin{bmatrix}
B \\
D
\end{bmatrix} =
\begin{bmatrix}
A \\
C
\end{bmatrix} \cdot
\begin{bmatrix}
B \\
D
\end{bmatrix}.
\]

Very much in the same way as we have two notions of correctness for designs, we shall have two different definitions of ‘valid’, viz. glass-box valid and black-box valid (written as gbv\( (d_1, d_2) \) and bbv\( (d_1, d_2) \) respectively).

Definition 3.3.13 Assume pf \& ds designs \( d_1 \) and \( d_2 \) such that \( d_1 \circ d_2 \) is defined and let \( d_1 \) and \( d_2 \) be given as in definition 3.3.5.

(i) the pair \((d_1, d_2)\) is glass-box valid (notation gbv\((d_1, d_2)\)) if

\[
\forall i \, (1 \leq i \leq n) \cdot \Gamma \vdash S_i \subseteq M_i
\]

where \( \Gamma := [z_1 \subseteq B_1], \ldots, [z_k \subseteq B_k], [x_{k+1} = A_{k+1}], \ldots, [z_l = A_l] \), assuming that \( h \) is the number of prim components of \( d_2 \).

(ii) the pair \((d_1, d_2)\) is black-box valid (notation bbv\((d_1, d_2)\)) if

\[
\forall i \, (1 \leq i \leq n) \cdot \Delta \vdash S_i \subseteq M_i
\]

where \( \Delta := [z_1 \subseteq B_1], \ldots, [z_l \subseteq B_l] \).

Before actually employing these notions gbv and bbv for formulating the correctness-preserving properties of \( \circ \), we have a look at a few simple properties.

For the left neutral element \( e_n \) we have for all \( d \) that both gbv\((e_n, d)\) and bbv\((e_n, d)\) provided the arities match. But gbv\((d, e_n)\), bbv\((d, e_n)\) need not hold in general. The proposition that for all \( d_1, d_2 \) we have bbv\((d_1, d_2) \Rightarrow (d_1, d_2) \) are bbc just does not hold in general. The counter-example is very simple. Use \( R_1 \) as before and take the \( d_2 \) with one component \( x := 2 \subseteq 1 \) and with system \( [ ] \). This is because bbv is about the “plug-ability” of \( d_2 \) with respect to \( d_1 \) rather than about the internals of \( d_1 \) and \( d_2 \). Similarly glass-box validation does not imply glass-box correctness - as is easily seen by considering the same counter-example.

As a non-trivial property we have bbv\((d_1, d_2) \Rightarrow gbv(d_1, d_2)\), provided \( d_2 \) is glass-box correct. This can be shown by similar techniques as employed to prove ‘bbc\((d) \Rightarrow gbc(d)\)’ in Chapter 2.

Next we investigate the behaviour of \( \circ \) with respect to gbc and bbc. Intuitively it is clear that the validation conditions gbv and bbv will play a role here.
Lemma 3.3.14 (Correctness-preserving properties of o). Let $d_1$ and $d_2$ be pf and da designs.

(i) $(d_1, d_2$ are gbc $\land$ gbv$(d_1, d_2)) \Rightarrow d_1 \circ d_2$ is gbc,

(ii) $(d_1, d_2$ are bbc $\land$ bbv$(d_1, d_2)) \iff d_1 \circ d_2$ is bbc.

Proof. Let $d_1$ and $d_2$ be given as in definition 3.3.5. Let $\Delta$ and $\Gamma$ be given by definition 3.3.13.

(i) Assume that $d_1$ and $d_2$ are gbc and that gbv$(d_1, d_2)$ holds. We must show that $d_1 \circ d_2$ is gbc. We investigate three kinds of components in $d_1 \circ d_2$.

- A component from $d_2$ has the same correctness condition in $d_1 \circ d_2$ as in $d_2$.

- For a component $(x_i := S_i \subseteq M_i)$ where $1 \leq i \leq n$ the correctness condition is $\Gamma_i[x_1 = S_1], \ldots, [x_{i-1} = S_{i-1}] \vdash S_i \subseteq M_i$. This correctness condition follows by weakening from the assumption that gbv$(d_1, d_2)$ holds.

- For a component $(y_j := P_j \subseteq Q_j)$ where $1 \leq j \leq m$ the correctness condition is $\Gamma_j[x_1 = S_1], \ldots, [x_n = S_n], [y_1 = P_1], \ldots, [y_{j-1} = P_{j-1}] \vdash P_j \subseteq Q_j$.

We have $\Gamma \vdash [S_1 \subseteq M_1], \ldots, [S_n \subseteq M_n]$ and hence $\Gamma, [x_1 = S_1], \ldots, [x_n = S_n] \vdash [x_1 \subseteq M_1], \ldots, [x_n \subseteq M_n]$ which at its turn together with the assumption that $d_1$ is gbc shows the condition (*).

(ii) We show ($\Rightarrow$) first. Assume that $d_1$ and $d_2$ are bbc and that bbv$(d_1, d_2)$ holds. We must show that $d_1 \circ d_2$ is bbc. We investigate three kinds of components in $d_1 \circ d_2$.

- A component from $d_2$ has the same correctness condition in $d_1 \circ d_2$ as in $d_2$.

- For a component $(x_i := S_i \subseteq M_i)$ where $1 \leq i \leq n$ the correctness condition is $\Delta_i[x_1 \subseteq M_1], \ldots, [x_{i-1} \subseteq M_{i-1}] \vdash S_i \subseteq M_i$. This correctness condition follows by weakening from the assumption that bbv$(d_1, d_2)$ holds.

- For a component $(y_j := P_j \subseteq Q_j)$ where $1 \leq j \leq m$ the correctness condition is $\Delta_j[x_1 \subseteq M_1], \ldots, [x_n \subseteq M_n], [y_1 \subseteq Q_1], \ldots, [y_{j-1} \subseteq Q_{j-1}] \vdash P_j \subseteq Q_j$, which follows by weakening from the assumption that $d_1$ is bbc.

Next we shall show ($\Leftarrow$). Assume that $d_1 \circ d_2$ is bbc. First we must show
that \( d_1 \) is gbc. For a non-prim component in \( d_1 \), \( (y_l := P_l \subseteq Q_l) \) say, the correctness condition is \([x_1 \subseteq M_{i_1}], \ldots, [x_n \subseteq M_{i_n}], [y_l \subseteq Q_{k_l}], \ldots, [y_{l-1} \subseteq Q_{k_{l-1}}] \Rightarrow P_l \subseteq Q_l\). Now we note that the assumptions in \( \Delta \) do not play an essential role in the derivation of \( P_l \subseteq Q_l \). This shows that \( d_1 \) is gbc.

For a component in \( d_2 \) the correctness condition is the same as in \( d_1 \circ d_2 \). This shows that \( d_2 \) is gbc. Finally we must show that \( \text{bbv}(d_1, d_2) \) holds, i.e. for arbitrary \( i \) such that \( 1 \leq i \leq n \) we must show \( \Delta \vdash S_i \subseteq M_i \). Now we note that the assumptions \([x_{i_1} \subseteq M_{i_1}], \ldots, [x_{i_{n-1}} \subseteq M_{i_{n-1}}] \) do not play an essential role in the derivation of the correctness condition for the component \((x_i := S_i \subseteq M_i)\) in \( d_1 \circ d_2 \).

\[\square\]

Remark 3.3.15 The proposition that we have

\[d_1 \circ d_2 \text{ is gbc } \Rightarrow \text{ d}_1, \text{ d}_2 \text{ are gbc}\]

does not hold in general.

The counter-example is as follows. Consider \( R_1 \) as before. Let \( d_1 \) and \( d_2 \) respectively be given by

\[
\begin{align*}
y &:= \text{prim } \sqsubseteq 3 \\
z &:= y \sqsubseteq 2 \\
\text{system } [y, z],
\end{align*}
\]

We see that \( d_1 \) is not gbc. The design \( d_1 \circ d_2 \) is gbc. We give \( d_1 \circ d_2 \) below:

\[
\begin{align*}
x &:= 1 \sqsubseteq 2 \\
y &:= x \sqsubseteq 3 \\
z &:= y \sqsubseteq 2 \\
\text{system } [y, z].
\end{align*}
\]

We have a law for \( \text{bbv} \) which is somewhat similar to the interchange law of lemma 3.3.11.

Lemma 3.3.16 (Interchange law for \( \text{bbv} \)). Consider \( \text{pf } \wedge \text{ d} \) designs \( d_1, d_2, d_3 \) and \( d_4 \) such that the compositions \( d_1 \circ d_3 \) and \( d_2 \circ d_4 \) are defined.

\[
\text{bbv}(d_1 \ast d_2, d_3 \ast d_4) \Leftrightarrow (\text{bbv}(d_1, d_2) \wedge \text{bbv}(d_3, d_4)).
\]

Proof. The condition \( \text{bbv}(d_2, d_3) \) requires that one can prove that the system elements of \( d_3 \) satisfy the restrictions of the prim components of \( d_1 \) in a context given by \( d_2 \). The same facts must also be proved for \( \text{bbv}(d_1, d_2) \), but now in the combined context of \( d_2 \) and \( d_3 \). We note that the assumptions from \( d_4 \) do not play an essential role.

The condition \( \text{bbv}(d_2, d_4) \) requires that one can prove that the system ele-
ments of \( d_4 \) satisfy the restrictions of the \texttt{prim} components of \( d_2 \) in a context given by \( d_4 \). The same facts must also be proved for \( \texttt{bbv}(d_1 + d_2, d_3 + d_4) \), but now in the combined context of \( d_3 \) and \( d_4 \). We note that the assumptions from \( d_3 \) do not play an essential role.

The operation \( \circ \) on designs is interpreted (via \[ \]) as the operation \( \circ \) of the \( \lambda \pi \)-calculus with sequences (see appendix A).

Lemma 3.3.17 (Semantics of \( \circ \)). Let \( d_1 \) and \( d_2 \) be \( \texttt{pf} \wedge \texttt{ds} \) designs and let \( d_2 \) be \( \texttt{gbc} \).

\[ \Rightarrow \mathcal{G}[d_1 \circ d_2] = \mathcal{G}[d_1] \circ \mathcal{G}[d_2]. \]

Proof. As 3.3.4 (Semantics of \( \ast \)).

3.3.3 The operations \texttt{bot} and \texttt{top}

We would like to isolate the parts of \( d_1 \) which for given \( d_2 \) play a role in \( \texttt{bbv}(d_1, d_2) \) and the parts of \( d_2 \) which for given \( d_1 \) play a role in \( \texttt{bbv}(d_1, d_2) \). We cast these parts in the form of designs.

Definition 3.3.18 (\texttt{bot}, \texttt{top}). Consider a \( \texttt{pf} \wedge \texttt{ds} \) design \( d \) which is given as

\[
\begin{align*}
  x_1 & := \text{prim} \subseteq M_1 \\
  \ldots \\
  x_n & := \text{prim} \subseteq M_n \\
  y_1 & := P_1 \subseteq Q_1 \\
  \ldots \\
  y_m & := P_m \subseteq Q_m \\
  \text{system} & \left[ S_1, \ldots, S_l \right]
\end{align*}
\]

where \( P_1, \ldots, P_m \) are not equal to \text{prim}. We define \( \texttt{bot}(d) \) and \( \texttt{top}(d) \) respectively as the designs

\[
\begin{align*}
  x_1 & := \text{prim} \subseteq M_1 & x_1 & := \text{prim} \subseteq M_1 \\
  \ldots \\
  x_n & := \text{prim} \subseteq M_n & x_n & := \text{prim} \subseteq M_n \\
  \text{system} & \left[ \right], & \text{system} & \left[ \right]
\end{align*}
\]

\[
\begin{align*}
  y_1 & := \text{prim} \subseteq Q_1 & y_1 & := \text{prim} \subseteq Q_1 \\
  \ldots \\
  y_m & := \text{prim} \subseteq Q_m & y_m & := \text{prim} \subseteq Q_m \\
  \text{system} & \left[ S_1, \ldots, S_l \right], & \text{system} & \left[ S_1, \ldots, S_l \right]
\end{align*}
\]
where it is understood that in $\text{top}(d)$ only those components are retained whose name $x_i$ for some $i$ with $1 \leq i \leq n$ or $y_j$ for some $j$ with $1 \leq j \leq m$ occurs in the system $[S_1, \ldots, S_l]$. Note that $\text{bot}(d)$ and $\text{top}(d)$ are $\text{pf} \land \text{ds}$ designs again. Note also that if $\text{arity}(d) = (n, l)$ then $\text{arity}(\text{bot}(d)) = (n, 0)$ and $\text{arity}(\text{top}(d)) = (n', l)$ where $n' \leq n + m$. We call $\text{bot}(d)$ the bottom of $d$ and we call $\text{top}(d)$ the top of $d$.

**Remark 3.3.19** $\text{bot}(d)$ and $\text{top}(d)$ are not related to the idea of having terms $\bot$ and $\top$ which are minimal and maximal with respect to $\sqsubseteq$ in $\lambda\pi$-calculus. A motivation of the terms bottom and top will be given in remark 3.3.22 below.

We have a number algebraic properties.

**Lemma 3.3.20** (Algebraic properties of bot, top). Consider a $\text{pf} \land \text{ds}$ design $d$ and let $e = \text{system} [\cdot]$.

(i) $\text{bot}(\text{bot}(d)) = \text{bot}(d),$

(ii) $\text{top}(\text{top}(d)) = \text{top}(d),$

(iii) $\text{top}(\text{bot}(d)) = e.$

**Proof.** Directly from the definition 3.3.18.

The following lemma gives some algebraic properties which relate the unary operations $\text{bot}$ and $\text{top}$ with the binary operations $\ast$ and $\circ$.

**Lemma 3.3.21** Assume $\text{pf} \land \text{ds}$ designs $d_1$ and $d_2$ and let $d_1 \ast d_2$ and $d_1 \circ d_2$ be defined.

(i) $\text{bot}(d_1 \ast d_2) = \text{bot}(d_1) \ast \text{bot}(d_2),$

(ii) $\text{top}(d_1 \ast d_2) = \text{top}(d_1) \ast \text{top}(d_2),$

(iii) $\text{bot}(d_1 \circ d_2) = \text{bot}(d_2),$

(iv) $\text{top}(d_1 \circ d_2) = \text{top}(d_1).$

**Proof.** (i) and (ii) follow directly from the definitions 3.3.18 and 3.3.1, whereas (iii) and (iv) follow directly from the definitions 3.3.18 and 3.3.5.

A $\text{pf} \land \text{ds}$ design $d$ for which $\text{bot}(d) = d$ is called a bottom design and a design $d$ for which $\text{top}(d) = d$ is called a top design.

**Remark 3.3.22** We have chosen the terms bottom and top because in our view these notions are related to the so-called top-down and bottom-up models of the software development process. For example in a top-down
development process one starts with a given top design and then adds components and implements components, until the remaining prim components correspond to the primitives which actually are available. In a bottom-up development process one starts with a given bottom design and then adds components and adds system elements until the resulting design meets the actual requirements of the user of the design. We shall investigate the top-down and bottom-up models of the software development process in Sections 3.4.4 and 3.4.5 respectively.

The following lemma confirms our intuition that bot(d) is precisely the part of d which is relevant for bnv(d, d1) and that top(d) is precisely the part of d which is relevant for bnv(d2, d).

**Lemma 3.3.23** Consider pf ∧ ds designs d, d1 and d2.

(i) \( bnv(d, d1) \Rightarrow bnv(bot(d), d1) \),

(ii) \( bnv(d2, d) \Rightarrow bnv(d2, top(d)) \).

**Proof.** Let d be given as in definition 3.3.18. Let d1 and d2 respectively be given by

\[
\begin{align*}
  z_1 & := A_1 \cup B_1 & v_1 & := \text{prim} \cup R_1 \\
  \cdots & & \cdots \\
  z_k & := A_k \cup B_k & v_k & := \text{prim} \cup R_k \\
  \text{system} \ [C_1, \ldots, C_n] & & \text{system} \ [C_1, \ldots, C_n] \\
  w_1 & := X_1 \cup Y_1 & w_k & := X_k \cup Y_k \\
  \cdots & & \cdots \\
  w_n & := X_n \cup Y_n & \\
  \text{system} \ L & & 
\end{align*}
\]

(i) \( bnv(d, d1) \) holds iff \( \forall i \ (1 \leq i \leq n) \cdot [z_i \subseteq B_i, \ldots, z_k \subseteq B_k] \vdash C_i \subseteq M_i \), which is precisely the condition \( bnv(bot(d), d1) \).

(ii) \( bnv(d2, d) \) holds iff \( \forall i \ (1 \leq i \leq l) \cdot [x_1 \subseteq M_1, \ldots, x_n \subseteq M_n, [y_1 \subseteq Q_1], \ldots, [y_m \subseteq Q_m] \vdash S_i \subseteq R_i \). The condition \( bnv(d2, top(d)) \) requires that the same formulae \( S_i \subseteq R_i \) can be derived, but with fewer assumptions, since the following assumptions have been removed: those assumptions \( [x_1 \subseteq M_i] \) and those assumptions \( [y_1 \subseteq Q_i] \ (1 \leq i \leq n, 1 \leq j \leq m) \) for which the variable \( (x_i \text{ or } y_j) \) does not occur in the system \( [S_1, \ldots, S_l] \). Now \( (\Rightarrow) \) follows by weakening and \( (\Rightarrow) \) follows from the observation that the removed assumptions do not play an essential role in derivation of the formulae \( S_i \subseteq R_i \).

The following fallacy is essentially due to the fact that glass-box correct designs do not offer implementation freedom.
Remark 3.3.24 The proposition that for pf & ds designs $d_1$ and $d_2$ we have
\[ \text{gbv}(d_1, d_2) \Rightarrow \text{gbv}(d_1, \text{top}(d_2)). \]
just does not hold in general.

The counter-example is as follows
\[ \text{gbv}(x := \text{prim} \subseteq 1, y := 1 \subseteq 2) \text{ holds because } [y = 1] \models y \subseteq 1, \]
\[ \text{gbv}(x := \text{prim} \subseteq 1, y := \text{prim} \subseteq 2) \text{ does not, for } [y \subseteq 2] \forall y \subseteq 1. \]

Recall that we took the decision to cast the top of a design into the form of a design. This can be viewed as the choice of a very specific representation. For a given design $d_3$, we have $\text{top}(d_2)$ as the representation of the parts of $d_3$ which for given $d_1$ play a role in $\text{bbv}(d_1, d_2)$. First we note that the order of the prim components in $\text{top}(d_3)$ is not relevant (from the current point of view). This idea is formalised by introducing a binary predicate $=_{\text{pp}}$.

Definition 3.3.25 Let $d$ and $d'$ be pf & ds designs. We say that $d'$ is a prim permutation of $d$, notation $d =_{\text{pp}} d'$, if $d'$ can be obtained from $d$ by permuting the order of the prim components.

Example 3.3.26 Let $d$ and $d'$ respectively be given by
\[
x_1 := \text{prim} \subseteq M_1, \quad x_2 := \text{prim} \subseteq M_1, \quad x_3 := \text{prim} \subseteq M_2, \quad x_1 := \text{prim} \subseteq M_1, \quad x_3 := P \subseteq Q, \quad x_2 := P \subseteq Q, \quad \text{system } [x_1, x_2, x_3], \quad \text{system } [x_1, x_2, x_3].
\]
We have $d =_{\text{pp}} d'$. Of course we also have $d' =_{\text{pp}} d$.

Lemma 3.3.27 Consider pf & ds designs $d_1$, $d_2$ and $d'_2$ such that $d_2 =_{\text{pp}} d'_2$ and such that $d_1 \circ d_2$ is defined, then we have
\[ \text{bbv}(d_1, d_2) \Leftrightarrow \text{bbv}(d_1, d'_2). \]

Proof. Just note that $\text{bbv}(d_1, d_2)$ is defined (in definition 3.3.13) as $\forall i (1 \leq i \leq n) \cdot \Delta \models S_i \subseteq M_i$ for suitable $n$, $\Delta$, $S_i$ and $M_i$ where the elements of $\Delta$ are derived from the components of $d_1$, where the $S_i$ stem from the system.
of \( d_2 \) and the \( M_i \) stem from \( d_1 \). Finally recall that the context \( \Delta \) is a set (of assumptions) and that therefore \( \Delta \) is independent of the order of the components in \( d_0 \).

If we view a top design as a representation of the parts of a design \( d_2 \) which are relevant for \( \text{bbv}(d_1, d_2) \), then we might identify even more top designs than indicated by \( \equiv_{pp} \).

**Definition 3.3.28** Let \( d_2 \) and \( d_2' \) be top designs with arities \((h, n)\) and \((h', n)\) respectively. Then we define

\[
d_2 \equiv_{\text{top}} d_2' \iff \forall d_1 \text{ with } d_1 \circ d_2 \text{ defined:}
\]

\[
\text{bbv}(d_1, d_2) \Leftrightarrow \text{bbv}(d_1, d_2').
\]

If for two top designs \( d_2 \) and \( d_2' \) we have \( d_2 \equiv_{\text{top}} d_2' \) we say that \( d_2 \) and \( d_2' \) are \textit{top-equivalent}.

**Example 3.3.29** Consider an algebraic system \( \mathcal{R} \) which provides a binary function with symbol \(+\) which is written in infix notation. Let the top designs \( d_2 \) and \( d_2' \) respectively be given as

\[
x_1 := \text{prim } \subseteq M_1, \quad x_1 := \text{prim } \subseteq M_1
\]

\[
x_2 := \text{prim } \subseteq M_2, \quad x_2 := \text{prim } \subseteq M_2
\]

\[
x_3 := \text{prim } \subseteq M_3, \quad x_3 := \text{prim } \subseteq M_3
\]

\[
\text{system } [x_1 + x_2, x_1 + x_3], \quad \text{system } [x_1 + x_2, x_1' + x_3].
\]

Then \( d_2 \equiv_{\text{top}} d_2' \) because if we consider the composition of \( d_2 \) (or \( d_2' \)) with some design \( d_1 \) of the form

\[
y_1 := \text{prim } \subseteq R_1
\]

\[
y_2 := \text{prim } \subseteq R_2
\]

(constructing \( d_1 \circ d_2 \) or \( d_1 \circ d_2' \)), then the validity conditions \( \text{bbv}(d_1, d_2) \) and \( \text{bbv}(d_1, d_2') \) are equivalent:

\[
\text{bbv}(d_1, d_2) \Leftrightarrow [x_1 \subseteq M_1], [x_2 \subseteq M_2], [x_3 \subseteq M_3] \vdash x_1 + x_2 \subseteq R_3, x_1 + x_3 \subseteq R_2
\]

\[
\Leftrightarrow [x_1' \subseteq M_1], [x_2' \subseteq M_2], [x_3' \subseteq M_3] \vdash x_1 + x_2 \subseteq R_3, x_1' + x_3 \subseteq R_2
\]

\[
\Leftrightarrow \text{bbv}(d_1, d_2'). \quad \square
\]
In the above example we have transformed the top design \( d_2 \) into \( d'_2 \), where \( d'_2 \) can be viewed as the concatenation of two simpler designs. This idea is formulated in a general form in lemma 3.3.30 given below.

**Lemma 3.3.30** Let \( d_2 \) be a top design with arity \((k, n)\). Then for all \( n', n'' \in \mathbb{N} \) with \( n' + n'' = n \) we can find \( d'_{2}, d''_{2}, k' \) and \( k'' \) such that \( d'_{2} \) and \( d''_{2} \) have arities \((k', n')\) and \((k'', n'')\) and furthermore

\[
d'_{2} \ast d''_{2} = \top d_{2}.
\]

**Proof.** As in example 3.3.29 introduce additional prim components and rewrite system elements in terms of the names of these new components. In this way it is possible to make the set of component names in the first \( n' \) system elements disjoint with respect to the set of component names in the remaining \( n'' \) system elements. Then permute the order of the components such that first we have the components whose name is in the first \( n' \) system elements, followed by the remaining components. Finally take \( d'_{2} \) and \( d''_{2} \) in the obvious way such that \( d'_{2} \ast d''_{2} = \top d_{2} \). \( \square \)

**Example 3.3.31.** Consider an algebraic system \( \mathcal{R} \) which provides a binary function with symbol \(+\) which is written in infix notation. Let the top design \( d_{2} \) be given as in example 3.3.29. Then for \( n' = n'' = 1 \) lemma 3.3.30 says that there exist \( d'_{2}, d''_{2} \) with \( d'_{2} \ast d''_{2} = \top d_{2} \). Indeed, we can take \( d'_{2} \) and \( d''_{2} \) as follows

\[
\begin{align*}
    z_1 &:= \text{prim} \subseteq M_1 & x'_1 &:= \text{prim} \subseteq M_1 \\
    z_2 &:= \text{prim} \subseteq M_2 & x_2 &:= \text{prim} \subseteq M_2 \\
    \text{system} [z_1 + z_2] & & \text{system} [x'_1 + x_2].
\end{align*}
\]

**Remark 3.3.32** For bottom designs the situation is much easier. The most interesting notion of equality is \( =_{\parallel} \). Furthermore each bottom design \( d \) of arity \((m, 0)\) can be split for every \( n' \) and \( n'' \) with \( n' + n'' = n \) into \( d' \) and \( d'' \) with arities \((n', 0)\) and \((n'', 0)\) such that \( d' \ast d'' = d \). \( \square \)

We see that \( =_{\parallel} \) is nothing but a special case of \( =_{\top} \) and lemma 3.3.27 just says that \( d =_{pp} d' \Rightarrow d =_{\top} d' \). The advantage of considering \( =_{pp} \) is the fact that \( =_{pp} \) is a simple syntactical notion.

### 3.3.4 The operation \( \parallel \)

We shall briefly discuss a binary operation denoted by \( \parallel \). The operation is suggested by the transitivity of the implementation relation \( \subseteq \). This operation will turn out to be somewhat disappointing but nevertheless we believe
that it is illustrative to see what goes wrong. Yet the idea behind the operation seems attractive and we shall sketch one possible line of formalisation to exploit it. The results of this section will not be used in the remainder of this chapter (Section 3.6.5 being the only exception).

Definition 3.3.33 (\(\boxplus\)) Assume \(pf \land ds\) designs \(d_1\) and \(d_2\). Let \(d_1\) and \(d_2\) respectively be given by

\[
x_1 := \text{prim} \subseteq M_1 \\
\vdots \\
x_n := \text{prim} \subseteq M_n \\
y_1 := P_1 \subseteq Q_1 \\
\vdots \\
y_m := P_m \subseteq Q_m \quad \text{system } L,
\]

So \(d_1, d_2\) have the same component names, the same black-box descriptions for the \(\text{prim}\) components, the same system and furthermore they have one 'column' in common. Then we define \(d_1 \boxplus d_2\) as

\[
x_1 := \text{prim} \subseteq M_1 \\
\vdots \\
x_n := \text{prim} \subseteq M_n \\
y_1 := P_1 \subseteq Q_1 \\
\vdots \\
y_m := P_m \subseteq Q_m \quad \text{system } L.
\]

Note that \(d_1 \boxplus d_2\) is a \(pf \land ds\) design again. Note also that if \(\text{arity}(d_1) = (n, k)\) and \(\text{arity}(d_2) = (n, k)\), then \(\text{arity}(d_1 \boxplus d_2) = (n, k)\). \(\Box\)

Lemma 3.3.34 (Algebraic properties of \(\boxplus\)).

\[(d_1 \boxplus d_2) \boxplus d_3 = d_1 \boxplus (d_2 \boxplus d_3).\]

Proof. Directly from the definition 3.3.33 \(\Box\)

The following remark shows that \(\boxplus\) lacks a desirable property.

Remark 3.3.35 The proposition that we have

\[d_1, d_2 \text{ are bbc } \Rightarrow \text{ } d_1 \boxplus d_2 \text{ is bbc}\]

just does not hold in general.
3.3. ALGEBRAIC OPERATIONS ON DESIGNS

The counter-example is as follows. Consider \( R_1 \) as before. Let \( d_1 \) and \( d_2 \) respectively be given by

\[
x := 1 \quad \subseteq \quad 2 \\
y := x \quad \subseteq \quad 2
\]

system \([y]\).

\[
x := 2 \quad \subseteq \quad 3 \\
y := 2 \quad \subseteq \quad 2
\]

system \([y]\).

So both \( d_1 \) and \( d_2 \) are both bbc but \( d_1 \upharpoonright d_2 \) is not bbc. We give \( d_1 \upharpoonright d_2 \) below.

\[
x := 1 \quad \subseteq \quad 3 \\
y := x \quad \subseteq \quad 2
\]

system \([y]\).

The above is a bit disappointing, but when adopting glass-box correctness, there is no problem.

Lemma 3.3.36 Assume pf and ds designs \( d_1, d_2 \). Then we have

\( d_1, d_2 \) are gbc \( \Rightarrow \) \( d_1 \upharpoonright d_2 \) is gbc

provided \( d_1 \upharpoonright d_2 \) is defined.

Proof. For each component use rule (trans.) and the fact that neither the black-box descriptions nor the glass-box descriptions of \( d_2 \) can contain component names.

As a direct consequence of this lemma we also have \( d_1, d_2 \) are bbc \( \Rightarrow \) \( d_1 \upharpoonright d_2 \) is gbc. Just use ‘\( d_1 \) is bbc \( \Rightarrow \) \( d_1 \) is gbc’ and ‘\( d_2 \) is bbc \( \Rightarrow \) \( d_2 \) is gbc’.

It is not hard to see why \( \upharpoonright \) fails to preserve bbc. The obvious idea – which is to suppose that rule (trans.) applies – fails because the facts \( P_i \subseteq Q_i \) and \( Q_i \subseteq R_i \) are not given for the same set of assumptions. In particular the assumptions \( y_j \subseteq Q_j \quad (j < i) \) are stronger than the \( y_i \subseteq R_i \) which determine the black-box correctness of \( d_1 \upharpoonright d_2 \).

As a possible solution, we could generalise the notions of correctness so as not to have two notions gbc and bbc, but just one notion, ‘correct’ say, which is parameterised over contexts – writing ‘\( \Gamma \)-correct’ for context \( \Gamma \).

Definition 3.3.37 (\( \Gamma \)-correct) Let the pf and ds design \( d \) be given as

\[
x_1 := \text{prim} \quad \subseteq \quad M_1 \\
\ldots
x_n := \text{prim} \quad \subseteq \quad M_n \\
y_1 := P_1 \quad \subseteq \quad Q_1 \\
\ldots
\]
\[ y_m := P_m \sqsubseteq Q_m \]

system \( L \).

For a given context \( \Gamma \) we say that \( d \) is \( \Gamma \)-correct, if for every component \( y_i := P_i \sqsubseteq Q_i \), we have

\[ \Gamma \vdash P_i \sqsubseteq Q_i. \]

\[ \square \]

Recall that \( \Gamma_{gb}(d) \) denotes the glass-box context of \( d \) and that \( \Gamma_{bb}(d) \) denotes the black-box context of \( d \) (see definition 3.2.18). Now we get the specialised notions gbc and bbc back in the sense that \( d \) is gbc \( \iff d \) is \( \Gamma_{gb}(d) \)-correct and similarly that \( d \) is bbc \( \iff d \) is \( \Gamma_{bb}(d) \)-correct. (To see these equivalences is not completely trivial: for \( \Rightarrow \) we must use weakening and for \( \Leftarrow \) we must use that \( d \) is wf and that for each \( P_i \sqsubseteq Q_i \), the assumptions \( y_j = Q_j \) or \( y_j \sqsubseteq Q_j \) for \( j \geq i \) are non-essential).

With the above machinery of generalised correctness it is straightforward to see precisely in which way \( \triangleright \) preserves black-box correctness.

**Lemma 3.3.38** Assume \( \psi \land \phi \) designs \( d_1, d_2 \). Then we have

\[ d_1 \triangleright d_2 \text{ are } \Gamma_{bb}(d_2) \text{-correct } \Rightarrow d_1 \triangleright d_2 \text{ is bbc} \]

provided \( d_1 \triangleright d_2 \) is defined.

**Proof.** Just use transitivity for each component. \( \square \)

Using transitivity of \( \sqsubseteq \) means to work by means of a certain type of stepwise refinement. For example, when constructing \( d_1 \triangleright d_2 \) we could consider a \( y_i := P_i \sqsubseteq Q_i \) from \( d_1 \) and a \( y_i := Q_i \sqsubseteq R_i \) from \( d_2 \). Now this situation has the interpretation that \( R_i \) is a 'high-level' specification which is refined by the 'intermediate-level' specification \( Q_i \), which at its turn is refined by the 'low-level' description \( P_i \).

A typical development process could be to construct the \( R_i \) first, followed by the \( Q_i \) and after that the \( P_i \). Another option is to work the other way around, i.e. first the \( P_i \), then the \( Q_i \) and the \( R_i \). Of course the length of these sequences can also be extended to more than three.

The analysis of \( \triangleright \) also suggests a somewhat different line of development where we do not generalise the notions of correctness, but rather generalise the notion of design. This suggestion arises naturally from the observation that when \( d_1 \triangleright d_2 \) is defined, \( d_1 \) and \( d_2 \) must have already much in common. Therefore we could view them as parts of one generalised design. We do not give formal definitions here, but we rather give a sketch. We could adopt *generalised designs* which are of the following form:
3.3. ALGEBRAIC OPERATIONS ON DESIGNS

\[
\begin{align*}
x_1 &:= \text{prim} \subseteq M_1 \\
\ldots \\
x_n &:= \text{prim} \subseteq M_n \\
y_1 &:= P_1 \subseteq Q_1 \subseteq R_1 \\
\ldots \\
y_m &:= P_m \subseteq Q_m \subseteq R_m
\end{align*}
\]

We conclude this section with mentioning the advantages and disadvantages of these generalised designs over our earlier designs.

The main advantage is that generalised designs carry more information for each component. The intermediate-level specifications (the \( Q_i \)) can play the role of "part of a correctness proof" for each component. The second advantage is that certain types of step-wise refinement – just as sketched above – become correctness-preserving transformations of designs in a natural way.

The main disadvantage is that many different generalisations are conceivable: what about \( n \) instead of three descriptions for each component (the \( P_i, Q_i, R_i \))? what about a different \( n \), for each component? how should we generalise ds and pf? do we besides gbc and bfc also allow for something like "intermediate-box" correctness?. We expect that these technicalities tend to complicate the theory significantly.

A second disadvantage is that it is not entirely clear how to adapt the mapping \( \| \) from designs to lambda terms to make it work for generalised designs.

3.3.5 Summary

We end this section with a summary of the algebra of designs. The signature of the algebra of designs is summarised by the picture given below. We did not include all predicates. E.g. the predicate \( \Rightarrow \) is not shown. The collection of all pf \& ds designs is shown as a circle. For the sake of the picture we view predicates as functions to Boolean values. The set of Boolean values is shown as a circle named Bool. The operations on designs are shown as arrows. The arrow which has been labeled \( \Downarrow \) corresponds with the possibility to construct a design directly, without using algebraic operations on designs. The constants \( c \) and \( c_n \) are given in lemma 3.3.2 and remark 3.3.8 (ii). We also give a number of equations and equivalences which correspond
with some of our lemmas. We write \( d_1 =_I d_2 \) if \( \llbracket d_1 \rrbracket = \llbracket d_2 \rrbracket \).

\[
\begin{align*}
(\bot, \top) &\rightarrow \text{bot, top} \\
\epsilon, \epsilon_n &\rightarrow \text{designs} \\
pf &\rightarrow \text{leads} \\
\{\ast, \circ, \$\} &\rightarrow \text{bf, gbf, bbv, gbv, Bool} \\
\end{align*}
\]

\[
\begin{align*}
d + e &= e + d = d \\
(d_1 * d_2) * d_3 &= d_1 * (d_2 * d_3) \\
\epsilon_n * d &= d \]
\]

\[
\begin{align*}
(d_1 \circ d_2) \circ d_3 &= d_1 \circ (d_2 \circ d_3) \\
(d_1 \circ d_2) \circ (d_2 * d_3) &= (d_1 \circ d_2) * (d_2 \circ d_3) \\
\text{bbv}(d_1) \land \text{bbv}(d_2) &\iff \text{bbv}(d_1 * d_2) \\
\text{bbv}(d_1) \land \text{bbv}(d_2) \land \text{bbv}(d_1, d_2) &\iff \text{bbv}(d_1 \circ d_2) \\
\text{bot} &\text{bot}(d) = \text{bot}(d) \\
\text{top}(\text{top}(d)) &\text{top}(d) \\
\text{top}(\text{bot}(d)) &\text{top}(d) \\
\text{bot}(d) &\text{bot}(d) = \text{bot}(d) \\
\text{top}(d_1 \circ d_2) &\text{top}(d_1) \\
\text{bot}(d_1 + d_2) &\text{bot}(d_1) \land \text{bot}(d_2) \\
\text{top}(d_1 + d_2) &\text{top}(d_1) \land \text{top}(d_2) \\
\text{bbv}(d_1, d_2) &\text{bbv}(\text{bot}(d_1), d_2) \\
\text{bbv}(d_1, d_2) &\text{bbv}(d_1, \text{top}(d_2)) \\
\text{bbv}(d_1 * d_2, d_3 * d_4) &\text{(bbv}(d_1, d_2) \land \text{bbv}(d_3, d_4)) \\
\end{align*}
\]

Fig 3.1. The algebra of designs.
3.4 Design Creation

3.4.1 General

In this section we want to derive several design-programs which describe the creation of a design. We shall use the design-development language of appendix B for expressing these design-programs. A design-program can be executed by a developer (or a team of developers). Since in general our design-programs will be highly non-deterministic, the developer(s) must make choices during the execution of a design-program. The fact that these choices exist, corresponds with the necessity of 'creative freedom' for the developer(s). Sometimes we shall use the term 'model of the development process' as a synonym for 'design-program'.

The derivation of such design-programs is quite non-trivial. The control structure of a design-program is not essentially complexer than that of a classical simple 'while program', but the point is that the design-programs describe the manipulation of complex data types: designs, components etc.

Therefore we shall spend quite some effort to the presentation of an elaborate example about a top-down development process first and only after that treat the formalisation of the top-down design-program in a more general setting. This has the advantage that during the formal treatment of the top-down design-program we can refer to the example. Once we have done the top-down design-program, the bottom-up design program is relatively easy.

The next section (Section 3.4.2) is devoted to the example and after that we proceed with the derivation of two design-programs, which we begin in Section 3.4.3. Section 3.4.3 presents the common setting of our design-programs which applies to the next two sections (3.4.4 and 3.4.5). Section 3.4.4 is about top-down development. Section 3.4.5 is about bottom-up development.

3.4.2 Top-down example

Before embarking on a formal treatment of top-down and bottom-up developments, we first give an elaborate example of a top-down development which is taken from the area of electronic digital hardware [3]. We have no claim whatsoever that the hardware circuit of the example is efficient or fast. In fact our main interest is not in hardware design at all, and the purpose of the example is to illustrate the notion of design and some of the transformation steps operating on designs. Of course we could take examples using the class-algebra CA of COLD-K, but for illustrating the design concept it is beneficial to have a kind of 'stand-alone' example as well.
The example is about so-called logical circuits and the composition mechanisms for these are interconnection wirings. We have two kinds of descriptions for logical circuits, viz. equations and interconnection diagrams. The equations are based on two-valued Boolean logic with constants 0, 1, multiplication (= and) such that 0.0 = 0.1 = 1.0 = 0, 1.1 = 1, addition (= or) such that 0+0 = 0, 1+0 = 0+1 = 1+1 = 1 and inverting (= not) such that \( \overline{0} = 1 \) and \( \overline{1} = 0 \). We write \( x \oplus y \) for \( x\overline{y} + \overline{x}y \). Logical circuits have so-called ports, which act as a kind of variables \( (a, b, c, \ldots) \) and which are grouped into two categories, viz. input ports and output ports. An equation is always written with the output ports occurring at the left-hand side and the input ports at the right-hand side of the equation. E.g. \( z = (a, b) \) is an equation which specifies a 'nand' logical circuit with input ports \( a, b \) and output port \( z \). The interconnection diagrams represent algebraic terms corresponding to some algebraic approach to logical-circuit composition. We do not provide a formalisation of these representation issues — although of course this could be done. The interconnection diagrams may contain component names and we assume that this is also the case for the terms represented by the interconnection diagrams. Intuitively, the interconnection diagrams will speak for themselves.

We shall use numbers such as 7400, 7404, \ldots to act as component names. The use of these names is consistent with the terminology of the well-known transistor transistor logic (TTL) family of integrated logical circuits [3].

Let us assume — for sake of the example — that it is our task to develop a so-called four-bit adder. We need some notation to specify the four-bit adder. For a bit-sequence \( \overline{b} \) we write \( \text{int}(\overline{b}) \) to denote the integer which is binary represented by \( \overline{b} \). In particular, \( \text{int}(0, 0, 0, 0) = 0 \), \( \text{int}(0, 0, 0, 1) = 1 \), \( \text{int}(0, 0, 1, 0) = 2 \) and \( \text{int}(0, 0, 1, 1) = 3 \). A four-bit adder is a logical circuit with input ports \( a_3, a_2, a_1, a_0, b_3, b_2, b_1, b_0 \), output ports \( a_4, a_3, a_2, a_1, a_0 \) and \( b_4, b_3, b_2, b_1, b_0 \) which is specified by the equation \( \text{int}(a_4, a_3, a_2, a_1, a_0) = \text{int}(a_3, a_2, a_1, a_0) + \text{int}(b_3, b_2, b_1, b_0) \) which we abbreviate as \( \text{int}(\overline{a}) = \text{int}(\overline{a}) + \text{int}(\overline{b}) \). The component name of the four-bit adder is 74283. Therefore the top of the design \( d \) to be developed is given by the following design.

\[
\begin{align*}
74283 & := \text{prim} \subseteq (\text{int}(\overline{a}) = \text{int}(\overline{a}) + \text{int}(\overline{b})) \\
\text{system } [74283] & 
\end{align*}
\]

This is the initial design of the top-down development. With respect to the available primitive building blocks, we adopt a minimalistic approach by restricting ourselves to two simple logical circuits called nand-gate and ground-connection. It is a well-known fact that these are sufficient to construct all other logical circuits. A nand-gate has two input ports \( a, b \) and
one output port $x$. It is specified by $x = (a, b)$. A ground-connection has no input ports and one output port $g$, specified by $g = 0$. Therefore the bottom of the design $d$ to be developed is given by the following design.

$$
\begin{align*}
7400 & := \text{prim} \quad (s = (a, b)) \\
\text{GND} & := \text{prim} \quad (g = 0) \\
\text{system} & [ ]
\end{align*}
$$

Now the top-down development can really begin. In order to decompose the 74283 four-bit adder we employ a component providing for a so-called single-bit full adder. A single-bit full adder is a logical circuit with three input ports $a$, $b$, $c_i$ and two output ports $s$ and $c_o$. The ports $c_i$ and $c_o$ are usually known as 'carry-in' and 'carry-out' respectively. It is specified by the equations $s = a \oplus b \oplus c_i$ and $c_o = a \cdot b + c_i \cdot (a \oplus b)$. So $s = 1$ if and only if the number of inputs that equal 1 is odd. Also $c_o = 1$ if and only if there are two or more inputs that equal 1. We shall introduce a new component named 74183 to provide the functionality of a single-bit full adder.

Using four instances of the 74183 and one ground-connection, the four-bit adder can be implemented. This is known as a ripple-carry configuration [3] (p. 87). We refer to the following interconnection diagram as 74283IMPL.

![Interconnection diagram 74283IMPL](image)

The design-step to be taken involves two modifications. First of all, the new components 74183 and GND must be added to the design. These new components can be viewed as a two-components 'mini-design', $d'$ say.

$$
\begin{align*}
\text{GND} & := \text{prim} \quad (g = 0) \\
74183 & := \text{prim} \quad (s = a \oplus b \oplus c_i, \ c_o = a \cdot b + c_i \cdot (a \oplus b)) \\
\text{system} & [ ]
\end{align*}
$$
We must concatenate $d'$ with our initial design $d$, i.e. construct $d' \ast d$. Secondly, the old 74283 must be updated by means of a kind of overwriting insert operation, inserting $74283 := 74283\text{IMPL} \sqsubseteq (\text{int}(\overline{z}) = \text{int}(\overline{a}) + \text{int}(\overline{b}))$. After this design step, our design $d$ is as follows:

\[
\begin{align*}
\text{GND} & := \text{prim} \quad \sqsubseteq \quad (g = 0) \\
74183 & := \text{prim} \quad \sqsubseteq \quad (s = a \oplus b \oplus c_i, \quad c_o = a.b + c_i.(a \oplus b)) \\
74283 & := 74283\text{IMPL} \quad \sqsubseteq \quad (\text{int}(\overline{z}) = \text{int}(\overline{a}) + \text{int}(\overline{b})) \\
\text{system} & \quad \sqsubseteq \quad 74283
\end{align*}
\]

In order to decompose the 74183 single-bit full adder, we employ and-gates, or-gates and xor-gates which are provided by new components 7408, 7432 and 7486 respectively. These are specified by $z = x.y$, $x = x + y$ and $z = x \oplus y$ respectively.

Using and-gates, or-gates and xor-gates the single-bit full adder can be implemented. We refer to the following interconnection diagram as 74183\text{IMPL}. We can easily verify that this diagram satisfies the equations of 74183. In particular, it is helpful to note that the output of the leftmost 7408 equals $a.b$ and that the output of the leftmost 7486 equals $a \oplus b$. The output of the rightmost 7408 equals $c_i.(a \oplus b)$. Having noticed this, it follows that for the output $s = a \oplus b \oplus c_i$ and $c_o = a.b + c_i.(a \oplus b)$.

![Fig 3.3. Interconnection diagram 74183IMPL.](image)

As a matter of fact, verifying that this diagram satisfies the equations of its
3.4. DESIGN CREATION

specification essentially means to verify a pair in the implementation relation \( \sqsubseteq \). The black-box correctness of \( d \) requires that each pair (glass-box description, black-box description) is in this relation -- for a suitable context. Here and in the general formulation of the top-down model in Section 3.4.4, black-box correctness of the current design serves as an invariant of the development process. The current section is one of the few places in this monograph where we give detailed proofs for the proof obligations arising in connection with \( \sqsubseteq \). In Chapter 5, most proofs remain more or less implicit.

Again the design-step to be taken involves two modifications. First, the new components must be added to the design. These new components can be viewed as a three-components mini-design, \( d' \), say.

\[
\begin{align*}
7408 & \triangleq \text{ prim} \quad \sqsubseteq \quad (x = x.y) \\
7432 & \triangleq \text{ prim} \quad \sqsubseteq \quad (z = x + y) \\
7486 & \triangleq \text{ prim} \quad \sqsubseteq \quad (z = x \oplus y) \\
\text{system} & \quad [ ]
\end{align*}
\]

We must concatenate \( d' \) with our current design \( d \). Secondly, we must insert a new version of 74183 which has 74183IMPL instead of prim. After this design step, our design \( d \) is as follows:

\[
\begin{align*}
7408 & \triangleq \text{ prim} \quad \sqsubseteq \quad (x = x.y) \\
7432 & \triangleq \text{ prim} \quad \sqsubseteq \quad (z = x + y) \\
7486 & \triangleq \text{ prim} \quad \sqsubseteq \quad (z = x \oplus y) \\
\text{GND} & \triangleq \text{ prim} \quad \sqsubseteq \quad (g = 0) \\
74183 & \triangleq \quad 74183\text{IMPL} \quad \sqsubseteq \quad \begin{align*}
\varepsilon & = a \odot b \oplus c_t, \\
\varepsilon & = a.b + c_t(a \odot b)
\end{align*} \\
74283 & \triangleq \quad 74283\text{IMPL} \quad \sqsubseteq \quad \begin{align*}
\text{int}(\varepsilon) & = \text{int}(a) + \text{int}(b)
\end{align*} \\
\text{system} & \quad [ 74283 ]
\end{align*}
\]

We could select the last prim-component as the next candidate to be implemented -- which typically is a kind of default in top-down development. However, we have no intention to decompose GND, so we permute some prim components, putting GND first (i.e. constructing a design that is pp-equivalent, cf. definition 3.3.25).

In order to decompose the 7486 xor-gate, we employ inverters, which are provided by a new component 7404. An inverter is a logical circuit with one input port \( p \) and one output port \( q \). It is specified by \( q = \overline{p} \). Using and-gates, or-gates and inverters it is easy to implement a xor-gate. We refer to the following interconnection diagram as 7486IMPL. For its verification it is useful to note that the output of the upper 7408 is \( \overline{x.y} \) and that the output of the lower \( 7408 \) is \( x.\overline{y} \). Therefore the final output \( x \) is \( x.y + x.\overline{y} \) which equals \( x \oplus y \).
Fig 3.4. Interconnection diagram 7486IMPL.

Now the design-step is the addition of the 7404 and the insertion of the implemented 7486 with 7486IMPL instead of prim. Once more we put GND first. We do not show the resulting d and we proceed immediately with the next design-step.

In order to decompose the 7432 or-gate, we shall employ the 7400 nand-gate – which is an available primitive. Furthermore we employ two 7404 inverters. We refer to the following interconnection diagram as 7432IMPL. It is useful to note that the inputs of its 7400 equal $\overline{x}$ and $\overline{y}$. Therefore the final output $z$ is $\overline{x} \overline{y}$ which equals $x + y$.

Fig 3.5. Interconnection diagram 7432IMPL.

The design-step is to add the 7400 component and to insert a modified 7432
component, with 7432IMPL instead of prim. To decompose the 7408
and-gate we need no new components. It can be done easily with one 7400
nand-gate and one 7404 inverter. We refer to the following interconnection
diagram as 7408IMPL. It is useful to note that the output of its 7400 equals
$x.y$. Therefore the final output $z$ is $x.y$.

![Fig 3.6. Interconnection diagram 7408IMPL.]

Finally we implement the 7404 inverter. We employ one 7400 whose two
inputs are connected. We refer to the following interconnection diagram as
7404IMPL.

![Fig 3.7. Interconnection diagram 7404IMPL.]

The resulting design is finished because its bottom equals the agreed bottom
design with 7400 and GND. We show the resulting design $d$ below.

```
7400 := prim \cup \{z = (a \cdot \bar{b})\}
GND := prim \cap \{g = 0\}
7404 := 7404IMPL \cap \{q = p\}
7408 := 7408IMPL \cup \{z = x.y\}
7432 := 7432IMPL \cup \{x = x + y\}
7486 := 7486IMPL \cap \{x = x \oplus y\}
74183 := 74183IMPL \cup \{s = a \oplus b \oplus c, c_s = a . \bar{b} + c . (a \oplus b)\}
74283 := 74283IMPL \cup \{(int(\delta) = int(\delta) + int(\bar{\delta}))\}
system \{ 74283 \}
```

This concludes the example. A systematic discussion of top-down develop-
ment in general will be undertaken in Section 3.4.4.
3.4.3 Design-programs

In this section we turn our attention to design-programs for design creation in a very general setting. We shall use the design-development language of appendix B. The design-programs will be highly non-deterministic. The data that are manipulated by the developer(s) during the execution of a design-program may include designs, components, terms of $\lambda x$ and names. In this section we view these data as belonging to given data types which bring with them certain predicates and operations. In particular, we have predicates $bbc$, $bbv$, $=$ etc. and operations $\ast$, $\circ$, $\bot$, $\top$ etc.

We shall briefly sketch the main ingredients of the design-development language. For the details we refer to appendix B. In the design-development language we have assignment statements (an example of an assignment statement is $d', d'' := \text{bot}(d), \text{top}(d)$). Statements can be composed with sequential composition, a non-deterministic choice construct (symbol $[ ]$) and a repetition construct (keywords while, do and od). We have expressions and expression lists. Expressions may contain operation symbols and variables (an example of an expression is $\text{top}(d)$). Expression lists may contain operation symbols, variables and procedure calls. Assertions may contain expressions, predicate symbols, logical connectives and quantifiers (an example of an assertion is $\forall d (\text{bbc}(d))$). Procedures have a list of input parameters and sometimes a list of result parameters. A procedure is either given axiomatically (keywords pre and post) or it is defined explicitly (keyword def). If a procedure is intended to be executed by a developer, it is called a technique (keyword technique). If a procedure is meant as a description of an event which is not performed by a developer it is called an event (keyword event).

We assume that in the design-development language we have variables of distinct sorts: $d, \ldots$ for pf & ds designs, $c, \ldots$ for components, $v, w, \ldots$ for names and $P, Q, \ldots$ for terms.

In order to have a systematic approach for deriving design-programs we shall use methods which come from the field of classical sequential programming [4], [7]. In particular, if we want to derive a design-program with a repetition construct then we shall first look for a suitable invariant.

We are interested in design-programs which describe the creation of a design which is valid in a given context. Therefore we must start with a formalisation of what it means that a design is valid in a given context.

Definition 3.4.1 (machine&user-context, d valid in w)

(i) A machine&user-context $w$ is a pair $(d_m, d_s)$ of pf & ds designs where
\( d_m \) is called the **machine** of \( w \) and \( d_r \) is called the **system user** of \( w \).

(ii) Let \( w = (d_m, d_r) \), then we say that \( d \) is **valid** in \( w \) if \( \text{bbv}(d_m, d_r) \land \text{bbv}(d_r, d) \).

In a certain way, a machine&user-context \( w \) constitutes a (simplified) view of the external world — at least from the viewpoint of a developer who has to create a design which is valid in \( w \). The definition can be motivated as follows. The **prim components** of \( d \) can be viewed as a specification of all building blocks that are needed by \( d \). When the product described by the design \( d \) becomes somehow operational, then the actual building blocks are provided. We view such a collection of actual building blocks as an underlying ‘machine’. The design \( d \) itself can be viewed as a description of a product to be delivered to the user of \( d \). In general this user has certain requirements to the product described by the design. In our view both providing a machine for a design and providing a product to its user can be described by the binary operation \( \circ \). In particular, the components provided by the the machine can (via \( \circ \)) be plugged into the prims of the design. Similarly we can imagine that the system user is assuming a number of primitives (also prims) which become available by means of the system of the design.

Of course, in many practical situations it is not the case that the system user and the machine are formalised as designs. Often the system user is not formalised at all and validation becomes a matter of informal reasoning and negotiating. Nevertheless we believe that our abstraction might provide some insight for such situations also.

If we call the activity of showing that a design \( d \) is (black-box) correct **verification** and if we call the activity of showing that a design \( d \) is valid in a machine&user-context \( w \) **validation**, then our terminology is consistent with the usual terminology \([1]\): verification \( = \) ‘are we building the product right?’ and validation \( = \) ‘are we building the right product?’.

Let us illustrate this with the example of the four-bit adder of Section 3.4.2. In this example the machine&user-context consists of two “designs”. The first is the design of the 7400 and GND; typically this is a design where transistors and resistors occur as components. This 7400 + GND design is the underlying machine. The second is the design of a higher layer, where the 74283 four-bit adder is just a primitive building block; typically this could be the design of a digital computer. This digital-computer design is the system user.

We see that the validity of a design \( d \) in a given machine&user-context \( w \) depends only on \( \text{bot}(d) \), \( \text{top}(d) \) and on \( w \). This observation can from an intuitive point of view be explained as follows. First of all \( \text{bot}(d) \) contains precisely all prim components of \( d \). The prim components of \( d \) can be
viewed as a summary of all building blocks that are needed by \( d \). Secondly \( \text{top}(d) \) has the same system as \( d \) and \( \text{top}(d) \) contains a number of prim components which can be viewed as specifications of the names occurring in the system of \( d \). The system of \( d \) (with these names eliminated) can be viewed as the top-level product to be delivered to the user of \( d \).

In general, it is the task of the developer to (re-)establish an invariant, \( \text{INV} \) say, which depends both on the design \( d \) and on the machine&user-context \( (d_m, d_u) \). This invariant should consist of two parts where the first part deals with validation and the second part deals with verification. So \( \text{INV} \) must express that \( d \) is valid in its machine&user-context \( (d_m, d_u) \) and that \( d \) is black-box correct and therefore we define it as follows:

\[
\text{INV} \equiv \text{bbv}(d, d_m) \text{ and } \text{bbv}(d_u, d) \text{ and } \text{bbc}(d).
\]

Now we turn our attention to the problem of design creation, which in its most general form is to create a design \( d \) such that \( \text{INV} \) is established. In order to keep things simple, we shall study the problem of design creation in a restricted setting where we focus on the verification aspect. Therefore we assume that before the actual execution of a design-program starts the boundaries of the design to be created are already fixed, by which we mean that somehow the bottom and the top of the design to be created are determined. Since the bottom and the top of a design are designs themselves we can model this situation by assuming that there are two given designs \( d_s \) and \( d_t \). In view of lemma 3.3.27 we consider the top as given up to permutation of prim components. Therefore we have the following postcondition of the of design creation.

\[
\text{POST} \equiv \text{bot}(d) = d_s \text{ and } \text{top}(d) = d_t \text{ and } \text{bbc}(d).
\]

It is possible to give a criterion for the selection of \( d_s \) and \( d_t \). We see that if the machine&user-context \( w = (d_m, d_u) \) then \( d_s \) and \( d_t \) should be chosen such that \( \text{bbv}(d_s, d_m) \) and \( \text{bbv}(d_u, d_t) \) hold. Using lemma 3.3.23 and lemma 3.3.27 one can verify that \( \text{bbv}(d_s, d_m) \) and \( \text{bbv}(d_u, d_t) \) and \( \text{POST} \) implies \( \text{INV} \). Of course it is always possible to derive a \( d_s \) and \( d_t \) from \( (d_m, d_u) \) mechanically, but our approach is somewhat more general.

Let us illustrate this with the example of the four-bit adder of Section 3.4.2. Indeed, in this example we have chosen a certain \( d_t \), viz. the design with one prim-component 74283 and with system \( 74283 \). Also we have chosen a certain \( d_s \), viz. the design with two prim-components 7400 and GND and with system \( 7400 \).

It is not reasonable to assume that the developer can create a large design in one step; instead he adds components and modifies existing components.
one at a time. Therefore we assume that there is one variable \( d \) which always contains the 'current design' and we shall focus on design-programs that modify this \( d \) in a repetitive manner.

There are many (loop-) invariants which could be derived from POST. We shall investigate two possibilities. The first possibility will be investigated in Section 3.4.4 and it leads to the derivation of a design-program which corresponds to the top-down approach. The second possibility will be investigated in Section 3.4.5 and it leads to the derivation of a design-program which corresponds to the bottom-up approach. These design-programs are related to the models of the development process given in [5].

We shall consider the partial correctness of these design-programs; it may very well be the case that a (partially) correct design-program cannot successfully terminate for a given input. In this case we say that the execution of the design-program fails. From this it should be clear that a correctness formula \( \{ \{ A_1 \} \}_{s(\{ A_2 \)} \) does not say anything about failure or successful termination of \( s \). We shall not use an axiomatisation in the style of Hoare's logic [6] for reasoning about these formulae \( \{ A_1 \} s(\{ A_2 \). Of course this could be done, but we think it would push the level of formalisation too far.

3.4.4 Top-down development

In order to obtain an invariant, we take the postcondition POST as a starting point. POST consists of three conjuncts and a candidate invariant is obtained by simply omitting the first conjunct (as suggested in [7] Section 16.2). This yields \( \text{top}(d) = d \) and \( \text{bbc}(d) \), i.e. during the development process the top of the design remains constant (up to permutation of primitive components) and furthermore black-box correctness is adopted as a methodological principle. We strengthen this assertion by requiring that all components (except possibly those in \( d_1 \)) play a role in the system of the design \( d \). In order to formulate this precisely we need an auxiliary definition:

Definition 3.4.2 Let \( d \) be a \( \text{pf} \wedge \text{ds} \) design.

(i) The binary relation \( \leq^s \) on the set of component names of \( d \) is defined by \( x_1 \leq^s x_2 \iff x_1 \) occurs freely in the glass-box description of the component named \( x_2 \).
(ii) The binary relation \( \leq^d \) is defined as the transitive closure of \( \leq^s \).
(iii) Let \( \gamma \) be some subset of \( \text{cast}(d) \) then we define

\[
x_1 \leq^d \gamma \iff x_1 \in \gamma \vee \exists x_2 \in \gamma : x_1 \leq^d x_2.
\]
Roughly speaking, we can view a design \( d \) as a set of components, identified by names, where some components are part of other components. Therefore we shall sometimes refer to \( \prec^d \) as the 'part of' relation.

Now we can express the requirement that all components (except possibly those in \( d_i \)) play a role in the system of the design \( d \) as a condition for all \( v \) \( (v \in \text{cset}(d) \rightarrow (v \in \text{cset}(d_i) \text{ or } v \preceq^d \text{sys}(d))) \). This condition guarantees that no implementation effort is spent on components which will not be used. This yields the following invariant (where we use the notation \( \text{sys}(d) \) from definition 3.2.4):

\[
\text{TD INV} \equiv \top(d) =_{\text{mp}} d_i \text{ and } \\
\text{hbc}(d) \text{ and } \\
\text{forall } v (v \in \text{cset}(d) \rightarrow (v \in \text{cset}(d_i) \text{ or } v \preceq^d \text{sys}(d))).
\]

Now we can develop a design-program based on this invariant. The technique \( \text{td} \) given below has two input parameters \( (d_i \text{ and } d_i) \). It is given by an explicit definition and it uses one variable \( d \). After execution of an initialisation statement and a repetition construct it yields the value of \( d \) as its result.

\[
\text{td} := \text{technique } d_i, d_i \\
def \ d := d_i; \\
\text{while not } \text{bot}(d) = d_i \text{ do } d := \text{td step}(d); \text{ od; } \\
d
\]

where \( \text{td step} \) satisfies the partial-correctness assumption \( \{\text{TD INV} \land \text{bot}(d) \neq d_i\} \ d := \text{td step}(d); \{\text{TD INV}\} \).

Let us illustrate this with the example of the four-bit adder of Section 3.4.2. Recall that in the example the assignment \( d := d_i \) was executed indeed, viz. at the point where we said "This is the initial design of the top-down development." Recall also that in the example we had several points where we said "In order to decompose the ... we shall employ the ..." etc.; each such point marks the beginning of the execution of \( d := \text{td step}(d) \). Finally recall also that in the example we reached a point where we could say: "The resulting design is finished because its bottom equals the agreed bottom design with 7400 and GND." This corresponds with a positive outcome of the step-criterion \( \text{bot}(d) = d_i \).

**Remark 3.4.3** Under the given assumption for \( \text{td step} \) we have

\[
\{\text{bot}(d_i) = d_i \text{ and } \top(d_i) = d_i\} \ d := \text{td}(d_i, d_i); \{\text{POST}\}
\]

(where the precondition simply expresses that \( d_i \) is a bottom design and that \( d_i \) is a top design). This can be proved as follows. First we verify that
after execution of \( d := d_t \) the invariant TD_INV holds. We have \( d = d_t \), so \( \text{top}(d) =_{\text{eq}} d_t \). Since \( d_t \) is a top design it contains only \text{prim} components and therefore it is trivially bbe. Take some \( v \in \text{cas}(d) \), then \( v \leq^d \text{sys}(d) \) because \( d \) is a top design. As a next step we must show that execution of \( d := \text{td\_step}(d) \); does not violate TD_INV. This follows from the assumption for \text{td\_step}. Finally we note that \( \text{bot}(d) = d_t \) and TD_INV implies POST.

We now turn our attention to \text{td\_step}. It will turn out that there exist several techniques which satisfy the assumption for \text{td\_step}. Therefore we shall investigate several techniques which we shall call \text{td\_step}_0, \text{td\_step}_1, etc.

The techniques \text{td\_step}_0 and \text{td\_step}_1 describe the transformation of a \text{prim} component into a non-\text{prim} component. The latter component should be provably correct in a suitable context. Therefore we must describe how the developer should select a \text{prim} component and its context (represented as a design) from a design \( d \). Actually the developer has no choice in this selection: he must select the last \text{prim} component, otherwise the result of re-inserting it (in its original relative position) as an implemented component could violate the condition \text{pf}.

**Definition 3.4.4** Let the \( \text{pf} \wedge \text{ds} \) design \( d \) be given as follows:

\[
\begin{align*}
x_1 &:= \text{prim} \sqsubseteq M_1 \\
\vdots \\
x_n &:= \text{prim} \sqsubseteq M_n \\
y_1 &:= P_1 \sqsubseteq Q_1 \\
\vdots \\
y_m &:= P_m \sqsubseteq Q_m \\
\text{system } [S_1, \ldots, S_l].
\end{align*}
\]

where \( P_1, \ldots, P_m \) are not equal to \text{prim}.

(i) \text{last\_prim}(d) is defined as the component \((x_n := \text{prim} \sqsubseteq M_n)\),

(ii) \text{last\_prim\_context}(d) is defined as the bottom design given by

\[
\begin{align*}
x_1 &:= \text{prim} \sqsubseteq M_1 \\
\vdots \\
x_{n-1} &:= \text{prim} \sqsubseteq M_{n-1} \\
\text{system } [\cdot].
\end{align*}
\]

We assume that there is an operation 'insert' which serves for inserting a component into a design by overwriting an existing component (which has
the same name as the component to be inserted). We do not give a formal definition of ‘insert’. For a name v and a term P we write v ∈ P if v occurs freely in P.

Of course the developer cannot select a prim component from d if d does not contain at least one prim component. The fact that d contains at least one prim component can be expressed as not bot(d) = system []

The technique td_step₀ is a kind of naïve approach. It uses an auxiliary technique called td_impl which takes a design d and which describes the selection of the last prim component and its transformation into a non-prim component (c'). We use the notation Γ_bb from definition 3.2.18.

\[
\begin{align*}
\text{td_impl} & := \text{technique } d \rightarrow c' \\
\text{pre not bot}(d) & = \text{system } [] \\
\text{post exists } v & \text{ (exists } P \text{ (exists } Q \\
& \text{(last_prim}(d) = (v := \text{prim } \sqsubseteq Q) \text{ and } \\
& c' = (v := P \sqsubseteq Q) \text{ and } \\
& \forall w \in P \rightarrow w \in \text{cset(last_prime_context}(d)) \text{ and } \\
& \Gamma_{bb}(\text{last_prime_context}(d)) = P \sqsubseteq Q)\text{))}
\end{align*}
\]

\[
\begin{align*}
\text{td_step}_0 & := \text{technique } d \\
\text{def insert}(d, \text{td_impl}(d))
\end{align*}
\]

Let us illustrate this with the example of the four-bit adder of Section 3.4.2. Indeed, at a certain point we said: “To decompose the 7408 and-gate we need no new components. It can be done easily with one 7400 nand-gate and one 7404 inverter. We refer to the following interconnection diagram as 7408IMPL.” etc. Formally this corresponds with a last_prim(d) which is (7408 := prim ⊆ (s = z,y)). The last_prime_context(d) here contains three prim-components, viz. GND, 7400 and 7404.

Lemma 3.4.5 (Partial correctness of td_step₀).

\[
\{\text{TD.INV}\} \; d := \text{td_step}_0(d); \{\text{TD.INV}\}.
\]

Proof. Assume that initially d is the design given in definition 3.4.4. We assume that this design satisfies TD.INV. If this design has no prim component then td_step₀ fails because the preconditions of td_impl does not hold. Recall that we deal with partial correctness, so td_step₀ is allowed to fail. Therefore we proceed with the assumption that there is at least one prim component and hence last_prim(d) = (xₙ := prim ⊆ Mₙ). Then it follows that td_impl returns a component (xₙ := P ⊆ Mₙ) for some P. This P contains only names from \{x₁,...,xₙ₋₁\} and it satisfies [x₁ ⊆ M₁],..., [xₙ₋₁ ⊆ Mₙ₋₁] ⊨ P ⊆ Mₙ. Execution of ‘insert’ yields the design
\[ x_1 := \text{prim} \subseteq M_1 \\
\ldots \\
x_{m-1} := \text{prim} \subseteq M_{m-1} \\
x_m := P \subseteq M_m \\
y_1 := P_1 \subseteq Q_1 \\
\ldots \\
y_m := P_m \subseteq Q_m \\
\text{system} \{S_1, \ldots, S_l\} \]

which is \( pf \land ds \) and \( bcc \). We note that the top of the latter design equals precisely the top of the initial design. Finally we show that all names from \( \{x_1, \ldots, x_m, y_1, \ldots, y_m\} \) which are not in \( \text{cset}(d) \) are part of the system. This follows from the fact that \( \text{TD_INV} \) holds for the initial design. \( \square \)

The definition of \( \text{td_step}_0 \) is a kind of naive approach because \( \text{td_step}_0 \) does not allow for the creation and insertion of new prim components. We shall now describe an improvement with respect to \( \text{td_step}_0 \). We could define a technique which describes the creation and insertion of one new prim component; however, we shall not do this because it is essential for the top-down approach as expressed by \( \text{TD_INV} \) that no new prim component is introduced unless it is used immediately. We prefer a technique which describes both the creation of new prim components and the transformation of an existing prim component into a non-prim component whose glass-box description uses all new prim components. This leads us to \( \text{td_step}_1 \) which is an improved version of \( \text{td_step}_0 \). It uses an auxiliary technique called \( \text{td_spec_impl} \) which describes the selection of the last prim component, the creation of a set of new prim components and the transformation of the selected prim component into a non-prim component. The set of new prim components is represented as a bottom design (\( d' \)).

\[
\begin{align*}
\text{td_spec_impl} & := \text{technique } d \rightarrow d', c' \\
\text{pre not bot}(d) = \text{system} [] \\
\text{post exists } u \ (\exists ) P \ (\exists ) Q \\
& \quad \text{cset}(d') \cap \text{cset}(d) = \emptyset \text{ and} \\
& \quad \text{bot}(d') = d' \text{ and} \\
& \quad c' = (v := P \subseteq Q) \text{ and} \\
& \quad \forall w \ w \in \text{cset}(d') \rightarrow w \in P \text{ and} \\
& \quad \forall w \ w \in P \rightarrow w \in \text{cset(last_prim_context}(d) \cup \text{cset}(d'))) \\
& \quad \text{and } \Gamma_{bb}(d') \cup \Gamma_{bb}(\text{last_prim_context}(d)) \vdash P \subseteq Q) \\
\end{align*}
\]

We can use the binary operation \( * \) of Section 3.3 for describing the addition of the new prim components to the current design.
\text{td\_step}_1 := \text{technique } d \\
def d', c' := \text{td\_spec\_impl}(d) \\
\quad \text{insert}(d' \ast d, c')

Let us illustrate this with the example of the four-bit adder of Section 3.4.2. At some point of that development we said: "In order to decompose the 74183 single-bit full adder, we employ ... which are provided by new components 7408, 7432 and 7486 respectively." Then these new components were viewed as a mini-design $d'$ and we concatenated $d'$ with $d$ and furthermore we said: "Secondly, we must insert a new version of 74183 which has 74183IMPL instead of prim." In fact this was an example of the execution of $d', c' := \text{td\_spec\_impl}(d); \text{insert}(d' \ast d, c')$.

**Lemma 3.4.6** (Partial correctness of \text{td\_step}_1).

\begin{align*}
\{\text{TD\_INV}\} & \quad d := \text{td\_step}_1(d); \{\text{TD\_INV}\}. \\
\end{align*}

\textbf{Proof.} Assume that initially $d$ is the design given in definition 3.4.4. We assume that this design satisfies TD\_INV. If this design has no prim components then \text{td\_step}_1 fails because the precondition of \text{td\_spec\_impl} does not hold. Therefore we can assume that there is at least one prim component and hence last\_prim$(d) = \{x_n := \text{prim} \subseteq M_n\}$. Therefore \text{td\_spec\_impl} yields a bottom design $(d')$ with new names, say

\[
\begin{align*}
z_1 & := \text{prim} \subseteq R_1 \\
\cdots \\
z_k & := \text{prim} \subseteq R_k \\
\text{system} & \mid \\
\end{align*}
\]

and it also yields a component $(x_n := P \subseteq M_n)$ for some $P$. This $P$ contains only names from $\{z_1, \ldots, z_{n-1}, x_1, \ldots, z_k\}$ and it satisfies $[z_1 \subseteq R_1], \ldots, [z_k \subseteq R_k], [x_1 \subseteq M_1], \ldots, [x_{n-1} \subseteq M_{n-1}] \vdash P \subseteq M_n$. Furthermore each $z_i$ $(1 \leq i \leq k)$ occurs in $P$. This bottom design is added to the current design, using $\ast$, and execution of 'insert' yields the design

\[
\begin{align*}
z_1 & := \text{prim} \subseteq R_1 \\
\cdots \\
z_k & := \text{prim} \subseteq R_k \\
z_1 & := \text{prim} \subseteq M_1 \\
\cdots \\
z_{n-1} & := \text{prim} \subseteq M_{n-1} \\
z_n & := P \subseteq M_n \\
y_1 & := P_1 \subseteq Q_1 \\
\end{align*}
\]
... 
\[ y_m := P_m \subseteq Q_m \]

which is pf \wedge ds and bbc. We note that the top of the latter design equals precisely the top of the original design. Finally we must show that all names from \( \{x_1, \ldots, x_{m}, y_1, \ldots, y_{m}, z_1, \ldots, z_k\} \) which are not in \( \text{cs}(d_i) \) are 'part of' the system. For the names from \( \{x_1, \ldots, x_{m}, y_1, \ldots, y_{m}\} \) this follows from the fact that \( \text{TD}_\text{INV} \) holds for the original design. For the names from \( \{z_1, \ldots, z_k\} \) this follows from the fact that \( x_n \) is 'part of' the system and the fact that \( z_1, \ldots, z_k \) occur in the new glass-box description of the component named \( x_n \).

\( \square \)

d_\text{step}_1 \) raises two related problems. The first problem is that the developer has no choice in selecting the prim component to be implemented (although he can influence later choices by 'thinking in advance'). For the prim components which are in the top design \( d_i \) he has no influence at all on the order in which they are selected. The second problem is that it may be hard to make sure that the order of the prim components will match the order of the prim components in the bottom design \( d_i \). We shall remedy these problems by describing the possibility that the developer modifies the order of the prim components. We formalise this by defining a technique \( d_\text{step}_2 \).

\[
\begin{align*}
  d_\text{step}_2 := & \text{technique } d \to d' \\
  \text{pre} & \text{ true} \\
  \text{post} & d' =_{pp} d
\end{align*}
\]

Let us illustrate this with the example of the four-bit adder of Section 3.4.2. Indeed, at a certain point we said: "We could select the last prim-component as the next candidate to be implemented – which typically is a kind of default in top-down development. However, we have no intention to decompose \( \text{GND} \), so we permute some prim components, putting \( \text{GND} \) first." This was a typical execution of \( d_\text{step}_2 \).

**Lemma 3.4.7** (Partial correctness of \( d_\text{step}_2 \).)

\[ \{\text{TD}_\text{INV}\} \ d := d_\text{step}_2(d) \; ; \{\text{TD}_\text{INV}\} \]

**Proof.** We assume that \( \text{TD}_\text{INV} \) holds for \( d \) and we must show that it holds also for \( d' \) where \( d' =_{pp} d \). Since we have \( \text{top}(d) =_{pp} d_i \) we also have \( \text{top}(d') =_{pp} d_i \). The fact that \( d' \) is bbc follows from the fact that \( d \) is bbc. Finally we note that the 'part of' relation does not depend on the order of the prim components.

\( \square \)
It is possible to execute \texttt{td\_step} by choosing between \texttt{td\_step\_1} and \texttt{td\_step\_2}. Using the non-deterministic choice construct \( [] \) from the design-development language we define \texttt{td\_step} as

\[
\texttt{td\_step} := \text{technique } d \\
\text{def } d' := \texttt{td\_step\_1}(d) []; d'' := \texttt{td\_step\_2}(d);
\]

Now we can combine the results on \texttt{td\_step\_1} and \texttt{td\_step\_2}.

\textbf{Theorem 3.4.8} (Partial correctness of the top-down technique).

(i) \( \{ \text{TD\_INV} \} d := \texttt{td\_step}(d); \{ \text{TD\_INV} \} \)

(ii) \( \{ \text{bot}(d_4) = d_4 \text{ and } \text{top}(d_4) = d_4 \} d := \texttt{td}(d_4, d_4); \{ \text{POST} \}. \)

\textbf{Proof.} (i) By the semantics of \( [] \) and by lemmas 3.4.6 and 3.4.7. (ii) By (i) and remark 3.4.3, noting that \( \text{TD\_INV} \land \text{bot}(d) \neq d_4 \) implies \( \text{TD\_INV} \).

\textbf{Remark 3.4.9} (i) Let us consider the total-correctness question for \texttt{td}, which is as follows: can every execution sequence which is according to the top-down model but which is not ready yet, be completed to a full execution sequence? In other words: is it the case that a top-down development process essentially never can get stuck? The answer is positive: every execution sequence can be completed. But the main reason for this is the reflexivity of \( \subseteq \). From a practical point of view, this answer is of little interest. As soon as executability considerations or efficiency considerations (at the product level) play a role, either "thinking in advance" or "backtracking" are needed.

(ii) In the postconditions of the techniques \texttt{td\_step\_0} and \texttt{td\_step\_1} we have clauses

\[
\Gamma_{bb}(\text{last\_prim\_context}(d)) \vdash P \subseteq Q \text{ and } \Gamma_{bb}(d') \cup \Gamma_{bb}(\text{last\_prim\_context}(d)) \vdash P \subseteq Q. \text{ It is interesting to note that if in these clauses we replace } \Gamma_{bb} \text{ by } \Gamma_b \text{ the resulting top-down technique is still partially correct with respect to POST. We can interpret this fact as follows: as long as the developer works according to the top-down technique, it does not matter if he knows the difference between black-box correctness and glass-box correctness.}
\]

(iii) Our description of the top-down technique should be considered as open-ended. One can think of techniques \texttt{td\_step\_3}, \texttt{td\_step\_4} etc. E.g. \texttt{td\_step\_3} could describe the possibility of back-tracking where components are removed and where non-prim components are transformed into prim components. \texttt{td\_step\_4} could describe the possibility of adding prim components which need not be 'part of' the system of the current design, but which happen to be present in \( d_4 \). A general form of the top-down technique could be based
3.4. DESIGN CREATION

on a technique $\text{td.step}$ given as follows.

$$
\text{td.step} := \text{technique } d
$$

$$
def \bigwedge_{i=1,2,...,n} d' := \text{td.step}_i(d);
$$

Finally we consider a kind of completeness question: can every $\text{pf} \land d_s \land \text{bbc}$ design $d$ where all components play a role in its system, be obtained by means of a top-down development? The answer is positive, for a given $d$ is easily obtained from a $d_i = \text{top}(d)$ by a sequence of $\text{td.step}_i$ steps.

3.4.5 Bottom-up development

In order to obtain another invariant we take again the postcondition POST as a starting point. POST consists of three conjuncts and a candidate invariant is obtained by simply omitting the second conjunct. This yields $\text{bot}(d) = d_s$ and $\text{bbc}(d)$, i.e. during the development process the bottom of the design remains constant and black-box correctness is adopted as a methodological principle. We need not strengthen this assertion by requiring that all components are built in terms of primitive components, since this is taken care of by the fact that we consider only designs which are $\text{pf} \land d_s$ and hence $\text{wf}$. We adopt the following invariant:

$$
\text{BU.INV} := \text{bot}(d) = d_s \land \text{bbc}(d).
$$

Now we can develop a design-program based on this invariant. The technique 'bu' given below has two input parameters ($d_s$ and $d_i$). It is given by an explicit definition and it uses one variable ($d$). After execution of an initialisation statement and a repetition construct it yields the value of $d$ as its result.

$$
\text{bu} := \text{technique } d_s, d_i
$$

$$
def \ d := d_s;
$$

$$\quad \text{while not } \text{top}(d) = d_i \text{ do } d := \text{bu.step}(d); \text{ od};
$$

$$
d
$$

where $\text{bu.step}$ satisfies $\{\text{BU.INV} \land \text{top}(d) \neq d_s \} d := \text{bu.step}(d); \{\text{BU.INV}\}.

Remark 3.4.10 Under the given assumption for $\text{bu.step}$ we have

$$
\{\text{bot}(d) = d_s \text{ and } \text{top}(d) = d_i \} \ d := \text{bu}(d_s, d_i); \ \{\text{POST}\}.
$$

This can be proved as follows. First we verify that after execution of $d := d_s$;
the invariant \( \text{BU.INV} \) holds. We have \( d = d_4 \), so \( \text{bot}(d) = d_4 \). Since \( d_4 \) is a bottom design it contains only \( \text{prim} \) components and therefore it is trivially bbc. As a next step we must show that execution of \( d := \text{bu.step}(d) \) does not violate \( \text{BU.INV} \). This follows from the assumption for \( \text{bu.step} \). Finally we note that \( \text{top}(d) = \text{pp} \ d_4 \) and \( \text{BU.INV} \) implies \( \text{POST} \).

We now turn our attention to \( \text{bu.step} \). It will turn out that there exist several techniques which satisfy the assumption for \( \text{bu.step} \). Therefore we shall investigate several techniques which we shall call \( \text{bu.step}_0, \text{bu.step}_1 \), etc.

Let us assume an operation 'add.system.element' which takes a design, \( d \), say, and a term \( P \). If \( d \) has system \( [S_1, \ldots, S_i] \), then \( \text{add.system.element}(d, P) \) yields the design which has the same components as \( d \), but which has the sequence \( [S_1, \ldots, S_i, P] \) as its system. We do not give a formal definition of 'add.system.element'.

The technique \( \text{bu.step}_0 \) describes the creation of a term and its addition to the system of the current design.

\[
\text{bu.step}_0 := \text{technique} \ d \rightarrow d' \\
\text{pre} \ true \\
\text{post} \ exists \ P \\
\quad \left( \forall w \ (w \in P \rightarrow w \in \text{set}(d)) \text{ and } \ d' = \text{add.system.element}(d, P) \right)
\]

**Lemma 3.4.11** (Partial correctness of \( \text{bu.step}_0 \)).

\( \{\text{BU.INV}\} d := \text{bu.step}_0(d); \{\text{BU.INV}\} \).

**Proof.** Assume that initially \( d \) is the design given in definition 3.4.4. We assume that this design satisfies \( \text{BU.INV} \). Since \( P \) contains only names from \( \{x_1, \ldots, x_n, y_1, \ldots, y_m\} \), evaluation of \( \text{add.system.element}(d, P) \) yields the design

\[
\begin{align*}
  x_1 & := \text{prim} \in M_1 \\
  \ldots \\
  x_n & := \text{prim} \in M_n \\
  y_1 & := P_1 \in Q_1 \\
  \ldots \\
  y_m & := P_m \in Q_m \\
  \text{system} \ [S_1, \ldots, S_i, P]
\end{align*}
\]

which is \( \text{pf} \land \text{ds} \) and bbc. We note that the bottom of the latter design equals precisely the bottom of the initial design. \( \Box \)
3.4. DESIGN CREATION

Let us assume an operation 'add_component' which takes a design, \( d \) say, and a component \( c \) and which yields the design which has all components of \( d \) and which has furthermore \( c \) as its last component. We do not give a formal definition of 'add_component'. The technique \( \text{bu.step}_1 \) describes the creation of a non-\( \text{prim} \) component and its addition to the current design.

\[
\begin{align*}
\text{bu.step}_1 & := \text{technique } d \to d' \\
\text{pre} & \text{ true} \\
\text{post} & \text{ exists } v \text{ (exists } P \text{ (exists } Q \\
& \quad \text{ (not } v \in \text{cset}(d) \text{ and} \\
& \quad \quad \forall w \ (w \in P \rightarrow w \in \text{cset}(d)) \text{ and} \\
& \quad \quad \Gamma_{bh}(d) = P \subseteq Q \text{ and} \\
& \quad \quad d' = \text{add.component}(d, (v := P \subseteq Q))))
\end{align*}
\]

Note that we did not give the relative order in which \( P \) and \( Q \) have to be constructed.

**Lemma 3.4.12** (Partial correctness of \( \text{bu.step}_1 \)).

\[\{\text{BU.INV}\} \ d := \text{bu.step}_1(d); \{\text{BU.INV}\}.\]

**Proof.** Assume that initially \( d \) is the design given in definition 3.4.4. We assume that this design satisfies BU.INV. The result \( d' \) is the result of evaluating \( \text{add.component} \) with \( d \) as its first argument and with a component \((v := P \subseteq Q)\) as its second argument. From the postcondition of \( \text{bu.step}_1 \) we have that \( v \) is a new name and that all names occurring in \( P \) are from \( \{x_1, \ldots, x_m, y_1, \ldots, y_n\} \). Furthermore this component is correct in the black-box context of \( d \). Evaluation of \( \text{add.component}(d, (v := P \subseteq Q)) \) yields the design

\[
\begin{align*}
x_1 & := \text{prim} \sqsubseteq M_1 \\
\vdots \\
x_n & := \text{prim} \sqsubseteq M_n \\
y_1 & := P_1 \sqsubseteq Q_1 \\
\vdots \\
y_m & := P_m \sqsubseteq Q_m \\
v & := P \subseteq Q \\
\text{system} & [S_1, \ldots, S_l]
\end{align*}
\]

which is \( \text{pf} \wedge \text{ds} \) and \( \text{bbc} \). We note that the bottom of the latter design equals precisely the bottom of the initial design. \( \square \)

It is possible to execute \( \text{bu.step} \) by choosing between \( \text{bu.step}_0 \) and \( \text{bu.step}_1 \). Using the non-deterministic choice construct [ ] from the design-development
language we define $bu \_ step$ as

\[
bu \_ step := \text{ technique } d \\
\text{ def } d' := bu \_ step_0(d); [\cdot] d' := bu \_ step_1(d); \\
\text{ d'}
\]

Now we can combine the results on $bu \_ step_0$ and $bu \_ step_1$.

**Theorem 3.4.13** (Partial correctness of the bottom-up technique).

(i) \{BU \_ INV\} $d := bu \_ step(d); \{BU \_ INV\}$,

(ii) $\{\text{bot}(d_i) = d_i \text{ and top}(d_i) = d_i\} \ d := bu(d_i, d_i); \{\text{POST}\}$.

**Proof.** (i) By the semantics of $\cdot$ and by 3.4.11 and 3.4.12. (ii) By (i) and remark 3.4.10, noting that $BU \_ INV \land \text{top}(d) \neq_{pp} d_i$ implies $BU \_ INV$. □

**Remark 3.4.14** (i) Let us consider the total-correctness question for $bu$, which is as follows: assume a given $d_i$ and $d_i$; can every execution sequence which is according to $bu(d_i, d_i)$ but which is not ready yet, be completed to a full execution sequence? The answer is negative because we can only add system elements and not remove them. Once we have too many system elements, we have no way left to arrive at the same system as $d_i$.

(ii) In the postcondition of the technique $bu \_ step_1$ we have a clause $\Gamma_{bb}(d) \vdash P \subseteq Q$. It is interesting to note that if in this clause we replace $\Gamma_{bb}$ by $\Gamma_{gb}$, the resulting bottom-up technique is not partially correct with respect to POST. Instead its execution will yield a glass-box correct design.

(iii) Our description of the bottom-up technique should be considered as open-ended. One can think of techniques $bu \_ step_0, bu \_ step_1$, etc. E.g. $bu \_ step_3$ could describe the possibility of back-tracking, where components or system elements are removed. □

Finally we consider a kind of completeness question: can every pf and ds and bbc design $d$ be obtained by means of a bottom-up development? The answer is positive, for a given $d$ is easily built-up from a $d_i = \text{bot}(d_i)$ by means of a number of $bu \_ step_1$ steps followed by a number of $bu \_ step_0$ steps.

As an illustration of the design program $bu$, we show a sequence of designs corresponding with a possible bottom-up development of the four-bit adder. The first design $d_i$ is $d_i$, which is

\[
7400 := \text{prim} \quad \subseteq \quad (x = (a,b)) \\
\text{GND} := \text{prim} \quad \subseteq \quad (g = 0) \\
\text{system} \quad | \quad |
\]
3.4. DESIGN CREATION

\( d_3 \) is obtained by a \texttt{bu_step1}:

\[
\begin{align*}
7400 & := \text{prim} & (x = (a \cdot b)) \\
\text{GND} & := \text{prim} & (g = 0) \\
7404 & := \text{7404IMPL} & (q = \overline{p}) \\
\text{system} & []
\end{align*}
\]

\( d_5 \) is obtained by a \texttt{bu_step1}:

\[
\begin{align*}
7400 & := \text{prim} & (x = (a \cdot b)) \\
\text{GND} & := \text{prim} & (g = 0) \\
7404 & := \text{7404IMPL} & (q = \overline{p}) \\
7408 & := \text{7408IMPL} & (z = x \cdot y) \\
7432 & := \text{7432IMPL} & (z = x + y) \\
\text{system} & []
\end{align*}
\]

\( d_6 \) is obtained by a \texttt{bu_step1}:

\[
\begin{align*}
7400 & := \text{prim} & (x = (a \cdot b)) \\
\text{GND} & := \text{prim} & (g = 0) \\
7404 & := \text{7404IMPL} & (q = \overline{p}) \\
7408 & := \text{7408IMPL} & (z = x \cdot y) \\
7432 & := \text{7432IMPL} & (z = x + y) \\
7486 & := \text{7486IMPL} & (z = x \oplus y) \\
\text{system} & []
\end{align*}
\]

\( d_9 \) is obtained by a \texttt{bu_step1}:

\[
\begin{align*}
7400 & := \text{prim} & (x = (a \cdot b)) \\
\text{GND} & := \text{prim} & (g = 0) \\
7404 & := \text{7404IMPL} & (q = \overline{p}) \\
7408 & := \text{7408IMPL} & (z = x \cdot y) \\
7432 & := \text{7432IMPL} & (z = x + y) \\
7486 & := \text{7486IMPL} & (z = x \oplus y) \\
74183 & := \text{74183IMPL} & (s = a \oplus b \oplus c, c = a \cdot b + c, (a \oplus b)) \\
\text{system} & []
\end{align*}
\]
$d_1$ is obtained by a bu_step$_i$:

\[
\begin{align*}
7400 & := \text{prim} & (x = (a \oplus b)) \\
\text{GND} & := \text{prim} & (g = 0) \\
7404 & := 7404\text{IMPL} & (q = \overline{p}) \\
7408 & := 7408\text{IMPL} & (z = x \cdot y) \\
7432 & := 7432\text{IMPL} & (z = x + y) \\
7486 & := 7486\text{IMPL} & (z = x \oplus y) \\
74183 & := 74183\text{IMPL} & (s = a \oplus b \oplus c, c_o = a \cdot b + c_r (a \oplus b)) \\
74283 & := 74283\text{IMPL} & (\text{int}(z) = \text{int}(\bar{a}) + \text{int}(\bar{b})) \\
\text{system} & | 74283 \\
\end{align*}
\]

and after a bu_step$_0$ we get $d_0$ which is

\[
\begin{align*}
7400 & := \text{prim} & (x = (a \oplus b)) \\
\text{GND} & := \text{prim} & (g = 0) \\
7404 & := 7404\text{IMPL} & (q = \overline{p}) \\
7408 & := 7408\text{IMPL} & (z = x \cdot y) \\
7432 & := 7432\text{IMPL} & (z = x + y) \\
7486 & := 7486\text{IMPL} & (z = x \oplus y) \\
74183 & := 74183\text{IMPL} & (s = a \oplus b \oplus c, c_o = a \cdot b + c_r (a \oplus b)) \\
74283 & := 74283\text{IMPL} & (\text{int}(z) = \text{int}(\bar{a}) + \text{int}(\bar{b})) \\
\text{system} & | 74283 \\
\end{align*}
\]

This concludes the bottom-up example.

### 3.5 Design Evolution

#### 3.5.1 General

In Section 3.4 we investigated design creation, but it would be wrong to assume that in realistic software development the 'machine&user-context' in which the design is supposed to be valid is always a constant machine&user-context. In current software development practice it may very well be the case that about 50% of the development costs of a product are spent on so-called 'maintenance' (see e.g. [1] pages 540, 541). Traditionally, maintenance was classified into software update and software repair, where software repair includes a corrective aspect (see e.g. [1] page 536). In this chapter we shall not investigate this corrective aspect. From now on we shall use the term 'design evolution' (because 'maintenance' suggests that there might be something like 'wear', which of course is not the case for software products).
3.5. DESIGN EVOLUTION

If we want to discuss design evolution then we must have an abstraction of the (variable) machine&user-context in which a (variable) design 'evolves'. We adopt the definitions of 3.4.1 where a machine&user-context $w$ is a pair $(d_m, d_s)$ of pf & da designs where $d_m$ is called the machine of $w$ and $d_s$ is called the system user of $w$. The design $d$ is said to be valid in $w$ if $bbv(d, d_m) \land bbv(d, d_s)$. This definition was motivated by viewing the prim components of $d$ as a specification of all building blocks that are needed by $d$ and by viewing the design $d$ as a description of a product to be delivered to the user of $d$.

We consider design evolution to be the evolution of a design in a changing machine&user-context. The developer operates on a variable design which is part of a global state. Also part of this global state is a variable machine&user-context $w = (d_m, d_s)$. The machine&user-context is modified, let us say at certain points in time. We formalise this view by assuming that there are three variables $d_m, d_s$ and $d$. We have the following intuitions for these variables: $d_m$ = 'current machine', $d_s$ = 'current system user', $d$ = 'current design'.

It is the task of the developer to (re-)establish the invariant INV, which depends both on the design $d$ and on the machine&user-context $(d_m, d_s)$ and which expresses that $d$ is valid in the machine&user-context $(d_m, d_s)$ and that $d$ is black-box correct.

$$INV \equiv bbv(d, d_m) \land bbv(d, d_s) \land bbc(d).$$

The following scenario is adopted. We assume a state in which $d_m, d_s$ and $d$ are such that INV holds. We now assume that the machine&user-context of the next state has been modified and we say that an external event has happened. The developer must find a design $d'$ such that after establishing the state modification $d := d'$; the invariant INV holds again. The developer may do this by acting according to some technique. Let us assume that the change of the machine&user-context is such that either the machine is modified or the system user is modified, but not both. We shall discuss both kinds of machine&user-context change separately. One might devise many techniques addressing the problem of design evolution; we only show some of the simplest techniques. In Section 3.5.2 we shall give a technique which deals with a changing machine. In Section 3.5.3 we shall give a technique which deals with a changing system user.

### 3.5.2 Changing machine

The following procedure can be used for modeling an external event:
change := event $d \to d'$
pre true
post true

After the external event $d_m := \text{change}(d_m)$ has happened there is a new machine but there is still the old system user. The condition $\text{bbv}(d, d_m)$ may be false but $\text{bbv}(d', d)$ and $\text{bbc}(d)$ hold. The developer must restore the invariant and therefore we look for a suitable statement $s_m$ such that

$$\{ \text{INV} \} \ d_m := \text{change}(d_m); \ s_m \ \{ \text{INV} \}.$$  

One possible technique is based on the idea of an emulator, which is nothing but a 'layer' interpolating between the new machine and the old design. Finding an emulator is described by a technique which takes designs $d_m$ (the new machine) and $d$ (the old design) and yields an emulator $d_{em}$.

emulator := technique $d_m, d \to d_{em}$
pre true
post $\text{bbc}(d_{em})$ and $\text{bbv}(d_{em}, d_m)$ and $\text{bbv}(d, d_{em})$

Now if emulator$(d_m, d)$ yields $d_{em}$, then we might say that $d_{em} \circ d_m$ is 'd-equivalent' with the old machine, by which we mean that $\text{bbv}(d, d_{em} \circ d_m)$ holds. So it is possible to make the old design 'run' upon $d_{em} \circ d_m$ by constructing the composition $d \circ (d_{em} \circ d_m)$. By lemma 3.3.7 (associativity of $\circ$) this is the same as $(d \circ d_{em}) \circ d_m$. If we assume that the developer has modification rights with respect to $d$ but not with respect to the machine, he should replace the old design $d$ by $(d \circ d_{em})$. This indicates that we can take the following statement for $s_m$:

$$d := d \circ \text{emulator}(d_m, d);$$

Let us illustrate this with the example of the four-bit adder of Section 3.4.2. Suppose that the supply of 7400 nand-gates gets exhausted whereas there is a rich supply of 7402 nor-gates, say. Then we could have an emulator design as follows:

\[
\begin{align*}
7402 & := \text{prim} & \Downarrow & (z = (a + b)) \\
GND & := \text{prim} & \Downarrow & (y = 0) \\
7400 & := 7400\text{IMPL} & \Downarrow & (z = (\overline{a.b})) \\
system & [7400,GND]
\end{align*}
\]

where 7400IMPL implements the functionality of the nand-gate using nor-gates and ground-connections only.
3.5. DESIGN EVOLUTION

Theorem 3.5.1 \{INV\} $d_m := \text{change}(d_m); d := d \circ \text{emulator}(d_m, d); \{\text{INV}\}$

Proof. Assume that initially $d$ satisfies INV. After $d_m := \text{change}(d_m); d$ has happened we have $\text{bbv}(d_m, d)$ and $\text{bbc}(d)$. Execution of the emulator technique yields some design $d_{em}$ such that $\text{bbc}(d_{em})$ and $\text{bbv}(d_{em}, d_m)$ and $\text{bbv}(d, d_{em})$ hold. We must investigate each conjunct of INV for the new design $d \circ d_{em}$.

- $\text{bbv}(d \circ d_{em}, d_m)$ holds because $\text{bbv}(d_{em}, d_m)$ holds.
- $\text{bbv}(d_m, d \circ d_{em})$ holds because $\text{bbv}(d_m, d)$ holds.
- $\text{bbc}(d \circ d_{em})$ follows by lemma 3.3.14 (ii) from $\text{bbc}(d), \text{bbc}(d_m)$ and $\text{bbv}(d, d_{em})$. \qed

Remark 3.5.2 We shall briefly sketch an alternative solution for the statement $s_m$. It is based on the idea of (re-)starting a top-down development process. Let us assume that we have an operation 'remove_unused' which takes a design and which yields the design which is obtained by removing all components which are not 'part of' the system (in the sense of $\leq^d \text{sys}(d)$). Actually, for top-down made designs this means that no implementations are thrown away. Let us also assume a technique 'determine_bottom' which for given machine $d_m$ yields a bottom design $d_b$ such that $\text{bbv}(d_b, d_m)$. After the external event $d_m := \text{change}(d); d$ has happened we are in a state in which $\text{bbv}(d, d)$ and $\text{bbc}(d)$ holds. If in this state the statement $d := \text{remove_unused}(d); d := \text{determine_bottom}(d_m); d := \text{top}(d)$; is executed then the invariant TD.INV as given in Section 3.4.4 holds (where $d := \text{top}(d)$ means to fix the existing top, rather than throwing something away). When TD.INV holds, the developer can start executing the repetition statement which we know already from the top-down approach, viz. while not bot(d) := $d$ do $d := \text{td_step}(d)$; od; If the latter statement terminates then INV holds again.

Under certain conditions it may be possible to derive an emulator from the result of this top-down development process. \qed

3.5.3 Changing system user

After the external event $d_e := \text{change}(d_e); d$ has happened there is a new system user but there is still the old machine. The conditions $\text{bbv}(d, d_m)$ and $\text{bbc}(d)$ hold but $\text{bbv}(d_e, d)$ may be false. The developer must restore the invariant and therefore we look for a suitable statement $s_e$ such that

\{INV\} $d_e := \text{change}(d_e); s_e \{\text{INV}\}$. 
One possible technique is based on the idea of a simulator. Finding a ‘simulator’ is described by the following technique:

\[
\text{simulator} ::= \text{technique } d_s, d \rightarrow d_{si} \\
\text{pre true} \\
\text{post } \text{bbc}(d_{si}) \text{ and } \text{bbv}(d_{si}, d) \text{ and } \text{bbv}(d_s, d_{si})
\]

Now we can take the following statement for \( s_i \):

\[
d := \text{simulator}(d_s, d) \circ d;
\]

**Theorem 3.5.3** \{INV\} \( d_s := \text{change}(d_s); d := \text{simulator}(d_s, d) \circ d; \{\text{INV}\} \).

**Proof.** Assume that initially \( d \) satisfies INV. After \( d_s := \text{change}(d_s) \); has happened we have \( \text{bbv}(d, d_{si}) \) and \( \text{bbc}(d) \). Execution of the simulator technique yields some design \( d_{si} \) such that \( \text{bbc}(d_{si}) \) and \( \text{bbv}(d_{si}, d) \) and \( \text{bbv}(d_s, d_{si}) \) hold. We must investigate each conjunct of INV for the new design \( d_{si} \circ d \).

- \( \text{bbv}(d_{si} \circ d, d_{si}) \) holds because \( \text{bbv}(d, d_{si}) \) holds.
- \( \text{bbv}(d_{si}, d_{si} \circ d) \) holds because \( \text{bbv}(d_s, d_{si}) \) holds.
- \( \text{bbc}(d_{si} \circ d) \) follows from Lemma 3.3.14 (ii) from \( \text{bbc}(d_{si}) \), \( \text{bbc}(d) \) and \( \text{bbv}(d_{si}, d) \).

**Remark 3.5.4** We shall briefly sketch an alternative solution for the statement \( s_i \). It is based on the idea of (re-)starting a bottom-up development process. Let us assume that we have an operation ‘empty system’ which takes a design and yields the design which is obtained by making the system equal to \( \emptyset \). Let us also assume a technique ‘determine.top’ which for given system user \( d_s \) yields a top design \( d_t \) such that \( \text{bbv}(d_s, d_t) \). After the external event \( d_s := \text{change}(d_s) \); has happened we are in a state in which \( \text{bbv}(d_s, d_{si}) \) and \( \text{bbc}(d) \) holds. If in this state the statement \( d := \text{empty.system}(d); d_t := \text{determine.top}(d_s); d_b := \text{bot}(d); \) is executed then the invariant BU.INV as given in Section 3.4.5 holds. Therefore the developer can start executing the repetition statement which we know already from the bottom-up approach, viz. while \( \text{not } \text{top}(d) = \emptyset \) \( d_t \) do \( d := \text{bu.step}(d); \) od; If the latter statement terminates then INV holds again.

Under certain conditions it may be possible to derive a simulator from the result of this bottom-up development process. \( \square \)
3.6 Design Partition

3.6.1 General

In this section we want to investigate parallel development. Often it is desirable to have several developers doing their work simultaneously such that their interaction is limited. The result of their joint effort should be a valid and correct design. The condition that a design \( d \) is valid and correct can be expressed by \( \text{INV} \) as given in Sections 3.4.3 and 3.5. The motivation behind this approach is that one wants to achieve a reduction in development time compared with the approach in which there is only one active developer at a time.

The design-development language of appendix B does not have a parallel composition construct for statements but some design-programs offer a possibility of parallelism at the 'implementation level'. Because the procedures can not have side-effects and because we have expression lists, there exists a kind of parallelism in the following sense: let \( p_1 \) and \( p_2 \) be procedures (techniques) and consider the statement

\[
d_1, d_2 := p_1(d_1), p_2(d_2);
\]

Now the execution of \( p_1(d_1) \) can be done simultaneously with (and independently of) the execution of \( p_2(d_2) \).

We could try to devise design-programs dealing with design creation and design evolution, making special 'parallel' versions of the techniques given in Sections 3.4 and 3.5. However, in order to keep matters simple, we shall in this chapter focus only on design-programs \( s \) which satisfy \( \{\text{INV}\} s \{\text{INV}\} \). Such an \( s \) may range over a very large class of implementation techniques. E.g. \( s \) might correspond with optimisations of the current design e.g with respect to performance requirements which are not expressed as black-box descriptions; however, \( s \) can also correspond with an entire re-design.

One of the ideas behind the notion of a design is the locality principle that it should be possible to implement each component in a design without worrying about the implementation of the other components in the design. As one of the results of the formalisation of this idea (Section 2.5.4) we have the so-called bbc-preserving glass-box modifications, abbreviated as \( \text{bbc-gb-mod} \). We write this as a binary predicate on \( pf \land ds \) designs: \( \text{bbc-gb-mod}(d, d') \) means that \( d' \) is a bbc-preserving glass-box modification of \( d \). Because for \( d \) and \( d' \) satisfying \( \text{bbc-gb-mod}(d, d') \) we have \( \text{bbc}(d) \Rightarrow \text{bbc}(d') \) and since one design can contain many components, these modifications offer an opportunity for parallel development. This will be formalised in Section 3.6.2.
Modifications of black-box descriptions can not always be dealt with in a similar way: if the black-box description of one component is modified, this modification may destroy the correctness of several other components. Therefore we must somehow limit the scope of such modifications. For example, if the design \( d \) consists in fact of two unrelated parts, then a black-box modification in one part can not endanger the correctness of components which are in the other part. It follows that there is a possibility for parallel development if the current design \( d \) can be partitioned into \( d_1 \) and \( d_2 \) such that \( d = d_1 \ast d_2 \). This approach will be investigated in Section 3.6.3.

Similarly there is a possibility for parallel development if the current design \( d \) can be partitioned into \( d_1 \) and \( d_2 \) such that \( d = d_1 \circ d_2 \). This approach will be investigated in Section 3.6.4. We briefly mention several approaches based on \( \ast \) in Section 3.6.5.

The application of the techniques which we shall discuss in Sections 3.6.3 and 3.6.4 sometimes needs preparations in the sense that the developer must restructure (transform) the current design. (e.g. using lemmas 3.2.9 and 3.2.16 and remark 3.2.17). In many situations the application of the techniques from Sections 3.6.3 and 3.6.4 is doomed to fail unless suitable preparatory transformations are performed first. The theory of these preparatory transformations probably must be based on the use of algebraic operations on designs such as \( \ast \) and \( \circ \), but may require additional operations. We have not investigated this yet.

### 3.6.2 Splitting into components

In this section we want to formalise the idea that the locality principle of components can be exploited for parallel development. First we need some notation.

**Definition 3.6.1** Let \( v \) be a name and \( d \) a pf \& ds design such that \( v \in \text{coct}(d) \). Then we define \( d[v] := \) 'the unique component in \( d \) with name \( v \').

The technique called 'impl' given below describes the activity of one developer who gets a design \( d \) and the name of a component in this design (\( v \)). If the execution of this technique succeeds, then the developer has performed a bbc-gb-mod and the result is one component. Formally the technique does not exclude the possibility that the developer does nothing but taking an existing old component, but for the partial correctness of the parallel development technique this does not matter. As before, we use the design-development language of appendix B and we have variables of distinct sorts:
3.6. Design Partition

... for pf \& ds designs, c, ... for components and v, ... for names.

\[ \text{impl}_c := \text{technique } d, v \rightarrow c \]
\[ \text{pre } v \in \text{cset}(d) \]
\[ \text{post } \exists d' (d'[v] = c \text{ and bbc-\text{gb-mod}}(d, d') \text{ and } \]
\[ \forall u ((u \in \text{cset}(d) \text{ and } u \neq v) \rightarrow d'[u] = d[u]) \]

There is a very simple splitting to be done before the actual parallelism can start. Two distinct component names must be selected such that the corresponding components are non-prim. This is described by the technique 'split,' given below.

\[ \text{split}_c := \text{technique } d \rightarrow v_1, v_2 \]
\[ \text{pre } \text{true} \]
\[ \text{post } v_1 \neq v_2 \text{ and } \]
\[ v_1 \in \text{cset}(d) \text{ and } v_2 \in \text{cset}(d) \text{ and } \]
\[ v_1 \notin \text{cset}(\text{bot}(d)) \text{ and } v_2 \notin \text{cset}(\text{bot}(d)) \]

As in Section 3.4.4 we assume that there is an operation 'insert' which serves for inserting a component by overwriting an existing component (which has the same name as the component to be inserted). The following technique describes the splitting, the parallel development and the insertion of the results.

\[ \text{pardev}_c := \text{technique } d \]
\[ \text{def } v_1, v_2 := \text{split}_c(d) \]
\[ c_1, c_2 := \text{impl}_c(d, v_1), \text{impl}_c(d, v_2); \]
\[ \text{insert}(\text{insert}(d, c_1), c_2) \]

The partial correctness of this technique with respect to INV is stated in the following theorem.

Theorem 3.6.2 \{INV\} \( d := \text{pardev}_c(d); \{\text{INV}\} \).

Proof. Let initially \( d \) be a pf \& ds design given as

\[ x_1 := \text{prim} \subseteq M_1 \]
\[ \ldots \]
\[ x_n := \text{prim} \subseteq M_n \]
\[ y_1 := P_1 \subseteq Q_1 \]
\[ \ldots \]
\[ y_m := P_m \subseteq Q_m \]

system \([S_1, \ldots, S_l]\)
and assume that initially INV holds. If the execution of \( \text{split}_i(d) \) does not fail then it results two names, \( y_i \) and \( y_j \), say, \( (1 \leq i \leq m, 1 \leq j \leq m \) and \( i \neq j \)). Execution of \( \text{impl}_i(d,y_i) \) results a component \( (y_i := P'_i \subseteq Q_i) \) such that \( [x_1 \subseteq M_1], \ldots, [x_n \subseteq M_n], [y_i \subseteq Q_{i-1}], \ldots, [y_{i-1} \subseteq Q_{i-2}] \vdash P'_i \subseteq Q_i \). Execution of \( \text{impl}_j(d,y_j) \) results a component \( (y_j := P'_j \subseteq Q_j) \) such that \( [x_1 \subseteq M_1], \ldots, [x_n \subseteq M_n], [y_1 \subseteq Q_1], \ldots, [y_{j-1} \subseteq Q_{j-2}] \vdash P'_j \subseteq Q_j \). The insertion of these components yields the design:

\[
\begin{align*}
  x_1 &:= \text{prim} \subseteq M_1 \\
  \cdots \\
  x_n &:= \text{prim} \subseteq M_n \\
  y_1 &:= P_1 \subseteq Q_1 \\
  \cdots \\
  y_i &:= P'_i \subseteq Q_i \\
  \cdots \\
  y_j &:= P'_j \subseteq Q_j \\
  \cdots \\
  y_m &:= P_m \subseteq Q_m \\
  \text{system } [S_1, \ldots, S_i]
\end{align*}
\]

where we assumed that \( i < j \). If \( i > j \) we can have a similar reasoning. The latter design is bbc. Finally we note that the bottom of the latter design equals precisely the bottom of the initial design and that the top of the latter design equals precisely the top of the initial design. This shows that after execution of \( d := \text{pardev}_i(d) \), the condition INV holds. \( \square \)

The technique \( \text{pardev}_i \) can be generalised towards \( n \)-ary splittings. The number of non-prim components is an upperbound to \( n \) and hence also to the degree of parallelism.

### 3.6.3 Splitting according to \(*\)

If we split a design \( d \) into two designs \( d_1 \) and \( d_2 \) such that \( d = d_1 \ast d_2 \) then we also must split the validation of \( d \) into two parts.

Let 'impl' be an arbitrary technique which has the property that it does not violate the invariant INV and let us assume that this technique 'impl' describes the activity of one single developer.

\[
\begin{align*}
  \text{impl} &:= \text{technique } d_m, d_n, d \rightarrow d' \\
  \text{pre} &\quad \text{bbv}(d, d_m) \text{ and } \text{bbv}(d_n, d) \text{ and } \text{bbc}(d) \\
  \text{post} &\quad \text{bbv}(d', d_m) \text{ and } \text{bbv}(d_n, d') \text{ and } \text{bbc}(d')
\end{align*}
\]
Note that this ‘impl’ may modify black-box descriptions; so ‘impl’ certainly is not restricted to just bbc-gp-modes. There is a splitting to be done before the actual parallelism can start. This is described by the technique ‘split,’ given below.

\[
\text{split.} \; := \; \text{technique} \; d \rightarrow d_1, d_2 \\
\text{pre} \; \text{true} \\
\text{post} \; d_1 \ast d_2 = d
\]

Based on the splitting of \( d \) into \( d_1 \) and \( d_2 \) it is possible to indicate which parts of the machine\&user-context are relevant for \( d_1 \) and which parts of the machine\&user-context are relevant for \( d_2 \). Of course in many practical situations it is not the case that the system user and the machine are formalised as designs. Nevertheless, in such situations we probably still have that certain parts of the machine\&user-context are relevant for \( d_1 \) and that other parts of the machine\&user-context are relevant for \( d_2 \).

In view of lemma 3.3.23, the developers need not know \( d_m \) and \( d_s \) completely; it is sufficient if they have access to the relevant parts of \( \text{top}(d_m) \) and \( \text{bot}(d_s) \). In fact \( \text{top}(d_m) \) only needs to be given up to \( \text{top-equivalence} \) (see definition 3.3.28). The splitting of the bottom of the system user \( d_s \) is a matter of counting prim components and system elements whereas the splitting of the top of the machine \( d_m \) might be more complicated. The corresponding techniques are given below. We define \( \text{def}_e(d_1, d_2) \) \( \Leftrightarrow \) “the composition \( d_1 \circ d_2 \) is defined”.

\[
\text{msplit} \; := \; \text{technique} \; d_m, d_1, d_2 \rightarrow d'_m, d''_m \\
\text{pre} \; \text{def}_e(d_1 \ast d_2, d_m) \; \text{and} \; \text{top}(d_m) = d_m \\
\text{post} \; d'_m \ast d''_m = \text{top}(d_m) \; \text{and} \; \text{def}_e(d_1, d'_m) \; \text{and} \; \text{def}_e(d_2, d''_m)
\]

\[
\text{ssplit} \; := \; \text{technique} \; d_s, d_1, d_2 \rightarrow d'_s, d''_s \\
\text{pre} \; \text{def}_e(d_s, d_1 \ast d_2) \; \text{and} \; \text{bot}(d_s) = d_s \\
\text{post} \; d'_s \ast d''_s = d, \; \text{and} \; \text{def}_e(d'_s, d_1) \; \text{and} \; \text{def}_e(d''_s, d_2)
\]

The condition \( \text{top}(d_m) = d_m \) in the precondition of ‘msplit’ says that \( d_m \) must be a top design. The condition \( \text{bot}(d_s) = d_s \) in the precondition of ‘ssplit’ says that \( d_s \) must be a bottom design. Note that if ‘msplit’ is invoked with arguments satisfying its precondition then by lemma 3.3.30 its execution need not fail. Note also that if ‘ssplit’ is invoked with arguments satisfying its precondition then by remark 3.3.32 its execution need not fail.
The following technique describes the splitting, the parallel development and the recombination of the results.

\[
\text{pardev} := \text{technique } d_m, d_s, d \\
\text{def } d_1, d_2 := \text{split}_s(d); \\
\quad d'_m, d'_s := \text{msplit}(\text{top}(d_m), d_1, d_2); \\
\quad d'_s, d''_s := \text{ssplit}(\text{bot}(d_s), d_1, d_2); \\
\quad d_1, d_2 := \text{impl}(d'_m, d'_s, d_1), \text{impl}(d'_m, d''_s, d_2); \\
\quad d_1 \circ d_2
\]

The partial correctness of this technique with respect to INV is stated in the following theorem.

\textbf{Theorem 3.6.3} \{INV\} \( d := \text{pardev}(d_m, d_s, d); \}\{INV\}.

\textbf{Proof.} Use Lemma 3.3.3 (ii), Lemma 3.3.23 and Lemma 3.3.16. \qed

\section{3.6.4 Splitting according to \( \circ \)}

Let again \text{impl} be a technique as given in Section 3.6.3 with the property that it does not violate the invariant INV and let us assume that this technique \text{impl} describes the activity of one single developer.

There is a splitting to be done before the actual parallelism can start. This is described by the technique \text{split}_s given below.

\[
\text{split}_s := \text{technique } d \rightarrow d_1, d_2 \\
\text{pre true} \\
\text{post } d_1 \circ d_2 = d
\]

As before it is possible to indicate which parts of the machine&user-context are relevant for \( d_1 \) and which parts of the machine&user-context are relevant for \( d_2 \). This is relatively simple, due to the fact that we can view \( d_1 \circ d_2 \) as a layered design with layers \( d_1 \) and \( d_2 \). We see directly that \( d_1 \) must be validated with respect to \text{top}(d_2) and the bottom of the system user, i.e. \text{bot}(d_s). Similarly we see directly that \( d_2 \) must be validated with respect to \text{top}(d_m) and \text{bot}(d_s).

It is tempting to propose a technique similar to \text{pardev} given below as a description of the splitting, the parallel development and the recombination of the results.

\[
\text{pardev} := \text{technique } d_m, d_s, d \\
\text{def } d_1, d_2 := \text{split}_s(d);
\]
3.6. DESIGN PARTITION

\[ d'_{m} := \text{top}(d_{4}); \]
\[ d''_{m} := \text{top}(d_{m}); \]
\[ d'_{s} := \text{bot}(d_{s}); \]
\[ d''_{s} := \text{bot}(d_{l}); \]
\[ d_{1}, d_{2} := \text{impl}(d'_{m}, d'_{s}, d_{1}), \text{impl}(d''_{m}, d''_{s}, d_{2}); \]
\[ d_{1} \circ d_{2} \]

Remark 3.6.4 The proposition that we have

\{INV\} \( d := \text{pardev}_{d}(d_{m}, d_{s}, d) \); \{INV\}

just does not hold in general.

The counter-example is as follows. In this counterexample we do not worry about the validation with respect to \( d_{m} \) and \( d_{s} \) but we focus on the interface between \( d_{1} \) and \( d_{2} \). Consider the algebraic system with preorder \( \mathcal{W} \), as before. Assume that the designs \( d_{1} \) and \( d_{2} \) which are obtained by 'split,' respectively are given by

\[ y := \text{prim} \sqsubseteq 4 \]
\[ x := 1 \sqsubseteq 1 \]
\[ \text{system} [], \]
\[ \text{system} [x]. \]

Indeed \( \text{bbv}(d_{1}, d_{2}) \) because \( [x \sqsubseteq 1] \vdash x \sqsubseteq 4 \). In this case \( d_{1} \) happens to be a bottom design and \( d_{2} \) happens to be a top design, so after the execution of the assignments to \( d'_{m}, d''_{m}, d'_{s} \) and \( d''_{s} \) we have \( d'_{m} = d_{2} \) and \( d''_{m} = d_{1} \). Now one execution of 'impl' yields a design which is valid with respect to the design \( d_{2} \) as given above and the other execution of 'impl' yields a design which is valid with respect to the design \( d_{1} \) as given above. Assume that the designs yielded by these two executions respectively are given as

\[ y := \text{prim} \sqsubseteq 2 \]
\[ x := 1 \sqsubseteq 3 \]
\[ \text{system} [], \]
\[ \text{system} [x]. \]

Let us refer to the latter designs as the new value of \( d_{1} \) and the new value of \( d_{2} \) respectively. Note that \( \text{bbv}(\text{new value of } d_{1}, \text{old value of } d_{2}) \) holds because \( [x \sqsubseteq 1] \vdash x \sqsubseteq 2 \) and that \( \text{bbv}(\text{old value of } d_{1}, \text{new value of } d_{2}) \) holds because \( [x \sqsubseteq 3] \vdash x \sqsubseteq 4 \). But \( \text{bbv}(\text{new value of } d_{1}, \text{new value of } d_{2}) \) does not hold.

Therefore the result of \( \text{pardev}_{d} \) is not \( \text{bbc}. \)

We shall now propose a technique \( \text{pardev}^* \), which is an improved version of \( \text{pardev}_{d} \). The idea is that either the bottom of \( d_{1} \) or the top of \( d_{2} \) should remain constant in order to avoid the problems shown in remark 3.6.4. We investigate the solution in which the top of \( d_{1} \) remains constant. In fact it only needs to remain constant up to top-equivalence (see definition 3.3.28).
Therefore we need a slightly modified version of ‘impl’, which we shall call ‘impl+’ which is like ‘impl’ as used above but has the additional property that its output design has the same top as its input design. We obtain the definition of ‘impl+’ from the definition of ‘impl’ by simply adding a conjunct to the postcondition (note that this makes one of the other conjuncts redundant).

\[
\text{impl+} := \text{technique } d_m, d_a, d \rightarrow d' \\
\text{pre } \text{bbv}(d, \text{d}_m) \text{ and } \text{bbv}(d_a, d) \text{ and } \text{bbc}(d) \\
\text{post } \text{bbv}(d', \text{d}_m) \text{ and } \text{bbv}(d_a, d') \text{ and } \text{bbc}(d') \text{ and } \text{top}(d') =_{\text{top}} \text{top}(d)
\]

The technique ‘pardev’ given below describes the splitting, the parallel development and the recombination of the results.

\[
\text{pardev} := \text{technique } d_m, d_a, d \\
\text{def } d_1, d_2 := \text{split}_s(d); \\
\quad d_{m}' := \text{top}(d_2); \\
\quad d_{m} := \text{top}(d_m); \\
\quad d_{a} := \text{bot}(d_1); \\
\quad d_{a}' := \text{bot}(d_1); \\
\quad d_1, d_2 := \text{impl}(d_{m}', d_{a}', d_1), \text{impl+}(d_{m}, d_{a}', d_2); \\
\quad d_1 \circ d_2
\]

The partial correctness of this technique with respect to INV is stated in the following theorem.

**Theorem 3.6.5** \{INV\} \(d := \text{pardev}(d_m, d_a, d); \{\text{INV}\}\).

**Proof.** Use lemma 3.3.21 (iii), (iv) lemma 3.3.23 and lemma 3.3.14 (ii). \(\square\)

Of course there is also a version of this technique based on the solution in which the bottom of \(d_1\) remains constant.

### 3.6.5 Splitting according to \(\|=\)

Splitting according to \(\|=\) for parallel development is not as straightforward as according to \(\ast\) and \(\circ\). We do not give a formalisation, but we sketch a few approaches.

The first approach is to adopt black-box correctness. However we must be careful for the proposition ‘\(d_1, d_2\) are bbc’ does not hold. But of course we could split a given \(d\) into \(d_1\) and \(d_2\) in the sense that \(d_1 \|= d_2\) is defined (and equal to \(d\)) and that both \(d_1\) and \(d_2\) are \(\Gamma_{bb}(d_2)\)-correct. Now at first sight it seems attractive to have two independent developments for
3.7. LOOKING BACK

$d_1$ and $d_2$, preserving their $\Gamma_{bb}(d_2)$-correctness and their $\mathcal{F}$-definedness. But on second thought we see that their interface encompases $d_2$ completely: the black-box descriptions of $d_2$ must be kept fixed because these determine $\Gamma_{bb}(d_2)$ and the glass-box descriptions of $d_2$ must be kept fixed because $\mathcal{F}$-definedness requires $d_1$ and $d_2$ to have that column in common. As it turns out, the technique works in principle, but there are so many constraints on $d_2$ that there is in fact no parallelism at all.

The second approach is to adopt glass-box correctness. Now we can exploit the fact that $d_1, d_2$ are gbc $\Rightarrow$ $d_1 \gg d_2$ is gbc. There are two points worth noting. The first point is that the validation of $d_1 \gg d_2$ with respect to the machine&user-context is determined by $d_2$ alone. The second point is that $d_1$ and $d_2$ must have one column in common. Therefore one could cast that column into the shape of a bottom design which is kept fixed during the parallel development.

The third approach is to employ the generalised designs sketched in Section 3.3.4. This does not give any new approach to parallelism, but there are generalisations of the splitting according to $\ast$, $\diamond$ and $\mathcal{F}$ as discussed above.

3.7 Looking Back

Section 3.2 presents an extension of the $\lambda\pi$-calculus, which is necessary in order to define algebraic operations on designs. The interpretation (in $\lambda\pi$) of each algebraic operation gives rise to an extension of $\lambda\pi$ — each extension requiring the introduction of additional rules.

By defining binary operations ($\ast$, $\circ$) and unary operations (bot, top) on designs, an algebra of designs is obtained. The algebraic laws that hold for these operations are investigated and — as it turns out — certain laws of great simplicity hold (see e.g. remark 3.3.12). Under certain conditions it is possible to apply a binary operation to two bbc designs in a bbc design again. For $\ast$ this condition is simply 'true' and for $\circ$ it is formulated as a binary predicate bbc; we have the following laws:

\[
\begin{align*}
\text{bbc}(d_1) \land \text{bbc}(d_2) & \iff \text{bbc}(d_1 \ast d_2), \\
\text{bbc}(d_1) \land \text{bbc}(d_2) \land \text{bbc}(d_1, d_2) & \iff \text{bbc}(d_1 \circ d_2).
\end{align*}
\]

The transitivity of $\mathcal{F}$ suggests a binary operation $\mathcal{F}$. It has the property that $d_1, d_2$ are bbc $\Rightarrow$ $d_1 \mathcal{F} d_2$ is bbc and the positive result that $d_1, d_2$ are gbc $\Rightarrow$ $d_1 \mathcal{F} d_2$ is gbc. Several generalisations of our theory are discussed — suggested by this $\mathcal{F}$ — which are interesting for a particular kind of stepwise refinement.
To describe models of the software development process, a simple design-development language is introduced, leading us to models of the development process as highly non-deterministic design-programs of an imperative nature.

A postcondition for design creation is formulated and from this postcondition two invariants can be derived in a natural way. The first invariant is investigated in Section 3.4.4 and it leads to the derivation of a design-program which corresponds to the top-down approach. The second invariant is investigated in Section 3.4.5 and it leads to the derivation of a design-program which corresponds to the bottom-up approach. It is remarkable that the top-down and bottom-up models of the development process can be derived in a systematic manner by applying (at the design-program level) an approach which comes from the field of classical sequential programming. The possibilities for deriving invariants and design-programs describing design creation by this approach have by no means been exhausted in Section 3.4. It certainly is interesting to investigate other possibilities.

As it turns out the formulation of the design-programs of Section 3.4 requires several ad-hoc operations on designs; we think this is mainly due to the fact that both top-down and bottom-up development processes take place within the scope of one variable design and that therefore one must be very explicit about names, contexts, components, etc. In remarks 3.4.9 (ii) and 3.4.14 (ii) an interesting difference between the top-down approach and the bottom-up approach is revealed: the top down approach always yields a black-box correct design (even if the developer does not know the difference between black-box correctness and glass-box correctness) but there exist two variants of the bottom-up approach where the first variant yields a black-box correct design and the second variant yields a glass-box correct design.

Using the algebra of designs, we discuss the validation of a given design with respect to a given machine/user-context and the related problems of design evolution. In Section 3.5 we discuss some simple models of the development process which deal with design evolution.

The fact that it is possible to split designs and reassemble them again, gives rise to models of the development process where two (or more) developers each operate on a part of a design such that when each of them has finished his part, their results are fitted together to yield a new design which is both bbc and valid.

Most of the basic ideas behind the models of the development process which are given in Sections 4, 5 and 6 are not really new, but their formalisation is very useful because it makes it possible to reason about the development process in a precise way. The relatively subtle (but important) points addressed in remark 3.2.17, remark 3.3.35, remark 3.6.4 and remarks 3.4.9
(ii) and 3.4.14 (ii) demonstrate that one needs to be careful indeed when discussing the development process.

It should be noted that most sections in this chapter are open-ended. Appendix A and Section 3 are open-ended because one can always imagine new useful operations (both in terms of λσ and in terms of designs). Appendix B is open-ended because one might extend the design-development language with almost any language construct from classical specification languages and classical programming languages. Sections 4, 5 and 6 are open-ended because one can always devise other design-programs; in fact we only investigate some of the simplest design-programs.

It is important that the concepts investigated in this chapter are applied in one or more case studies. The language COLD-K can be employed for such case studies. This is possible because COLD-K is based on an algebraic system (viz. the algebra CA of class descriptions) and has λσ for parameterisation and because it has components and designs as built-in language constructs [8]. For a very first example using a technique from this chapter we refer to [9]. A large example will be presented in Chapters 4 and 5.
Bibliography

Appendix A

$\lambda\pi$-Calculus with sequences

In order to define algebraic operations on designs it is useful to use sequences within $\lambda\pi$. For that purpose we shall make an extension of the calculus $\lambda\pi$ in this appendix. We shall not give the complete definition of the resulting calculus, but rather describe the modifications with respect to the calculus given in Chapter 2 in an incremental manner.

As a first step we shall adapt the definition of $\lambda\pi$-calculus for an algebraic system with preorder $\mathbb{R}$ by adding sequences to the language of $\mathbb{R}$. We shall denote sequences by using square brackets; e.g. the sequence with two elements $P$ and $Q$ is denoted by $[P, Q]$. We shall have projection functions $\pi_i$ for $i = 1, 2, \ldots$ We shall avoid the problems that arise when a projection function $\pi_i$ can be applied to a sequence whose length is less than $i$ by adapting the type system. Furthermore we shall have a binary operation $*$ for concatenation. The laws that describe the operations on sequences will be given as rules of $\lambda\pi$.

We do not define these constructions in $\lambda\pi$ as one would do in classical lambda calculus. Some of the techniques one uses in classical lambda calculus for defining 'pairing' are based on having $\eta$-reduction and it is not clear (yet) what $\eta$ reduction in $\lambda\pi$-calculus should look like.

Definition A.1.1 (i) Add to the inductive definition of the set of type symbols:

- if $\sigma_1, \ldots, \sigma_m$ are type symbols $(m \geq 0)$, then also is $(\sigma_1, \ldots, \sigma_m)$.

(ii) Add to the definition of the alphabet:

- Symbols for operations on sequences: $[\cdot, \cdot]$, $\pi_i$ (for $i = 1, 2, \ldots$), $\ast$. 
(iii) Add to the inductive definition of the set $\Lambda_R$ of lambda terms for $R$, and their type:

- (i) if $P_1 \in \Lambda_R, \ldots, P_n \in \Lambda_R$ for $n \geq 0$ with types $\sigma_1, \ldots, \sigma_n$ then $[P_1, \ldots, P_n] \in \Lambda_R$ with type $[\sigma_1, \ldots, \sigma_n]$.

(ii) if $P \in \Lambda_R$ with type $[\sigma_1, \ldots, \sigma_n]$ and $1 \leq i \leq n$ then $\pi_i(P) \in \Lambda_R$ with type $\sigma_i$.

Definition A.1.2 (length). If a lambda term $P$ has type $[\sigma_1, \ldots, \sigma_n]$ then the length of $P$ is defined as $n$.

Definition A.1.3 (Rules for sequences). We adopt the rules of $\lambda\pi$ ($|=_1$, $|=_2$, context, refl., trans., $\lambda I_1$, $\lambda I_2$, ap., $\pi = I$, subst.) and we add the following rules:

\[(\pi_1) \quad \Gamma \vdash \pi_i([P_1, \ldots, P_n]) = P_i\]

\[(\pi_2) \quad \Gamma \vdash [P_1, \ldots, P_n] \ast [Q_1, \ldots, Q_m] = [P_1, \ldots, P_n, Q_1, \ldots, Q_m]\]

\[(\sqsubseteq \text{-seq}) \quad \Gamma \vdash P_i \sqsubseteq Q_1, \ldots, \Gamma \vdash P_n \sqsubseteq Q_n\]

We could add a rule for surjectivity of sequence construction, but since we do not need it we shall not add it. From now on we write $\lambda\pi$ to denote the extended version of the calculus.

Lemma A.1.4 (Properties of $\ast$). Let $P \equiv [P_1, \ldots, P_m]$, $Q \equiv [Q_1, \ldots, Q_n]$ and $R \equiv [R_1, \ldots, R_p]$.

(i) $\vdash (P \ast Q) \ast R = P \ast (Q \ast R)$,

(ii) $\vdash [] \ast P = P \ast [] = P$.

Proof. (i) We use rule $(\pi_2)$.

\[\vdash (P \ast Q) \ast R \equiv ([P_1, \ldots, P_m] \ast [Q_1, \ldots, Q_n]) \ast [R_1, \ldots, R_p]\]

\[\equiv [P_1, \ldots, P_m, Q_1, \ldots, Q_n] \ast [R_1, \ldots, R_p]\]

\[\equiv [P_1, \ldots, P_m, Q_1, \ldots, Q_n, R_1, \ldots, R_p]\]

\[\equiv [P_1, \ldots, P_m] \ast [Q_1, \ldots, Q_n, R_1, \ldots, R_p]\]
\( \begin{array}{c}
\vdash [P_1, \ldots, P_m] * ([Q_1, \ldots, Q_n] * [R_1, \ldots, R_n]) \\
\equiv P * (Q * R).
\end{array} \)

\((i) \vdash P = [P_1, \ldots, P_m] \equiv [P_1, \ldots, P_m] \equiv P \) and finally \( \vdash P = [P_1, \ldots, P_m] \equiv [P_1, \ldots, P_m] \equiv P. \)

**Remark A.1.5** It is tempting to think that every term \( P \) of type \((\sigma_1, \ldots, \sigma_m)\) can be written as \([P_1, \ldots, P_m]\) for suitable \( P_1, \ldots, P_m \) such that \( \vdash P = [P_1, \ldots, P_m]. \) This is not the case however. As an example consider the algebraic system with preorder \( \mathbb{R}_1 = (\mathbb{N}, \leq, (+), (0,1,2,\ldots)) \) and \( P := (\lambda x \leq 1, 0, 0)2. \) An even simpler example is the term \( x^n \) for \( \sigma = (\sigma_1, \ldots, \sigma_m). \)

**Remark A.1.6** Instead of sequence construction using \( | \) and \( | \) we could have chosen constructors \( | \) and cons. In that case we might have chosen projection functions 'hd' and 'tl'.

We can define the relations \( \rightarrow, \rightarrow \) and \( =_s \) in an obvious way by viewing the rules \((\pi_1, \pi_i, \ast_1)\) as basic reduction steps.

**Definition A.1.7** (Reduction).

The relation \( \rightarrow \) is defined inductively by:

1. \( \Gamma \vdash R \subseteq A \Rightarrow \Gamma \vdash (\lambda x \subseteq A.B)R \rightarrow B[x := R], \)
2. \( \Gamma \vdash \pi_i([P_1, \ldots, P_n]) \rightarrow P_i \quad (1 \leq i \leq n), \)
3. \( \Gamma \vdash [P_1, \ldots, P_n] * [Q_1, \ldots, Q_m] \rightarrow [P_1, \ldots, P_n, Q_1, \ldots, Q_m], \)
4. \( \Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash f_j(\ldots, M, \ldots) \rightarrow f_j(\ldots, N, \ldots), \)
5. \( \Gamma \vdash P_i \rightarrow P'_i \Rightarrow \Gamma \vdash [P_1, \ldots, P_i, \ldots, P_n] \rightarrow [P_1, \ldots, P'_i, \ldots, P_n] \quad (1 \leq i \leq n), \)
6. \( \Gamma \vdash P \rightarrow P' \Rightarrow \Gamma \vdash P * Q \rightarrow P'* Q, \)
7. \( \Gamma \vdash Q \rightarrow Q' \Rightarrow \Gamma \vdash P * Q \rightarrow P * Q', \)
8. \( \Gamma \vdash P \rightarrow P' \Rightarrow \Gamma \vdash \pi_i(P) \rightarrow \pi_i(P'), \)
9. \( \Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash ZM \rightarrow ZN, \)
10. \( \Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash MZ \rightarrow NZ, \)
11. \( \Gamma \vdash P \rightarrow Q \Rightarrow \Gamma \vdash (\lambda x \subseteq P.M) \rightarrow (\lambda x \subseteq Q.M), \)
12. \( \Gamma, [x \subseteq P] \vdash M \rightarrow N, x \notin \Gamma \Rightarrow \Gamma \vdash (\lambda x \subseteq P.M) \rightarrow (\lambda x \subseteq P.N). \)

The relation \( \rightarrow \rightarrow \) is defined inductively by:

1. \( \Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash M \rightarrow N, \)
2. $\Gamma \vdash M \rightarrow M$,
3. $\Gamma \vdash M \rightarrow N, \Gamma \vdash N \rightarrow L \Rightarrow \Gamma \vdash M \rightarrow L$.

The relation $=_\pi$ is defined inductively by:

1. $\Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash M =_\pi N$, provided $M \rightarrow N$ only by $\pi$-reductions, i.e. excluding the clauses 2. and 3. from the definition of $\rightarrow$,
2. $\Gamma \vdash M =_\pi N \Rightarrow \Gamma \vdash N =_\pi M$,
3. $\Gamma \vdash M =_\pi N, \Gamma \vdash N =_\pi L \Rightarrow \Gamma \vdash M =_\pi L$.

And of course we have the following lemma, justifying $\rightarrow, \rightarrow^*$ and $=_\pi$.

**Lemma A.1.8**

(i) $\Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash M = N$,
(ii) $\Gamma \vdash M \rightarrow N \Rightarrow \Gamma \vdash M = N$,
(iii) $\Gamma \vdash M =_\pi N \Rightarrow \Gamma \vdash M = N$.

**Proof.**

(i) By induction over the definition of $\rightarrow$.
(ii) By induction over the definition of $\rightarrow^*$.
(iii) By induction over the definition of $=_\pi$.

This enriched $\lambda\pi$-calculus has reasonable properties: In $\lambda\pi$ with the extensions of definitions A.1.1 and A.1.3 every term strongly normalises (SN). This can be shown by adapting the computability argument of Chapter 2. Furthermore in $\lambda\pi$-calculus with the extensions of definitions A.1.1 and A.1.3, $\rightarrow^*$ satisfies the diamond property i.e. $\Gamma \vdash M \rightarrow M_1, M \rightarrow M_2 \Rightarrow \exists M_3 : (\Gamma \vdash M_1 \rightarrow M_3, M_2 \rightarrow M_3)$. This can be shown by adapting the proof of the weak diamond property and using Newman's lemma again, just as was done for $\lambda\pi$ in Chapter 2.

In order to have an interpretation for the algebraic operations on designs, we extend $\lambda\pi$ once more by introducing two operations on closed terms, viz. $\ast$ and $\circ$. One can view $\ast$ as a generalisation of the concatenation of sequences ($\ast$). One can view $M \circ N$ as functional composition, where an automatic un-Currying takes place. We shall not give the complete definition of the resulting calculus, but rather describe the extensions with respect to the calculus described so far in an incremental manner. We shall use the conventions that $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau$ abbreviates $\sigma_1 \rightarrow (\ldots (\sigma_n \rightarrow \tau) \ldots)$ and that $\lambda x_1 \leq \bar{P}_1 T$ is shorthand for $\lambda x_1 \leq P_1 \cdot \ldots \lambda x_n \leq P_n T$.
Definition A.1.9 ($\Lambda_\infty$). (i) Add to the definition of the alphabet a clause:

- A binary operation symbol: $o$.

(ii) Add to the inductive definition of the set of lambda terms and their types:

- if $P \in \Lambda_\infty$ with type $\sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow (\xi_1, \ldots, \xi_t)$ and $Q \in \Lambda_\infty$ with type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow (\eta_1, \ldots, \eta_t)$ then $P * Q \in \Lambda_\infty$ with type $\sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow (\xi_1, \ldots, \xi_t, \eta_1, \ldots, \eta_t)$,

- if $P \in \Lambda_\infty$ with type $\sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow (\xi_1, \ldots, \xi_t)$ and $Q \in \Lambda_\infty$ with type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow (\sigma_1, \ldots, \sigma_m)$ then $P \circ Q \in \Lambda_\infty$ with type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow (\xi_1, \ldots, \xi_t)$.

Definition A.1.10 (rules for $\ast, \circ$). We adopt the rules of $\lambda \pi$ ([m_1, \equiv_2, context, refl., trans., $\lambda I_1$, $\lambda I_2$, ap., $\pi = I$, subst., $\pi_1, \ast_1, \sqsubseteq$-seq) and we add the rules:

1. $\frac{\lambda \bar{x} \equiv \bar{P} \cdot S \ast \lambda \bar{y} \equiv \bar{Q} \cdot T = \lambda \bar{z} \equiv \bar{P} \cdot \bar{Q} \cdot S \ast T}{(S, T \text{ not of a type } \sigma \rightarrow \tau)}$

2. $\frac{\lambda \bar{x} \equiv \bar{B} \cdot N = \lambda \bar{z} \equiv \bar{B} \circ (M \circ N)}{(\bar{z} \neq M, N \text{ not of type } \sigma \rightarrow \tau)}$

3. $\frac{(\lambda \bar{x} \equiv \bar{M} \cdot N) \circ [\bar{L}] = (\lambda \bar{x} \equiv \bar{M} \cdot N)[\bar{L}]}{\text{where it is understood that the conclusion of rule } o_2 \text{ is an abbreviation of the following:}}$

   $$(\lambda x_1 \equiv M_1, \ldots, \lambda x_m \equiv M_m \cdot N) \circ [L_1, \ldots, L_m] = (\lambda x_1 \equiv M_1, \ldots, (\lambda x_m \equiv M_m \cdot N)L_m \ldots) L_1.$$ 

Since we have extended the set of terms and added rules, we must reconsider the properties of $\ast$ and $\circ$ again:

Lemma A.1.11 (Properties of $\ast$).
Let $P = \lambda \bar{x} \equiv \bar{P} \cdot [P_1, \ldots, P_m]$, $Q = \lambda \bar{y} \equiv \bar{Q} \cdot [Q_1, \ldots, Q_n]$ and $R = \lambda \bar{z} \equiv \bar{R} \cdot [R_1, \ldots, R_p]$.

1. $\vdash (P \ast Q) \ast R = P \ast (Q \ast R)$,
2. $\vdash [] \ast R = P \ast [] = P$.

Proof. (i) We use the rules $(s_1)$ and $(s_2)$.

$$\vdash (P \ast Q) \ast R \equiv ((\lambda \bar{x} \equiv \bar{P} \cdot [P_1, \ldots, P_m]) \ast (\lambda \bar{y} \equiv \bar{Q} \cdot [Q_1, \ldots, Q_n])) \ast R$$
\[\vdash (\lambda \vec{x}\vec{y} \subseteq \vec{P}\vec{R} \cdot \{[P_1, \ldots, P_m] \cdot [Q_1, \ldots, Q_n]\}) \ast R\]
\[\vdash (\lambda \vec{x}\vec{y} \subseteq \vec{P}\vec{Q} \cdot \{[P_1, \ldots, P_m, Q_1, \ldots, Q_n]\}) \ast R\]
\[\vdash \lambda \vec{x}\vec{y} \subseteq \vec{P}\vec{Q} \cdot \vec{R} \cdot \{[P_1, \ldots, P_m, Q_1, \ldots, Q_n] \ast [R_1, \ldots, R_p]\}\]
\[\vdash \lambda \vec{x}\vec{y} \subseteq \vec{P}\vec{Q} \vec{R} \cdot \{[P_1, \ldots, P_m, Q_1, \ldots, Q_n, R_1, \ldots, R_p]\}\]
\[\vdash \lambda \vec{x}\vec{y} \subseteq \vec{P}\vec{Q} \vec{R} \cdot \{[P_1, \ldots, P_m] \ast [Q_1, \ldots, Q_n, R_1, \ldots, R_p]\}\]
\[\vdash (\lambda \vec{x} \subseteq \vec{P} \cdot [P_1, \ldots, P_m]) \ast \lambda \vec{x} \subseteq \vec{Q} \vec{R} \cdot [Q_1, \ldots, Q_n, R_1, \ldots, R_p]\]
\[\vdash P \ast \lambda \vec{x} \subseteq \vec{Q} \vec{R} \cdot ([Q_1, \ldots, Q_n] \ast [R_1, \ldots, R_p]\}
\[\vdash P \ast ([\lambda \vec{y} \subseteq \vec{Q} \cdot [Q_1, \ldots, Q_n]) \ast \lambda \vec{x} \subseteq \vec{R} \cdot [R_1, \ldots, R_p])\]
\[\equiv P \ast (Q \ast R).\]

(ii) First we prove \[\vdash [] \ast P = P,\]
\[\vdash [] \ast P \equiv [] \ast \lambda \vec{x} \subseteq \vec{P} \cdot [P_1, \ldots, P_m]\]
\[\vdash \lambda \vec{x} \subseteq \vec{P} \cdot ([] \ast [P_1, \ldots, P_m])\]
\[\vdash \lambda \vec{x} \subseteq \vec{P} \cdot [P_1, \ldots, P_m] \equiv P.\]

Finally we prove \[\vdash P \ast [] = P.\]

\[\vdash P \ast [] \equiv (\lambda \vec{x} \subseteq \vec{P} \cdot [P_1, \ldots, P_m]) \ast []\]
\[\vdash \lambda \vec{x} \subseteq \vec{P} \cdot ([P_1, \ldots, P_m] \ast [])\]
\[\vdash \lambda \vec{x} \subseteq \vec{P} \cdot [P_1, \ldots, P_m] \equiv P.\]

Lemma A.1.12 (Properties of \(\circ\)). The following properties hold only under certain conditions. A sufficient condition is that both the left-hand side and the right-hand side of the stated equation are fully reducible (by which we mean that all candidate-redexes are redexes). We consider closed terms \(P\), \(Q\), and \(R\). Let \(P \equiv \lambda x_1 \subseteq P_1, \ldots, \lambda x_m \subseteq P_m, [A_1, \ldots, A_j] \equiv \lambda \vec{x} \subseteq \vec{P} \cdot [\vec{A}]\). Let \(Q \equiv \lambda y_1 \subseteq Q_1, \ldots, \lambda y_n \subseteq Q_n, [B_1, \ldots, B_m] \equiv \lambda \vec{y} \subseteq \vec{Q} \cdot [\vec{B}]\). Let \(R \equiv \lambda z_1 \subseteq R_1, \ldots, \lambda z_p \subseteq R_p, [\vec{C}] \equiv \lambda \vec{z} \subseteq \vec{R} \cdot [\vec{C}]\).

(i) \[\vdash (P \circ Q) \circ R = P \circ (Q \circ R),\]

(ii) \[\vdash [] \circ P = P \circ ([]).\]

Proof. (i) \(M[\vec{x} := \vec{B}]\) abbreviates \(M[x_m := B_m] \cdots [x_1 := B_1]\). We freely use \(\pi\)-equality.

\[\vdash (P \circ Q) \circ R \equiv ((\lambda \vec{x} \subseteq \vec{P} \cdot [\vec{A}] \circ \lambda \vec{y} \subseteq \vec{Q} \cdot [\vec{B}]) \circ R)\]
\[ (\lambda \bar{y} \in \bar{G}.((\lambda \bar{x} \in \bar{F}.[\bar{A}]) \circ [\bar{B}])) \circ R \]
\[ (\lambda \bar{y} \in \bar{G}.(\lambda \bar{x} \in \bar{F}.[\bar{A}][\bar{B}]) \circ R \]
\[ (\lambda \bar{y} \in \bar{G}.((\lambda \bar{x} \in \bar{F}.[\bar{A}][\bar{B}]) \circ \lambda \bar{x} \subseteq \bar{R}.[\bar{C}]) \]
\[ \lambda \bar{x} \subseteq \bar{R}.((\lambda \bar{y} \in \bar{G}.(\lambda \bar{x} \in \bar{F}.[\bar{A}][\bar{B}]) \circ \lambda \bar{x} \subseteq \bar{R}.[\bar{C}])) \]
\[ \lambda \bar{x} \subseteq \bar{R}.((\lambda \bar{y} \in \bar{G}.(\lambda \bar{x} \in \bar{F}.[\bar{A}][\bar{B}]) \circ \lambda \bar{x} \subseteq \bar{R}.[\bar{C}]) \subseteq \bar{R}.((\lambda \bar{y} \in \bar{G}.(\lambda \bar{x} \in \bar{F}.[\bar{A}][\bar{B}]) \circ \lambda \bar{x} \subseteq \bar{R}.[\bar{C}])) \]
\[ \lambda \bar{x} \subseteq \bar{R}.((\lambda \bar{y} \in \bar{G}.(\lambda \bar{x} \in \bar{F}.[\bar{A}][\bar{B}]) \circ \lambda \bar{x} \subseteq \bar{R}.[\bar{C}]) \subseteq \bar{R}.((\lambda \bar{y} \in \bar{G}.(\lambda \bar{x} \in \bar{F}.[\bar{A}][\bar{B}]) \circ \lambda \bar{x} \subseteq \bar{R}.[\bar{C}])) \]

(ii) \( \vdash P \circ [\bar{A}] = P \) is nothing but rule (c2). For the converse, note that \( P \) must be of type \( r_1 \rightarrow \ldots r_m \rightarrow \). Since \( P \) is closed and fully reducible, it must be \( \pi \) convertible to a term of the form \( \lambda \bar{x} \subseteq \bar{F}.[\bar{A}] \). Therefore \( \vdash [\bar{A}] \circ P = [\bar{A}] \circ \lambda \bar{x} \subseteq \bar{F}.[\bar{A}] \). Therefore \( \vdash [\bar{A}] \circ P = [\bar{A}] \circ \lambda \bar{x} \subseteq \bar{F}.[\bar{A}] = P. \)

\( \square \)
Appendix B

Design-development language

In this appendix we shall define a simple language which can be employed for expressing certain models of the software development process. It is particularly tuned to the setting of our study of growing, combining and modifying designs. The models of the development process we aim at, are abstract in the sense that they leave room for creative freedom by a human developer.

Essentially the design-development language provides for procedures with input and output parameters. The procedures can be defined by either imperative programming-language constructs (assignment, while, sequential composition, etc.) or by first-order predicate logic constructs (not, and, for all, etc.). This language is kept small and simple. Its semantics is relational, i.e. the meaning of a procedure is just a relation on input-domain X output-domain.

The motivation for these choices is as follows. The imperative style is chosen because we view it as natural and intuitive to support a notion like ‘current design’. The logical constructs and the relational semantics are required by the abstractness of our models. The imperative constructs are required because sometimes we want to be very specific in our models - like in ‘first do this, then do that’, or in case of repetitive models. The procedures themselves serve as a simple modularisation mechanism for our models. At the end of this appendix we shall discuss some more aspects of the use of this language. There we shall also show a few examples.

We assume the following sets

Var: variables with typical elements x, y, ...
Pred: n-ary predicate symbols with typical elements P_n, ...
Op: n-ary operation symbols (n \in \mathbb{N}) with typical elements O_n, ...
We shall define the following sets

\[ \text{Dexp: deterministic expressions with typical elements } e, \ldots \]
\[ \text{Asn: assertions with typical elements } A, \ldots \]
\[ \text{Elist}_k (k \in \mathbb{N}^+): \text{expressions lists of length } k \text{ with typical elements } l_k, \ldots \]
\[ \text{Stat: statements with typical elements } s, \ldots \]
\[ \text{Proc}_{n,m} (n \in \mathbb{N}, m \in \mathbb{N}^+): \text{procedures with typical elements } p_{n,m}, \ldots \]

We shall define mappings yielding the sets of ‘free variables’ (denoted as \( FV \)) and the so-called ‘assign-set’ (denoted as \( AS \)). The assign-set of a statement will contain those variables for which it follows on syntactical grounds that assignments are made to them.

\[ FV : \text{Dexp} \rightarrow \mathcal{P}(\text{Var}) \]
\[ FV : \text{Asn} \rightarrow \mathcal{P}(\text{Var}) \]
\[ FV : \text{Elist}_k \rightarrow \mathcal{P}(\text{Var}) \]
\[ FV : \text{Stat} \rightarrow \mathcal{P}(\text{Var}) \]
\[ AS : \text{Stat} \rightarrow \mathcal{P}(\text{Var}) \]

We give the definitions now, where it is understood that \( \text{Dexp}, \text{Asn}, \text{Elist}_k, \text{Stat} \) and \( \text{Proc}_{n,m} \) are the smallest sets which are closed under the syntax rules given below, under the indicated restrictions related to \( FV \) and \( AS \) and where it is also understood that the mappings \( FV \) and the mapping \( AS \) are simultaneously inductively defined.

\[
\begin{align*}
   \text{e} & ::= \varepsilon \\
   & \mid O_n(e_1, \ldots, e_n) \\

   A & ::= P_n(e_1, \ldots, e_n) \\
   & \mid \text{true} \\
   & \mid \text{not } A' \\
   & \mid (A' \text{ or } A'') \\
   & \mid (A' \text{ and } A'') \\
   & \mid (A' \rightarrow A'') \\
   & \mid (A' \leftrightarrow A'') \\
   & \exists x (A') \\
   & \forall x (A') \\

   l_k & ::= \varepsilon \quad (k = 1) \\
   & \mid O_n(l'_n) \quad (k = 1) \\
   & \mid l'_n, l''_m \quad (k = n + m) \\
   & \mid p_{n,m}(l'_n) \quad (k = m)
\end{align*}
\]
\textit{s ::=} \quad (\text{i.e. empty})
\begin{align*}
| & x_1, \ldots, x_k := l_i; \quad (x_1, \ldots, x_k \text{ distinct}) \\
| & s' ; s'' \\
| & s' \parallel s'' \\
| & \text{while } A \text{ do } s' \text{ od;}
\end{align*}

\begin{align*}
\text{proc } x_1, \ldots, x_n \rightarrow y_1, \ldots, y_m \text{ axiom } A \\
& (\text{FV}(A) \subseteq \{\overline{x}\} \cup \{\overline{y}\}, \ x_1, \ldots, x_n, \ y_1, \ldots, y_m \text{ distinct})
\end{align*}

\begin{align*}
| & \text{proc } x_1, \ldots, x_n \text{ def } s \ l_k \\
& (k = m, \ \text{FV}(l_k) \subseteq \{\overline{x}\} \cup \text{AS}(s), \ \text{FV}(s) \subseteq \{\overline{x}\}, \ x_1, \ldots, x_n \text{ distinct})
\end{align*}

\text{proc } x_1, \ldots, x_n \rightarrow y_1, \ldots, y_m \text{ pre } A \text{ post } A' \text{ abbreviates proc } x_1, \ldots, x_n \rightarrow y_1, \ldots, y_m \text{ axiom } (A \text{ and } A') \text{ provided FV}(A) \subseteq \{\overline{x}\}, \ \text{FV}(A') \subseteq \{\overline{x}\} \cup \{\overline{y}\}, \ x_1, \ldots, x_n, \ y_1, \ldots, y_m \text{ distinct.}

\text{FV : } \text{Dexp} \rightarrow \mathcal{P}(\text{Var}) \text{ is defined by}
\begin{align*}
& \text{FV}(x) = \{x\} \\
& \text{FV}(O_n(\overline{x})) = \bigcup_{i=1,\ldots,n} \text{FV}(a_i)
\end{align*}

\text{FV : } \text{Ass} \rightarrow \mathcal{P}(\text{Var}) \text{ is defined by}
\begin{align*}
& \text{FV}(P_n(\overline{x})) = \bigcup_{i=1,\ldots,n} \text{FV}(a_i) \\
& \text{FV}(\text{true}) = \emptyset \\
& \text{FV}(\text{not } A) = \text{FV}(A) \\
& \text{FV}((\text{A or } A')) = \text{FV}(A) \cup \text{FV}(A') \\
& \text{FV}((\text{A and } A')) = \text{FV}(A) \cup \text{FV}(A') \\
& \text{FV}((A \rightarrow A')) = \text{FV}(A) \cup \text{FV}(A') \\
& \text{FV}((A \leftrightarrow A')) = \text{FV}(A) \cup \text{FV}(A') \\
& \text{FV}(\text{exists } x (A)) = \text{FV}(A) \setminus \{x\} \\
& \text{FV}(\text{forall } x (A)) = \text{FV}(A) \setminus \{x\}
\end{align*}

\text{FV : } \text{Elist}_k \rightarrow \mathcal{P}(\text{Var}) \text{ is defined by}
\begin{align*}
& \text{FV}(x) = \{x\} \\
& \text{FV}(O_n(l_n)) = \text{FV}(l_n) \\
& \text{FV}(l_n, l_m') = \text{FV}(l_n) \cup \text{FV}(l_m') \\
& \text{FV}(p_{n,m}(l_n)) = \text{FV}(l_n)
\end{align*}

\text{FV : } \text{Stat} \rightarrow \mathcal{P}(\text{Var}) \text{ is defined by}
**B. DESIGN-DEVELOPMENT LANGUAGE**

\[
\begin{align*}
FV(\ ) &= \emptyset \\
FV(x := t) &= FV(t) \\
FV(s \cdot s') &= FV(s) \cup (FV(s') - AS(s)) \\
FV(s \mid s') &= FV(s) \cup FV(s') \\
FV(\text{while } A \text{ do } s \text{ od}) &= FV(A) \cup FV(s)
\end{align*}
\]

\[
\text{AS} : \text{Stat} \to \mathcal{P}(\text{Var}) \text{ is defined by}
\]

\[
\begin{align*}
\text{AS}(\ ) &= \emptyset \\
\text{AS}(x := t) &= \{x\} \\
\text{AS}(s \cdot s') &= \text{AS}(s) \cup \text{AS}(s') \\
\text{AS}(s \mid s') &= \text{AS}(s) \cap \text{AS}(s') \\
\text{AS}(\text{while } A \text{ do } s \text{ od}) &= \emptyset
\end{align*}
\]

We assume a set \(D\) (the domain of the built-in data type) with typical elements \(d_1, \ldots\) and we assume meaning functions

\[
\begin{align*}
\llbracket\cdot\rrbracket & : \text{Pred}_n \to \mathcal{P}(D^n) \\
\llbracket\cdot\rrbracket & : \text{Op}_n \to D^n \to D
\end{align*}
\]

The set \(\Sigma\) of states, with typical elements \(\sigma_1, \ldots\) is defined by

\[
\Sigma = \text{Var} \to D
\]

We shall define the following meaning functions

\[
\begin{align*}
\llbracket\cdot\rrbracket : \text{Dexp} & \to \Sigma \to D \\
\llbracket\cdot\rrbracket : \text{Asn} & \to \mathcal{P}(\Sigma) \\
\llbracket\cdot\rrbracket : \text{Ellst}_n & \to \Sigma \to \mathcal{P}(D^n) \\
\llbracket\cdot\rrbracket : \text{Stat} & \to \Sigma \to \mathcal{P}(\Sigma) \\
\llbracket\cdot\rrbracket : \text{Proc}_{n,m} & \to \mathcal{P}(D^{n+m})
\end{align*}
\]

We shall identify \(D \times (D \times D)\) with \((D \times D) \times D\). We write \(\sigma(d/x)\) for the modified state \(\lambda y \cdot (\text{if } y = x \text{ then } d \text{ else } \sigma(y))\).

The meaning of a deterministic expression yields for a given state the result value. The meaning of an assertion is the set of states in which it holds. The meaning of an expression list yields for a given state the set of possible result sequences. A statement is viewed as a non-deterministic state transformer. A procedure is viewed as a relation between an input sequence and a result sequence.

\[
\begin{align*}
\llbracket\cdot\rrbracket : \text{Dexp} & \to \Sigma \to D \text{ is defined by} \\
\llbracket x \rrbracket(\sigma) &= \sigma(x)
\end{align*}
\]
\[ [O_n(x)](\sigma) = [O_n([x]([\sigma]))] \]

\[ [\ ] : Asn \rightarrow \mathcal{P}(\Sigma) \text{ is defined by} \]

\[ [P_n(x)] = \{ \sigma \mid \sigma \in [P_n] \} \]

\[ \text{true} = \Sigma \]

\[ \text{not } A = \Sigma \setminus [A] \]

\[ [(A \text{ or } A')] = [A] \cup [A'] \]

\[ [(A \text{ and } A')] = [A] \cap [A'] \]

\[ [(A \rightarrow A')] = \{ \sigma \mid \sigma \in \Sigma \setminus [A] \cup \{ \sigma' \in [A'] \} \}

\[ [(A \leftrightarrow A')] = \{ \sigma \mid \sigma \in [A] \leftrightarrow \sigma \in [A'] \} \]

\[ [\text{exists } x (A)] = \{ \sigma \mid \exists \alpha \cdot \sigma(d/x) \in [A] \} \]

\[ [\forall x (A)] = \{ \sigma \mid \forall \alpha \cdot \sigma(d/x) \in [A] \} \]

\[ [\ ] : Elist_k \rightarrow \Sigma \rightarrow \mathcal{P}(D^k) \text{ is defined by} \]

\[ [x](\sigma) = \{ \sigma(x) \} \]

\[ [O_n(t)](\sigma) = \{ [O_n](d) \mid d \in [t](\sigma) \} \]

\[ [\text{l}, \text{l}'](\sigma) = \{ \alpha \cdot \beta \cdot \gamma \mid \alpha \cdot \beta \cdot \gamma \in [l](\sigma) \land \exists \gamma' \cdot \gamma' \in [l'](\sigma) \} \]

\[ [\text{proc } \text{def } s \text{ t}](\sigma) = \{ \alpha \cdot \beta \cdot \gamma \mid \alpha \cdot \beta \cdot \gamma \in [s](\sigma) \land \exists \gamma' \cdot \gamma' \in [\text{t}](\sigma) \} \]

\[ [\ ] : \text{Stat} \rightarrow \Sigma \rightarrow \mathcal{P}(\Sigma) \text{ is defined by} \]

\[ [\ ](\sigma) = \{ \sigma \} \]

\[ [x := \text{l}; \ldots](\sigma) = \{ \alpha \cdot \beta \cdot \gamma \mid \alpha \cdot \beta \cdot \gamma \in [l](\sigma) \} \]

\[ [s \cdot x](\sigma) = \{ \sigma'' \mid \exists \sigma'. \sigma' \cdot \sigma'' \in [s](\sigma) \land \exists \sigma'' \in [s](\sigma) \} \]

\[ [s \cdot x](\sigma) = \{ \sigma'' \mid \exists \sigma'. \sigma' \cdot \sigma'' \in [s](\sigma) \land \exists \sigma'' \in [s](\sigma) \land \exists \sigma'' \in [s](\sigma) \} \]

\[ [\text{while } A \text{ do } s \text{ od}](\sigma) = \bigcup_{i=0,1,\ldots} \Phi_i(\sigma) \]

\[ \Phi_0 = \emptyset \]

\[ \Phi_{i+1} = \emptyset \]

\[ \Phi_i = \emptyset \]

\[ \{ \sigma'' \mid \exists \sigma' \in \Phi_i(\sigma) \cdot \sigma' \in [A] \land \sigma'' \in [s](\sigma') \land \sigma'' \in [not \ A] \} \]

\[ [\ ] : \text{Proc}_{n,m} \rightarrow \mathcal{P}(D^{n+m}) \text{ is defined by} \]

\[ [\text{proc } \text{def } s \text{ t}](\sigma) = \{ \alpha \cdot \beta \cdot \gamma \mid \alpha \cdot \beta \cdot \gamma \in [s](\sigma) \land \exists \gamma' \cdot \gamma' \in [\text{t}](\sigma) \} \]

Now we shall discuss the use of this language. Typically the domain of the built-in data type is the set of all (pl \wedge ds) designs, but we shall also use other domains, however without formally having defined this. We take
\[ \mathcal{O}_2 = \{ \text{top, bot,} \ldots \} \]
\[ \mathcal{O}_2 = \{ \circ, * \ldots \} \]
\[ \text{Pred}_1 = \{ \text{bbc, gbc,} \ldots \} \]
\[ \text{Pred}_2 = \{ =, \neq, \pm, \div, \times, \text{top, bbv, gbv, bbc-gb-mod,} \ldots \} \]

where \( = \) denotes equality on \( D \) and where \( \equiv \) is defined by \( d_1 \equiv d_2 : \Leftrightarrow \exists x. [d_1] = [d_2]. \) The dots (\( \ldots \)) mean that we shall feel free to extend the set of operations and predicates when necessary. Both \( \circ \) and \( * \) are written in infix notation. Also \( = \) and \( \equiv \) are written in infix notation.

If no confusion can arise we sometimes omit parentheses e.g. writing \( A \) and \( A' \) instead of \( (A \text{ and } A') \). We shall freely give names to procedures, writing \( \alpha := p \) where \( \alpha \) is the name to be given to the procedure \( p \). We do not allow recursion (yet). Sometimes we replace the keyword proc by event or technique.

We give three simple examples of design-programs written in this language. In these examples we have three procedures (techniques) which are named 'split', 'impl' and 'pardev' respectively. The procedure 'split' is given in a pre- and postcondition style. The result of executing split\( (d) \) can be any pair of designs \( d_1 \) and \( d_2 \) for which \( d_1 \circ d_2 = d \) and in that sense 'split' is non-deterministic.

\[
\text{split} := \text{technique } d \rightarrow d_1, d_2 \\
\text{pre } \text{bbc}(d) \\
\text{post } d_1 \circ d_2 = d
\]

The second example is about a procedure 'impl' which also is given in a pre- and postcondition style. The technique 'impl' should be considered as an instruction to a developer to perform a bbc-preserving glass-box modification (bbc-gb-mod) upon a bbc input design and to yield the design resulting from this modification.

\[
\text{impl} := \text{technique } d \rightarrow d' \\
\text{pre } \text{bbc}(d) \\
\text{post } \text{bbc-gb-mod}(d, d')
\]

The third example is about a procedure 'pardev' which is given as a very simple algorithm consisting of an assignment statement and a result expression. It invokes the two procedures of the other examples. The technique 'pardev' can be viewed as a model of the development process offering a possibility for parallel development. It says that first of all the input design \( d \) should be split using the technique 'split' and that the resulting parts of \( d \) should be temporarily stored in \( d_1 \) and \( d_2 \). Then the technique 'impl' should be applied
to each of these parts (possibly in parallel) and the results should be fitted
together again with $\ast$ to form the final result design.

$$\text{pardev} := \text{technique } d$$
$$\text{def } d_1, d_2 := \text{split}(d);$$
$$\text{impl}(d_1) \ast \text{impl}(d_2)$$

After these examples, we can give some more motivation for this procedure
concept. Procedures have to yield a sequence of results because we want to
view splitting as a procedure (as shown in the ‘split’ example). We need the
non-determinism since we must be able to describe the creative freedom of
the developer. The fact that procedures have no side-effects makes reasoning
about them relatively easy.

In order to state properties of our design-programs we shall write \{A_1\} s \{A_2\}
iff the following holds:

$$\forall \sigma_1, \sigma_2 \cdot ((\sigma_1 \in \llbracket A_1 \rrbracket \land \sigma_2 \in \llbracket s \rrbracket \sigma_1) \Rightarrow \sigma_2 \in \llbracket A_2 \rrbracket)$$

i.e. if we start in a state $\sigma_1$ in which $A_1$ holds and if execution of $s$ stops in
state $\sigma_2$ then $A_2$ holds in $\sigma_2$. Note that the execution of $s$ starting in $\sigma_1$ can
go wrong for it may be the case that there is no $\sigma_2 \in \llbracket s \rrbracket(\sigma_1)$. In this case
we say that the execution of $s$ fails. It follows that non-termination is viewed
as a failure as well. We shall refer to \{A_1\} s \{A_2\} as the partial correctness of
$s$ with respect to $A_1$ and $A_2$. As an easy example of this notation we have
\{bbc(d)\} $d_1, d_2 := \text{split}(d); \{d_1 \ast d_2 = d\}$. 
Appendix C

List of symbols

In this appendix we give a list of the symbols used. For each symbol the list contains a very short informal description. The list has been subdivided into a number of sub-lists. The first sub-list contains general mathematical symbols. The second sub-list contains the symbols which are introduced in Chapter 2. The third sublist contains the symbols which are introduced and/or used first in Section 2. In a similar way the fourth sub-list contains the symbols which are introduced and/or used first in Section 3, and so on.

For some symbols the list contains a relevant page number – usually the defining occurrence of the symbol.

General mathematical symbols

\(\Rightarrow, \Leftarrow\) \hspace{1cm} Logical implication
\(\Leftrightarrow\) \hspace{1cm} Logical equivalence
\(\land, \lor\) \hspace{1cm} Conjunction, disjunction
\(\forall, \exists\) \hspace{1cm} Universal, existential quantification
\(\equiv\) \hspace{1cm} Syntactical equality
\(=\) \hspace{1cm} Equality
\(:=\) \hspace{1cm} Abbreviation
\(\{ \}\) \hspace{1cm} Set construction
\(\{ | \} \) \hspace{1cm} Set comprehension
\(\in\) \hspace{1cm} Set membership
\(\emptyset\) \hspace{1cm} Empty set
\(\cup, \cap\) \hspace{1cm} Set union, set intersection
\(\setminus\) \hspace{1cm} Set difference
\(\subseteq, \supseteq\) \hspace{1cm} Set inclusion
\(\mathcal{P}\) \hspace{1cm} Poweset
\(C^*\) \hspace{1cm} The set of sequences with elements from \(C\)
\(\times\) \hspace{1cm} Cartesian product
( , ) Pair
( , . . . , ) Tuple (= sequence)
\( \mathbb{N}, \mathbb{N}^+ \) Natural numbers, positive nat. numbers
\(<, \leq \) Less than, less than or equal to
+ Addition
0, 1, 2, . . . Natural numbers
\( f : A \rightarrow B \) \( f \) is a function from \( A \) to \( B \)
\( \lambda \) Lambda notation for functions
[ := ] Substitution

Symbols from Chapter 2

\( \mathbb{R} \) Typical algebraic system with preorder 55
\( \mathbb{R}_1 \) \( \mathbb{N} \) as algebraic system 58
\( \lambda \) Abstractor (symbol) 61
\( \lambda \pi \) Lambda calculus based on rule (\( \pi \)) 60
\( \pi_\mathbb{R} \) Set of terms for \( \lambda \pi \) 62
\( 0 \) Basic type symbol 61
\( \rightarrow \) Constructor for type symbols 61
\( \sigma, \sigma_1, . . . , r, r_1, . . . \) Typical type symbols 61
\( x_0, x_1, . . . \) Variables 62
\( x^r \) Typical variable of type \( r \) 62
\( \equiv \) Partial order (symbol) 62
\( A, B, C, L, M, P, . . . \) Typical lambda terms 63
\( x, y, z, u, v, w \) Typical variables 63
\( [\varphi] \) Formula \( \varphi \) viewed as an assumption 64
\( \Gamma, \Delta \) Typical contexts 64
\( \vdash \) Derivation symbol 64
\( \vdash_1 \) Rule of \( \lambda \pi \) (algebraic system oracle) 65
\( \vdash_2 \) Rule of \( \lambda \pi \) (algebraic system oracle) 65
\( (\text{context.}) \) Rule of \( \lambda \pi \) (context rule) 65
\( (\text{refl }) \) Rule of \( \lambda \pi \) (reflexivity) 66
\( (\text{trans.}) \) Rule of \( \lambda \pi \) (transitivity) 66
\( (\lambda I_1) \) Rule of \( \lambda \pi \) (lambda introduction) 66
\( (\lambda I_2) \) Rule of \( \lambda \pi \) (lambda introduction) 66
\( (\text{ap}.) \) Rule of \( \lambda \pi \) (application) 66
\( (\pi) \) Rule of \( \lambda \pi \) (partial contraction rule) 67
\( (=I) \) Rule of \( \lambda \pi \) (= introduction) 68
\( (\text{subst.}) \) Rule of \( \lambda \pi \) (substitution) 68
\( \Diamond \) Diamond (Church Rosser) property 80
SN Strong normalisation property 76
\( C_\mathbb{R} \) The set of components 84
### C. LIST OF SYMBOLS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c, c_1, \ldots$</td>
<td>Typical components</td>
</tr>
<tr>
<td>$D \subseteq$</td>
<td>The set of designs</td>
</tr>
<tr>
<td>$d, d_1, \ldots$</td>
<td>Typical designs</td>
</tr>
<tr>
<td>$\equiv, \subseteq$</td>
<td>Symbols used in concrete syntax of designs</td>
</tr>
<tr>
<td>\textbf{prim}, \textbf{system}</td>
<td>Symbols used in concrete syntax of designs</td>
</tr>
<tr>
<td>$\text{wf}$</td>
<td>Well-formed</td>
</tr>
<tr>
<td>$\lbrack \rbrack$</td>
<td>Meaning function for designs</td>
</tr>
<tr>
<td>gbc, bbc</td>
<td>Glass-box correct, black-box correct</td>
</tr>
<tr>
<td>bbc-gb-mod</td>
<td>bbc-preserving glass-box modification</td>
</tr>
</tbody>
</table>

**Symbols concerning designs**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tilde{M}$</td>
<td>Abbreviated form of e.g. $M_1, \ldots, M_n$</td>
</tr>
<tr>
<td>$D' \subseteq$</td>
<td>Designs where the system is a sequence</td>
</tr>
<tr>
<td>$\text{cs}( )$</td>
<td>Set of names of a design</td>
</tr>
<tr>
<td>$\text{sys}( )$</td>
<td>Set of names in the system of a design</td>
</tr>
<tr>
<td>$\text{arity}( )$</td>
<td>Arity of a design</td>
</tr>
<tr>
<td>$\equiv_l$</td>
<td>Semantic equivalence on designs</td>
</tr>
<tr>
<td>$p_f$</td>
<td>prims-first</td>
</tr>
<tr>
<td>$d_s$</td>
<td>Directly specified</td>
</tr>
<tr>
<td>$\Gamma_{gb}, \Gamma_{bb}$</td>
<td>Glass-box context, black-box context</td>
</tr>
<tr>
<td>$L$</td>
<td>Typical sequence of abstractions</td>
</tr>
<tr>
<td>$\mathcal{S}$</td>
<td>Typical Sequence of substitutions</td>
</tr>
</tbody>
</table>

**Symbols concerning algebraic operations on designs**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ast$</td>
<td>Concatenation operation on designs (binary)</td>
</tr>
<tr>
<td>$\circ$</td>
<td>Composition operation on designs (binary)</td>
</tr>
<tr>
<td>$e$</td>
<td>Empty design</td>
</tr>
<tr>
<td>$e_n$</td>
<td>Identity design of arity $(n, n)$</td>
</tr>
<tr>
<td>gbc, bbv</td>
<td>Glass-box valid, black-box valid</td>
</tr>
<tr>
<td>$\perp, \top$</td>
<td>Min. and max. elements (cf. $\subseteq$)</td>
</tr>
<tr>
<td>bot, top</td>
<td>Bottom and top operations on designs</td>
</tr>
<tr>
<td>$\equiv_{vp}$</td>
<td>Equality modulo permutation of prims</td>
</tr>
<tr>
<td>$=_{top}$</td>
<td>Top-equivalence</td>
</tr>
<tr>
<td>$%$</td>
<td>Column-wise operation on designs</td>
</tr>
<tr>
<td>${ }$</td>
<td>Embedding</td>
</tr>
</tbody>
</table>

**Symbols concerning design creation**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0, 1, \ldots, \oplus, \ominus, -$</td>
<td>Symbols of Boolean logic (in example)</td>
</tr>
<tr>
<td>GND, 7400, 7401, \ldots</td>
<td>TTL component names (in example)</td>
</tr>
<tr>
<td>$\hat{b}, \hat{s}$</td>
<td>Typical bit-sequences (in example)</td>
</tr>
</tbody>
</table>
int(δ)  Integer represented by δ (in example) 156
w  Typical machine&user-context 162
INV  General invariant to be (re-) established 164
POST  Postcondition of design creation 164
γ  Typical set of component names 165
<ₜˣ  One-step ‘part of’ relation 165
<ₜ  ‘Part of’ relation 165
<ₜᵈ  Occurs (directly or via <ₜ) in ... 165
TD_INV  Top-down invariant 166
td  Top-down technique 166
td_step  One-step top-down technique 166
last_prim  Operation yielding component 167
last_prim_context  Operation yielding ‘context’ design 167
td_impl  Technique implementing one component 168
insert  Operation for inserting a component 168

td_step₀, td_step₁,...  Alternative techniques for td_step 167
td_spec_impl  Combined specification & implementation 169
BU_INV  Bottom-up invariant 173
bu  Bottom-up technique 173
bu_step  One-step bottom-up technique 173
bu_step₀, bu_step₁,...  Alternative techniques for bu_step 174
add_system_element  Operation for adding a system element 174
add_component  Operation for adding a component 175

Symbols concerning design evolution

change  Procedure describing arbitrary event 179
emulator  Technique describing emulation 180
remove_unused  Operation to remove unused components 181
determine_bottom  Technique yielding a valid bottom design 181
simulator  Technique describing simulation 182
empty_system  Operation for making the system empty 182
determine_top  Technique yielding a valid top design 182

Symbols concerning design partition

d[v]  The unique component in d with name v 184
implᵥ  One-developer bbc-gb-mod technique 184
splitᵥ  Technique for splitting into components 185
pardevᵥ  Component-based parallel technique 185
impl  Arbitrary technique respecting INV 186
splitₚ  Technique for splitting according to * 187
defᵥ  Definedness of composition 187
C. List of Symbols

\text{msplit} \quad \text{Technique for splitting machine} \quad 187
\text{ssplit} \quad \text{Technique for splitting system user} \quad 187
\text{pardev}_s \quad \text{Parallel technique based on } * \quad 187
\text{split}_s \quad \text{Technique for splitting according to } o \quad 188
\text{pardev}_c \quad \text{Fallacious version of pardev}_s \quad 188
\text{pardev}_\circ \quad \text{Parallel development technique based on } o \quad 190
\text{impl}^+ \quad \text{Technique as impl not affecting top} \quad 190

Symbols concerning \( \lambda \text{-calculus with sequences} \)

\((, , )\) \quad \text{Constructor for type symbols} \quad 195
\([, , ]\) \quad \text{Constructor for terms (sequence)} \quad 196
\pi_i \quad \text{Constructor for terms (projection)} \quad 195
\pi \quad \text{Constructor for terms (concatenation)} \quad 195
\eta \quad \text{Classical notion of reduction} \quad 195
\((\pi_i), (\pi_1), ([\pi]-\text{seq})\) \quad \text{Rules of } \lambda \pi \text{ with sequences} \quad 196
\rightarrow \quad \text{One step reduction} \quad 197
\vdash \quad \text{Reduction} \quad 197
\equiv_\pi \quad \text{\( \pi \)-convertibility} \quad 198
\equiv_\pi \quad \text{Equality because of rule } (\pi_1) \quad 196
\tau, \sigma, \eta, \zeta \quad \text{Typical type symbols} \quad 195
\circ \quad \text{Constructor for terms} \quad 199
\((\circ_1), (\circ_2), (\circ_3)\) \quad \text{Rules of } \lambda \pi \text{ with sequences} \quad 199
\lambda \bar{x} \subseteq M. \quad \text{Abbreviating } \lambda x_1 \subseteq M_1, \ldots, \lambda x_m \subseteq M_m. \quad 199
M \quad \text{Abbreviating } M_n \ldots M_l \quad 199

Symbols concerning the design-development language

\text{Var} \quad \text{Set of variables} \quad 202
\text{Pred}_n \quad \text{Set of } n \text{-ary predicate symbols} \quad 202
\text{Op}_n \quad \text{Set of } n \text{-ary operation symbols} \quad 202
\text{Dexp} \quad \text{Set of deterministic expressions} \quad 203
\text{Asn} \quad \text{Set of assertions} \quad 203
\text{Elist}_k \quad \text{Set of expressions lists of length } k \quad 203
\text{Stat} \quad \text{Set of statements} \quad 203
\text{Proc}_{n,m} \quad \text{Set of procedures} \quad 203
x, y, \ldots \quad \text{Typical elements of } \text{Var} \quad 202
P_n \ldots \quad \text{Typical elements of } \text{Pred}_n \quad 202
O_n \ldots \quad \text{Typical elements of } \text{Op}_n \quad 202
c_n \ldots \quad \text{Typical elements of } \text{Dexp} \quad 203
A_n \ldots \quad \text{Typical elements of } \text{Asn} \quad 203
I_k \ldots \quad \text{Typical elements of } \text{Elist}_k \quad 203
s, ..., p_{a,m}, ... FV( ) AS( ) ::= | true, not, or, and [] while, do, od proc, axiom, def := [ ] D d ... σ, ... σ(d/x) Φ, {A_1} \circ \{A_2\} pre, post event, technique split, pardev, impl

Typical elements of Stat
Typical elements of Proca,m
Set of free variables of ...
Assignset of ...
Production symbol of syntax
Separator for alternatives in syntax
Keywords for logical connectives
Symbol for non-deterministic choice
Keywords for repetition construct
Statement terminator
Keywords for procedures
Symbol for giving a name to a procedure
Semantics of design-development language
Data domain (e.g. pf \& ds designs)
Typical elements of the data domain
Set of states
Typical states
Modification of a state
Semantics of i steps in repetition
Partial correctness (design-program level)
Alternative keywords for procedures
Alternatives keywords for proc
Examples of techniques
Chapter 4

Formal Specification of a Text Editor

4.1 Introduction

Many of the classical problems of software construction are caused by the absence of specifications or — if there are specifications — by the ambiguities in their formulation. Therefore the use of formal specification techniques is considered worthwhile. The ability to construct large formal specifications supports the applicability of the notions of component, black-box description and design of Chapter 2 and the correctness-preserving transformations of designs investigated in Chapter 3. This chapter deals with the construction of a formal specification and by way of example we specify a display-oriented text editor.

A text editor is a good example for illustrating formal specification techniques because of the following reasons. First, most software designers are familiar with at least one text editor, so the subject in itself may raise some interest and there is no need for an introduction into some less known application area. Secondly an editor is complex enough to give opportunity for illustrating many specification techniques and design principles.

In [1] the language COLD is proposed, in which one can formally express the design of a complete software system in various stages of its development. This language will appear in several user-oriented versions. There exists a kernel-language COLD-K [1,2,3] which contains all essential semantic features of the design language. We use this kernel language for expressing all formal definitions and axioms throughout our case study. This has the advantage that no ambiguities can arise and that a mechanical syntax-checking and type-checking can take place. However, sometimes when we want to
state derivable properties of certain operations, we use a somewhat more liberal style of notation.

There are several related reasons that make the undertaking of performing this editor case study worthwhile. The first reason is that it shows the usefulness of formal specification techniques in general and of \( \lambda \)-calculus and COLD-K in particular. The second reason is that we get an opportunity to annotate the specification and to formulate some guidelines which might be of help for other users of formal specification techniques. The third reason is that it will provide us with a starting-point for a large and typical application of the design methods studied in Chapter 3. The application consists of a systematic top-down development of the editor, which will be undertaken in Chapter 5.

The construction of a formal specification typically involves the following activities: (1) formalisation of application domain-specific concepts: in our case the most important concept being that of text, and (2) writing actual specifications of the system under consideration, which in our case is the editor. Typically one starts with (1) and then proceeds with (2). However during the writing of the actual specifications, one might discover the need of additional operations etc. and these should be added to the application domain-specific concepts formalised already. This is precisely what has happened during the construction of the specification presented in this chapter. The chapter is a rational reconstruction of this process; we have collected the application-domain specific concepts and put them before the description of the editor. A central role will be played by the concept of text. We shall devote some effort to an investigation of this concept by introducing a collection of algebraic operations on texts. In this way we develop a machinery which will turn out to be useful, without making any specification decisions about the actual text editor itself yet. Section 4.2 gives an overview of the entire editor specification. Text and algebraic operations on text are the subject of Section 4.3.

The formal specification presented in this chapter will focus on the functional aspects of a text editor, by which we mean those aspects that are not related to performance issues. Rather than creating a completely new editor we take over several concepts from the EMACS family of editors [4,5,6]. Of course, we have to make choices about the way our editor behaves and choose among the many different possible features. We do not claim that our choices are preferable over the many other possibilities. Instead, the points we want to make in this chapter are (1) that it is possible to make a precise description of the behaviour of an editor and (2) that writing a formal specification helps in constructing an editor that is based on a few well-understood concepts, rather than a large collection of ad-hoc features. In Section 4.4 we describe
a display, some data types for interfacing and a file system. The actual text editor is the subject of Section 4.5. In Section 4.6 we discuss some related work. Section 4.7 is devoted to conclusions.

In Appendix A we provide an overview of all sorts, functions, predicates and procedures used in this chapter. For each symbol there is a short informal description. In Appendix B we give a number of standard class descriptions.

4.2 Overview of the Formal Specification

In this section we give an overview of the formal specification presented in the remainder of this chapter. This formal definition will consist of many definitions of sorts, functions, predicates, procedures and axioms. These definitions are grouped into a number of modules, or class descriptions as they are called in COLD-K. Of course we must avoid that this becomes an almost endless list of definitions without its goal being clear in advance. Therefore the overview presented in this section is goal-oriented in the sense that it begins with the top-level definition of the entire formal editor specification. This editor specification will not be finished until Section 4.5.14 where its top-level definition turns out to be a procedure called key. We show a fragment of the definition of key, where it is understood that the sort Char corresponds with a character-set.

```
PROC key: Char ->
VAR c:Char
DEF ( printable(c) ?; insert_character(c)
  | ord(c) = 0 {"\0"} ?; set_mark
  | ord(c) = 1 {"\n"} ?; beginning_of_line
  ...
  | ord(c) = 20 {"\?"} ?; insert-file
  ...
  etc.
  )
```

This procedure will be made available to the user of the editor and things must be arranged such that when the user hits some key on his keyboard, producing character c say, then key(c) is invoked which activates one of the editor operations insert_character(c), set_mark, beginning_of_line, insert_file etc. Most of these operations result in a modification of an edit-buffer and this modification at its turn can be observed on the screen of the user's video display unit. The procedure key is contained in a class description called KEYBIND_SPEC which imports two other class descriptions called
WITEFA_SPEC and MOREDOP_SPEC. The names of the latter class descriptions are derived from Window-and-Text-Facility and More-Editing-Operations. The window and text facility is the kernel of the editor whereas MOREDOP_SPEC provides a few more operations which can be described easily in terms of operations from this kernel. More precisely, WITEFA_SPEC provides a layer of general-purpose editing primitives, whereas MOREDOP_SPEC turns them into one particular editor -- using the general-purpose primitives in a specialized way. These two class descriptions are presented in Sections 4.5.1-4.5.12 and 4.5.13 respectively. The window and text facility provides operations such as

PROC insert_character: Char ->

The insert_character operation has roughly speaking the effect that an edit-buffer corresponding with a notion of 'current text' is modified, that the screen of the display is updated and that the cursor moves one position to the right. When we try to explain the meaning of insert_character in more detail, it becomes clear that we need a lot of preparations. In particular we must have descriptions of the conceptual organisation of the edit-buffers, of the notion of 'current text' and of the relation between the contents of the edit-buffers and the visible contents of the screen. There are also operations that deal with file handling. For example we have

PROC insert_file: ->

which indicates a need to describe a file system as well. Since files are addressed by names, we must also be prepared to model the strings which are used for that. The above discussions explain why WITEFA_SPEC and MOREDOP_SPEC at their turn are based on several lower level class descriptions, amongst which:

- DISPLAY_SPEC: a model of the video display unit,
- 'SEG_SPEC' and 'STRING_SPEC': sequences and as a special case of these, strings (the quotes are formally part of the names).
- FILE_SPEC: a file-system.

These are presented in Sections 4.4.2, 4.4.3 and 4.4.4 respectively. By way of example we shall have a closer look at DISPLAY_SPEC, which specifies the video display unit -- at least as much of it as we need to specify the editor. The state space of DISPLAY_SPEC is spanned by two variable functions

FUNC screen: -> Text VAR
FUNC cursor: -> Nat # Nat VAR

where Text denotes a sort of texts and where Nat refers to the natural num-
bers. These numbers represent the vertical and horizontal co-ordinates of the
cursor. All display operations are described by their effect on either screen
or cursor or both. For example

PROC nl : -> NEW screen, cursor

serves for sending a new-line command to the display, thereby possibly mod-
ifying the screen and the cursor. The sort Text is in fact an application
domain-specific concept and it plays a key role throughout this case study.
Therefore the formal specification begins with a study of texts and opera-
tions on texts. This study comes before the specification of the display and
the file system. We view a text as a sequence of lines. The sort of lines is
denoted as Line and each line consists of a sequence of characters. We shall
have the following sorts introduced formally:

SORT Line
SORT Text

and there are operations to select a line from a text and to select a character
from a line.

FUNC sel: Text # Nat -> Line
FUNC sel: Line # Nat -> Char

Starting from this very simple model, a rich collection of operations on texts
is defined, including a variety of cut and paste operations. For example, there
is a paste operation such that paste(t, u, k, l) means to take a text t and
to insert another text u into it immediately before the position with given
coordinates (k, l).

FUNC paste: Text # Text # Nat # Nat -> Text

As a kind of inverse of paste there is an operation cut such that cut(t, k, l, m, n)
means to take a text t and to cut out the piece of text beginning at posi-
tion (k, l) and ending at position (m, n). It yields a pair (t₁, t₂) where t₁ =
'remaining text' and t₂ = 'deleted text'.

FUNC cut: Text # Nat # Nat # Nat # Nat -> Text # Text

We shall not only give the definitions of these and similar operations, but
we shall also study some of their properties, which are quite elegant from
an algebraic point of view. The operations are grouped into a number of
class descriptions among which LINE_SPEC, TEXT_SPEC, TEXT_OPS1_SPEC,
TEXT_OPS2_SPEC, TEXT_OPS3_SPEC, STRING_SPEC, and PROFILE_SPEC. This
is done in Section 4.3. No real decisions about the editor are taken yet
in Section 4.3; the only thing which happens is that those operations are introduced formally which are needed to discuss text editing. In fact, at the end of Section 4.3 there is hardly any clue whether we aim at an EMACS-like editor, an ED-like editor or a VI-like editor. We could even specify a MacPaint-like system, although our collection of operations is quite useless in that case.

This concludes the overview of the formal specification of this chapter. The overview has been written afterwards, in a goal-oriented fashion, but the actual presentation in the remainder of this chapter will be organised the other way around. This is because we want to formally introduce each sort and operation before it is used.

4.3 Text and Algebraic Operations On Text

4.3.1 Introduction

There will be several approaches to modeling texts, which differ both with respect to the style of description and the level of abstraction. E.g. consider the following text:

```
first line of text
second line
```

then we can model it as a sequence of lines where the first line has 18 characters viz. "first line of text" and where the second line has 11 characters, viz. "second line".

An equally valid, but somewhat more abstract approach is to model texts by focusing on their 'contour' only. In this approach the above text is modeled by just the sequence (18,11) alone. Of course there is a kind of forgetful mapping from the first model to the second model in the sense that the information conveyed by the actual characters in the text, is lost in the second model. Such approaches at distinct levels of abstraction will play a role in our formalisation of the notion of text and as it turns out there will be operations on texts at several levels of abstraction, such that the corresponding forgetful mapping behaves homomorphically.

In fact there are several more ways of modeling texts and we mention two of these now. First, we can describe texts as strings, provided we adopt some line-separator. This approach is, roughly speaking, at the same level of abstraction as the first approach mentioned above, where a text is modeled
by a sequence of lines. Secondly we can describe texts just by their length which we could define as the length of the string representation. In a sense this is the most abstract non-trivial model, since it forgets about both the structure and the contents of the text.

The above discussion may seem a bit vague and it is the very purpose of our formalisation of the notion of text to cast such ideas into definitions and propositions which are described with mathematical rigour. In this formalisation we will introduce several algebraic operations on texts. For example there will be an operation called cut which serves for taking apart texts and an operation paste which serves for fitting together texts. As it turns out, there are many meaningful operations on texts and it is possible to build a rich collection of them. We shall build our collection with the guideline that an operation is included when we foresee that it can be used when specifying an editor. In case of doubt about the usefulness of an operation, we just include it, because there is not much harm in having too much operations; this is because we are not specifying any system to be built yet but we are just gathering a vocabulary for speaking about such a system.

A second guideline is that an operation is more likely to be useful when it has nice algebraic properties. When introducing a new operation, it is a good idea to consider the questions: does associativity hold? does commutativity hold? does idempotency hold? does a neutral element exist? etc. Note that it depends on the signature of an operation whether these questions are meaningful. Sometimes such a question is not meaningful but we can invent a suitable variation of it which does make sense. For the applicable properties we then investigate whether they hold or not.

In order to get started, we need specifications describing several data types of a mathematical nature. For each such data type there is one Cold-K class description. We do not start completely from the beginning, but instead we use data types such as NAT_SPEC, CHAR_SPEC, SEQ_SPEC, TUPLE_SPEC, SET_SPEC, MAP_SPEC etc. We adopt these from [?] and we only add a few simple constants such as 127 and 'a'. Just for completeness, these class descriptions are given in Appendix B. This chapter is organised in such a way that when all formal Cold-K texts below are appended and Appendix B is put before that, we get a well-formed list of abbreviation-type components.

We start with the formal introduction of the sorts Line and Text. This is the subject of Section 4.3.2. Next, we introduce strings and we have a look at the relation between texts and strings. This is the subject of Sections 4.3.3 and 4.3.4. Furthermore we say a few words about printability, which is in Section 4.3.5.

After that, our systematic study of algebraic operations on texts begins. This
study of algebraic operations covers the Sections 4.3.6 to 4.3.11.

4.3.2 Texts

We pay some attention to the question 'what is text?'. It is tempting to say that a text is a sequence of characters, separated by new-line symbols. However, if you ask this question to someone without programming experience, he or she will probably tell you about characters on some two-dimensional medium like paper or a screen. Therefore it is more natural approach to say that a text is a sequence of lines where each line is a sequence of characters. This leads us to a formalisation of lines first.

We use SEQ_SPEC from Appendix B which is a parameterised description of sequences (sort Seq) with constructor operations empty and cons and also operations hd, tl, eq, cat, len, rev, bag for head, tail, selection, concatenation, length, reversal and bag construction respectively. Its parameter restriction only mentions a sort Item. LINE_SPEC is obtained as an instantiation of SEQ_SPEC where the items are replaced by characters (sort Char) and where the resulting sequences are named Lines. The specification CHAR_SPEC provides among other things this sort Char. It is taken from Appendix B and it serves as an actual parameter here.

It is a peculiarity of COLD-K that when establishing such an instantiation the renaming takes place in the parameterised specification (SEQ_SPEC); the resulting renamed version of it is applied to the actual parameter (CHAR_SPEC).

LET LINE_SPEC :=
APPLY RENAME
  SORT Seq TO Line,
  SORT Item TO Char
IN SEQ_SPEC TO CHAR_SPEC;

In a similar way we introduce the sort Text as sequences of lines. It is important to realise that a text is something to be distinguished from the internal representation of a text inside a text editor.

LET TEXT_SPEC :=
APPLY RENAME
  SORT Seq TO Text,
  SORT Item TO Line,
  FND empty: -> Seq TO niltext
IN SEQ_SPEC TO LINE_SPEC;
Since we used an instantiation of \texttt{SEQ\_SPEC} we have already the operations \texttt{nil\_text, cons, hd, tl, sel, cat, len and rev}. Note that \texttt{nil\_text} is the text having no lines at all.

We shall introduce some more operations now and they will be made part of a class description called \texttt{TEXT\_OPS1\_SPEC}. We want to add a remark about structuring our specification here. The purpose of \texttt{TEXT\_OPS1\_SPEC} and similar class descriptions to be introduced later is to structure our collection of sorts and operations. Each class description typically begins with a collection of imports, followed by a \texttt{CLASS \ldots END} type class description containing the newly introduced operations. The class descriptions serve as a grouping mechanism for sort definitions, function definitions, predicate definitions etc. Of course we aim at a functional grouping by which we mean that we put together those operations which are closely related with respect to their purpose and their technical contents.

\begin{verbatim}
LET TEXT\_OPS1\_SPEC :=
IMPORT NAT\_SPEC INTO
IMPORT CHAR\_SPEC INTO
IMPORT LIL\_SPEC INTO
IMPORT TEXT\_SPEC INTO
CLASS

Sometimes is is desirable to consider non-nltext texts only; later we shall introduce a predicate \texttt{ok} on texts, such that \texttt{ok(t)} will imply that \( t \neq \text{nil}\text{text} \). It would be convenient to have a unique text with the intuition of 'empty text'. However, there are two candidates for this, viz. \texttt{nil\text{text}} and the text that consists of one empty line. Let us call the latter \texttt{text zero}.

We introduce some auxiliary functions, grouped into functions to construct texts (\texttt{zero, add\_empty and add\_char}), and functions to take texts apart (\texttt{first and rest}). The function \texttt{zero} yields the text that consists of one empty line. The function \texttt{add\_empty} adds an empty line in front of a text. The function \texttt{add\_char} adds one character at the beginning of the first line of a text. \texttt{first} and \texttt{rest} are unary functions on texts. The function \texttt{first} yields the first character of a text. The function \texttt{rest} yields a text that is the input text with its first character removed. Note that \texttt{hd, tl} and \texttt{cons} are overloaded in the sense that there are two operations called \texttt{hd} (viz. one \texttt{one on texts} and one on lines) and similarly two \texttt{tl} and \texttt{two cons} operations. Later we shall come back to this overloading mechanism.

F\texttt{UNC} zero: \( \rightarrow \text{Text} \\
D\texttt{EF} \ cons(\text{empty.niltext})
\end{verbatim}
FUNC addempty: Text -> Text
PAR t:Text
DEF cons(empty, t)

FUNC addchar: Char # Text -> Text
PAR c:Char,t:Text
DEF cons(cons(c,hd(t)),tl(t))

FUNC first:Text -> Char
PAR t:Text
DEF hd(hd(t))

FUNC rest: Text -> Text
PAR t:Text
DEF cons(tl(hd(t)),tl(t))

We have the obvious properties
\[
\forall t: Text, c: Char \ ( t \neq \text{niltext} \Rightarrow \\
\quad \text{first}(\text{addchar}(c, t)) = c \land \text{rest}(\text{addchar}(c, t)) = t)
\]

and
\[
\forall t: Text \\
\quad (\text{hd}(\text{addempty}(t)) = \text{empty} \land \text{tl}(\text{addempty}(t)) = t)
\]

where we took some obvious notational freedom with respect to COLD-K in the sense that we wrote \(\forall\) rather than FORALL, \(t \neq \text{niltext}\) instead of NOT \(t = \text{niltext}\) etc. This kind of syntactic sugar is formally not part of COLD-K and when syntax- and type-checking the formal texts of this chapter, we excluded the propositions which are based on this syntactic sugar. We only use this syntactic sugar when discussing properties of definitions and never to give the formal definitions themselves. Throughout this chapter the actual formal definitions are in COLD-K. These formal COLD-K texts are recognisable by their somewhat smaller font (first rather than \text{first}) and by the complete absence of mathematical type-setting (\(\Rightarrow\) rather than \(\Rightarrow\)).

END: (of TEXT_DEF1_SPEC)

As a next step we relate texts with strings, by which we mean sequences of characters. This is the subject of our next two sections.
4.3.3 Strings

We introduce the sort String as sequences of characters. Again we use SEQ_SPEC from Appendix B. In informal writing we freely denote strings using double quotes, e.g. "this is a string". We introduce the usual lexicographical ordering on strings as a predicate less. It is defined by recursion.

It is interesting to point out that the strictness principles of COLD-K are essential for this definition. In particular, consider the case where t = empty, then hd(t) is undefined and then by strictness so is ord(hd(t)). Therefore less(ord(hd(s)),ord(hd(t))) is false and so is the equation hd(s) = hd(t). Therefore in that case less(s,t) is false.

LET STRING_SPEC :=
EXPORT

SORT Char,
SORT Nat,
SORT String,
FUNC empty : String -> String,
FUNC cons : Char # String -> String,
FUNC hd : String -> Char,
FUNC tl : String -> String,
FUNC len : String -> Nat,
FUNC sel : String # Nat -> Char,
FUNC cat : String # String -> String,
FUNC rev : String -> String,
PRED less : String # String

FROM

IMPORT APPLY RENAME
SORT Seq TO String,
SORT Items TO Char
IN SEQ_SPEC TO CHAR_SPEC INTO

IMPORT NAT_SPEC INTO
IMPORT CHAR_SPEC INTO
CLASS

PRED less : String # String
PAR s:String,t:String
DEF s = empty AND NOT t = empty
OR (less(ord(hd(s)),ord(hd(t)))
OR hd(a) = hd(t) AND less(tl(a), tl(t))

END;

4.3.4 Relating Texts and Strings

Let us now discuss how texts and strings are related. By adopting some special separator character, control-j say, we can define a bijection between the sort String and the set of non-nil text texts not containing this special separator character. We call these texts ok and for this purpose we introduce a predicate ok. We formally introduce control-j as a constant \texttt{ctr\_j} first and we use the operation \texttt{chr: Nat \rightarrow Char} from \texttt{CHAR\_SPEC} for that.

\begin{verbatim}
LET TEXT\_OPS2\_SPEC :=
IMPORT NAT\_SPEC INTO
IMPORT CHAR\_SPEC INTO
IMPORT LIST\_SPEC INTO
IMPORT TEXT\_SPEC INTO
IMPORT TEXT\_OPS1\_SPEC INTO
IMPORT STRING\_SPEC INTO
CLASS

FUNC \texttt{ctr\_j}: \rightarrow \texttt{Char}
DEF \texttt{chr(10)}

PRE ok: Text
IND FORALL c:Char, t:Text
  \begin{align*}
  \text{ok} & (\text{zero}:\text{Text}) ; \\
  \text{ok} & (\text{t} \Rightarrow \text{ok(addempty(t))} ; \\
  \text{ok} & (\text{t} \text{ AND NOT } c = \text{ctr\_j} \Rightarrow \text{ok(addchar(c, t))} )
  \end{align*}

Note that \text{ok(t)} ⇒ t \neq \text{nil\_text}. We have the following induction principle for ok texts. To prove an assertion \texttt{A} for ok texts, it suffices to show \texttt{A(zero)} and two induction steps. The first induction step is \forall t: Text (A(t) ⇒ A(addempty(t))). The second induction step is \forall t: Text, c: Char (A(t) \land c \neq \text{ctr\_j} ⇒ A(addchar(c, t))).

We have the following decomposition principle for ok texts. Each ok text is of one of three possible forms, viz. \texttt{zero}, \texttt{addempty(...)} and \texttt{addchar(...)}. We can distinguish these three possibilities also as \{1\} \texttt{hd(t)} = \texttt{empty} \land \texttt{tl(t)} = \texttt{nil\_text}, \{2\} \texttt{hd(t)} = \texttt{empty} \land \texttt{tl(t)} \neq \texttt{nil\_text} and \{3\} \texttt{hd(t)} \neq \texttt{empty}.

The functions \texttt{text} and \texttt{string} to be introduced below establish a bijection
between the sort String and the set of ok texts. These functions text and string are similar to Meertens' Line and Unline functions [10]. Although it is possible to give inductive or axiomatic characterisations of these functions, we use recursive definitions. This has the advantage that it is easy to experiment with these definitions using any classical imperative or functional programming language. The function text converts a string to a text by interpreting ctr_j as a separator. The function string applied to a text t converts t to a string by putting a ctr_j between the lines. In the definition of the function string we distinguish three cases, where the cases (2) and (3) give rise to a recursive call of string. We easily verify that the guard (2) implies ok(tl(t)) and that the guard of (3) implies ok(rest(t)).

    FUNC text: String -> Text
    PAR s: String
    DEF ( (1) s = empty ?;
        zero
        | (2) hd(s) = ctr_j ?;
            addempty(text(tl(s)))
        | (3) NOT (s = empty OR hd(s) = ctr_j) ?;
            addchar(hd(s), text(tl(s)))
    )

    FUNC string: Text -> String
    PAR t: Text
    DEF ok(t) ?;
    ( (1) hd(t) = empty AND tl(t) = niltext ?;
        empty
        | (2) hd(t) = empty AND NOT tl(t) = niltext ?;
        cons(ctr_j, string(tl(t)))
        | (3) NOT hd(t) = empty ?;
        cons(first(t), string(rest(t)))
    )

We leave several options un-investigated here currently such as embedding Char \{ ctr_j \} into Char, or having a literal character in the string representation.

### 4.3.5 Printability

Let us say a few words about the problems arising because in our formalisation texts may contain non-printable characters. There are two such problems. The first problem is obvious: non-printable characters cannot be printed or displayed as such by a display-oriented editor. Of course it is
always possible to print some substitute, e.g. \texttt{\textquotesingle\textquotesingle}.

The second problem is that inside a text editor some representation for texts must be chosen. If the text editor is only meant for manipulating ok texts, then its designer is in a comfortable position because he has the option of choosing the string-representation discussed before or some other related representation. We need not make such a choice here, but we introduce a suitable predicate denoting printability. We use the fact that in ASCII the printable characters are in the range \texttt{\textquotesingle	extquoteright} \texttt{. . . . }\texttt{\textquoteright}. We consider the \texttt{\textbackslash t} character as non-printable.

\begin{verbatim}
FUNC blank: -> Char
DEF chr(32)

FUNC tilde: -> Char
DEF chr(126)

PRED printable: Char
PAR c:Char
DEF leq(ord(blank),ord(c)) AND leq(ord(c),ord(tilde))
\end{verbatim}

For later use we elaborate on the idea of replacing non-printables by some substitute, and in order to keep things simple, we choose \texttt{\textbackslash t}ilde as a substitute indeed. Therefore we introduce by overloading three functions called \texttt{printify} with the intuition of 'make printable'.

\begin{verbatim}
FUNC printify: Char -> Char
PAR c:Char
DEF ( printable(c) ... c
\mid NOT printable(c) ... tilde
\)

FUNC printify: Line -> Line
PAR l:Line
DEF ( l = empty ... empty
\mid NOT l = empty ... cons(printify(hd(l)),printify(tl(l)))
\)

FUNC printify: Text -> Text
PAR t:Text
DEF ( t = niltext ... niltext
\mid NOT t = niltext ... cons(printify(hd(t)),printify(tl(t)))
\)
\end{verbatim}

We used the mechanism of overloading which means that we have several
functions with the same identifier (printify) but with different argument types and/or result types. This is allowed in COLD-K and it has the advantage that we are not forced to invent new names ourselves (printify_char, printify_line and printify_text, say). When using overloaded functions or predicates, we must make sure that our expressions and assertions can be analysed in precisely one way. We give a simple example: AXION printify(printify(nil_text)) = nil_text is well-typed because there is only one way of analysing it, viz. by assuming that both printify invocations refer to the function printify: Text \rightarrow Text. Note that under this assumption the axiom becomes well-typed.

4.3.6 Natural Operations on Text

One of the best ways to get an understanding of the concept of text as formalised by the sort Text, is to look for algebraic operations operating on texts. In particular, we shall start with a simple binary operation of function-type Text # Text → Text. Of course it is not hard to define the insertion of a character into a text, which would be an operation of function-type Text # Char # Nat^2 → Text, where we used Nat^2 as an obvious shorthand for Nat # Nat. However, we avoid such heterogeneous operations, by which we mean those operations that deal both with texts and with characters. We must be prepared to let the domain Nat^2 play a role when we define operations dealing with co-ordinate pairs.

We shall introduce an operation called add below and we refer to it as (natural) addition because it can be viewed as a natural way of adding texts. Its definition is based on the idea that people are supposed to read texts line after line, scanning lines from left to right. To read the addition of two texts, one first reads the first text and after that, immediately proceeds with the second text. In Section 4.3.9 we shall also encounter other ways of adding texts for which we want to use names such as vertical addition and horizontal addition. Let us give an example showing the natural addition of two texts and postpone the formalisation for a moment.

\[
\text{add}(\begin{array}{c}
\text{aaaaaa} \\
\text{aaaaaa} \\
\text{a}
\end{array}, \begin{array}{c}
\text{bbb} \\
\text{bbb} \\
\text{bbb}
\end{array}) = \begin{array}{c}
\text{aaaaaa} \\
\text{aaaaaa} \\
\text{abbb} \\
\text{bb} \\
\text{bb}
\end{array}
\]

From the example it can be seen that this addition is not the same as concatenation; the subtle difference is that the addition does not introduce a
jump to a new line at the point where the two texts are joined.

Although the above addition operation seems too trivial for being interesting, it will serve as a kind of starting point for finding other operations which can be employed in the description of cut and paste capabilities for an editor. An interesting question which poses itself immediately is whether add has an inverse. Since addition is not injective there can not be an inverse in the strict sense, but we can propose an operation whose intuition is related to the idea of 'undoing the effect of an addition'. This operation will be called split.

The operation split describes the splitting of a text into two parts, where the splitting-point is just before some given position. We add one remark about the use of co-ordinates. Whenever we use pairs (i, j) where i and j are co-ordinates – or similar quantities – it is understood that i is the vertical co-ordinate and j is the horizontal co-ordinate. The following picture may convey some intuition for both split and add.

![Diagram of split and add operations]

The fact that an arbitrary co-ordinate pair may indicate a position that is non-existing in the given text has as consequences that split is a partial operation and that the result of split need not be a pair of ok texts. Also add is a partial operation, which is the effect of an explicit guard testing the ok-ness of the input texts. We give the formal definitions below using recursion.

```
FUNC split: Text # Nat # Nat -> Text # Text
PAR t:Text, i:Nat, j:Nat
DEF ( (1) i = 0 AND j = 0 ?:
    zero, t
  | (2) i = 0 AND NOT j = 0 ?
    LET t1,Text,t2,Text;
    t1,t2 := split(xest(t),0,pred(j));
    addchar(first(t),ti), t2
```
4.3. TEXT AND OPERATIONS ON TEXT

{|3|} \textbf{ok} \ i = 0 ?:
  \begin{align*}
  \text{LET } & t1 : \text{Text}, t2 : \text{Text}; \\
  & t1, t2 := \text{split}(t1, \text{pred}(t1), j); \\
  & \text{cons}(\text{hd}(t), t1), t2
  \end{align*}
\}

\textsc{func add} : \text{Text} \# \text{Text} \rightarrow \text{Text}
\text{par} t1 : \text{Text}, t2 : \text{Text}
\text{def ok}(t1) \land ok(t2) ?:
  \begin{align*}
  (\{1\} & \text{hd}(t1) = \text{empty} \land t1(t1) = \text{niltext} ?; \\
  & t2 \\
  \{2\} & \text{hd}(t1) = \text{empty} \land \text{not} t1(t1) = \text{niltext} ?; \\
  & \text{addempty}(\text{add}(t1(t1), t2)) \\
  \{3\} & \text{not} \text{hd}(t1) = \text{empty} ; \\
  & \text{addchar}(\text{first}(t1), \text{add}(\text{rest}(t1), t2))
  \end{align*}

We state several properties of these operations. Addition of ok texts is associative, i.e.
\[ \forall t1, t2, t3 : \text{Text} \ ( \text{ok}(t1) \land \text{ok}(t2) \land \text{ok}(t3) \Rightarrow \\
\text{add}(t1, \text{add}(t2, t3)) = \text{add}(\text{add}(t1, t2), t3) \) \]

which can be shown by induction on \( t1 \), using the induction principle of Section 4.3.4. Now we can also explain why we used the name \text{zero} for the text with one empty line: it behaves both as a left and a right neutral element with respect to the operation \text{add}:
\[ \forall t : \text{Text} \ ( \text{ok}(t) \Rightarrow \\
\text{add}(t, \text{zero}) = t \land \text{add}(\text{zero}, t) = t ) \]

where the first equality is a consequence of the induction principle of Section 4.3.4 and where the second equality holds just by definition. The function \text{add} is the inverse of \text{split} in the following sense:
\[ \forall t : \text{Text}, c : \text{Nat} \ ( \text{ok}(t) \land \text{split}(t, c) = \Rightarrow \\
\text{add}(\text{split}(t, c)) = t ) \]

which can be shown by induction on the vertical co-ordinate of the splitting point. The basis of the induction is \( c = (0, j) \) which at its turn is shown by induction on \( j \).

Next, we describe two operations which are somewhat more complicated, but which can be defined easily in terms of the operations \text{add} and \text{split}. These operations are called \text{cut} and \text{paste} and they can be viewed as a
complementary pair of operations—just like split and add. The operation cut deletes a text-region from a given position (inclusive) unto another given position (exclusive) if possible. It yields a pair \((t_1, t_2)\) where \(t_1\) = ‘remaining text’ and \(t_2\) = ‘deleted text’.

The operation paste takes a text \(t\) and inserts another text \(u\) into it immediately before the position with given co-ordinates \((k, l)\) say, if possible. The following picture may convey some intuition for both cut and paste.

![Diagram of cut and paste operation]

Formally we define cut and paste by using the simpler operations split and add. These split and paste are partial operations and as a consequence, so are cut and paste.

\[
\begin{align*}
\text{FUNC cut}: & \quad \text{Text} \times \text{Nat} \times \text{Nat} \times \text{Nat} \rightarrow \text{Text} \times \text{Text} \\
\text{PAR} & \quad t : \text{Text}, n : \text{Nat}, i : \text{Nat}, j : \text{Nat} \\
\text{DEF} & \quad \text{LET} \ t_1 : \text{Text}, t_2 : \text{Text}; t_1, t_2 := \text{split}(t, i, j); \\
& \quad \text{LET} \ t_{11} : \text{Text}, t_{12} : \text{Text}; t_{11}, t_{12} := \text{split}(t_1, n, m); \\
& \quad \text{add}(t_{11}, t_{12}), t_{12}
\end{align*}
\]

\[
\begin{align*}
\text{FUNC paste}: & \quad \text{Text} \times \text{Text} \times \text{Nat} \times \text{Nat} \rightarrow \text{Text} \\
\text{PAR} & \quad t : \text{Text}, u : \text{Text}, k : \text{Nat}, l : \text{Nat} \\
\text{DEF} & \quad \text{LET} \ t_1 : \text{Text}, t_2 : \text{Text}; t_1, t_2 := \text{split}(t, k, l); \\
& \quad \text{add}(t_1, \text{add}(u, t_2))
\end{align*}
\]

For appropriate choices of the \(\text{Nat}^2\) argument, the zero text behaves also as a left and a right neutral element with respect to the operation paste.

\[
\forall t : \text{Text}, c : \text{Nat}^2 \ (\text{ok}(t) \land \text{paste}(t, \text{zero}, c) \Rightarrow \text{paste}(t, \text{zero}, c) = t)
\]

which holds because zero is the left neutral element with respect to add and add is the inverse of split.

\[
\forall t : \text{Text}, c : \text{Nat}^2 \ (\text{ok}(t) \land (0, 0) \Rightarrow \text{paste}(\text{zero}, t, c) = t)
\]
The latter proposition follows from the fact that zero is the left- and right neutral element with respect to add. The operation paste is the inverse of cut in the following sense:

\[
\forall t : \text{Text}, \ c, d : \mathbb{Nat}^+ \ (\text{ok}(t) \land \text{cut}(t, c, d)!) \Rightarrow \\
\text{paste}(\text{cut}(t, c, d), c) = t
\]

which follows from the fact that add is the inverse of split and furthermore from the property \(\text{split}(t_1, c) = (t_{i_1}, \ldots) \Rightarrow \text{split}(\text{add}(t_{i_1}, t_2), c) = (t_{i_1}, t_2)\), which at its turn is shown by induction on the vertical and horizontal coordinates of the splitting point — just as before. In a certain way, split is also the inverse of add and cut is the inverse of paste, but before we can state this precisely, we first need another concept, viz. that of the reach of a text. This is the subject of our next section.

### 4.3.7 The Reach of a Text

The function reach applied to a text \(t\) results the pair \((d_1, d_2)\) where \(d_1\) is the co-ordinate of the last line of \(t\) and where \(d_2\) is the number of characters in the last line of \(t\). We give an example:

\[
\text{reach}(\begin{array}{c}
\text{a}
\end{array}) = (2, 1)
\]

We call such a pair \((d_1, d_2)\) a reach. We have chosen the term reach because of the intuition that \(\text{reach}(t)\) indicates ‘how far you can get’, starting at position \((0, 0)\) and trying to get as far downwards and rightwards as possible within this text \(t\). In a certain sense \(\text{reach}(t)\) can be viewed as a ‘size’ of \(t\). Note that \(d_1\) and \(d_2\) play a somewhat different role: \(d_1\) is the co-ordinate of the last line whereas \(d_2\) is the co-ordinate of the last character in the last line plus one. We sketch the intuition behind reach by a picture:
We give the formal definition using recursion below. There are three cases marked as 1', 2' and 3' to indicate that we employ a decomposition principle for texts different from that of Section 4.3.4.

\[
\text{FUCN} \text{reach} : \text{Text} \rightarrow \text{Nat} \times \text{Nat}
\]

\[
\text{PAR} \quad \text{t:Text}
\]

\[
\text{DEF} \quad \text{or(t)} \; ? ;
\]

\[
\begin{align*}
\{1\}' & : \text{tl(t)} = \text{niltext AND hd(t)} = \text{empty} \; ? ; \\
& \quad \{0, 0\} \\
\{2\}' & : \text{tl(t)} = \text{niltext AND NOT} \; \text{hd(t)} = \text{empty} \; ? ; \\
& \quad \text{LET} \; \text{d1: Nat, d2: Nat; d1, d2 := reach(rest(t)); (d1, succ(d2))}
\end{align*}
\]

\[
\begin{align*}
\{3\}' & : \text{NOT} \; \text{tl(t)} = \text{niltext} \; ? ; \\
& \quad \text{LET} \; \text{d1: Nat, d2: Nat; d1, d2 := reach(tl(t)); (succ(d1), d2)}
\end{align*}
\]

Note that reach(\text{zero}) = (0,0). The function add defined below can be viewed as the 'addition' of two reaches. Sometimes we refer to it as natural addition. Let us give an example first. We show two texts and their reaches, and after that we determine the reach of their addition simply by adding the reaches and applying the reach operator.

\[
\text{let} \; t1 = ( \begin{array}{c}
\text{aaaaaaa} \\
\text{aaaaaaa} \\
\text{a}
\end{array} ), \text{ so reach}(t1) = (2,1)
\]

\[
\text{let} \; t2 = ( \begin{array}{c}
\text{bbbbbbb} \\
\text{bbbbbbb} \\
\text{b}
\end{array} ), \text{ so reach}(t2) = (2,1)
\]

\[
\text{add}(t1, t2) = ( \begin{array}{c}
\text{aaaaaaa} \\
\text{aaaaaaa} \\
\text{aaaaaaa} \\
\text{abbbbbbb} \\
\text{bbbbbbb} \\
\text{b}
\end{array} ), \text{ so reach}(\text{add}(t1, t2)) = (4,1)
\]

The interesting observation now is that we could have derived the value of reach(\text{add}(t1, t2)) also from the reaches of t1 and t2 alone. This is precisely the purpose of having the function add on reaches.

\[
\text{FUCN} \text{add} : \text{Nat} \times \text{Nat} \times \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \times \text{Nat}
\]

\[
\text{PAR} \quad c1: \text{Nat}, c2: \text{Nat}, \text{d1: Nat}, \text{d2: Nat}
\]

\[
\text{DEF} \quad \{ \text{d1 = 0} \; ? ; \text{c1, add(c2, d2)} \\
\text{NOT d1 = 0} \; ? ; \text{add(c1, d1), d2} \}
\]
4.3. TEXT AND OPERATIONS ON TEXT

Let us enter the data of the above example: \( \text{add}((2,1),(2,1)) = (4,1) \) indeed. Addition on reaches is associative and the pair \( (0,0) \) behaves as a left- and right neutral element with respect to \( \text{add} \) as can be shown by a case-analysis using the definition of \( \text{add} \). The following proposition confirms our intuition as sketched by the above example.

\[
\forall t,u : \text{Text} \ ( \text{ok}(t) \land \text{ok}(u) \Rightarrow \\
\text{reach}(\text{add}(t,u)) = \text{add}(	ext{reach}(t),\text{reach}(u)) )
\]

This can be shown by induction on \( t \) using the induction principle of Section 4.3.4 and where for the step \( t = \text{addchar}(c,t') \) one distinguishes two cases: either \( \text{len}(t) = \text{len}(u) = 1 \), so \( c \) contributes to the reach of the added texts, or \( \text{len}(t) + \text{len}(u) > 2 \), in which case \( c \) does not contribute to it. Now we have the \( \text{reach} \) function, we can also formulate the proposition that in a certain sense \( \text{split} \) is the inverse of \( \text{add} \).

\[
\forall t,u : \text{Text} \ ( \text{ok}(t) \land \text{ok}(u) \Rightarrow \\
\text{split}(\text{add}(t,u),\text{reach}(t)) = (t,u) )
\]

which can be shown by induction on \( t \) using the induction principle of Section 4.3.4. We can also perform splitting on reaches.

**FUNC** \( \text{split} : \text{Nat} \times \text{Nat} \times \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \times \text{Nat} \times \text{Nat} \times \text{Nat} \\
\text{PAR} r : \text{Nat}, s : \text{Nat}, i : \text{Nat}, j : \text{Nat} \\
\text{DEF} (i,j) \rightarrow (\text{sub}(r,i), (\text{ins}(i,r)?; s | i = r ?; \text{sub}(s,j)))
\]

This function \( \text{split} \) is useful for calculating the reaches of the texts obtained by splitting a text with a given reach when also the splitting-point is given.

\[
\forall t,t_1,t_2 : \text{Text}, c : \text{Nat} \times \text{Nat} \times \text{Nat} \times \text{Nat} \\
( \text{ok}(t) \land \text{ok}(t_1) \land \text{ok}(t_2) \land \text{split}(t,c) = (t_1,t_2) \Rightarrow \\
\text{split}(\text{reach}(t),c) = (\text{reach}(t_1),\text{reach}(t_2)) )
\]

which can be shown by referring to the definition of \( \text{split} \) on reaches and to the fact that \( \text{split}(t,c) \) yields the two texts \( t_1,t_2 \) that are obtained by splitting \( t \) with the splitting point at \( c \). Hence the reach of \( t_1 \) equals \( c \) and we can get the reach of \( t_2 \) by a simple case-analysis. In any case, the vertical co-ordinate of \( t_2 \) is the difference of the vertical co-ordinates of \( \text{reach}(t) \) and \( c \). If the splitting point is before the last line of \( t \), then the length of the last line of \( t_2 \) is simply the length of the last line of \( t \). If the splitting point is in the last line, then the horizontal co-ordinate of \( t_2 \) is the length of this last line minus the horizontal co-ordinate of the splitting point. We can also perform a kind of cut and paste on reaches.
CHAPTER 4. SPECIFICATION OF A TEXT EDITOR

\texttt{FUNC} \texttt{cut} : \texttt{Nat} \times \texttt{Nat} \times \texttt{Nat} \times \texttt{Nat} \times \texttt{Nat} \rightarrow \texttt{Nat} \times \texttt{Nat} \times \texttt{Nat} \times \texttt{Nat}

\texttt{PAR} \ t: \texttt{Nat} \times \texttt{Nat}, \ n: \texttt{Nat}, \ m: \texttt{Nat}, \ i: \texttt{Nat}, \ j: \texttt{Nat}

\texttt{DEF} \ \texttt{LET} \ \texttt{t1} : \texttt{Nat} \times \texttt{Nat}, \texttt{t2} : \texttt{Nat} \times \texttt{Nat}; \ \texttt{t1} \cdot \texttt{t2} := \texttt{split}(\texttt{t}, \texttt{i}, \texttt{j})

\texttt{LET} \ \texttt{t11} : \texttt{Nat} \times \texttt{Nat}, \texttt{t12} : \texttt{Nat} \times \texttt{Nat}; \ \texttt{t11} \cdot \texttt{t12} := \texttt{split}(\texttt{t11}, \texttt{n}, \texttt{m})

\texttt{add}(\texttt{t11} \cdot \texttt{t12}, \texttt{t12})

\texttt{FUNC} \texttt{paste} : \texttt{Nat} \times \texttt{Nat} \times \texttt{Nat} \times \texttt{Nat} \rightarrow \texttt{Nat} \times \texttt{Nat}

\texttt{PAR} \ t: \texttt{Nat} \times \texttt{Nat}, \ u: \texttt{Nat} \times \texttt{Nat}, \ k: \texttt{Nat}, \ l: \texttt{Nat}

\texttt{DEF} \ \texttt{LET} \ \texttt{t1} : \texttt{Nat} \times \texttt{Nat}, \texttt{t2} : \texttt{Nat} \times \texttt{Nat}; \ \texttt{t1} \cdot \texttt{t2} := \texttt{split}(\texttt{t}, \texttt{k}, \texttt{l})

\texttt{add}(\texttt{t1} \cdot \texttt{add}(\texttt{u}, \texttt{t2}))

This function \texttt{cut} is useful for calculating the reaches of the texts obtained by cutting a text with a given reach.

\[ \forall t, t_1, t_2 : \texttt{Text}, c, d : \texttt{Nat}^2 \]
\[ (\text{ok}(t) \land \text{ok}(t_1) \land \text{ok}(t_2) \land \text{cut}(t, c, d) = (t_1, t_2) ) \Rightarrow \text{cut}(\text{reach}(t), c, d) = (\text{reach}(t_1), \text{reach}(t_2)) \]

which follows from the fact that \texttt{add} and \texttt{split} on texts commute via \texttt{reach} with \texttt{add} and \texttt{split} on reaches, using the obvious similarity in both definitions of \texttt{cut}. In the same way one can show the following proposition, which expresses that the latter function \texttt{paste} can be used for calculating the reach of the text obtained by pasting two texts with given reaches.

\[ \forall t, u : \texttt{Text}, c : \texttt{Nat}^2 \]
\[ (\text{ok}(t) \land \text{ok}(u) ) \Rightarrow \text{reach}(\text{paste}(t, u, c)) = \text{paste}(\text{reach}(t), \text{reach}(u), c) \]

We can summarize some of the above propositions by saying that \texttt{reach} : \texttt{Text} \rightarrow \texttt{Nat}^2 is a homomorphic mapping from the algebra of texts to the algebra of reaches. So, we can view the algebra of reaches as a simplified version of the algebra of texts. In general, \texttt{reach}(t) does not contain enough information for reconstructing \texttt{t}, but still it tells us something about the behaviour of \texttt{t} under the application of \texttt{split}, \texttt{add}, \texttt{cut} and \texttt{paste} operations.

Let us formulate that in a certain sense \texttt{cut} is the inverse of \texttt{paste}.

\[ \forall t, u : \texttt{Text}, c : \texttt{Nat}^2 \]
\[ (\text{ok}(t) \land \text{ok}(u) \land \texttt{paste}(t, u, c)! ) \Rightarrow \text{cut}(\text{paste}(t, u, c), c, \texttt{add}(c, \texttt{reach}(u))) = (t, u) \]

which follows from the definitions of \texttt{cut} and \texttt{paste} and from the homomorphism property of \texttt{reach}. Sometimes it is necessary to compare two reaches, so we introduce predicates \texttt{lls} : \texttt{Nat}^2 \neq \texttt{Nat}^2 and \texttt{leq} : \texttt{Nat}^2 \neq \texttt{Nat}^2.
4.3. TEXT AND OPERATIONS ON TEXT

\[ \text{PRED \ lss: \ Nat \#\ Nat \#\ Nat \#\ Nat} \]
\[ \text{PAR\ i1: Nat, i2: Nat, j1: Nat, j2: Nat} \]
\[ \text{DEF\ lss(i1, j1) OR (i1 = j1 AND lss(i2, j2))} \]

\[ \text{PRED \ leq: \ Nat \#\ Nat \#\ Nat \#\ Nat} \]
\[ \text{PAR\ i1: Nat, i2: Nat, j1: Nat, j2: Nat} \]
\[ \text{DEF\ lss(i1, j1) OR (i1 = j1 AND leq(i2, j2))} \]

Notice that a position in a given text can be described precisely by the reach of the text preceding this position; hence reaches can also be viewed as position indicators.
So if \( \text{lss}((c_1, c_2), (d_1, d_2)) \), then this can be interpreted as 'the position \((c_1, c_2)\) comes before the position \((d_1, d_2)\)'.

\[ \text{END\ \{\text{TEXT\_OPSG2\_SPEC}\}} \]

Thus \text{TEXT\_OPSG2\_SPEC} adds to the notions of texts and strings of Sections 4.3.2 and 4.3.3 several conversion operations (Sections 4.3.4 and 4.3.5), a range of adding and splitting operations (Section 4.3.6) and their images under the operation reach (Section 4.3.7).

4.3.8 The Profile of a Text

When we use co-ordinate pairs for indicating positions in a text – as we do in split, cut and paste – then we have to be careful. The point is that our co-ordinate pairs are based on natural numbers and thus may indicate positions that are non-existing in a given text. We develop some machinery so that later, in the context of a text editor, we can deal with these problems formally. This will happen in Section 4.5.5, when we shall postulate a so-called text-invariant. If we have a text \( t \), then we might collect all information that is relevant for 'addressing' in \( t \). For this purpose we introduce so-called profiles. The profile of a text \( t \) is a sequence of natural numbers, one for each line of \( t \) such that the \( i \)-th element in the sequence is precisely equal to the length of the \( i \)-th line in \( t \). We start with an example and then do the formalisation.

\[
\text{profile(} \text{aaa\aaa\aaa} \text{\}) = \langle 8, 7, 1 \rangle
\]

Now we formally introduce a sort \text{Profile}. An object of sort \text{Profile} is nothing but a sequence \text{(Seq)} of natural numbers \text{(Nat)}.
LET PROFILE :=
APPLY RENAME
    SORT Seq TO Profile,
    SORT Nat TO Nat,
    FUNC empty: -> Seq TO nil_profile
IN SEQ_SPEC TO NAT_SPEC;

And of course we must define the profile corresponding with a given text,
which we do by introducing a function profile. Therefore we write another
class description called PROFILE_SPEC which imports a number of class
descriptions introduced before. We want to add a remark about such imports
here. In general, when writing specifications, we need not adopt some
minimality principle. Instead of that we begin each new class description with
a relatively rich collection of imports – so that we have enough notations
at our disposal. There is no harm in importing too much, whereas it is a
nuisance if the imports turn out to be insufficient.

LET PROFILE_SPEC :=
IMPORT PROFILE INTO
IMPORT NAT_SPEC INTO
IMPORT CHAR_SPEC INTO
IMPORT LINE_SPEC INTO
IMPORT TEXT_SPEC INTO
IMPORT TEXT_GPS1_SPEC INTO
IMPORT TEXT_GPS2_SPEC INTO
CLASS

    FUNC profile: Text -> Profile
    PAR t:Text
    DEF ok(t) ?:
        \{ 1\} hd(t) = empty AND tl(t) = niltext ?;
            cons(0,nil_profile)
        | \{ 2\} hd(t) = empty AND NOT tl(t) = niltext ?;
            cons(0,profile(tl(t)))
        | \{ 3\} NOT hd(t) = empty ?;
            LET p:Profile: p := profile(rest(t));
            cons(succ(hd(p)).tl(p))
    \}

END;

It is possible to define a kind of addition called add and operations like
split, cut and paste on profiles. For suitable choice of these operations,
profile: Text -> Nat' becomes a homomorphic mapping from the algebra
of texts to the algebra of profiles. Furthermore one can define a function
reach: \( \text{Nat}^* \rightarrow \text{Nat}^2 \) such that it is a homomorphic mapping from the
algebra of profiles to the algebra of reaches. The following picture sketches
this situation.

![Diagram](image)

**Fig 4.1.** The algebras of texts, profiles and reaches.

So, we can view the algebra of profiles as being intermediate between the
algebra of texts and the algebra of reaches. We leave these things for later
investigation, because we do not need them really yet.

```plaintext
LET TEXT_OPS3_SPEC :=
IMPORT NAT_SPEC INTO
IMPORT CHAR_SPEC INTO
IMPORT LINE_SPEC INTO
IMPORT TEXT_SPEC INTO
IMPORT TEXT_OPS1_SPEC INTO
IMPORT TEXT_OPS2_SPEC INTO
IMPORT STRING_SPEC INTO
IMPORT PROFILE_SPEC INTO
CLASS

We define a predicate called `intext`, thereby showing a typical application
of profiles. We use the standard selection function `sel`. If we have a given
text \( t \) and a co-ordinate pair \( (i, j) \) then `intext(t, i, j)` holds if either \( (i, j) \)
indicates a position that exists in \( t \) or \( (i, j) \) corresponds with the very end
of a line – which is an acceptable position for splitting a text into parts, and
therefore for pasting by the function `paste`.

```plaintext
PRED intext: Text \# Nat \# Nat
PAR t:Text, i:Nat, j:Nat
DEF LET m:Nat, n:Nat; m := reach(t);
     leq(i, m) AND leq(j, sel(profile(t), i))
```
In particular, we have `intext(zero,(0,0))`.

### 4.3.9 Vertical and Horizontal Composition of Text

In this section we introduce two ways of adding texts for which we want to use the names *vertical addition* and *horizontal addition*. These operations become important when one wants to describe multi-window capabilities of a text editor. Although multi-window capabilities are not included in the text editor described in this chapter, we believe that it is instructive to have a look at the necessary operations at an algebraic level.

We introduce an operation called `v_add` below and we refer to it as *vertical addition*. Actually there is nothing new here since vertical addition is nothing but *concatenation*.

![Diagram of vertical addition]

We give the formal definition below.

```
FUNC v_add: Text # Text -> Text
PAR t1:Text, t2:Text
DEF cat(t1,t2)
```

We state a few well-known properties. Vertical addition is associative.

\[
\forall t_1, t_2, t_3 : \text{Text} \\
\quad ( v_{\text{add}}(t_1, v_{\text{add}}(t_2, t_3)) = v_{\text{add}}(v_{\text{add}}(t_1, t_2), t_3) )
\]

`niltext` behaves both as a left and a right neutral element with respect to the operation `v_add`.

\[
\forall t : \text{Text} \\
\quad ( v_{\text{add}}(t, \text{niltext}) = t \land v_{\text{add}}(\text{niltext}, t) = t )
\]

The `len` function from texts to natural numbers behaves homomorphically.
as stated below.

\[ \forall t_1, t_2 : \text{Text} \]
\[ \left( \text{len}(v \_\text{add}(t_1, t_2)) = \text{add}(\text{len}(t_1), \text{len}(t_2)) \right) \]

We introduce another operation called \( \text{h\_add} \) below and we refer to it as horizontal addition.

We give the formal definition below.

\[
\text{FUNC} \quad \text{h\_add} : \text{Text} \# \text{Text} \rightarrow \text{Text} \\
\text{PAR} \quad t_1 : \text{Text}, \ t_2 : \text{Text} \\
\text{DEF} \quad \left( t_1 = \text{niltext} \ \text{AND} \ t_2 = \text{niltext} \quad \Rightarrow \quad \text{niltext} \right) \\
\left( \text{NOT} \ (t_1 = \text{niltext} \ \text{AND} \ t_2 = \text{niltext}) \quad \Rightarrow \quad \\
\text{cons} (\text{cat} (\text{hd}(t_1), \text{hd}(t_2)), \text{h\_add}(\text{tl}(t_1), \text{tl}(t_2))) \right)
\]

From this definition it can be seen that when two texts have different lengths, their horizontal addition is undefined. Horizontal addition is associative.

\[ \forall t_1, t_2, t_3 : \text{Text} \left( \text{len}(t_1) = \text{len}(t_2) = \text{len}(t_3) \Rightarrow \right) \\
\text{h\_add}(t_1, \text{h\_add}(t_2, t_3)) = \text{h\_add}(\text{h\_add}(t_1, t_2), t_3) \]

which follows by induction on the length of the texts involved and using the well-known associativity of list-concatenation. For each length \( i \), there is a left- and right neutral element and we introduce the notation \( \text{empties}(i) \) for it:

\[
\text{FUNC} \quad \text{empties} : \text{Nat} \rightarrow \text{Text} \\
\text{PAR} \quad n : \text{Nat} \\
\text{DEF} \quad \left( n = 0?; \quad \text{niltext} \right) \\
\left( \text{NOT} \ n = 0?; \quad \text{cons} (\text{empty}, \text{empties}(\text{pred}(n))) \right)
\]
so we can denote this property as follows:

\[ \forall l : \text{Text}, l : \text{Nat} \ (\text{len}(l) = l \Rightarrow \text{h\_add}(t, \text{empties}(l)) = t \wedge \text{h\_add}(\text{empties}(l), t) = t) \]

which follows again by induction on \( l \), using the fact that \text{empties} is neutral with respect to concatenation. The set of \text{texts} of a given length is closed under horizontal addition:

\[ \forall l, l_1, l_2 : \text{Text}, l : \text{Nat} \ (\text{len}(l_1) = \text{len}(l_2) = l \Rightarrow \text{len}(\text{h\_add}(l_1, l_2)) = l) \]

which follows again by induction on \( l \). Horizontal and vertical addition are related by the interchange law. We have met this interchange law before at another level, viz. at the level of composition mechanisms for designs. We refer to Chapter 3 lemma 3.3.11 and remark 3.3.12

\[ \forall l, l_1, l_2, l_3, l_4 : \text{Text} \ (\text{len}(l_1) = \text{len}(l_2) \wedge \text{len}(l_3) = \text{len}(l_4) \Rightarrow \text{v\_add}(\text{h\_add}(l_1, l_2), \text{h\_add}(l_3, l_4)) = \text{h\_add}(\text{v\_add}(l_1, l_2), \text{v\_add}(l_3, l_4))) \]

which follows by induction on \( \text{len}(l_1) \).

### 4.3.10 Operations for Searching

In this section we do some preparations such that later we can specify the search-command of an editor. The formalisation of searching is done in terms of strings, which is most natural. In typical editing sessions one searches for a word or for a short term. However, there is no clear intuition for searching an entire sentence or an entire formula that extends over more than one line. So we describe searching such that it becomes conceptually clear for \text{ctrl}\_\text{j}-less strings. We adopt the same formalisation then for all strings. After that, we transfer the formalisation by homomorphic mappings to the algebra of texts. We introduce a predicate \text{match} such that \text{match}(t, s, i) holds in the situation where in string \( t \) there is an occurrence of a substring \( s \) at position \( i \).

\[ \text{FRED match} : \text{String} \# \text{String} \# \text{Nat} \]

\[ \text{PAR} \ t : \text{String}, s : \text{String}, i : \text{Nat} \]

\[ \text{DEF} \ s = \text{empty} \ \text{OR} \]

\[ \text{hd}(s) = \text{sel}(t, i) \ \text{AND} \ \text{match}(t, t_1(s), \text{succ}(i)) \]

By way of preparation for a kind of sentinel technique, we introduce the following slightly modified version of \text{match}, which we call \text{match}’.
4.3. TEXT AND OPERATIONS ON TEXT

PRED match': String # String # Nat
PAR t: String, s: String, i: Nat
DEF i = len(t) OR match(t, s, i)

Now we define the result of a search operation as the least position at which
a match – in the sense of match’ – occurs.

FUNC search: String # String -> Nat
PAR t: String, s: String
DEF SOME i: Nat
   { match'(t, s, i) AND
     FORALL j: Nat ( match'(t, s, j) => leq(i, j) ) } )

Note that if in t there is no occurrence of s, then search(t, s) ‘falls through’
in the sense that it yields len(t), which is the sentinel technique announced
before. Now we transfer the above formalisation to the algebra of texts by
employing the mappings string and text.

FUNC search: Text # Text -> Nat # Nat
PAR t: Text, s: Text
DEF LET t1: String, t2: String:
   t1, t2 := split(string(t), search(string(t), string(s)));
   reach(text(t1))

where we used an obvious split operation on strings.

FUNC split: String # Nat -> String # String
PAR s: String, i: Nat
DEF ( i = 0 .
   empty, s
| NOT i = 0 .:
   LET t1: String, t2: String; t1, t2 := split(t1(s), pred(i));
   cons(hd(s), t1), t2
   )

The approach followed here could be considered as a part of our methodology
of writing formal specifications: choose among the different possible represen-
tations the one which yields the simplest and most elegant definitions.
If the remainder of the specification is in terms of another representation,
just use suitable mappings to transfer the results from one representation to
another.

END: {of TEXT_OPS3_SPEC}

Thus, TEXT_OPS3_SPEC adds to TEXT_OPS2_SPEC some operations for pro-
filing texts (Section 4.3.8), for vertical and horizontal composition of texts (Section 4.3.9) and for searching (Section 4.3.10).

4.3.11 Procrustean Operations

In this section we shall introduce some machinery for reducing the size of texts that are too large for some purpose and also some machinery for stretching texts that are too small. However, we not stick to this Procrustean terminology and later in this section we introduce operations called look and fill where the names of the operations have been derived from their typical applications.

One of the most important problems associated with the specification and design of an editor is the fact that in general the text being edited is too large to fit on the screen of the display. The text itself may be too long and some of the lines may be too wide. One – currently rather obsolete – solution is to make the editor line-oriented. This solves the problem of the text being too long and then the problem of the lines being too wide can be solved by e.g. wrap-around. A better solution is adopted in the so-called display-oriented editors where the screen contains one or more rectangular windows. These windows can be used for 'looking' to the text(s) being edited. In order to keep things simple, we restrict ourselves to the case where there is just one window. This window corresponds with a 'subtext' of the text being edited. The cursor on the screen corresponds with the 'current position' (= dot) in the text being edited. The editor maintains a kind of 'window-invariant' viz. that always this dot and a suitable subtext surrounding the dot are visible in the window.

A second important problem is that the text may be shorter than the vertical size of the window; also some of the lines in the text may be shorter than the horizontal size of the window. The obvious solution is filling, by which we mean the positions within the window for which there is no character in the text are displayed as blank.

In this chapter we choose to discuss a display-oriented editor, but as before, we postpone the description of the actual editor and we develop some machinery for formulating the ideas of looking and filling as mentioned above. This machinery takes the shape of a collection of algebraic operations on texts. We start with filling and we put everything dealing with filling in a class description called FILL_SPEC. Again we begin with an ample collection of imports.

\[
\text{LET FILL_SPEC := IMPORT NAT_SPEC INTO}
\]
We have to provide for filling in both vertical and horizontal directions. This observation leads us directly to two functions vfill and hfill. The function vfill serves for filling in the vertical direction. Somehow it seems natural to choose our definitions such that the text gets ‘upwards adjusted’, so that filling takes place at the end of the text.

We give the formal definition of vfill below. We can use the function empties which we introduced in Section 4.3.9. Recall that empties(l) is the text having precisely l empty lines.

```
FNC vfill: Text # Nat -> Text
PAR t:Text, n:Nat
DEF LET i:Nat; i := len(t);
   LET te:Text;
   te := empties( geq(n,i)?; sub(n,i) | lss(n,i)?; 0 );
cat(t,te)
```

The function hfill serves for filling in the horizontal direction. As an auxiliary for the description of hfill we have a procedure which describes the construction of lines consisting of blanks only. We refer to the constant blank for the ‘ ’ character which we introduced earlier.

```
FNC blanks: Nat -> Line
PAR j:Nat
```
DEF ( j = 0 ?; empty
       | NOT j = 0 ?; cons(blank, blanks(pred(j)))
     )

Somehow it seems natural to choose our definitions such that the text gets 'left adjusted', so that filling takes place at the end of the lines.

We give the formal definition of hfill below.

FUNC hfill: Text # Nat -> Text
PAR t:Text, n:Nat
DEF ( t = niltext ?;
       niltext
     |
     NOT t = niltext ?;
     LET l:Line; l := hd(t);
     LET i:Nat; i := len(l);
     LET j:Nat; j := ( geq(n,i)?; sub(n,i) | lseq(n,i)?; 0 );
     cons(cat(l, blanks(j)), hfill(tl(t), n))
   )

And we combine these to get a function fill which does both vertical and horizontal filling.

Note that vfill must be applied first, so that a subsequent application of hfill will make sure that the empties introduced by vfill get the desired length. It just does not work the other way around. We give the formal definition of fill below.
4.3. TEXT AND OPERATIONS ON TEXT

FUCG fill: Text # Nat # Nat -> Text
PAR t:Text, i:Nat, j:Nat
DEF hfill(vfill(t,i),j)

END; {of FILL_SPEC}

This completes our formalisation of filling and as a next step we introduce some machinery for the description of looking. As stated before, the application we have in mind is to describe how a window is used for ‘looking’ to a text. We put everything related to looking in a class description called LOOK_SPEC.

LET LOOK_SPEC :=
IMPORT NAT_SPEC INTO
IMPORT CHAR_SPEC INTO
IMPORT LINE_SPEC INTO
IMPORT TEXT_SPEC INTO
IMPORT TEXT_OPS1_SPEC INTO
IMPORT TEXT_OPS2_SPEC INTO
IMPORT TEXT_OPS3_SPEC INTO
IMPORT STRING_SPEC INTO
IMPORT PROFILE_SPEC INTO
CLASS

To begin with, we have again an auxiliary. It describes the splitting at line-level.

FUCG hsplit: Line # Nat -> Line # Line
PAR s:Line, n:Nat
DEF \( \begin{align*}
  (n = 0) \Rightarrow \\
  \text{empty}, \
  \text{n NOT n = 0?} \\
  \text{LET s:Line, s2:Line; s1.s2 := hsplit(tl(s),pred(n));} \\
  \text{(cons(hd(s),s1)).s2} \\
\end{align*} \)

The way we define the algebraic operations for ‘looking’ is somewhat similar to the way we defined the cut operation before. There it turned out that it was convenient to define the more elementary split operation first. We have to provide for splitting both in horizontal and vertical direction. These considerations lead us directly to two functions hsplit and vsplit. The function hsplit serves for splitting in the horizontal direction.
We give the formal definition of hasplit below. We use recursion for propagating the effect of hasplit: \text{Line} \rightarrow \text{Line}^2 \text{ over an entire text.}

\textbf{FUNC hasplit:} \text{Text} \ # \ \text{Nat} \rightarrow \text{Text} \ # \ \text{Text}
\textbf{PAR} \ t: \text{Text}, \ n: \text{Nat}
\textbf{DEF} \ (t = \text{niltext} ?; \ \text{niltext}, \ \text{niltext}
\hline
| \ NOT n = \text{niltext} ?; 
\hline
| \ \text{LET} l1: \text{Line}, l2: \text{Line}; \ l1, l2 := \text{hasplit}(\text{hd}(t), n); 
\hline
| \ \text{LET} t1: \text{Text}, t2: \text{Text}; \ t1, t2 := \text{hasplit}(t1(t), n); 
\hline
| \ \text{cons}(l1, t1), \ \text{cons}(l2, t2) 
\hline
) 
\hline

Note that \text{h_add} is the inverse of \text{hasplit}. The function \text{vsplit} serves for splitting in the vertical direction.

\textbf{FUNC vsplit:} \text{Text} \ # \ \text{Nat} \rightarrow \text{Text} \ # \ \text{Text}
\textbf{PAR} \ t: \text{Text}, \ n: \text{Nat}
\textbf{DEF} \ (n = 0 ?; \ \text{niltext}, t
\hline
| \ NOT n = 0 ?; 
\hline
| \ \text{LET} t1: \text{Text}, t2: \text{Text}; \ t1, t2 := \text{vsplit}(t1(t), \text{pred}(n)); 
\hline
| \ \text{cons}(\text{hd}(t), t1), \ t2 
\hline
) 
\hline

Note that \text{v_add} is the inverse of \text{vsplit}. Splitting horizontally twice yields what we call a horizontal look (function hlook).
We give the formal definition of \texttt{hlook} below.

\begin{verbatim}
FUNC hlook: Text # Nat # Nat -> Text
PAR t:Text, j1:Nat, j2:Nat
DEF LET t1:Text, t2:Text; t1 ,t2 := hsplit(t,j2);
    LET t11:Text,t12:Text; t11,t12 := hsplit(t1,j1);
    t12
\end{verbatim}

Splitting vertically twice yields what we call a vertical look (function \texttt{vlook}).

\begin{verbatim}
FUNC vlook: Text # Nat # Nat -> Text
PAR t:Text, j1:Nat, j2:Nat
DEF LET t1:Text, t2:Text; t1 ,t2 := vsplit(t,j2);
    LET t11:Text,t12:Text; t11,t12 := vsplit(t1,j1);
    t12
\end{verbatim}

We give the formal definition of \texttt{vlook} below.

And we combine them in an obvious way to get a function \texttt{look} which performs both a vertical and a horizontal 'subsetting' of a given text. \texttt{look}(i,c,d) will denote the contents of a window whose leftmost uppermost corner is at \texttt{c} and whose rightmost lowermost corner is determined by \texttt{d}.
We give the formal definition of `look` below. We prefer to define it such that the application of `vlook` is done first, followed by `hlook` because in this way the result of `look` is more often defined.

```plaintext
F func look : Text # Nat # Nat # Nat # Nat -> Text
par t : Text, i1 : Nat, i2 : Nat, j1 : Nat, j2 : Nat
def hlook(vlook(t, i1, j1), i2, j2)
end
```

Both class descriptions `FILL_SPEC` and `LOOK_SPEC` are intended to play a role in formulating precisely how the text being edited should be shown on the screen of a display device. This will be done in Section 4.5.12.

### 4.4 Interfacing an Editor with its Environment

#### 4.4.1 Introduction

In this section we describe several class descriptions that are important for interfacing the editor with its environment. First of all we construct our own model of a physical display device. This is the subject of Section 4.4.2. After that we introduce special kinds of sequences and strings. This is the subject of Section 4.4.3. Finally we describe a simple file-system, which is the subject of Section 4.4.4.

#### 4.4.2 DISPLAY: an Abstract Display

In this section we shall look for a formal description of a *display device*. The device that we have in mind is a classical display of the so-called video-display-unit type – a VT102 or so, say. We do not investigate all the details of
one or more concrete displays and instead we start with an abstract display. We abstract from the character-sequences that should be sent to the display for 'clear screen', 'cursor motion' etc. In the abstract display the sending of these concrete character sequences is replaced by abstract commands cl, cm etc. The choice of the names and the functionality of the abstract commands have been inspired by the so-called termcap facility (abbreviating terminal capabilities) as used on UNIX systems. We decided that we shall not try to include all possible features and capabilities that various displays may have. Instead we restrict our description to those display capabilities that are minimally required by the EMACS editor described in \cite{4}.

The corresponding class description is called DISPLAY_SPEC and it is given below. There are two variable functions which span the state-space. The first function is called screen and it corresponds with the observable contents on the screen of the display, which is the text that you see when you sit in front of the display. The screen has a size given by two constants: 11 and co, abbreviating lines and columns.

![Fig 4.2 The window of the display.](image)

The second function is called cursor and it corresponds with the observable position of the cursor on the screen of the display. There are a few operations that are not considered executable, but which we export just for reasoning purposes. In particular, these are the procedure display_op and the functions screen and cursor.

\begin{verbatim}
LET DISPLAY_SPEC :=
EXPORT
  SORT Char,
  SORT Nat,
  SORT Text,
  FUNC li: -> Nat,
  FUNC co: -> Nat,
  PROC cr: -> ,
  PROC nl: -> ,
  PROC bc: -> ,
  PROC ce: -> .
\end{verbatim}
PROC cl: -> ,
PROC nd: -> ,
PROC up: -> ,
PROC cm: Nat # Nat -> ,
PROC print: Char -> .
PROC displ_op: -> ,
FUNC screen: -> Text ,
FUNC cursor: -> Nat # Nat

FROM
IMPORT NAT_SPEC INTO
IMPORT CHAR_SPEC INTO
IMPORT LINE_SPEC INTO
IMPORT TEXT_SPEC INTO
IMPORT TEXT_OPS1_SPEC INTO
IMPORT TEXT_OPS2_SPEC INTO
IMPORT STRING_SPEC INTO
IMPORT PROFILE_SPEC INTO
IMPORT FILL_SPEC INTO

CLASS

FUNC screen: -> Text VAR
FUNC cursor: -> Nat # Nat VAR

FUNC li: -> Nat % number of lines
FUNC co: -> Nat % number of columns

PROC cr: -> MOD screen, cursor % carriage return
PROC nl: -> MOD screen, cursor % newline
PROC bc: -> MOD screen, cursor % backwards cursor
PROC ce: -> MOD screen, cursor % clear to end-of-line
PROC cl: -> MOD screen, cursor % erase display
PROC nd: -> MOD screen, cursor % move cursor right
PROC up: -> MOD screen, cursor % move cursor up
PROC cm: Nat # Nat -> MOD screen, cursor % cursor motion
PROC print: Char -> MOD screen, cursor % character processing

Of course we have to specify the effect of the procedures upon the screen (function screen) and upon the cursor (function cursor). This effect is such that a certain invariant is maintained and we refer to it as display-invariant. We believe that it is a good methodological principle to formulate this invariant first. We introduce predicates D11, D12 and D13.

- D11 states that the screen is rectangular and that its size is given by li and co.
4.4. INTERFACING WITH THE ENVIRONMENT

- DI12 states that the cursor is on the screen.
- DI13 states that the screen contains only printable characters.

We add one remark related to DI13. Of course we know that many text editors offer the possibility of entering non-printable characters into texts and that such characters still somehow get displayed on the screen (e.g., as tilde), but this matter is not settled within the display but within the editor. We use the predicate printable on Char from Section 4.3.5.

**PRED DI1:**
DEF len(screen) = li AND
FORALL i:Nat ( len(i,li) =>
(len(scl(screen,i)) = co ) )

**PRED DI2:**
DEF LET c1:Nat,c2:Nat; c1,c2 := cursor;
len(c1,li) AND len(c2,co)

**PRED DI3:**
DEF FORALL i:Nat,j:Nat
(len(i,li) AND len(j,co) => printable(scl(screen,i),j))

The invariance of DI1 ∧ DI2 ∧ DI3 can be expressed as follows:

**PROC displ_op:** ->
DEF ( cr
| nl
| bc
| ce
| e1
| md
| up
| cm(SOME i:Nat,j:Nat ()
| print(SOME c:Char (printable(c)))
)

**AXIOM INIT => DI1 AND DI2 AND DI3**
**AXIOM DI1 AND DI2 AND DI3 =>
[ displ_op ] DI1 AND DI2 AND DI3**

Next we turn our attention to the semantical description of the display operations which amounts to giving the axioms of the CLASS ... END part of DISPLAY_SPEC. To begin with, we have definedness axioms and termination axioms.
AXIOM \text{geq}(11,1) \text{ AND } \text{geq}(co,1)

AXIOM \langle cr \rangle > \text{TRUE};
\langle nl \rangle > \text{TRUE};
\langle bc \rangle > \text{TRUE};
\langle ce \rangle > \text{TRUE};
\langle cl \rangle > \text{TRUE};
\langle nd \rangle > \text{TRUE};
\langle up \rangle > \text{TRUE}

AXIOM \text{FORALL } i:\text{Nat}, j:\text{Nat}
\langle \text{cm}(i,j) \rangle > \text{TRUE}

AXIOM \text{FORALL } c:\text{Char}
\langle \text{printable}(c) \Rightarrow
\langle \text{print}(c) \rangle > \text{TRUE} \rangle

The effect of each procedure is described by a postcondition. Before giving
these postconditions, we introduce by overloading two simple auxiliary func-
tions called \text{blank_text}. The first function called \text{blank_text} yields one-line
texts consisting of blank characters only. The second function \text{blank_text}
takes two arguments \text{i}, \text{j} say, and then it yields the text with blanks of dimen-
sions \text{i} and \text{j}. We use the function \text{blanks}: \text{Nat -> Line} from \text{FILL_SPEC}.

\text{FUNC} \text{blank_text}: \text{Nat -> Text}
\text{PAR} n: \text{Nat}
\text{DEF} \text{cons(\text{blanks}(n)),niltext)}

\text{FUNC} \text{blank_text}: \text{Nat # Nat -> Text}
\text{PAR} i: \text{Nat}, j: \text{Nat}
\text{DEF} \langle \begin{array}{l}
\text{i = 0?; niltext}
\text{\mid NOT i = 0?; cons(\text{blanks}(j)),\text{blank_text}(\text{pred}(i),j)}
\end{array} \rangle

We introduce several postcondition predicates, one for each procedure. These
predicates have formal parameters \text{s}' for the previous screen, \text{c}' for the
previous cursor, \text{s} for screen and \text{c} for cursor. The first predicate describes
the effect of 'carriage return'. This case is simple: the second co-ordinate of
the cursor becomes 0.

\text{PRED} \text{post_cr: Text # Nat # Nat # Text # Nat # Nat}
\text{PAR} s': \text{Text}, c': \text{Nat # Nat}, s: \text{Text}, c: \text{Nat # Nat}
\text{DEF} \text{LET } i: \text{Nat}, j: \text{Nat}; i, j := c';
\text{c} = (i, 0);
\text{s} = s'
After we shall have defined this (and several similar) predicates, we state axiomatically that \texttt{post-cr} is the postcondition of \texttt{cr} indeed. We shall not write all axioms in full detail, since it should be clear from the axiom for \texttt{cr} what the remaining axioms should look like.

The second predicate describes what happens if a 'new-line' is sent to the display. At first sight this case seems simple too: the first co-ordinate of the cursor must be incremented by 1. However there is a complication because the cursor may already be on the last line of the screen. In the latter situation so-called scrolling takes place, i.e. one line of blanks is added to the end of the screen, whereas the first line of the screen is removed.

\begin{verbatim}
PRED post_ml: Text # Nat # Nat # Text # Nat # Nat
PAR s':Text, c':Nat # Nat, s:Text, c: Nat # Nat
DEF LET i: Nat, j: Nat; i, j := c';
    c = (( lns(i,pred(li))? succ(i) | i = pred(li)? i,j) AND
         s = ( lns(i,pred(li))? s' | i = pred(li) ? t1(v.add(s',blank_text(co))))

The predicates for 'backwards cursor', 'clear to end-of-line', 'erase display', 'move cursor right', 'move cursor up' and 'cursor motion' speak for themselves.

PRED post_bc: Text # Nat # Nat # Text # Nat # Nat
PAR s':Text, c':Nat # Nat, s:Text, c: Nat # Nat
DEF LET i: Nat, j: Nat; i, j := c';
    s = s';
    c = (i,(gte(j,0)? pred(j) | j = 0?: 0))

PRED post_ce: Text # Nat # Nat # Text # Nat # Nat
PAR s':Text, c':Nat # Nat, s:Text, c: Nat # Nat
DEF LET i: Nat, j: Nat; i, j := c';
    LET tt:Text, kk:Text; tt, kk := cut(s',i,j,i,co);
    s = paste(tt,blank_text(sub(co,j)),i,j);
    c = c'

PRED post_cl: Text # Nat # Nat # Text # Nat # Nat
PAR s':Text, c':Nat # Nat, s:Text, c: Nat # Nat
DEF s = blank_text(li,co);
    c = (0,0)

PRED post_nd: Text # Nat # Nat # Text # Nat # Nat
PAR s':Text, c':Nat # Nat, s:Text, c: Nat # Nat
DEF LET i: Nat, j: Nat; i, j := c';
\end{verbatim}
\[c = (i, \text{lss}(j, \text{pred}(co)) \vee \text{succ}(j) \mid j = \text{pred}(co)) \vee (j))\]

\[s = s'\]

**PRED post.up:** Text # Nat # Nat # Text # Nat # Nat

**PAR s': Text, c': Nat # Nat, s: Text, c: Nat # Nat**

**DEF LET i: Nat, j: Nat; i,j := c':**

\[c = ((\text{gtr}(i, 0) \vee \text{pred}(i) \mid i = 0 \vee 0), j);\]

\[s = s'\]

**PRED post.cn:** Text # Nat # Nat # Nat # Nat # Text # Nat # Nat # Nat

**PAR s': Text, c': Nat # Nat, i: Nat, j: Nat, s: Text, c: Nat # Nat**

**DEF s = s':**

\[c = (\text{lss}(i, 1, i); i \mid \text{geq}(i, 1, i); \text{pred}(i));\]

\[(\text{lss}(j, c); j \mid \text{geq}(j, c); \text{pred}(c));\]

And finally we have one important predicate dealing with printable-character processing.

**PRED post.print:** Text # Nat # Nat # Char # Text # Nat # Nat

**PAR s': Text, c': Nat # Nat, c: Char, s: Text, c: Nat # Nat**

**DEF printable(c) :=**

\[
(\text{LET } i: \text{Nat}, j: \text{Nat}; i,j := c';\]

\[
(\text{LET } tt: \text{Text}, kk: \text{Text}; tt, kk := \text{cut}(s', i, j, \text{succ}(j));\]

\[
c = (i, (\text{lss}(j, \text{pred}(co)) \vee \text{succ}(j)) \mid j \text{pred}(co)) \vee (j));\]

\[s = \text{paste}(tt, \text{addchar}(c, \text{zero}), i, j))\)

We state axiomatically that post_cr is the postcondition of cr.

**AXIOM (cr)**

**FORALL screen': Text, cursor': Nat # Nat**

**('screen = screen' AND cursor = cursor' \implies**

**[ cr ] post.cr(screen', cursor', screen, cursor) )**

We do not write all the axioms in full detail, since it should be clear from the axiom for `cr` what they should look like.

% **AXIOM nl.bc.ce.cl.nd.up as cr**

**AXIOM (cnw) FORALL screen': Text, cursor': Nat # Nat, i:Nat, j:Nat**

**('screen = screen' AND cursor = cursor' \implies**

**[ cn(i, j) ] post.cn(screen', cursor', i, j, screen, cursor) )**

**AXIOM (print) FORALL screen': Text, cursor': Nat # Nat, c:Char**

**('screen = screen' AND cursor = cursor' \implies**
[ print(c) ] post_print(screen',cursor',c,screen,cursor)

END;

Now the invariance of the display-invariant under the display operations
should be derivable in the sense that DI1 \land DI2 \land DI3 \Rightarrow [ displ_op ]
DI1 \land DI2 \land DI3. This can be shown by verifying its invariance under each
operation separately. By way of example, we shall present this for one of
them and we choose cr.

Assume that in a given state DI1 \land DI3 \land DI3 holds and that in this state we
have executed cr. It is important to observe that DI1 and DI3 depend only
on the screen and not on the cursor. DI2 at its turn depends only on the
cursor and not on the screen. From the axiom about cr and the postcondition
predicate of cr, we can see that in this new state screen = PREV screen
and this immediately establishes DI1 and DI3. So now we have to check for
DI2. The vertical co-ordinate of the cursor has kept its old value (in the
postcondition predicate denoted as 1) which thus is less than 11, as was to
be shown. Finally we consider the horizontal co-ordinate of the cursor. We
see that it has become 0, so we should check 0 < co which holds by the
axiom saying 11 \geq 1 \land co \geq 1.

This concludes the description of the abstract display. In the specification
of the editor, we shall have to describe the relation between the text being
edited and the dot on one hand and the screen and the cursor of the display
on the other hand. We introduce such a relation, called the window-invariant
in Section 4.5.12.

4.4.3 'SEQ' and 'STRING': Data types for Interfacing

Before we can specify editing operations, we must decide about the way in
which we have to invoke our editing operations. It is hard to avoid that these
operations will take arguments and we must be precise about the types of
these arguments. A frequently occurring data type is the sort of strings, or
more precisely, some data type representing strings.

In our window-and-text facility to be described in Sections 4.5.1-4.5.12 we
shall introduce so-called buffers which have names associated with them and
we shall need some kind of strings for that. For example, there will be a
procedure yank_buffer which takes a string argument for addressing a buffer
to be 'yanked' and a procedure current_buffer_name which yields a string.
Also certain operations for file handling will deal with string arguments.
Now we want to be able to choose a suitable representation of these strings
by means of conventional data reification techniques. For example, the string "hello, world" could be represented by the address of a memory location containing the 'h' where subsequent locations contain the 'e', the 'l' and so on until some termination value indicates the end of the string.

Of course it would not be wise to make such representation choices now already during the specification phase, but in order to avoid problems in a later design phase, we must make sure that our specification does not exclude certain reasonable choices.

Consider the above representation and assume that the memory contains "hello, world" starting at address 1024 but also at address 1037, say. Now there is a problem with equality: the test 1024 = 1037 yields FALSE, although both string representations are equivalent in the sense that they represent the same sequence of characters.

There is also a problem with the dynamic allocation of strings. In STRING_SPEC we only have functions such as FUNC empty: -> String and FUNC cons: Char # String -> String. This implies that formally an implementation of this cons is not allowed to have some side-effect such as storing characters in memory locations. The two problems sketched above indicate that, at least for certain applications, we better not adopt the sort String, but a slightly different sort, which we shall name 'String'. The quotes in 'String' are formally part of the identifier. For the latter sort we introduce an eq predicate which is meant for an equivalence relation as indicated above and we introduce a procedure called cons rather than a function. Furthermore we explicitly introduce the possibility that a representation invariant has to be maintained. We shall refer to the objects of this sort 'String' as implementable strings.

Of course the introduction of this special type of strings is a complication for our task of formally specifying an editor, especially because the use of the sort 'String' will propagate throughout large parts of our formal specification. However we think that data reification is an important topic and therefore we try to deal with it explicitly, rather than running away from the problem. Readers who prefer not to get involved with this topic, should just skip the current section and continue reading at Section 4.4.4, replacing everywhere 'String' by String, and replacing eq by equality (=).

The purpose of having 'String' rather than String is not to add an implementation bias. Instead we like to refer to our approach as specification for implementability. First we introduce a somewhat more general concept, by specifying a data type of implementable sequences. These are very much like sequences, except that they allow for dynamic implementations. They are given by the class description 'SEQ_SPEC'.
The parameter restriction is not the usual ITEM, but a somewhat more general form 'ITEM', which provides an eq predicate rather than just equality. The advantage of 'ITEM' over ITEM is that with 'ITEM' it is possible to have implementable sequences of implementable sequences of ..., rather than only implementable sequences of items with equality.

LET 'ITEM' :=
IMPORT ITEM INTO
CLASS

SORT 'Item'
FREE
PRED eq : 'Item' $ 'Item'
FREE
PRED item_inv:
FREE
FUNC f : 'Item' $ Item
FREE

AXIOM item_inv $>
FORALL i:'Item',j:'Item'
{ eq(i,j) $ f(i) = f(j) }

END;

The body of the class description 'SEQ_SPEC' introduces a sort 'Seq'. The specification 'SEQ_SPEC' is based on the use of an abstraction function f -- by overloading -- and a representation invariant seq_inv. Notice that we do not really specify the invariant; we just allow the implementation to maintain one. An implementer who does not need this, may just define seq_inv DEF TRUE. Furthermore in 'SEQ_SPEC' we have procedures cons and cat rather than functions. There is an eq predicate which should be viewed as the external notion of equality on 'Seq'.

Not all sorts and operations are considered executable, but some of the non-executable sorts and operations are exported nevertheless because it is convenient to have them for reasoning purposes. In particular, the sort Seq, the predicate seq_inv and the function f are needed for reasoning purposes only. The remaining exported sorts and operations are considered executable. Formally, COLD-K does not provide a program-execution model and when we make statements about executability, these do not refer to any formal notion of COLD-K.

LET 'SEQ_SPEC':=

LAMBDA X : 'ITEM' OF
EXPORT

SORT Seq.
PRED seq_inv:
FUNC f: 'Seq' -> Seq.

SORT 'Seq',
SORT Nat,
SORT 'Item',
FUNC empty: -> 'Seq',
PROC cons: 'Item' # 'Seq' -> 'Seq',
FUNC hd: 'Seq' -> 'Item',
FUNC tl: 'Seq' -> 'Seq',
PRED eq: 'Seq' # 'Seq',
FUNC seal: 'Seq' # Nat -> 'Item',
PROC cat: 'Seq' # 'Seq' -> 'Seq',
PROC rev: 'Seq' -> 'Seq'

FROM

IMPORT x INTO
IMPORT nat_spec INTO
IMPORT apply_seq_spec TO x INTO

CLASS

SORT 'Seq' VAR

FUNC empty: -> 'Seq'
PROC cons: 'Item' # 'Seq' -> 'Seq' MOD 'Seq'
FUNC hd: 'Seq' -> 'Item'
FUNC tl: 'Seq' -> 'Seq'
PRED eq: 'Seq' # 'Seq'

FUNC f: 'Seq' -> Seq
PRED seq_inv: VAR

AXIOM {INVARIANCE}

INIT AND item_inv => seq_inv;
item_inv AND seq_inv => [ p ] seq_inv

PROC p: ->
DEF ( FLUSH cons(SOME i:'Item' (), SOME s:'Seq' ()) |
| FLUSH cat (SOME s:'Seq' (), SOME t:'Seq' ())
4.4. INTERFACING WITH THE ENVIRONMENT

| FLUSH \text{rev} \ (\text{SOME} \ s: \text{Seq} \ ()

AXIOM \{ABSTRACTION\}

item\_inv \ \text{AND} \ \text{seq}\_\text{inv} \ \Rightarrow
\ \text{FORALL} \ t: \text{Seq} \ ( \ f(t) ! ) ;

item\_inv \ \text{AND} \ \text{seq}\_\text{inv} \ \Rightarrow
\ \text{FORALL} \ s: \text{Seq} , \ t: \text{Seq} , \ i: \text{Item} , \ j: \text{Item}
\ f(\text{empty}) = \text{empty};
\ ( \ [ \ \text{LET} \ \text{u}: \text{Seq}; \ \text{u} := \text{cons}(\text{i}, \text{s}) \ ] \ f(\text{u}) = \text{cons}(f(\text{t}), f(\text{e})) \ ) ;
\ \text{MUT} \ f(\text{s}) \ \Rightarrow \ f(\text{hd}(\text{h})) = \text{hd}(f(\text{s}));
\ \text{MUT} \ f(\text{s}) = \text{empty} \Rightarrow \ f(\text{tl}(\text{s})) = \text{tl}(f(\text{s}));
\ \text{eq}(s, t) \ \iff \ f(\text{s}) = f(\text{t})

AXIOM \{Termination\}

item\_\text{inv} \ \text{AND} \ \text{seq}\_\text{inv} \ \Rightarrow
\ \text{FORALL} \ \text{i}: \text{Item} , \ s: \text{Seq} \ ( \ < \text{cons}(\text{i}, \text{s}) > \text{TRUE} )

PROC \text{cat}: \text{Seq} \ # \ \text{Seq} \ \rightarrow \ \text{Seq} \ \text{MOD} \ \text{Seq}
PROC \text{rev}: \text{Seq} \ \rightarrow \ \text{Seq} \ \text{MOD} \ \text{Seq}
FUNC \text{sel}: \text{Seq} \ # \ \text{Nat} \ \rightarrow \ \text{Item}

AXIOM \{\text{Abstraction}\}

item\_\text{inv} \ \text{AND} \ \text{seq}\_\text{inv} \ \Rightarrow
\ \text{FORALL} \ s: \text{Seq} , \ t: \text{Seq} , \ n: \text{Nat}
\ ( \ [ \ \text{LET} \ \text{u}: \text{Seq}; \ \text{u} := \text{cat}(\text{s}, \text{t}) \ ] \ f(\text{u}) = \text{cat}(f(\text{s}), f(\text{t}));
\ [ \ \text{LET} \ \text{u}: \text{Seq}; \ \text{u} := \text{rev}(\text{s}) \ ] \ f(\text{u}) = \text{rev}(f(\text{s}));
\ [ \ \text{LET} \ \text{i}: \text{Item}; \ \text{i} := \text{sel}(\text{s}, \text{n}) \ ] \ f(\text{i}) = \text{sel}(f(\text{s}), \text{n}) \ )

AXIOM \{\text{Termination}\}

item\_\text{inv} \ \text{AND} \ \text{seq}\_\text{inv} \ \Rightarrow
\ \text{FORALL} \ s: \text{Seq} , \ t: \text{Seq}
\ ( \ < \text{FLUSH} \ \text{cat}(\text{s}, \text{t}) > \ \text{TRUE};
\ < \text{FLUSH} \ \text{rev}(\text{s}) > \ \text{TRUE} )

END;

Now implementable strings are nothing but implementable sequences of characters. We do not need the full generality offered by 'ITEM', but that does not matter. We also introduce the usual lexicographical ordering on
sequences as a predicate less. Again we can distinguish between non-exectutables and executables. In particular, String, f: 'String' → String and string_inv need not be executable. The remaining exported sorts and operations are considered executable.

LET 'STRING_SPEC' :=

EXPORT

SORT String,
FUNC f: 'String' → String,
PRED string_inv'.

SORT Char,
SORT Nat,
SORT 'String',
FUNC empty: Char # 'String' → 'String',
PROC cons : Char # 'String' → 'String',
FUNC hd : 'String' → Char,
FUNC tl : 'String' → 'String',
PRED eq : 'String' # 'String',
FUNC sel : 'String' # Nat → Char,
PROC cat : 'String' # 'String' → 'String',
PROC rev : 'String' → 'String',
PRED less : 'String' # 'String'

FROM
IMPORT NAT_SPEC INTO
IMPORT CHAR_SPEC INTO
IMPORT STRING_SPEC INTO

IMPORT APPLY RENAME

SORT Seq TO String,
SORT 'Seq' TO 'String',
SORT Item TO Char,
SORT 'Item' TO Char,
PRED seq_inv: TD string_inv

TN 'SEQ_SPEC' TO
IMPORT CHAR_SPEC INTO

CLASS

PRED eq: Char # Char PAR c:Char,d:Char DEF c = d
PRED item_inv: DEF TRUE
FUNC f: Char → Char PAR c:Char DEF c

END

INTO
4.4. INTERFACING WITH THE ENVIRONMENT

CLASS

PREM less : 'String' # 'String'

AXIOM string_inv =>
FORALL s:'String', t:'String'
( less(s, t) <=> less(f(s), f(t)) )

END;

4.4.4 FILE: a File system

Both the possibility to edit existing texts and the possibility to store the results of an edit-session are indispensable for text editing. We assume that the basic mechanisms for text storage and retrieval are provided by a file system. We cannot completely specify an editor unless we also have a description of the available file system. The class description FILE_SPEC below models a simple file system with just enough operations for our purposes.

We use names (sort 'String') as file-identifications. Since the file system can in fact be independent of any particular string implementation, we have parameterised the file system specification over implementations of 'STRING_SPEC'. The predicate valid describes the valid names of the file system, corresponding with the situation that for certain names there exist files in the system, whereas for other names there is no file (yet).

We restrict ourselves to character-files and we model the contents of a file as a string. This is described by the function file which for a given name yields the contents of the corresponding file. We allow for an arbitrary number of files in the file system, but in any state at most two files can be active, viz. one for input and one for output. So read and eof operate on the file that has been reset most recently. Similarly write operates on the most recently rewritten file. In the description below this idea of having at most two active files is modelled by the functions infile and outfile which correspond with the names of the input file and the output file respectively.

We suppose that the operations rewrite, reset, read, write and eof are executable. We also export valid, file and pos, but this is just because we might want to reason with them; they need not be executable.

LET FILE_SPEC :=
LAMDA X : 'STRING_SPEC' OF
EXPORT
SORT String.
PREM valid : String.
FUNC file : String -> String.
FUNC pos : String -> Nat.

SORT Nat.
SORT Char.
SORT 'String'.
PROC rewrite : 'String' -> ,
PROC reset : 'String' -> ,
PROC read : -> Char.
PROC write : Char -> ,
PREM eof : 

FROM
IMPORT X INTO
IMPORT NAT_SPEC INTO
IMPORT CHAR_SPEC INTO
IMPORT STRING_SPEC INTO
CLASS

FUNC file : String -> String VAR
FUNC pos : String -> Nat VAR

PREM valid : String
PAR s:String
DEF file(s)!

FUNC(infile : ) -> String VAR
FUNC (outfile: ) -> String VAR

PROC rewrite : 'String' -> NODE outfile

AXIOM FORALL s:'String' ( string_inv => ( < rewrite(s) > TRUE;
[ rewrite(s) ]
(output = f(s);
file(f(s)) = empty;
FORALL t:'String', u: String
( NOT f(t) = f(s) =>
  file(f(t)) = u <=> PREV file(f(t)) = u ) ) )

PROC reset : 'String' -> NODE pos, infile
AXIOM FORALL a: 'String' ( string_inv AND valid(f(a)) ) => ( 
  < reset(a) > TRUE;
  [ reset(a) ]
  ( infile = f(a);
    pos(f(n)) = zero;
    FORALL t:'String', i:Nat
      ( NOT f(t) = f(a) =>
        pos(f(t)) = i <=> PREV pos(f(t)) = i ) ) )

PROC read : -> Char MOD pos
    
AXIOM string_inv AND lea(pos(infile), len(file(infile))) => ( 
  < read > TRUE;
  [ LET c:Char. c := read ]
  ( c = sel(file(infile), PREV pos(infile));
    pos(infile) = succ(PREV pos(infile));
    FORALL t: String, i:Nat
      ( NOT t = infile =>
        pos(infile) = i <=> PREV pos(t) = i ) ) )

PROC write : Char -> MOD file

AXIOM FORALL c:Char ( string_inv AND valid(outfile) ) => ( 
  < write(c) > TRUE;
  [ write(c) ]
  ( outfile = cat((PREV outfile), cons(c, empty));
    FORALL t: String, u: String
      ( NOT t = outfile =>
        ( outfile = u <=> PREV outfile = u ))) )

PRED eof:

AXIOM string_inv AND valid(infile) AND pos(infile) => ( 
  ( eof <=> pos(infile) = len(file(infile)) )

END;

We tried to keep the specification of the file system as simple as possible. In particular, our description does not cover the phenomenon of output buffering and the related problems among which the need for a so-called *flush* operation.

Note also that we did not exclude the situation that *infile* and/or *outfile* are undefined - typically after system initialisation. In that case nothing is specified about the effect of *read* and *write*. For example if *infile*
is undefined then the assertion less(pos(infile), len(file(infile))) is false, so read can do anything – e.g. abort.

4.5 Text Editing

4.5.1 Introduction

We adopt the basic idea underlying display-oriented editors where the user of an editor need not give print or display commands himself. Instead, the editor more or less automatically makes sure that parts of one or more texts are displayed within windows on the screen of a display. We also adopt the idea that the user can edit several texts simultaneously. In combination with cut and paste capabilities, this makes it possible to move pieces of text from one text to another. It should be possible for the user to give names to texts. As a third idea we adopt the algebraic operations cut and paste of Section 4.3.6 as a starting point for the cut and paste capabilities of the editor. These ideas are also realised in the EMACS editor [4]. However, in [4] the algebraic aspects are somewhat hidden and instead there are simply operations copy-region-to-buffer and yank-buffer offering cut and paste capabilities.

We shall describe a kind of abstract machine which we refer to as window-and-text facility. The corresponding class description is named WITEFA_SPEC – for WINDOW and TEXT FACILITY). This facility allows for the manipulation of texts and it makes sure that a suitable part of one text is displayed on the screen of a display which serves as a window. It supports a notion of 'current text' and a mechanism for associating names with texts. For the time being the window management part of the window-and-text facility is kept simple: there is just one window with with dimensions 11 and co which thus covers the entire screen of the display (see Section 4.4.2). It should be stressed however that our approach would be suited for the inclusion of multi-window features: the algebraic operations h_add and v_add of Section 4.3.9 would be a good starting point for describing the partition of a multi-window screen.

We avoid the introduction of a notion of buffer, which in our view is best postponed until an implementation phase. We have operations whose names are based on a buffer-oriented terminology, but we do this only for maintaining a kind of name-compatibility with EMACS. In the formal specification of these operations we do not use the buffer-oriented terminology. Instead of that we have the notion of marked text, which is the subject of our next section.
The remainder of this section is organised as follows. In Section 4.5.2 we formally introduce marked texts. Section 4.5.3 is about formally importing the machinery developed before into the editor specification. Section 4.5.4 is about modelling the state-space. In Section 4.5.5 we formulate a so-called text-invariant. The definition of the editor operations will cover the Sections 4.5.6 to 4.5.10. The termination axioms are given for all operations together in Section 4.5.11. In Section 4.5.12 we discuss the connection between the editor and the display. In Section 4.5.13 we present a simple editor which can be built on top of our window-and-text facility. Finally in Section 4.5.14 we discuss the connection between the editor and the keyboard.

4.5.2 Marked Texts

A marked text is a composite object that consists of a text and a collection of co-ordinate pairs. These co-ordinate pairs will be called markers. We expect that we need the following markers.

- Dot: a kind of 'current' location in the text.
- Mark: a marker that can be put on any position in the text. This mark can be used for selecting a text-region for cut- and paste operations. In the typical applications, the mark indicates the beginning of this text region and the dot indicates its end. This application of the mark will be worked out in Section 4.5.7.

Furthermore we have considered the possibility of having the following additional markers, but at least for the time being we do not need these urgently, so we shall not include them in our specification: 1. bob: marker that is always at the beginning of the text. 2. eob: marker that is always at the end of the text. 3. searcher: a marker that plays a role in search- and replace commands. One could even go further and consider the introduction of a stack of markers (in [6] this is called a 'mark ring'), allowing for the manipulation of a large number of markers.

The following picture sketches a marked text.
We do not introduce the idea of locations for storing marked texts directly, but instead we focus now on the contents of such locations. We formalise markers as tuples \((i, j)\). This is done by introducing a class description called COPA_SPEC (COPA for CO-ordinate PAir). We have projection functions \(v\) for the vertical co-ordinate and \(h\) for the horizontal co-ordinate.

\[
\text{LET COPA_SPEC := APPLY APPLY RENAME}
\]

\[
\text{SORT Item1 TO Nat, SORT Item2 TO Nat,}
\text{SORT Tup TO Copa, FUNC proj1: Tup \rightarrow Item1 TO v,}
\text{FUNC proj2: Tup \rightarrow Item2 TO h, FUNC tup: Item1 \# Item2 \rightarrow Tup TO copa}
\]

\[
\text{IN TUP2_SPEC TO NAT_SPEC TO NAT_SPEC;}
\]

And now we can define a class description MTEXT_SPEC, which serves for formalising marked texts as triples of the form \((t, d, m)\) where \(t\) is the text and where \(d\) and \(m\) are the dot and the mark respectively.

\[
\text{LET MTEXT_SPEC := IMPORT APPLY APPLY RENAME}
\]

\[
\text{SORT Item1 TO Text, SORT Item2 TO Copa, SORT Item3 TO Copa, SORT Tup TO MText, FUNC proj1: Tup \rightarrow Item1 TO text,}
\text{FUNC proj2: Tup \rightarrow Item2 TO dot, FUNC proj3: Tup \rightarrow Item3 TO mark, FUNC tup: Item1 \# Item2 \# Item3 \rightarrow Tup TO mtext}
\]

\[
\text{IN TUP8_SPEC TO TEXT_SPEC TO COPA_SPEC TO COPA_SPEC INTO}
\]
4.5. TEXT EDITING

IMPORT NAT_SPEC INTO
IMPORT COPA_SPEC INTO

CLASS

FNC dot: MText -> Nat # Nat
PAR b: MText
DEF v(dot(b)), h(dot(b))

FNC mark: MText -> Nat # Nat
PAR b: MText
DEF v(mark(b)), h(mark(b))

This provides us with an extensional and functional specification of marked texts. We introduce a number of modification operations on marked texts - not in the sense of modifications on states of course.

FNC modtext: MText # Text -> MText
PAR m: MText, t: Text
DEF mttext(t, dot(m), mark(m))

FNC moddot: MText # Copa -> MText
PAR m: MText, c: Copa
DEF mttext(text(m), c, mark(m))

FNC modmark: MText # Copa -> MText
PAR m: MText, c: Copa
DEF mttext(text(m), dot(m), c)

END;

4.5.3 Stating the Application Domain

Most software products have a certain functionality associated with some specific application-domain. In our case this is text editing and therefore we have collected a number of definitions which are specifically related to the notion of text. This collection of definitions should be viewed as an application-domain specific notational framework, rather than as a product specification. It is convenient to have a name for this application-domain specific notational framework. Therefore we introduce a class description called APP_DOM_SPEC which encompasses many of the class descriptions given before.
4.5.4 Spanning the State-space

Our window-and-text facility manages a finite collection of marked texts. At the specification level, we describe this collection as a map. We shall use several operations from MAP_SPEC as given in Appendix B such as empty: → Map, app: Map # Item1 → Item2 and dom: Map → Set1 which yields a set of Item1. Sometimes we write m[i] for app(m, i). Of course we import the application-domain specific notational framework as collected in APP.DOM_SPEC.

LET WITEFA_SPEC :=

IMPORT APP.DOM_SPEC INTO
IMPORT 'STRING_SPEC' INTO
IMPORT DISPLAY_SPEC INTO

IMPORT (APPLY FILE_SPEC TO 'STRING_SPEC') INTO

IMPORT APPLY APPLY RENAME
    SORT Item1 TO String,
    SORT Item2 TO MText
IN MAP_SPEC TO STRING_SPEC TO MTEXT_SPEC INTO

IMPORT APPLY RENAME
    SORT Item TO String,
    SORT Set TO Set
IN SET_SPEC TO STRING_SPEC INTO
4.5. TEXT EDITING

CLASS

The collection of marked texts in WRITE.F_SPEC is formally described by a variable function mtextx. In this way marked texts will have names associated with them, where names are just strings.

FUNC mtextx: $\to$ Map VAR

Just to keep things concrete, let us briefly sketch one possible use of this variable map. In Section 4.5.13 we shall describe a simple editor on top of our window-and-text facility. In this editor we shall have three marked texts, denoted as mtextx["main"], mtextx["mini"] and mtextx["kill"]. In EMACS terminology these would be called the main-buffer, the mini-buffer and the kill-buffer respectively. The marked text named "main" contains essentially the text being edited, whereas the marked text named "mini" allows the user to input file names and search strings. The marked text named "kill" always contains the most recently deleted text.

Now we proceed with spanning the state-space. There is a notion of current marked text which formally is described by introducing a variable function.

FUNC current: $\to$ String VAR

The variables mtextx and current need not be considered executable. All operations to be described later are considered executable.

At first sight it seems attractive to postulate an initialisation condition by using the built-in initially assertion INIT of COLD-K, but this may be hard to implement and therefore we prefer the use of an initialisation procedure. Instead of INIT $\Rightarrow$ ... we must write INIT $\Rightarrow$ [ init(...) ] ...

We want to avoid the situation where current: String becomes undefined. Therefore the initialisation procedure gets a 'String' argument which represents the name of the first marked text. This marked text consists of the zero text with (0,0) as dot and mark. After the initialisation has been invoked, the user of the window-and-text facility can introduce more names — and hence more marked texts. The procedure init defined here has modification rights with respect to mtextx, current and a predicate named WT1.

The motivation for this WT1 and its definition will be provided later in Section 4.5.5. The procedure init also has use rights with respect to the procedure disp1_op which was defined in Section 4.4.2. In the description of the post-condition of init below we refer to the function f: 'String' $\to$ String which was defined in Section 4.4.3.

PROC init: 'String' $\to$
MOD mtexts, current, WII USE displ_op

AXIOM INIT => FORALL s: 'String' ( < init(s) > TRUE )

AXIOM INIT =>
FORALL s: 'String'
( [ init(s) ]
  current = f(s) AND
  mtexts = add( empty, f(s), mtext( zero, copa(0.0), copa(0.0)) )
)

We have classified the operations into the following groups:

- operations for dot and mark control,
- operations for text modification,
- operations for marked-text management,
- operations for searching,
- operations for string conversion.

4.5.5 The Text Invariant

We must make a decision about the behaviour of the editor. There are
a number of marked texts, and one of them, is the ‘current’ marked text.
Within this current marked text there is a point of interest, indicated by
dot. Probably this is close to the piece of text about which the user of the
editor is thinking and where he is going to do his next insertion or deletion.
The user can move this dot and the cursor in the sense that the cursor on
the screen somehow mirrors the position of the dot. The question now is
if one wants to allow the situation where the dot indicates a position that
is non-existent in the text of the current marked text. We believe that a
consequent and elegant approach is to avoid this situation entirely. Consider
the following text and suppose that dot is at the position corresponding with
the A at the end of the first line.

```
  this is a very long line of text rather long indeed 1
short line
  this is a very long line of text rather long indeed 2
  this is a very long line of text rather long indeed 3
  this is a very long line of text rather long indeed 4
```

Now according to our approach it should not be possible to move the dot
one line downwards while staying within the same column. Indeed, suppose
that it would be possible, then it is unclear what it means to do an insert
operation next.
We formulate the above ideas by introducing a so-called text-invariant. The text-invariant implies that for each marked text both the dot and the mark correspond with positions that exist in the text of that marked text; furthermore this text should be ok.

Considering ok texts only has the advantage that when this window-and-text facility has to be implemented, its designer is in a comfortable position in the sense that he has the option of choosing the string-representation of texts. Another advantage exists already in the specification phase: all operations such as cut and paste of Section 4.3.6 which are well-defined for ok texts only, are usable. Now we easily formulate this part of our text-invariant and we introduce it as a predicate T11.

\[
\text{PRED T11:}
\]
\[
\text{DEF \forall s: \text{String, } mt: \text{MText}}
\]
\[
(\text{mt} \in \text{app(atexts, } s)) \Rightarrow
\]
\[
(\text{LET } d: \text{Nat # Nat; } d := \text{dot(mt)};)
\]
\[
(\text{LET } m: \text{Nat # Nat; } m := \text{mark(mt);})
\]
\[
(\text{intext(text(mt), } d) \text{ AND intext(text(mt), } m) \text{ AND ok(text(mt)))}
\]

A second constraint on the state-space arises because we have a notion of current and this should be the name of an existing marked text.

\[
\text{PRED T12:}
\]
\[
\text{DEF app(atexts, current)!
}\]

\[
\text{PRED T1:}
\]
\[
\text{DEF T11 AND T12}
\]

We must formalise the requirement that the operations of the \text{WITEFA\_SPEC} respect T1. As a first attempt, we could express this requirement by saying that T1 is an invariant in the following weak sense, which we might refer to as \text{repetition-invariant}. Let \text{witefa\_op} denote an invocation of one of the operations of \text{WITEFA\_SPEC}.

\[
\text{% AXIOM INIT } \Rightarrow \; [\ \text{init(SOME s: 'String')}] \; [\ (\text{witefa\_op})^*] \; \text{T1}
\]
\[
\text{% (not adopted)}
\]

This first attempt fails, because it does not guarantee that T1 is preserved by an interleaving of \text{witefa\_op} invocations with invocations of other operations. As a second attempt, we could express this requirement as follows.

\[
\text{% AXIOM INIT } \Rightarrow \; [\ \text{init(SOME s: 'String')}] \; \text{T1;}
\]
\[
\text{% T1 } \Rightarrow \; [\ \text{witefa\_op}] \; \text{T1}
\]
\[
\text{% (not adopted)}
\]
We shall refer to this formulation by saying that TI is a *classical invariant*. This attempt fails too, because we must allow an implementation to have auxiliary variables and hence to maintain a stronger invariant WTI, say. The problem is that from the fact that WTI is a classical invariant one cannot conclude that TI is so too.

Therefore we introduce as a part of the specification another auxiliary predicate WTI (for Window and Text Invariant) which must be a classical invariant and which at its turn must imply TI. One might be tempted to think that the classical invariance of WTI under \texttt{witefa\_op} follows already automatically if we simply do not add WTI to the modification rights of the operations from the \texttt{witefa\_op} group. However, this would be too strong, because it would forbid an implementation of the procedures to change the truth-value of the invariant from false to true — just by luck. So we must add WTI to the modification rights of the operations from the \texttt{witefa\_op} group and we must state its invariance by an axiom.

\begin{verbatim}
PRED WTI: VAR

AXIOM (WTI1) INIT => [ init(SOME a:'String') ] WTI;
            WTI => [ witefa\_op ] WTI

AXIOM (WTI2) WTI => TI
\end{verbatim}

We adopt the latter solution and we might refer to its construction by saying that TI is an *observational invariant* with respect to \texttt{witefa\_op}. The intuition behind this term is as follows. Define an experiment as the execution of some interleaving of \texttt{witefa\_op} invocations with invocations of other operations. Now this construction guarantees that after any experiment, it can be observed that TI holds, although in fact something stronger may hold, which need not be observable.

Actually \texttt{witefa\_op} can be defined in terms of the operations from the next sections. Its definition refers to all but one procedures of \texttt{WITEFA\_SPEC}, the only exception being \texttt{init}.

\begin{verbatim}
PROC witefa\_op: ->
DEF (FLUSH belp | FLUSH celp | forward\_character | backward\_character | next\_line | previous\_line | beginning\_of\_line | end\_of\_line
\end{verbatim}
```lisp
| beginning_of_buffer
| end_of_buffer
| set_mark
| exchange_dot_and_mark
| insert_file(SOME a: 'String' (valid(f(a)))))
| insert_character(SOME c: Char printable(c))
| newline
| yank_buffer(SOME a: 'String' (app(atexts.f(a)))))
| delete_next_character
| erase_region
| erase_buffer(SOME a: 'String' ()))
| copy_region_to_buffer(SOME a: 'String' ()))
| FLUSH current_buffer_name
| write_named_file(SOME a: 'String' ()))
| switch_to_buffer(SOME a: 'String' ()))
| search_forward(SOME a: 'String' ()))
| FLUSH buffer_to_string (SOME a: 'String' (app(atexts.f(a)))))
```

In the specification of the operations, we must take care not to contradict the assertion that T1 is an observational invariant. Even better, we can try to make a kind of 'almost-invariance' of T1 derivable in the sense that init establishes T1 and that \( T1 \Rightarrow \{ \mathtt{wtefa_op \_ TI} \} \). We decided to do so and furthermore we shall show below that init establishes T1. We shall later show by way of example this almost invariance for one of the operations, which will be \( \mathtt{copy\_region\_to\_buffer} \) of Section 4.5.7.

Let us show now already that \( \mathtt{INIT} \Rightarrow \forall a: 'String' \{ \mathtt{init}(a) \} \) T1 is derivable from the given postcondition of \( \mathtt{init} \). This postcondition states that \( \mathtt{current} = f(a) \land \mathtt{atexts} = \mathtt{add}(\mathtt{empty}, f(a), \mathtt{mtext}(\mathtt{zero}, \mathtt{cota}(0,0), \mathtt{cota}(0,0))) \). For T1 we must check something for all marked texts \( m \) in the range of \( \mathtt{atexts} \), but we see that there is only one, which is the marked text \( \mathtt{mtext}(\mathtt{zero}, \mathtt{cota}(0,0), \mathtt{cota}(0,0)) \). We easily verify that \( \mathtt{intext}(\mathtt{zero}, (0,0)) \) and \( \mathtt{ok}(\mathtt{zero}) \) holds. Now we are left with T12 which requires that \( \mathtt{current} \) is in the domain of \( \mathtt{atexts} \). Since \( \mathtt{current} = f(a) \), it is even the only element in the domain. This shows that \( \mathtt{init} \) establishes T1.

The presentation of signatures and the the pre- and postcondition style axioms of the operations will be based on the classification of the operations into several groups as sketched in Section 4.5.4. This will cover the Sections 4.5.6 to 4.5.10. The termination axioms are given for all operations together in Section 4.5.11.
4.5.6 Operations for Dot and Mark control

To begin with, we give the signature of the operations from this group.

PROC bolp : -> Bool  MOD WTI
PROC eolp : -> Bool  MOD WTI
PROC forward_character : -> MOD mtext,WTI USE displ_op
PROC backward_character : -> MOD mtext,WTI USE displ_op
PROC next_line : -> MOD mtext,WTI USE displ_op
PROC previous_line : -> MOD mtext,WTI USE displ_op
PROC beginning_of_line : -> MOD mtext,WTI USE displ_op
PROC end_of_line : -> MOD mtext,WTI USE displ_op
PROC beginning_of_buffer : -> MOD mtext,WTI USE displ_op
PROC end_of_buffer : -> MOD mtext,WTI USE displ_op
PROC set_mark : -> MOD mtext,WTI USE displ_op
PROC exchange_dot_and_mark : -> MOD mtext,WTI USE displ_op

We describe the operations in pre- and postcondition style. TI must be respected, which for the operations of this group amounts to the restriction of TI1 that we may not move dot or mark to a position that is non-existing in the text. We have several similar axioms and in order to save space the following premise serves as a common clause for the specification of several operations:

AXIOM WTI => (  
  LET mtext':MText; mtext' = app(mtext, current);  
  LET i:Nat; j:Nat;  i,j := dot(mtext');
)

Our first operations are two Boolean procedures named bolp and eolp abbreviating beginning-of-line predicate and end-of-line predicate.

[ LET b:Bool; b := bolp ]
( b = true <-> j = 0 );

[ LET b:Bool; b := eolp ]
( b = true <-> len(sel(text(mtext'), i)) = j );

We used the selection operation sel: Text -> Line which – somewhat implicitly – was constructed in Section 4.3.2. The four next operations deal with moving the dot rightwards, leftwards, downwards and upwards respectively. Each operation has two clauses: one for the normal case and one for the situation where it has no effect because TI1 must be respected. The axioms below show a difference between our editor and EMACS [4]. The difference lies in the effect of trying to move rightwards when the dot is at the end of a line. In that case our editor does nothing whereas in EMACS
the dot moves to the beginning of the next line. Similar differences exist for leftward, downward and upward movements.

In view of the fact that \texttt{mtexts} is a map based on MAP SPEC of Appendix B, we can use the operation \texttt{add: Map # String -> MText} for describing how the marked text addressed by current is overwritten.

It is interesting to analyze the specification of \texttt{previous_line} below for the particular case that the vertical co-ordinate of the dot is 0, i.e. \( i = 0 \). In that case \texttt{pred(i)} is undefined and therefore \texttt{intext(\ldots ,\texttt{pred(i)},j)} is false. We see that the postcondition \texttt{mtexts = PREV mtext} applies - so nothing happens.

\begin{verbatim}
intext(text(mtext'),i,succ(j)) =>
  [ forward_character ]
  ( LET new_dot: Copa; new_dot := copa(i,succ(j)));
  LET new_mtext: MText; new_mtext := moddot(mtext',new_dot);
  mtext = add((PREV mtext),current,new_mtext) );
NUT intext(text(mtext'),i,succ(j)) =>
  [ forward_character ]
  ( mtext = (PREV mtext) ) ;

gtr(j,0) =>
  [ backward_character ]
  ( LET new_dot:Copa; new_dot := copa(i,pred(j))
    { rest as for forward_character} ) ;
NUT gtr(j,0) =>
  [ backward_character ]
  ( mtext = (PREV mtext) ) ;

intext(text(mtext'),succ(i),j) =>
  [ next_line ]
  ( LET new_dot:Copa; new_dot := copa(succ(i),j)
    { rest as for forward_character} ) ;
NUT intext(text(mtext'),succ(i),j) =>
  [ next_line ]
  ( mtext = (PREV mtext) ) ;

intext(text(mtext'),pred(i),j) =>
  [ previous_line ]
  ( LET new_dot:Copa; new_dot := copa(pred(i),j)
    { rest as for forward_character} ) ;
\end{verbatim}
\begin{verbatim}
NOT intext(text(mtext'), pred(i,j)) =>
  [ previous_line ]
  ( mtexts = (PREV mtexts) );

The next operations deal with moving the dot to the extreme positions in a
line. This is easy because there is no danger of moving dot to a non-existing
position.

  [ beginning_of_line ]
  ( LET new_dot:Copa; new_dot := copa(i,0) 
    {rest as for forward_character} ) ;

  [ end_of_line ]
  ( LET new_dot:Copa; new_dot := copa(i, len(sel(text(mtext'),i))) 
    {rest as for forward_character} ) ;

The next operations deal with moving the dot to the extreme positions within
the buffer. Again there is no danger of moving dot to a non-existing position.

  [ beginning_of_buffer ]
  ( LET new_dot:Copa; new_dot := copa(0,0) 
    {rest as for forward_character} ) ;

  [ end_of_buffer ]
  ( LET new_dot:Copa; new_dot := copa(reach(text(mtext'))) 
    {rest as for forward_character} ) ;

The following operations deal with controlling the mark. WTI and hence TII
hold in the begin-state. Therefore we know that both dot and mark are at
existing positions and this remains the case if we put mark at dot or if we
exchange dot and mark. This shows that there is no danger of violating TII.

  [ set_mark ]
  ( LET new_mark: Copa; new_mark := dot(mtext');
    LET new_mtext:MText; new_mtext := modmark(mtext',new_mark);
    mtexts = add(((PREV mtexts), current, new_mtext) ) ;
\end{verbatim}
4.5. TEXT EDITING

[ exchange_dot_and_mark ]
( LET new_dot :Copa; new_dot := mark(stext');
  LET new_mark:Copa; new_mark := dot(stext');
  LET new_stext :NText;
  new_stext := modmark(moddot(stext', new_dot), new_mark);
  stext = add((PREV stexts), current, new_stext) )
)

4.5.7 Operations for Text Modification

These operations serve for changing the text of some marked text and most of them modify the current marked text. To begin with, we give the signature of the operations from this group.

PROC insert_file: 'String' ->
MOD stexts, WTI USE displ_op, reset, read

PROC insert_character : Char -> MOD stexts, WTI USE displ_op
PROC newline                 -> MOD stexts, WTI USE displ_op
PROC yank_buffer: 'String'   -> MOD stexts, WTI USE displ_op
PROC delete_next_character:  -> MOD stexts, WTI USE displ_op
PROC erase_region            -> MOD stexts, WTI USE displ_op
PROC erase_buffer: 'String'  -> MOD stexts, WTI USE displ_op
PROC copy_region_to_buffer:  'String' -> MOD stexts, WTI USE displ_op

We describe the operations in pre- and postcondition style and we must make sure that WTI is respected. Again we have several similar axioms and we have a common clause for the specification of several operations.

AXIOM WTI =>

FORALL s: 'String', c: Char {

Most operations of this group have the effect that the buffer indicated by current becomes modified. In order to define the postcondition of the operation insert_file, we need a specification of the underlying file system and so we have imported FILE_SPEC as described in Section 4.4.4. Recall that for a: 'String' we have f(a) as the corresponding string and file(f(a)) as the string contents of the corresponding file. Finally text(file(f(a))) denotes the contents of this file viewed as a text. Of course all of this only holds under the assumption that this file exists, which is if valid(f(a)) holds. The main description tool for the operations insert_file, insert_character,
newline and yank_buffer is the paste operator on texts as described in Section 4.3.6. Some postconditions look complicated because we must re-calculate the co-ordinates of the dot; furthermore the co-ordinates of the mark may need re-calculation depending on the relative position of mark with respect to dot. In order to describe these re-calculations we use the add and paste operations on reaches from Section 4.3.7. The new dot is given as copa(add(d,reach(t))) but it is interesting to note that we could write copa(paste(d,reach(t),d)) alternatively because \[ \forall x,y \in \mathbb{N}^2 \{ paste(x,y,x) = add(x,y) \} \] is a simple property of the algebra of reaches (cf. the definitions of paste, split and add).

```ocaml
valid(f(s)) =>
[ insert_file(s) ]
( LET t:Text; t := text(file(f(s)));

LET mtext':NText; mtext' := app((PREV mtext),current);
LET d: Nat # Nat; d := dot(mtext');
LET m: Nat # Nat; m := mark(mtext');

LET new_text:Text; new_text := paste(text(mtext'),t,d);
LET new_dot :Copa; new_dot := copa(add(d,reach(t)));
LET new_mark:Copa;
new_mark := ( less(m,d) \?
| \ NOT less(m,d) \?
| \ NUT \ NOT less(m,d) \?
| \ NUT \ NOT less(m,d) \?
| \ NUT \ NOT less(m,d) \?
| \ NUT \ NOT less(m,d) \?
| \ NUT \ NOT less(m,d) \?)copa(m)

LET new_mtext:NText;
new_mtext := ntext(new_text,new_dot,new_mark);

printable(c) =>
[ insert_character(c) ]
( LET t:Text; t := addchar(c,zero)
   { rest as for insert_file(s) } );

[ newline ]
( LET t:Text; t := addempty(zero)
   { rest as for insert_file(s) } );

is_in(f(s),dom(mtext)) =>
[ yank_buffer(a) ]
( LET t:Text; t := text(app((PREV mtext),f(s)))
   { rest as for insert_file(s) } )
```
The main description tool for the operations delete_next_character and erase_region is the cut operator on texts as described in Section 4.3.6. Note that cut always yields a pair \((t_1, t_2)\) where \(t_1 = \text{'remaining text'}\) and \(t_2 = \text{'deleted text'}\). We apply cut, retaining its first result and throwing away its second result. Again the co-ordinates of the mark need re-calculation when cutting takes place at dot. As before, we have one premise serving the specification of several operations.

\[\text{AXIOM WTI} \Rightarrow\]

\[\text{FORALL } a : \text{'String'}, c : \text{Char} \]
\[\text{( LET } stext : \text{MText}; stext' := \text{app(stexts.current);}\]
\[\text{LET } i : \text{Nat}, j : \text{Nat}; i,j := \text{dot(stext')};\]
\[\text{LET } d : \text{Nat} \# \text{Nat}; d := \text{dot(stext')};\]
\[\text{LET } m : \text{Nat} \# \text{Nat}; m := \text{mark(stext')};\]

Again several operations have two clauses: one for the normal case and one for the case where the operation has no effect. For delete_next_character and erase_region it is not obvious what should happen with mark; we adopt some ad-hoc solution.

\[\text{intext}(\text{text}(\text{stext'}),i,i,\text{succ}(j)) \Rightarrow\]
\[\text{ [ delete_next_character ]}\]
\[\text{( LET new_text : Text, u : Text; new_text.u := \text{cut(text}(\text{atext'}),i,i,\text{succ}(j)); i : \text{Nat} \# \text{Nat}; new_dot := \text{copa}(d);\]
\[\text{LET new_mark : Copa; new_mark :=\}
\[\text{ ( leq(a,d) }\]
\[\text{ copa(m) \}}\]
\[\text{ | NOT leq(a,d)?}\]
\[\text{ LET x : \text{Nat} \# \text{Nat}, y : \text{Nat} \# \text{Nat}; x,y := \text{cut}(s,i,i,\text{succ}(j));\]
\[\text{ copa(x)}\]
\[\}];\]
\[\text{LET new_atext : MText; new_atext := \text{atext}(\text{new_text,new_dot,new_mark});\]
\[\text{ntexts} = \text{add}((\text{PREV ntext}),\text{current},\text{new_atext}) \};\]

\[\text{NOT intext}(\text{text}(\text{stext'}),i,i,\text{succ}(j)) \Rightarrow\]
\[\text{ [ delete_next_character ]}\]
\[\text{( ntexts }\Rightarrow\text{ (PREV ntext)})\];

\[\text{let}(m,d) \Rightarrow\]
\[\text{ [ erase_region ]}\]
\[\text{( LET new_text : Text, u : Text; new_text.u := \text{cut(text}(\text{stext'}),m,d);}
LET new_dot : Copa; new_dot := cops(n);
LET new_mark: Copa; new_mark := cops(n)
   (rest as for delete_next_character)

NOT lns(m,d) =>
   [ erase_region ]
   ( mtexts = (PREV mtexts) )

There are also a few operations that take a 'String' argument addressing a
certain buffer to be modified. Again it is not always clear what should happen
with mark and dot. In the postcondition of copy_region_to_buffer we use
again the cut operation, but now its first result is thrown away and its second
result is retained.

[ erase_buffer(n) ]
   ( mtexts
      = add((PREV mtexts), f(a), mtext(zero, cops(0,0), cops(0,0))) )

lns(m,d) =>
[ copy_region_to_buffer(s) ]
( LET t: Text, new_text: Text;
   t.new_text := cut(text(mtext'), n, d);
   mtexts
      = add((PREV mtexts), f(a), mtext(new_text, cops(0,0), cops(0,0))) )

NOT lns(m,d) =>
[ copy_region_to_buffer(s) ]
   ( mtexts = (PREV mtexts) )

Let us show now that WTI => \forall a: 'String' [ copy_region_to_buffer(s)
] TI is derivable from the given postcondition of copy_region_to_buffer,
as promised earlier. This postcondition states that if mark >= dot, nothing
happens, in which case we are done. So let us assume that mark < dot.
We know that mark and dot are in the old current text, so consider the
expression cut(text(mtext'), m, d) where mtext' is the previous marked
text and where m, d are its mark and dot respectively. Then this expression
yields two ok texts. The second of these is denoted as new_text and it is added to mtexts together with (0,0) for mark and dot. Note that
it is added by the 'overwriting' add of MAP_SPEC. Now TI holds because
intext(new_text, (0,0)) for arbitrary ok values of new_text. Furthermore
TI2 holds because the range of mtexts could only grow, whereas current
remained unaffected. This shows that copy_region_to_buffer preserves TI
when WTI holds as a precondition.

4.5.8 Operations for Marked-text Management

To begin with, we give the signature of the operations from this group. We considered a procedure `list_buffers`, but we decided not to include it because it is not clear how we should make it elegantly deliver its output. Similarly, we decided not to include a procedure `delete_buffer: 'String'` → because of the question of what should happen when the current marked text is deleted, which is problematic in view of TII.

```
PROC current_buffer_name: -> 'String'
MOD WTI

PROC write_named_file: 'String' ->
MOD WTI
USE rewrite, write

PROC switch_to_buffer: 'String' ->
MOD stexts, current, WTI
USE displ_op
```

The operations are described by their postconditions. As before, we have one premise serving the specification of several operations.

```
AXIOM WTI =>
```

```
FORALL s: 'String' ( |
```

We give the description of the operations below. In order to define the postcondition of `write_named_file` we refer again to the file system of Section 4.4.4. The operation `switch_to_buffer` may create a new marked text – not in the sense of dynamic object creation of course. We take the ad-hoc value zero for its text component and then TII dictates the value (0,0) for dot and mark.

```
[ LET t: 'String'; t := current_buffer_name ]
( f(t) = current );

[ write_named_file(s) ]
( valid(f(s));
  file(f(s)) = string(text(app(stexts, current)));
  FORALL t: 'String'
    ( NOT f(t) = f(s) => valid(f(s)) <-> PREV valid(f(s)) );
```
FORALL t:'String', u: 'String' ( NOT f(t) = f(u) => file(f(u)) - u <-> PREV file(f(u)) = u );

[ switch_to_buffer(s) ]
( current = f(s); is_in(f(s), dom(PREV mtexts)) => mtexts = (PREV mtexts);
 NOT is_in(f(s), dom(PREV mtexts)) => mtexts =
    add((PREV mtexts), f(s), mtext(zero.cops(0,0), cops(0,0))) )
)

4.5.9 Operations for Searching

We describe just one operation for searching. It takes a 'String' argument which is interpreted as the string representation of the search text. The main description tool is the search operation of Section 4.3.10 which was introduced precisely for this purpose.

PROC search_forward : 'String' => MOD mtexts,WTI USE displ_op

AXIOM WTI =>

FORALL s: 'String' ( [ search_forward(s) ]
    ( LET mtext:'MText; mtext' = app((PREV mtexts),current); let first_part:Text, second_part:Text;
        first_part, second_part := split(text(mtext'), dot(mtext'));
        LET i: Nat, j: Nat; i,j := search(second_part, text(f(s)));

        (i,j) = reach(second_part) => mtexts = (PREV mtexts);

        NOT (i,j) = reach(second_part) =>
            ( LET new_dot: Copa;
                new_dot := cops(add(reach(first_part),i,j));
                let new_mtext:'MText;
                new_mtext := mtext(text(mtext'), new_dot, mark(mtext'));
                mtexts = add((PREV mtexts), current, new_mtext ) )
    )
)

In the above precondition the clause (i,j) = reach(second_part) denotes the situation where the search text does not occur.
4.5.10 Operations for String Conversion

We expect that there will arise applications for a conversion of the text of a marked text into a string. One such application will be presented in Section 4.5.13 when we shall describe a simple editor, where the contents of the marked text named "mini" is converted to a string. The converse can be programmed by using the hd and tl operations on strings and the insert_character operation. We introduce an operation buffer_to_string taking a 'String' argument. This argument is interpreted as the name of the marked text whose text at its turn is to be converted to 'String'. The operation introduced here has no modification rights, except for WTI of course. We use the abstraction function f:'String' -> String and the function string, converting texts into strings.

PROC buffer_to_string: 'String' -> 'String'
MOD WTI

The effect of this operation is described again by a pre- and postcondition style axiom.

AXIOM WTI =>

FORALL s:'String' (app(texts,f(s)))! =>
[ LST t:'String'; t := buffer_to_string(s) ]
( f(t) = string(text(app(texts,f(s)))) )

4.5.11 Termination Axioms

We specify the termination of the procedures of the witef0_op group in the sense that for each procedure invocation there is a state that can be reached as its final state. This is a rather weak form of termination; but if furthermore we happen to know that the algorithmic constructs used in its implementation are deterministic, then it implies the strong form of termination in the sense that every procedure invocation terminates. Actually we relativize the termination axioms by the assertion WTI which was introduced in Section 4.5.5. As a naive attempt, we could try to write the termination axiom as AXIOM WTI => < witef0_op > TRUE, but this axiom is far too weak for being interesting; it just states that at least one operation can terminate for some argument value. So we have to provide one assertion for each operation.

AXIOM WTI => (}
< FLUSH bollp > TRUE;
< FLUSH elollp > TRUE;
< forward_character > TRUE;
< backward_character > TRUE;
< next_line > TRUE;
< previous_line > TRUE;
< beginning_of_line > TRUE;
< end_of_line > TRUE;
< beginning_of_buffer > TRUE;
< end_of_buffer > TRUE;
< set_mark > TRUE;
< exchange_dot_and_mark > TRUE;

FORALL s: 'String' ( valid(f(s)) => < insert_file(s) > TRUE );
FORALL c:Char < insert_character(c) > TRUE;

< newline > TRUE;

FORALL s: 'String' ( app(stxta.f(s))! => < yank_buffer(s) > TRUE );

< delete_next_character > TRUE;
< erase_region > TRUE;

FORALL s: 'String' < erase_buffer(s) > TRUE;
FORALL s: 'String' < copy_region_to_buffer(s) > TRUE;

< FLUSH current_buffer_name > TRUE;

FORALL s: 'String' < write_named_file(s) > TRUE;
FORALL s: 'String' < switch_to_buffer(s) > TRUE;
FORALL s: 'String' < search_forward(s) > TRUE;

FORALL s: 'String'
( app(stxta.f(s))! => < FLUSH buffer_to_string(s) > TRUE )

4.5.12 Connecting the Editor with the Display

The class descriptions FILL_SPEC and LOOK_SPEC of Section 4.3.11 provide enough machinery for formulating the ‘window-invariant’ of an editor. This invariant describes the relation between the current marked text on the one hand and the screen and the cursor of the display on the other hand. Note that WITEFA_SPEC imports APP_DOM_SPEC which encompasses LOOK_SPEC and FILL_SPEC. The main purpose of this section is to introduce one additional
4.5. TEXT EDITING

axiom that applies to the window-and-text facility.

We suppose that we have one window the size of which is given by the values of \( l_1 \) and \( c_0 \) and we leave the treatment of multi-window features as a generalisation for later. We start by introducing an auxiliary function size.

\[
\text{FUNC size: } \to \text{ Nat } \# \text{ Nat}
\]
\[
\text{DEF } l_1, c_0
\]

We introduce several more auxiliaries for notational purposes.

\[
\text{FUNC text: } \to \text{ Text}
\]
\[
\text{DEF } \text{text}(\text{app}(\text{stexts, current}))
\]

\[
\text{FUNC dot: } \to \text{ Nat } \# \text{ Nat}
\]
\[
\text{DEF } \text{dot}(\text{app}(\text{stexts, current}))
\]

Informally the window-invariant states that the window should correspond with a 'look' to the text, if necessary filled with blanks, such that the dot is visible as the cursor.

We do not need a separate concept of 'window'. Instead of that, it is sufficient to deal with the position and the size of a window. The position of a window can be given by a co-ordinate pair \((o_1, o_2)\) that indicates the position of the leftmost-uppermost corner of the window. We refer to this as the screen-origin or just as the origin. The following picture sketches part of the situation.

![Diagram](image)

**Fig 4.4.** The relation between window and text.

We introduce a function \( p_{\text{sub}} \) which describes pair-wise subtraction of co-ordinates. It serves for calculating the position of the cursor from given dot and given screen-origin. We also define a function \( p_{\text{add}} \) which describes pair-wise addition of co-ordinates.
CHAPTER 4. SPECIFICATION OF A TEXT EDITOR

FUNC p_sub: Nat # Nat # Nat # Nat -> Nat # Nat
PAR d1:Nat, d2:Nat, o1:Nat, o2:Nat
DEF sub(d1,o1), sub(d2,o2)

FUNC p_add: Nat # Nat # Nat # Nat -> Nat # Nat
PAR o1:Nat, o2:Nat, s1:Nat, s2:Nat
DEF add(o1,s1), add(o2,s2)

Now we must formulate our window-invariant which we define as a predicate W1. We must also deal with the problem that the text may contain non-printables. Fortunately we earlier (Section 4.3.5) introduced the printify: Text -> Text operation, so this problem gets solved by introducing an application of printify in W1.

PRED W1:
DEF LET origin: Nat # Nat: origin := p_sub(dot,cursor):
LET filled: Text: filled := fill(text,p_add(origin.size));
screen := printify(look(filled,origin,p_add(origin.size)))

The requirement that W1 is an observational invariant should be considered as a part of the specification of our window-and-text facility. This requirement should be added to the axioms WTI1 and WTI2 given in Section 4.5.5 and we do so now by writing another axiom labeled WTI3.

AXIOM (WTI3) WTI -> W1

We must add two remarks about this last axiom. The first remark is that we have chosen for a certain editor-behaviour that is based on similar concepts for the window-oriented subsetting of texts for the horizontal and for the vertical direction. When the text is too large in the vertical direction, it is shortened by the application of vlook. When the text is too large in the horizontal direction, it is shortened by the application of hlook. The user of the editor perceives this as a kind of leftward text-movement when typing long lines. Probably it is dependent on the kind of texts being edited whether this approach is convenient or not.

The second remark is that our specification still leaves a certain amount of implementation freedom, where some of this freedom is unacceptable from an ergonomic point of view. In particular, nothing has been said about the stability of the position of the window with respect to the text. For example there is no formal requirement that always should be tried to re-use the previous origin value unless this turns out impossible.
This concludes the specification of the window-and-text facility.

4.5.13 MOREDOP: More Editing Operations

In this section we shall present a simple editor which can be built on top of the window-and-text facility presented above. This is just a small example editor and more powerful and user-friendly editors can be built in a similar way. The editing primitives of WITEFA_SPEC are of a general-purpose nature in the sense that we can imagine that they are useful for any editor. Now we address the construction of one particular editor with characteristics that it has a fixed number of dedicated buffers and a particular kill/yank mechanism. Therefore we start with the definition of some additional operations which are specific for this editor. They are put together in a class description MOREDOP_SPEC (for MORE Editing Operations). Our simple editor uses precisely three marked texts:

- "mini" which serves for inputting search texts and file names,
- "kill" which will be used in combination with the operations yank_buffer and copy_region-to-buffer, thereby providing cut and paste capabilities at the user-level,
- "main" which essentially contains the text being edited.

The user can switch from "main" to "mini" by escape and from "mini" back again to "main" by either escape or return.

We introduce three 'String' constants which serve as the fixed names for the three marked texts. The procedure startup serves for initialisation. The remaining operations are typical editing commands.

\[
\text{LET MOREDOP_SPEC :=}
\]

\[
\text{IMPORT MAT_SPEC INTO}
\]

\[
\text{IMPORT RGCL_SPEC INTO}
\]

\[
\text{IMPORT CHAR_SPEC INTO}
\]

\[
\text{IMPORT WITEFA_SPEC INTO}
\]

\[
\text{CLASS}
\]

\[
\text{PROC mini: -> 'String'}
\]

\[
\text{DEF cons('n', cons('i', cons('n', cons('i', empty)))})
\]

\[
\text{PROC main: -> 'String'}
\]

\[
\text{DEF cons('n', cons('n', cons('i', cons('n', empty)))})
\]
PROC kill: -> 'String'
DEF cons('k',cons('i',cons('l',cons('l',empty))))

PROC startup: ->
DEF init(mini):
  switch_to_buffer(kill);
  switch_to_buffer(main)

PROC escape: ->
DEF (eq(current_buffer_name,mini) ?; switch_to_buffer(main)
  | NOT eq(current_buffer_name,mini) ?; switch_to_buffer(mini)
  )

PROC return: ->
DEF (eq(current_buffer_name,mini) ?; switch_to_buffer(main)
  | NOT eq(current_buffer_name,mini) ?; newline
  )

PROC delete_to_killbuffer: ->
DEF copy_region_to_buffer(kill);
  erase_region

PROC yank_from_killbuffer: ->
DEF yank_buffer(kill)

PROC search_forward: ->
DEF search_forward(buffer_to_string(mini))

PROC insert_file: ->
DEF insert_file(buffer_to_string(mini))

PROC write_named_file: ->
DEF write_named_file(buffer_to_string(mini))

PROC delete_previous_character: ->
DEF (blop = true ?; SKIP
  | blop = false ?;
      backward_character; delete_next_character
  )

END;

It is interesting to compare the definition of delete_previous_character with the way one would program it on top of EMACS [4] where we assume skip to be defined by (defun (skip (progn))):
(defun (delete-previous-character
  (if (help)
    (skip)
    (progn (backward-character) (delete-next-character))))

4.5.14 KEYBIND: Connecting the Editor with the Keyboard

In order to complete the simple editor, we need a procedure key, say which
takes characters as its argument and which invokes operations described by
WITEFA_SPEC and the additional editing operations from Section 4.5.13. Es-
tentially the procedure key is defined by one large case-statement. In order
to keep things simple, we omitted features such as escape-prefixes [4].

LET KEYBIND_SPEC :=

EXPORT

SORT Nat,
SORT Char,
SORT Text,
PROC startup: -> ,
PROC key : Char -> ,
FUNG screen : -> Text,
FUNG cursor : -> Nat # Nat

FROM
IMPORT NAT_SPEC INTO
IMPORT CHAR_SPEC INTO
IMPORT WITEFA_SPEC INTO
IMPORT MOBEDOP_SPEC INTO
CLASS

PROC key: Char ->
PAR c:Char
DEF ( printable(c) ?; insert_character(c)
| ord(c) = 0 ('@) ?; set_mark
| ord(c) = 1 ('A) ?; beginning_of_line
| ord(c) = 2 ('B) ?; backward_character
| ord(c) = 4 ('D) ?; delete_next_character
| ord(c) = 5 ('E) ?; end_of_line
| ord(c) = 6 ('F) ?; forward_character
| ord(c) = 13 ('M) ?; return
4.6 Related Work

Meyer et. al. [12] describe a strategy for displaying structured objects such as programs on a screen of limited size. They develop a formal model of the screen allocation, called 'calculus of windows'. Feldman [13] describes a text editor in a functional style using FP. Just as in our formalisation, text is considered to be a sequence of lines, each of which is a sequence of characters. Gutknecht [15] describes the text editor LARA. This seems to be a kind of combination between a formatting system and an editor, based on the what-you-see-is-what-you-get principle. If we compare this with our approach, it can be seen that Gutknecht puts much more (supposed) knowledge about the structure of documents into his editor than we did.

Sufrin [14] gives an elaborate specification of a display-oriented text editor. Unlike we do, he considers text as a sequence of characters with new-line symbols. His texts are somewhat similar to our marked texts in the sense that they have one pointer to a position in the text associated with them. This is done by defining text such that each text consists of two sequences, which should be appended and the splitting-point corresponds with the dot position.

Partsch [16] describes the specification and transformation process of a simple line-oriented editor using a sugared version of CIP-L. First of all, we should note that Partsch is in a comfortable position since he assumes a lot of syntactic sugar. Since COLD-K is a kernel language it is somewhat more Spartan and we assumed many features not to be available (yet). The text
editor in [16] is single-buffer and line-oriented. There is nothing comparable with our window-invariant W1. Furthermore he does not deal with file handling. However it should be remarked that he describes undo facilities, some of which in our case are absent. A text is considered as a sequence of lines, just as in our formalisation. In the formal model of the editor, the text being edited is called the “current text file” and it is modelled as a triple \((t_1, l, t_2)\), where \(t_1\) is the text before the current line, \(l\) is the current line and \(t_2\) is the text after the current line.

Guttag and Hornig [17] present a formal state-based model of a simple display, comparable with our DISPLAY_SPEC. They also present a formalisation of ‘text’ including pictures, views and line/paragraph-breaking facilities. In particular, a text is viewed as a sequence of paragraphs where each paragraph at its turn is a formatted English string.

### 4.7 Looking Back

In this section, we shall in retrospect summarise the main lines of the work presented in this chapter. One of the main purposes of constructing the formal specification presented in this chapter was to illustrate the use of formal specification techniques. In particular, we wanted to show how the language COLD-K can be used as a tool for describing complex systems. Let us explicitly point out some of the interesting points encountered during this case study.

The standard class descriptions of Appendix B provide us with the possibility to use standard mathematical data types such as natural numbers, sequences, maps etc. It is hard to imagine how we could specify an editor without using such data types. These standard class descriptions have been written in a certain axiomatic style and this style is different from the style of specification used during the rest of our editor specification. This observation is encouraging, for it suggests that the ability to re-invent the specification of all these standard mathematical data types need not be part of the skills of the average software developer willing to use formal specification techniques. This is why we put BOOL_SPEC, NAT_SPEC etc. in an appendix. These class descriptions are indispensable but atypical.

After the introduction of Section 4.1 we investigated in Section 4.3 the important concept of text and algebraic operations on text. At several occasions we used the mathematical data type of \(\text{Seq}\) from Appendix B. We introduced several operations on texts and instead of using an axiomatic approach, we just defined these operations. In particular, we used recursive definitions which are easily executable. This means that in an early phase of the spec-
ification construction some experiments can be done. The possibility to do
such experiments can be considered as a tool for establishing a close corre-
spondence between the necessary intuition about these operations and the
formal definitions. Of course this may not be the only tool and therefore we
immediately mention the tool of reasoning about the formal definitions. We
searched for suitable algebraic laws for our operations. It was our experi-
ence that this helps in establishing a set of well-understood operations with
useful notations. Let us mention one example: when we first defined the
operation named \texttt{hsplit} in Section 4.3.11 we thought of it as 'vertical split'
because there is a vertical splitting-line. Later, when looking for its inverse,
this turned out to be \texttt{h_add} of Section 4.3.9. But then we immediately saw
the inconsistency in the terminology and we changed from 'vertical split' to
'horizontal split'.

We showed that elementary and almost trivial operations form the basis for
powerful cut and paste operations, which indeed correspond with cut and
paste capabilities as found in editors. Similarly the elementary \texttt{hsplit} and
\texttt{vsplit} form the basis for more complicated operations that play a role in
the necessary subsetting of texts as required due to the physical limitations
of the display device.

We 'discovered' a wealth of algebraic systems related to text such as the alge-
bras of strings, reaches and profiles. We investigated the mappings between
them which turned out to have interesting properties. As has been shown,
the knowledge of these algebraic systems and the mappings between them
can be used for specification purposes. We also believe that this knowledge
is equally fruitful when implementing a text processing system.

In Section 4.4 we described several class descriptions which are important
for interfacing the editor with its environment. In Section 4.4.2 we described
a display, which can be viewed as a component that is not really part of
the actual editor, but which is indispensable for the editor. We think that
this phenomenon to include such descriptions is typical for the specification
of many realistic and relevant systems. An important methodological point
was demonstrated by introducing the display-invariant D1 A D12 A D13 first.
In this way we had a consistency-criterion for checking the pre- and postcon-
dition style specifications that were written after that.

We introduced the sorts 'Seq' and 'String', deliberately facing the prob-
lems that if we want to provide the implementation of a data type ourselves,
then this data type probably comes with an eq-predicate rather than with
just equality. Of course one is lucky if some data type is built-in into the
programming language and if it comes with a usable equality; but we wanted
to show the specification techniques required when this is not the case.
In Section 4.4.4 we introduced a file system, which also can be viewed as a component that is not really part of the actual editor, but that still is indispensable. The most important guideline when introducing this file system, was to restrict ourselves. The complete description of a file system is a serious enterprise in itself and we had to cut-off many possibilities.

In Section 4.5 we started describing the actual text editor. The study of algebraic operations on texts turned out to be fruitful. For example, the *copy-region-to-buffer* and *yank-buffer* operations are just direct applications of certain algebraic operators. We were able to use the Procrustean operations of Section 4.3.11 for formalising the display-management of the editor in a compact and abstract way.

In Section 4.5.5 we discussed several notions of 'invariant' and finally chose for the option of observational invariant. Still, the other notions of invariance are useful in their own right; e.g. the definition of observational invariant refers to the existence of a classical invariant. Furthermore we had a kind of 'almost invariance' of TT in the sense that \( \text{init} \) establishes TT and that \( \text{WTI} \Rightarrow [ \text{wita}_a \_\text{op} 1 \_\text{TI} \). The latter property constitutes already a consistency check on our specification. In Sections 4.5.6 to 4.5.10 we demonstrated another style of pre- and postcondition-style axioms, by writing the postconditions in-situ in the axioms.

The formal specification of Section 4.5 covers a number of important aspects of a text editor, although the editor described is relatively poor in its bells and whistles. However it is far from trivial and its functionality makes it a usable editor. We were able to cover several interesting features also present in other text editors and in particular in EMACS.

It is important to notice that the application-domain specific framework would have enabled several other choices for the editor behaviour as well. E.g. the question whether the dot should or should not stay within the boundaries of the current text could be discussed. Similarly we had got all notations and techniques for introducing an arbitrary number of markers at hand.

We deliberately did not model the current text as a pair of sequences \((t_1, t_2)\) where \( t_1 \) is the text before the current position and \( t_2 \) is the text after it. Essentially this is the model adopted by [14] and [16]. This model more or less collapses if one wants many markers or a marker that goes outside the text. The price we had to pay for our 'marked-text-as-triples' approach is that occasionally we had to describe clumsy re-calculation of co-ordinate pairs.

We showed how \( \lambda \tau \)-calculus and COLD-K can be used as a tool for the description of a relatively large and complex software system. By doing so
we illustrated a number of general-purpose specification techniques. This chapter presents one more example of a large formal specification and as such, it can be viewed as is a contribution to the advancement of formal specification techniques in general. Note that the ability to construct large formal specifications supports the applicability of the notions of component, black-box description and design (Chapter 2) and the correctness-preserving transformations of designs investigated in Chapter 3. In Chapter 5 we apply the techniques of Chapter 2 and 3 in an implementation activity where the current chapter serves as a starting point. We postpone a more complete evaluation of the editor case study until the end of this implementation activity.
Bibliography


[16] H. Partsch, From informal requirements to a running program: a case study in algebraic specification and transformational programming. Internal Report 87-7, Department of Informatics, Faculty of Science, University of Nijmegen.

Appendix A

List of Symbols

In this appendix we give a list of the sorts, functions, predicates and procedures used. For each symbol the list contains a short informal description. The list has been subdivided into a number of sub-lists. The first sub-list contains the symbols used from B. The second sub-list contains the symbols that are introduced in Section 4.3. The third sub-list contains the symbols that are introduced in Section 4.4 etc.

Symbols from the standard class descriptions

<table>
<thead>
<tr>
<th>Bool</th>
<th>Booleans</th>
</tr>
</thead>
<tbody>
<tr>
<td>true : → Bool</td>
<td>constant true</td>
</tr>
<tr>
<td>false: → Bool</td>
<td>constant false</td>
</tr>
<tr>
<td>not : Bool → Bool</td>
<td>negation</td>
</tr>
<tr>
<td>and : Bool² → Bool</td>
<td>conjunction</td>
</tr>
<tr>
<td>or : Bool² → Bool</td>
<td>disjunction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nat</th>
<th>natural numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero: → Nat</td>
<td>constant 0</td>
</tr>
<tr>
<td>succ : Nat → Nat</td>
<td>successor</td>
</tr>
<tr>
<td>pred: Nat → Nat</td>
<td>predecessor</td>
</tr>
<tr>
<td>less : Nat²</td>
<td>less than</td>
</tr>
<tr>
<td>leq : Nat²</td>
<td>less or equal</td>
</tr>
<tr>
<td>gtr : Nat²</td>
<td>greater than</td>
</tr>
<tr>
<td>geq : Nat²</td>
<td>greater or equal</td>
</tr>
<tr>
<td>add : Nat² → Nat</td>
<td>addition</td>
</tr>
<tr>
<td>sub : Nat² → Nat</td>
<td>subtraction</td>
</tr>
<tr>
<td>mul : Nat² → Nat</td>
<td>multiplication</td>
</tr>
<tr>
<td>div : Nat² → Nat</td>
<td>division</td>
</tr>
<tr>
<td>0: → Nat</td>
<td>constant 0</td>
</tr>
<tr>
<td>1: → Nat</td>
<td>constant 1</td>
</tr>
</tbody>
</table>
CHAPTER 4. SPECIFICATION OF A TEXT EDITOR

\[ 2 : \rightarrow \text{Nat} \]
\[ \text{etc.} \]

\[ \text{constant } 2 \]

\[ \text{Char} \]
\[ \text{ord: Char } \rightarrow \text{Nat} \]
\[ \text{chr: Nat } \rightarrow \text{Char} \]
\[ 'a' : \rightarrow \text{Char} \]
\[ '1' : \rightarrow \text{Char} \]
\[ \text{etc.} \]

\[ \text{ASCII characters} \]
\[ \text{conversion function} \]
\[ \text{conversion function} \]
\[ \text{constant 'a'} \]
\[ \text{constant '1'} \]

\[ \text{Tup} \]
\[ \text{tup : Item1 } \# \text{Item2 } \rightarrow \text{Tup} \]
\[ \text{proj1: Tup } \rightarrow \text{Item1} \]
\[ \text{proj2: Tup } \rightarrow \text{Item2} \]
\[ \text{Tup} \]
\[ \text{tup : Item1 } \# \text{Item2 } \# \text{Item3 } \rightarrow \text{Tup} \]
\[ \text{proj1: Tup } \rightarrow \text{Item1} \]
\[ \text{proj2: Tup } \rightarrow \text{Item2} \]
\[ \text{proj3: Tup } \rightarrow \text{Item3} \]

\[ \text{2-tuples (= pairs)} \]
\[ \text{pairing} \]
\[ \text{taking first field} \]
\[ \text{taking second field} \]
\[ \text{3-tuples (=triples)} \]
\[ \text{triple construction} \]
\[ \text{taking first field} \]
\[ \text{taking second field} \]
\[ \text{taking third field} \]

\[ \text{Set} \]
\[ \text{is_in: Item } \# \text{Set} \]
\[ \text{empty: } \rightarrow \text{Set} \]

\[ \text{finite sets} \]
\[ \text{element predicate} \]
\[ \text{empty set} \]

\[ \text{Seq} \]
\[ \text{empty: } \rightarrow \text{Seq} \]
\[ \text{cons : Item } \# \text{Seq } \rightarrow \text{Seq} \]
\[ \text{hd} : \text{Seq } \rightarrow \text{Item} \]
\[ \text{tl} : \text{Seq } \rightarrow \text{Seq} \]
\[ \text{len} : \text{Seq } \rightarrow \text{Nat} \]
\[ \text{sel : Seq } \# \text{Nat } \rightarrow \text{Item} \]
\[ \text{cat : Seq } \# \text{Seq } \rightarrow \text{Seq} \]
\[ \text{rev : Seq } \rightarrow \text{Seq} \]
\[ \text{bag : Seq } \rightarrow \text{Bag} \]

\[ \text{finite sequences} \]
\[ \text{empty sequence} \]
\[ \text{sequence construction} \]
\[ \text{head} \]
\[ \text{tail} \]
\[ \text{length} \]
\[ \text{selection} \]
\[ \text{concatenation} \]
\[ \text{reversal} \]
\[ \text{conversion to bag} \]

\[ \text{Map} \]
\[ \text{empty: } \rightarrow \text{Map} \]
\[ \text{add : Map } \# \text{Item1 } \# \text{Item2 } \rightarrow \text{Map} \]
\[ \text{rem : Map } \# \text{Item1 } \rightarrow \text{Map} \]
\[ \text{app : Map } \# \text{Item1 } \rightarrow \text{Item2} \]
\[ \text{dom : Map } \rightarrow \text{Set1} \]
\[ \text{ran : Map } \rightarrow \text{Set2} \]

\[ \text{finite mappings} \]
\[ \text{empty mapping} \]
\[ \text{'overwriting' addition} \]
\[ \text{removal} \]
\[ \text{map-application} \]
\[ \text{domain} \]
\[ \text{range} \]
Symbols concerning Text and Algebraic Operations on Texts

Line
Text
niltext : → Text
zero : → Text
addempty : Text → Text
addchar : Char # Text → Text
first : Text → Char
rest : Text → Text

String
empty : → String
cons : Char # String → String
hd : String → Char
tl : String → String
len : String → Nat
sel : String # Nat → Char
cat : String # String → String
less : String # String
ctr_j : → Char
ok : Text
text : String → Text
string : Text → String
blank : → Char
tilde : → Char
printable : Char
printify : Char → Char
printify : Line → Line
printify : Text → Text

split : Text # Nat^2 → Text^2
add : Text^2 → Text
cut : Text # Nat^2 # Nat^2 → Text^2
paste : Text^2 # Nat^2 → Text
reach : Text → Nat^2
add : Nat^2 # Nat^2 → Nat^2
split : Nat^2 # Nat^2 → Nat^2 # Nat^2
cut : Nat^2 # Nat^2 # Nat^2 → Nat^2 # Nat^2
paste : Nat^2 # Nat^2 # Nat^2 → Nat^2
less : Nat^2 # Nat^2
leq : Nat^2 # Nat^2

Profile
profile : Text → Profile

lines (= sequences of char)
texts (= sequences of lines)
the text with 0 line
the text with one empty line
put an empty line before ..
put a character in front of ..
first character
text except for its first char
strings (= sequences of char)
empty string
string construction
head
tail
length
selection
concatenation
lexicographical ordering
control-j
non-niltext and no control-j’s
conversion
conversion
constant ‘’
constant “”
printable (no control char)
make printable
make printable
make printable
inverse of natural addition
natural addition of texts
cutting a piece out of a text
pasting one text into another
position immediately after
addition of reaches
splitting of reaches
cutting of reaches
pasting of reaches
strictly less than
less or equal
profiles (= sequences of Nat)
the profile of a text
intext  : Text # Nat²
nilprofile: → Profile
v_add   : Text² → Text
h_add   : Text² → Text
empty:a : Nat → Text
match   : String² # Nat
match'  : String² # Nat
search  : String² → Nat
search' : Text² → Nat²
split   : String # Nat → String²
vfill   : Text # Nat → Text
blanks  : Nat → Line
hfill   : Text # Nat → Text
fill    : Text # Nat² → Text
hsplit  : Line # Nat → Line # Line
hsplit' : Text # Nat → Text²
vsplit  : Text # Nat → Text²
hlook   : Text # Nat² → Text
vlook   : Text # Nat² → Text
look    : Text # Nat² # Nat² → Text

position being in boundaries
profile of niltext
vertical addition
horizontal addition
text with empty lines
string matching
string matching with sentinel
search operation
search operation
splitting a string
vertical filling
line with blank characters
horizontal filling
filling in two directions
horizontal splitting
horizontal splitting
vertical splitting
horizontal look
vertical look
text subsetted by 'window'

Symbols concerning Interfacing the Editor with its Environment

li: → Nat
col: → Nat
PROC cr: →
PROC nl: →
PROC bc: →
PROC ce: →
PROC cl: →
PROC nd: →
PROC up: →
PROC cm: Nat² →
PROC print: Char →
PROC displ_op: →
screen: → Text
cursor: → Nat²

number of lines
number of columns
carriage return
newline
backwards cursor
clear to end-of-line
erase display
move cursor right
move cursor up
cursor motion
printable character processing
arbitrary display command
observable contents of screen
cursor position
display-invariant (conjunct 1)
display-invariant (conjunct 2)
display-invariant (conjunct 3)

blank_text: Nat → Text
blank_text: Nat² → Text

single-line text of blanks
text with blanks
post_cr: Text → Nat² # Text # Nat²
post_ml: Text → Nat² # Text # Nat²
post_bc: Text → Nat² # Text # Nat²
post_ce: Text → Nat² # Text # Nat²
post_cl: Text → Nat² # Text # Nat²
post_md: Text → Nat² # Text # Nat²
post_up: Text → Nat² # Text # Nat²
post_cm: Text → Nat² # Nat² # Text # Nat²
post_print: Text → Nat² # Char # Text # Nat²

'Item'
seq : 'Item' # 'Item'
item_inv:
seq_inv :
f : 'Item' → Item
f : 'Seq' → Seq

'Seq'
empty: → 'Seq'
PROC cons: 'Item' # 'Seq' → 'Seq'
hd : 'Seq' → 'Item'
tl : 'Seq' → 'Seq'
eq : 'Seq' # 'Seq'
sel: 'Seq' # Nat → 'Item'
PROC cat: 'Seq' # 'Seq' → 'Seq'
PROC rev: 'Seq' → 'Seq'

'String'
f : 'String' → String
string_inv:
empty: → 'String'
PROC cons : Char # 'String' → 'String'
hd : 'String' → Char
tl : 'String' → 'String'
eq : 'String' # 'String'
sel: 'String' # Nat → Char
PROC cat: 'String' # 'String' → 'String'
PROC rev: 'String' → 'String'
less: 'String' # 'String'
valid : String
file : String → String
pos : String → Nat
PROC rewrite: 'String' →
PROC reset : 'String' →

data type representing items
equivalence relation
invariant needed for 'Item'
invariant needed for 'Seq'
abstraction function
abstraction function
implementable sequences
representation of empty seq.
constructor procedure
head
tail
external notion of equality
selection
concatenation
reversal

implementable strings
abstraction function
invariant needed for 'String'
representation of empty string
constructor procedure
head
tail
external notion of equality
selection
concatenation
reversal
lexicographic ordering
existence of a file
contents of file
position of read-pointer
open file for writing
open file for reading
PROC read : → Char
PROC write : Char →
eof:

reading from file
writing to file
end-of-file predicate

Symbols concerning Text Editing

\(\text{Copa} : \text{Copa} \rightarrow \text{Nat} \)
\(\text{v} : \text{Copa} \rightarrow \text{Nat} \)
\(\text{h} : \text{Copa} \rightarrow \text{Nat} \)
\(\text{copia} : \text{Nat}^2 \rightarrow \text{Copa} \)

co-ordinate pairs
vertical co-ordinate
horizontal co-ordinate
pairing

\(\text{MText} \)
\(\text{text} : \text{MText} \rightarrow \text{Text} \)
\(\text{dot} : \text{MText} \rightarrow \text{Copa} \)
\(\text{mark} : \text{MText} \rightarrow \text{Copa} \)
\(\text{stext} : \text{Text} \# \text{Copa} \# \text{Copa} \rightarrow \text{MText} \)
\(\text{dot} : \text{MText} \rightarrow \text{Nat}^2 \)
\(\text{mark} : \text{MText} \rightarrow \text{Nat}^2 \)
\(\text{modtext} : \text{MText} \# \text{Text} \rightarrow \text{MText} \)
\(\text{moddot} : \text{MText} \# \text{Copa} \rightarrow \text{MText} \)
\(\text{modmark} : \text{MText} \# \text{Copa} \rightarrow \text{MText} \)
\(\text{atext} : \rightarrow \text{Map} \text{VAR} \)
\(\text{current} : \rightarrow \text{String} \text{VAR} \)
PROC init: 'String' →
PROC write_file_op: →

select text field
select dot field
select mark field
marked text construction
dot of a marked text
mark of a marked text
modification operation
modification operation
modification operation
collection of marked texts
name 'current' marked text
initialisation procedure
text-invariant (conjunct 1)
text-invariant (conjunct 2)
text-invariant
window and text invariant
arbitrary operation of ...

PROC bold: → Bool
PROC eolp: → Bool
PROC forward_character: →
PROC backward_character: →
PROC next_line: →
PROC previous_line: →
PROC beginning_of_line: →
PROC end_of_line: →
PROC beginning_of_buffer: →
PROC end_of_buffer: →
PROC set_mark: →
PROC exchange_dot_and_mark: →
PROC insert_file: 'String' →

'writefa' operation
'writefa' operation
'writefa' operation
'writefa' operation
'writefa' operation
'writefa' operation
'writefa' operation
'writefa' operation
'writefa' operation
'writefa' operation
A. List of Symbols

PROC insert_character : Char → 'witsfa' operation
PROC newline : → 'witsfa' operation
PROC yank_buffer : 'String' → 'witsfa' operation
PROC delete_next_character : → 'witsfa' operation
PROC erase_region : → 'witsfa' operation
PROC erase_buffer : 'String' → 'witsfa' operation
PROC copy_region_to_buffer : 'String' → 'witsfa' operation
PROC current_buffer_name : → 'String'
PROC write_named_file : 'String' →
PROC switch_to_buffer : 'String' → 'witsfa' operation
PROC search_forward : 'String' → 'witsfa' operation
PROC buffer_to_string : 'String' → 'String'
PROC size : → Nat²
PROC text : → Text
PROC dot : → Nat²
PROC p_sub : Nat² # Nat² → Nat²
PROC p_add : Nat² # Nat² → Nat²
PROC mini : → 'String'
PROC main : → 'String'
PROC kill : → 'String'
PROC startup : →
PROC escape : →
PROC return : →
PROC delete_to_kilbuffer : →
PROC yank_from_kilbuffer : →
PROC search_forward : →
PROC insert_file : →
PROC write_named_file : →
PROC delete_previous_character : →
PROC key : Char →

size : → Nat²

the pair (li,co)
current text
current dot
pair-wise subtraction
pair-wise addition
window-invariant

PROC mini : → 'String'
PROC main : → 'String'
PROC kill : → 'String'
PROC startup : →
PROC escape : →
PROC return : →
PROC delete_to_kilbuffer : →
PROC yank_from_kilbuffer : →
PROC search_forward : →
PROC insert_file : →
PROC write_named_file : →
PROC delete_previous_character : →
top-level operation of editor
Appendix B

Standard Class Descriptions

DESIGN

% This is a specification of the data type of booleans with
% inductive definitions for the non-constructor operations.
% The inductive definitions have the shape of truth tables.

LET BOOL_SPEC :=

EXPORT

SORT Bool,
FUNC true : -> Bool,
FUNC false: -> Bool,
FUNC not : Bool -> Bool,
FUNC and : Bool # Bool -> Bool,
FUNC or : Bool # Bool -> Bool,
FUNC isp : Bool # Bool -> Bool,
FUNC eqv : Bool # Bool -> Bool,
FUNC xor : Bool # Bool -> Bool

FROM

CLASS

SORT Bool
FUNC true : -> Bool
FUNC false: -> Bool

AXIOM \{BOOL1\} true!:
\{BOOL2\} false!:
\{BOOL3\} NOT true = false
B. STANDARD CLASS DESCRIPTIONS

PRED is_gen: Bool
IND is_gen(true);
is_gen(false)

AXIOM FORALL b:Bool
{BOOL4} is_gen(b)

FUNC not: Bool -> Bool
IND not(true ) = false;
not(false) = true

FUNC and: Bool # Bool -> Bool
IND and(false,false) = false;
and(false,true ) = false;
and(true ,false) = false;
and(true ,true ) = true

FUNC or: Bool # Bool -> Bool
IND or(false,false) = false;
or(false,true ) = true;
or(true ,false) = true;
or(true ,true ) = true

FUNC imp: Bool # Bool -> Bool
IND imp(false,false) = true;
imp(false,true ) = true
imp(true ,false) = false;
imp(true ,true ) = true

FUNC eqv: Bool # Bool -> Bool
IND eqv(false,false) = true;
eqv(false,true ) = false;
eqv(true ,false) = false;
eqv(true ,true ) = true

FUNC xor: Bool # Bool -> Bool
IND xor(false,false) = false;
xor(false,true ) = true;
xor(true ,false) = true;
xor(true ,true ) = false

END;
% This is a specification of the data type of natural numbers
% with inductive definitions for the non-constructor operations.

LET NAT_SPEC' :=

EXPORT

SORT Nat,
FUNC zero: -> Nat,
FUNC succ: Nat -> Nat,
FUNC pred: Nat -> Nat,
PREM les: Nat # Nat,
PREM leq: Nat # Nat,
PREM gtr: Nat # Nat,
PREM geq: Nat # Nat,
FUNC add: Nat # Nat -> Nat,
FUNC sub: Nat # Nat -> Nat,
FUNC mul: Nat # Nat -> Nat,
FUNC div: Nat # Nat -> Nat,
FUNC mod: Nat # Nat -> Nat,
FUNC exp: Nat # Nat -> Nat,
FUNC log: Nat # Nat -> Nat,
FUNC max: Nat # Nat -> Nat,
FUNC min: Nat # Nat -> Nat

FROM
CLASS

SORT Nat
FUNC zero: -> Nat
FUNC succ: Nat -> Nat

AXIOM
\( (\text{Nat1}) \) zero!
\( \forall n, m. \) Nat \( n \) Nat \( m \) Nat
\( (\text{Nat2}) \) succ(\( m \))!
\( (\text{Nat3}) \) NOT succ(\( m \)) = zero;
\( (\text{Nat4}) \) succ(\( m \)) = succ(\( n \)) \Rightarrow m = n \)

PREM is_gen: Nat
IND \( \forall \) Nat \( m \)
\( \forall \) Nat \( n \)
\( ( \text{is_gen(zero)} ) \)
\( \text{is_gen}(m) \Rightarrow \text{is_gen}(\text{succ}(n)) \)
AXIOM  FORALL  n:Nat  
(NATS)  is_gen(n)

FUNC  pred: Nat -> Nat  
IND  FORALL  n:Nat  
(n: pred(succ(n)) = n )

PRED  lss: Nat # Nat  
IND  FORALL  m:Nat,n:Nat  
(lss(m,succ(n));  
  lss(m,n) => lss(m,succ(n)) )

PRED  leq: Nat # Nat  
IND  FORALL  m:Nat,n:Nat  
(leq(n,m);  
  leq(m,n) => leq(m,succ(n)) )

PRED  gtr: Nat # Nat  
IND  FORALL  m:Nat,n:Nat  
(gtr(succ(m),m);  
  gtr(m,n) => gtr(succ(n),n) )

PRED  geq: Nat # Nat  
IND  FORALL  m:Nat,n:Nat  
(geq(m,m);  
  geq(n,m) => geq(succ(m),n) )

FUNC  add: Nat # Nat -> Nat  
IND  FORALL  n:Nat, n:Nat  
(add(n,zero) = m;  
  add(m,succ(n)) = succ(add(m,n)) )

FUNC  sub: Nat # Nat -> Nat  
IND  FORALL  m:Nat,n:Nat  
(sub(m,zero) = m;  
  gtr(m,n) => sub(m,succ(n)) = pred(sub(m,n)) )

FUNC  mul: Nat # Nat -> Nat  
IND  FORALL  n:Nat,n:Nat  
(mul(n,zero) = zero;  
  mul(m,succ(n)) = add(mul(m,n),n) )
\\textbf{FUNC} div: \textit{Nat} \times \textit{Nat} \rightarrow \textit{Nat} \\
\textbf{IND} \quad \textbf{FORALL} \quad m: \textit{Nat}, n: \textit{Nat}, q: \textit{Nat}, r: \textit{Nat} \\
\quad ( m = \text{add}(\text{mul}(m, q), r) \text{ AND } \text{lss}(r, n) \rightarrow \text{div}(m, n) = q )

\\textbf{FUNC} mod: \textit{Nat} \times \textit{Nat} \rightarrow \textit{Nat} \\
\textbf{IND} \quad \textbf{FORALL} \quad m: \textit{Nat}, n: \textit{Nat}, q: \textit{Nat}, r: \textit{Nat} \\
\quad ( m = \text{add}(\text{mul}(m, q), r) \text{ AND } \text{lss}(r, n) \rightarrow \text{mod}(m, n) = r )

\\textbf{FUNC} exp: \textit{Nat} \times \textit{Nat} \rightarrow \textit{Nat} \\
\textbf{IND} \quad \textbf{FORALL} \quad m: \textit{Nat}, n: \textit{Nat} \\
\quad ( \text{exp}(m, \text{zero}) = \text{succ}(\text{zero}) ; \\
\quad \text{exp}(m, \text{succ}(n)) = \text{mul}(m, \text{exp}(m, n)) )

\\textbf{FUNC} log: \textit{Nat} \times \textit{Nat} \rightarrow \textit{Nat} \\
\textbf{IND} \quad \textbf{FORALL} \quad m: \textit{Nat}, n: \textit{Nat}, p: \textit{Nat} \\
\quad ( \text{lseq}(\text{exp}(m, n), p) \text{ AND } \text{lss}(n, \text{exp}(m, \text{succ}(p))) \rightarrow \text{log}(m, n) = p )

\\textbf{FUNC} max: \textit{Nat} \times \textit{Nat} \rightarrow \textit{Nat} \\
\textbf{IND} \quad \textbf{FORALL} \quad m: \textit{Nat}, n: \textit{Nat} \\
\quad ( \text{geq}(m, n) \rightarrow \text{max}(m, n) = m ; \\
\quad \text{lseq}(m, n) \rightarrow \text{max}(m, n) = n )

\\textbf{FUNC} min: \textit{Nat} \times \textit{Nat} \rightarrow \textit{Nat} \\
\textbf{IND} \quad \textbf{FORALL} \quad m: \textit{Nat}, n: \textit{Nat} \\
\quad ( \text{lseq}(m, n) \rightarrow \text{min}(m, n) = m ; \\
\quad \text{geq}(m, n) \rightarrow \text{min}(m, n) = n )

\textbf{END};
% This is a specification of the data type of natural numbers
% based on NAT_SPEC', providing notations for some specific numbers.

LET NAT_SPEC :=

IMPORT NAT_SPEC' INTO

CLASS

FUNC 0 : -> Nat DEF zero
FUNC 1 : -> Nat DEF succ(zero)
FUNC 2 : -> Nat DEF succ(succ(zero))
FUNC 3 : -> Nat \% etc.
FUNC 4 : -> Nat
FUNC 5 : -> Nat
FUNC 6 : -> Nat
FUNC 7 : -> Nat
FUNC 8 : -> Nat
FUNC 9 : -> Nat

FUNC 10 : -> Nat
FUNC 11 : -> Nat
FUNC 12 : -> Nat
FUNC 13 : -> Nat
FUNC 14 : -> Nat
FUNC 15 : -> Nat
FUNC 16 : -> Nat
FUNC 17 : -> Nat
FUNC 18 : -> Nat
FUNC 19 : -> Nat

FUNC 20 : -> Nat
FUNC 21 : -> Nat
FUNC 22 : -> Nat
FUNC 23 : -> Nat
FUNC 24 : -> Nat
FUNC 25 : -> Nat
FUNC 26 : -> Nat
FUNC 27 : -> Nat
FUNC 28 : -> Nat
FUNC 29 : -> Nat
FUNC 30 : -> Nat
FUNC 31 : -> Nat
FUNC 32 : -> Nat
FUNC 72 : -> Nat
FUNC 74 : -> Nat
FUNC 89 : -> Nat
FUNC 97 : -> Nat
FUNC 98 : -> Nat
FUNC 99 : -> Nat
FUNC 100 : -> Nat
FUNC 101 : -> Nat
FUNC 102 : -> Nat
FUNC 103 : -> Nat
FUNC 104 : -> Nat
FUNC 105 : -> Nat
FUNC 108 : -> Nat
FUNC 107 : -> Nat
FUNC 106 : -> Nat
FUNC 109 : -> Nat
FUNC 110 : -> Nat
FUNC 111 : -> Nat
FUNC 112 : -> Nat
FUNC 113 : -> Nat
FUNC 114 : -> Nat
FUNC 115 : -> Nat
FUNC 116 : -> Nat
FUNC 117 : -> Nat
FUNC 118 : -> Nat
FUNC 119 : -> Nat
FUNC 120 : -> Nat
FUNC 121 : -> Nat
FUNC 122 : -> Nat

FUNC 126 : -> Nat
FUNC 127 : -> Nat
FUNC 128 : -> Nat
FUNC 129 : -> Nat

END:
B. STANDARD CLASS DESCRIPTIONS

% This is a specification of the data type of ASCII characters.

LET CHAR_SPEC :=

EXPORT

SORT Char,
SORT Nat,
FUNC ord: Char -> Nat,
FUNC chr: Nat -> Char,
FUNC minchar: -> Char,
FUNC maxchar: -> Char,
FRED isschar: Char # Char

FROM
IMPORT NAT_SPEC INTO
CLASS

SORT Char
FUNC min: -> Nat DEF 0
FUNC max: -> Nat DEF 127

FUNC ord: Char -> Nat
FUNC chr: Nat -> Char
(ord is an embedding and
chr is its inverse, i.e. a conversion function)

FRED dom: Nat
IND FORALL m:Nat
  ( leq(min,m) AND leq(m,max) => dom(m) )

FRED is_gen: Char
IND FORALL m:Nat ( dom(m) => is_gen(chr(m)) )

AXIOM (EMBEDDING)

{1} FORALL m:Nat ( chr(m):<=> dom(m) )
{2} FORALL m:Nat ( dom(m) => (ord(chr(m)) = m) )
{3} FORALL c:Char ( is_gen(c) )

FUNC minchar: -> Char DEF chr(min)

FUNC maxchar: -> Char DEF chr(max)
PRED lsschar: Char # Char
PAR c:Char, d:Char
DEF lss(ord(c),ord(d))

END;
This is a specification of the data type of ASCII characters based on CHAR_SPEC, providing notations for some specific characters.

```
LET CHAR_SPEC :=

IMPORT CHAR_SPEC INTO
IMPORT NAT_SPEC INTO
CLASS

FUNC bell: -> Char DEF chr(7)
FUNC tab: -> Char DEF chr(9)

FUNC 'a': -> Char DEF chr(97)
FUNC 'b': -> Char DEF chr(98)
FUNC 'c': -> Char DEF chr(99)
FUNC 'd': -> Char DEF chr(100)
FUNC 'e': -> Char DEF chr(101)
FUNC 'f': -> Char DEF chr(102)
FUNC 'g': -> Char DEF chr(103)
FUNC 'h': -> Char DEF chr(104)
FUNC 'i': -> Char DEF chr(105)
FUNC 'j': -> Char DEF chr(106)
FUNC 'k': -> Char DEF chr(107)
FUNC 'l': -> Char DEF chr(108)
FUNC 'm': -> Char DEF chr(109)
FUNC 'n': -> Char DEF chr(110)
FUNC 'o': -> Char DEF chr(111)
FUNC 'p': -> Char DEF chr(112)
FUNC 'q': -> Char DEF chr(113)
FUNC 'r': -> Char DEF chr(114)
FUNC 's': -> Char DEF chr(115)
FUNC 't': -> Char DEF chr(116)
FUNC 'u': -> Char DEF chr(117)
FUNC 'v': -> Char DEF chr(118)
FUNC 'w': -> Char DEF chr(119)
FUNC 'x': -> Char DEF chr(120)
FUNC 'y': -> Char DEF chr(121)
FUNC 'z': -> Char DEF chr(122)

END;
```
LET ITEM :=
CLASS
  SORT Item FREE
END;

LET ITEM1 :=
CLASS
  SORT Item1 FREE
END;

LET ITEM2 :=
CLASS
  SORT Item2 FREE
END;

LET ITEM3 :=
CLASS
  SORT Item3 FREE
END;
% This is a specification of a strict linear ordering.

LET SLO :=

CLASS

SORT Item  FREE
PRED x: Item \& Item

AXIOM FORALL i:Item,j:Item,k:Item (
  {SLO1} NOT r(i,i):
  {SLO2} r(i,j) AND r(j,k) \rightarrow r(i,k):
  {SLO3} r(i,j) OR r(j,i) OR (i = j)
)

END:
% This is an axiomatic specification of the 2-tuple data type
% with inductive definitions for the non-constructor operations.

LET TUP2_SPEC :=

LAMBDA X : ITEM1 OF
LAMBDA Y : ITEM2 OF
EXPORT

SORT Tup,
SORT Item1,
SORT Item2,
FUNC tup : Item1 # Item2 -> Tup,
FUNC proj1: Tup -> Item1,
FUNC proj2: Tup -> Item2

FROM
IMPORT X INTO
IMPORT Y INTO
CLASS

SORT Tup DEP Item1, Item2
FUNC tup: Item1 # Item2 -> Tup

AXIOM FORALL i1: Item1, j1: Item1, i2: Item2, j2: Item2
{ TUP1 } tup(i1, i2) ::
{ TUP2 } tup(i1, i2) = tup(j1, j2) => i1 = j1 AND i2 = j2 

PRED is_gen: Tup
IND FORALL i1: Item1, i2: Item2
{ is_gen(tup(i1, i2)) } 

AXIOM FORALL t: Tup
{ TUP3 } is_gen(t)

FUNC proj1: Tup -> Item1
IND FORALL i1: Item1, i2: Item2
{ proj1(tup(i1, i2)) = i1 } 

FUNC proj2: Tup -> Item2
IND FORALL i1: Item1, i2: Item2
{ proj2(tup(i1, i2)) = i2 } 

END;
This is an axiomatic specification of the 3-tuple data type with inductive definitions for the non-constructor operations.

LET TUP3_SPEC :=

LAMBDA X : ITEM1 OF
LAMBDA Y : ITEM2 OF
LAMBDA Z : ITEM3 OF
EXPORT

SORT Tup,
SORT Item1,
SORT Item2,
SORT Item3,
FUNC tup : Item1 # Item2 # Item3 -> Tup,
FUNC proj1: Tup -> Item1,
FUNC proj2: Tup -> Item2,
FUNC proj3: Tup -> Item3

FROM
IMPORT X INTO
IMPORT Y INTO
IMPORT Z INTO
CLASS

SORT Tup DEP Item1, Item2, Item3
FUNC tup: Item1 # Item2 # Item3 -> Tup

AXIOM
FORALL i1:Item1, j1:Item1, i2:Item2, j2:Item2, i3:Item3, j3:Item3 ( {TUP1} tup(i1,i2,i3); {TUP2} tup(i1,i2,i3) = tup(j1,j2,j3) => i1 = j1 AND i2 = j2 AND i3 = j3 )

PRED is_gen: Tup
IND FORALL i1:Item1, i2:Item2, i3:Item3 ( is_gen(tup(i1,i2,i3)) )

AXIOM FORALL t:Tup
{TUP3} is_gen(t)
CHAPTE 4. SPECIFICATION OF A TEXT EDITOR

FUNC proj1: Tup -> Item1
IND FORALL i1:Item1,i2:Item2,i3:Item3
    ( proj1(tup(i1,i2,i3)) = i1 )

FUNC proj2: Tup -> Item2
IND FORALL i1:Item1,i2:Item2,i3:Item3
    ( proj2(tup(i1,i2,i3)) = i2 )

FUNC proj3: Tup -> Item3
IND FORALL i1:Item1,i2:Item2,i3:Item3
    ( proj3(tup(i1,i2,i3)) = i3 )

END;
B. STANDARD CLASS DESCRIPTIONS

% This is an axiomatic specification of the data type of finite sets
% with inductive definitions for the non-constructor operations.

LET SET_SPEC :=

LAMBDA X:ITEM OF EXPORT

SORT Item,
SORT Nat,
SORT Set,
PRED is_in : Item # Set,
FUNC empty : -> Set,
FUNC ins : Item # Set -> Set,
FUNC rem : Item # Set -> Set,
FUNC union : Set # Set -> Set,
FUNC intersect : Set # Set -> Set,
FUNC diff : Set # Set -> Set,
PRED subset : Set # Set,
FUNC card : Set -> Nat

FROM
IMPORT X INTO
IMPORT NAT_SPEC INTO
CLASS

SORT Set DEP Item
PRED is_in : Item # Set,
FUNC empty : -> Set,
FUNC ins : Item # Set -> Set

AXIOM
{SET1} empty!;
{SET2} FORALL i:Item,s:Set ( ins(i,s) )

AXIOM FORALL i:Item,j:Item,s:Set
{SET3} NOT is_in(i,empty);
{SET4} is_in(i,ins(j,s)) <-> i = j OR is_in(i,s);
{SET5} ins(i,ins(j,s)) = ins(j,ins(i,s));
{SET6} ins(i,ins(i,s)) = ins(i,s)

PRED is_gen : Set
IND FORALL i:Item,s:Set
(is_gen(empty);
   is_gen(s) => is_gen(ins(i,s)) )
AXIOM FORALL s:Set
(SET7) is_gen(s)

FUNC ren: Item # Set -> Set
IND FORALL i:Item,j:Item,s:Set
( ren(i,empty) = empty:Set;
  ren(i,ins(i,s)) = ren(i,s);
  NOT i = j => ren(i,ins(j,s)) = ins(j,ren(i,s)) )

FUNC union: Set # Set -> Set
IND FORALL i:Item,s:Set,t:Set
( union(s,empty) = s;
  union(s,ins(i,t)) = ins(i,union(s,t)) )

FUNC isect: Set # Set -> Set
IND FORALL i:Item,s:Set,t:Set
( isect(s,empty) = empty;
  isect(ins(i,s),ins(i,t)) = ins(i,isect(s,t));
  NOT is_in(i,s) => isect(s,ins(i,t)) = isect(s,t) )

FUNC diff: Set # Set -> Set
IND FORALL i:Item,s:Set,t:Set
( diff(s,empty) = s;
  diff(s,ins(i,t)) = ren(i,diff(s,t)) )

PRED subset: Set # Set
IND FORALL i:Item,s:Set,t:Set
( subset(s,i);
  subset(t) => subset(s,ins(i,t)) )

FUNC card: Set -> Nat
IND FORALL i:Item,s:Set
( card(empty) = zero;
  NOT is_in(i,s) => card(ins(i,s)) = succ(card(s)) )

END;
% This is an axiomatic specification of the data type of finite
% bags, with inductive definitions for the non-constructor
% operations.

LET BAG_SPEC :=

LAMBDAX:ITEM OF

EXPORT

SORT Item,
SORT Nat,
SORT Set,
SORT Bag,
PRED is_in : Item # Bag,
FUNC empty : -> Bag,
FUNC ins : Item # Bag -> Bag,
FUNC rem : Item # Bag -> Bag,
FUNC union : Bag # Bag -> Bag,
FUNC intersect : Bag # Bag -> Bag,
FUNC diff : Bag # Bag -> Bag,
PRED subbag : Bag # Bag,
FUNC mult : Item # Bag -> Nat,
FUNC set : Bag -> Set

FROM
IMPORT X INTO
IMPORT NAT_SPEC INTO
IMPORT APPLY SET_SPEC TO X INTO
CLASS

SORT Bag DEP Item
PRED is_in: Item # Bag
FUNC empty: -> Bag
FUNC ins : Item # Bag -> Bag

AXIOM
{BAG1} empty:Bag!;
{BAG2} FORALL i:Item,b:Bag ( ins(i,b) )

AXIOM FORALL i:Item,j:Item,b:Bag,c:Bag ( {BAG3} NOT in_in(i,empty:Bag); {BAG4} is_in(i,ins(j,b)) <-> i = j OR is_in(i,b); {BAG5} ins(i,ins(j,b)) = ins(j,ins(i,b)); {BAG6} ins(i,b) = ins(i,c) => b = c )
PRED is_gen: Bag
IND FORALL i:Item,b:Bag
  ( is_gen(empty); 
  is_gen(b) IMPLIES is_gen(ins(i,b)) )

AXIOM FORALL b:Bag
  {BagT} is_gen(b)

FUNC rem: Item # Bag -> Bag
IND FORALL i:Item,j:Item,b:Bag
  ( rem(i,empty) = empty : Bag; 
  rem(i,ins(i,b)) = b; 
  NOT i = j => rem(i,ins(j,b)) = ins(j,rem(i,b)) )

FUNC union: Bag # Bag -> Bag
IND FORALL i:Item,b:Bag,c:Bag
  ( union(b,empty) = b; 
  union(b,ins(i,c)) = ins(i,union(b,c)) )

FUNC intersect: Bag # Bag -> Bag
IND FORALL i:Item,b:Bag,c:Bag
  ( intersect(b,empty) = empty; 
  intersect(ins(i,b),ins(i,c)) = ins(i,intersect(b,c)); 
  NOT is_in(i,b) IMPLIES intersect(b,ins(i,c)) = intersect(b,c) )

FUNC diff: Bag # Bag -> Bag
IND FORALL i:Item,b:Bag,c:Bag
  ( diff(b,empty) = b; 
  diff(b,ins(i,c)) = rem(i,diff(b,c)) )

PRED subbag: Bag # Bag
IND FORALL i:Item,b:Bag,c:Bag
  ( subbag(b,b); 
  subbag(b,c) IMPLIES subbag(b,ins(i,c)) )

FUNC mult: Item # Bag -> Nat
IND FORALL i:Item,j:Item,b:Bag
  ( mult(i,empty) = zero; 
  mult(i,ins(i,b)) = succ(mult(i,b)); 
  NOT i = j IMPLIES mult(i,ins(j,b)) = mult(i,b) )
FUNC set: Bag -> Set
IND FORALL i:Item,b:Bag
  ( set(empty) = empty;
    set(ins(i,b)) = ins(i,set(b)) )
END:
% This is an axiomatic specification of the data type of finite
% sequences, with inductive definitions for the non-constructor
% operations.

LET SEQ_SPEC :=

LAMBDA X : ITEM OF
EXPORT

SORT Item,
SORT Nat,
SORT Bag,
SORT Seq,
FUNC empty: Seq -> Seq,
FUNC cons : Item # Seq -> Seq,
FUNC hd : Seq -> Item,
FUNC tl : Seq -> Seq,
FUNC len : Seq -> Nat,
FUNC sel : Seq # Nat -> Item,
FUNC cat : Seq # Seq -> Seq,
FUNC rev : Seq -> Seq,
FUNC bag : Seq -> Bag

FROM
IMPORT X INTO
IMPORT NAT_SPEC INTO
IMPORT APPLY BAG_SPEC TO X INTO
CLASS

SORT Seq DEP Item
FUNC empty: Seq -> Seq
FUNC cons : Item # Seq -> Seq

AXION
{SEQ1} empty:Seq!
{SEQ2} FORALL i:Item,s:Seq ( cons(i,s) )

AXION FORALL i:Item,j:Item,s:Seq,t:Seq
{SEQ3} NOT cons(i,s) = empty;
{SEQ4} cons(i,s) - cons(j,t) => i = j AND s - t

PRED is_gen: Seq
IND FORALL i:Item,s:Seq
(is_gen(empty).
   is_gen(s) => is_gen(cons(i,s)) )
AXIOM FORALL s:Seq
  \{\text{is}_\text{gen}(s)\}

FUNC hd: Seq -> Item
IND FORALL i:Item,s:Seq
  ( hd(cons(i,s)) = i )

FUNC tl: Seq -> Seq
IND FORALL i:Item,s:Seq
  ( tl(cons(i,s)) = s )

FUNC len: Seq -> Nat
IND FORALL i:Item,s:Seq
  ( len(empty) = zero;
    len(cons(i,s)) = succ(len(s)) )

FUNC sel: Seq # Nat -> Item
IND FORALL i:Item,j:Item,a:Seq,n:Nat
  ( sel(cons(i,a),zero) = i;
    sel(cons(i,a),succ(n)) = j )

FUNC cat: Seq # Seq -> Seq
IND FORALL i:Item,s:Seq,t:Seq
  ( cat(empty,s) = s;
    cat(cons(i,s),t) = cons(i,cat(s,t)) )

FUNC rev: Seq -> Seq
IND FORALL i:Item,s:Seq
  ( rev(empty) = empty;
    rev(cons(i,s)) = cat(rev(s),cons(i,empty)) )

FUNC bag: Seq -> Bag
IND FORALL i:Item,s:Seq
  ( bag(empty) = empty;
    bag(cons(i,s)) = ins(i,bag(s)) )

END:
% This is an axiomatic specification of the data type of finite
% maps, with inductive definitions for the non-constructor
% operations.

LET MAP_SPEC :=

LAMBDA X : ITEM1 OF
LAMBDA Y : ITEM2 OF
EXPORT

SORT Item1.
SORT Item2.
SORT Set1.
SORT Set2.
SORT Map.
FUNC empty:   -> Map.
FUNC add : Map # Item1 # Item2 -> Map.
FUNC rem : Map # Item1       -> Map.
FUNC app : Map # Item1       -> Item2.
FUNC dom : Map               -> Set1.
FUNC ran : Map               -> Set2.

FROM
IMPORT X INTO
IMPORT Y INTO
IMPORT APPLY RENAME SORT Set TO Set1, SORT Item TO Item1
    IN SET_SPEC
            TO X

IMPORT APPLY RENAME SORT Set TO Set2, SORT Item TO Item2
    IN SET_SPEC
            TO Y

INTO

CLASS

SORT Map   DEP Item1,Item2
FUNC empty:   -> Map
FUNC add : Map # Item1 # Item2 -> Map
FUNC app : Map # Item1       -> Item2

AXIOM
{MAP1} empty:Map!;
{MAP2} FORALL m:Map,i:Item1,v:Item2
    ( add(m,i,v)! )
B. STANDARD CLASS DESCRIPTIONS

AXIOM FORALL i:Item1,j:Item1,v:Item2,w:Item2,m:Map
{MAP3} NOT app(empty,i)!
{MAP4} app(add(m,i,v),j) = w <=>
   (i = j AND v = w) OR (NOT i = j AND app(m,j) = w)
{MAP5} NOT i = j => add(add(m,i,v),j,w) = add(add(m,j,w),i,v)
{MAP6} add(add(m,i,v),i,w) = add(m,i,w)

PRED is_gen: Map
IND FORALL u:Map,i:Item1,v:Item2
   (is_gen(empty);
    is_gen(m) => is_gen(add(m,i,v)) )

AXIOM FORALL u:Map
{MAP7} is_gen(m)

FUNC rem: Map -> Item1 -> Map
IND FORALL m:Map,i:Item1,j:Item1,v:Item2
   (rem(empty,i) = empty;
    rem(add(m,i,v),i) = rem(m,i);
    NOT i = j => rem(add(m,i,v),j) = add(rem(m,j),i,v) )

FUNC dom: Map -> Set1
IND FORALL m:Map,i:Item1,v:Item2
   (dom(empty) = empty;
    dom(add(m,i,v)) = ins(i,dom(m)) )

FUNC ran: Map -> Set2
IND FORALL m:Map,i:Item1,v:Item2
   (ran(empty) = empty;
    ran(add(m,i,v)) = ins(v,ran(rem(m,i))) )

END;
Chapter 5

Systematic Design of a Text Editor

5.1 Introduction

This chapter describes the implementation of a display-oriented text editor. It serves for illustrating the notions of component, black-box description and design as described in Chapter 2. We shall follow one of the models of the software development process studied in Chapter 3, viz. the top-down model. We take the formal specification presented in Chapter 4 and we make it part of an initial design. This initial design serves as the starting point for a top-down development process to meet conditions of verification, validation and executability.

We shall use the language COLD-K [1] for our case study. The notions of component, black-box description and design as described in Chapter 2 are available in this language. One of the purposes of this chapter is to illustrate the use of design principles based on formal techniques. In particular, we want to show how the theory of Chapter 2 and Chapter 3 can be used as a tool for developing complex systems. We need a language as a vehicle and this will be COLD-K.

We begin with constructing a top design and a bottom design first and not before Section 5.5.1 our initial design will be constructed. After that the top-down development process starts. We describe this process as a sequence of modification steps, affecting one variable design. Since in the top-down development process most modifications amount to the addition of formal text to this variable design, we only give the newly added formal texts after each step. This chapter is organised such way that it is possible to concatenate the keyword DESIGN followed by the abbreviation-type components from Chapter
4 and of Sections 5.3.2 to 5.3.5 and all formal texts contained in Sections 5.5 and 5.6, thereby obtaining a final design which corresponds with the result of the development process and which can be syntax- and type-checked.

The presentation of this process corresponds more or less with the actual development process of the case study. Of course it is also a rational reconstruction in the sense that sometimes we had to do a little backtracking which is not reflected in this chapter. Also some thinking-in-advance and ‘throw-away’ preliminary implementation activities took place. These activities are typical for a top-down development process, but it is not always possible to describe them precisely. This means that sometimes we state “in order to implement ..., we postulate another component ...” although the motivation for certain details is incomplete. We did our best to avoid a ‘deus ex machina’ effect.

We adopt the methodological principles of conservativity, origin consistency and visibility consistency as discussed in [2]. The basic idea of conservativity is that when constructing a class description of the form \texttt{IMPORT P INTO Q}, the axioms of \texttt{Q} should not impose new restrictions upon the sorts, functions, predicates and procedures introduced in \texttt{P}. The principle of origin consistency serves for avoiding the situation where there are two or more defining occurrences of one name. The principle of visibility consistency amounts to a restriction for class descriptions of the form \texttt{IMPORT P INTO Q} when \texttt{p} is a procedure \texttt{p \in \Sigma(P)}. The restriction is that if some side-effect of \texttt{p} is exported by \texttt{Q}, then so it must be by \texttt{P}.

We adopt the implementation relation based on signature inclusion and theory inclusion, as argued in [2]. We shall implicitly assume the monotonicity of the COLD-K import, export and renaming operators – in fact without formal justification. We also adopt the condition \texttt{prime first} and the condition \texttt{directly specified} as discussed in Chapter 3. Finally we adopt the principles of \texttt{black-box correctness} (Chapter 2) and \texttt{black-box validation} (Chapter 3) – based on the exclusive use of specifications. COLD-K does not provide a program-execution model and therefore we have no formal criteria for deciding if a certain COLD-K text is executable or not. We aim at a manual translation from COLD-K to a classical imperative programming language (C) and with this idea in mind we shall sometimes (in informal speaking) distinguish between executable operations and non-executable operations: operations which need not be translated or which can not be translated in a straightforward manner are said to be non-executable.

This chapter is organised as follows. In Section 5.2 we discuss the top of the editor design. This is easy in view of the preparatory work of Chapter 4. In Section 5.3 we discuss the bottom of the editor design. This amounts
to the introduction of several new specifications. These describe instances, attributes, blocks and tables which will be assumed as available primitives. Section 5.4 contains a summary of the top-down approach. In Section 5.5 we begin a top-down development process. We implement the three components which constitute the system of the editor design; these are KEYBIND, MOREDOP and WITEFA. During the implementation of these system components, several new components are postulated. In Section 5.6 the top-down development process is continued; the newly postulated components are implemented which leads again to the introduction of new components etc. This goes on until at the end of Section 5.6 only the primitives of Section 5.3 are left.

In Section 5.7 we discuss some related work. Sections 5.8 and 5.9 are devoted to conclusions and evaluation. In Appendix A we introduce a lower design layer. In Appendix B we give a list of symbols used in this chapter. In Appendix C we give the C program resulting from the composition of the editor design and the design of Appendix A. Finally, in Appendix D we provide a 'reference chart'.

5.2 The Top of the Editor Design

In the specification presented in Chapter 4 we did not mention the structure of the design in which the various class descriptions of the formal specification fit. Now it is time to do so and, more precisely, we must cast these class descriptions into a design $d_4$. Recall from Chapter 3 that the top of a design $d$ is the design in which only those components are retained whose names occur in the system of $d$; it is denoted as $\text{top}(d)$. We must indicate a design $d_4$ which is considered as the top of a design $d_{\text{editor}}$, which at its turn must be constructed during the subsequent development process. The design $d_4$ is shown below. We assume that at the position of the dots in this design we have LET-constructs introducing the names BOOL_SPEC, NAT_SPEC, CHAR_SPEC ..., WITEFA_SPEC, MOREDOP_SPEC and KEYBIND_SPEC.

```
DESIGN
...

COMP WITEFA : WITEFA_SPEC;
COMP MOREDOP : MOREDOP_SPEC;
COMP KEYBIND : KEYBIND_SPEC

SYSTEM WITEFA, MOREDOP, KEYBIND
```
The above top design represents a certain view upon the editor design to be constructed in this chapter. More precisely, it is the view that is of interest to the user of the editor design. This top design is a kind of contract that specifies precisely the components of the design that are made available. In this case there are three available components, viz. WITEFA, MOREDOP and KEYBIND. These are specified by WITEFA_SPEC, MOREDOP_SPEC and KEYBIND_SPEC respectively. When the design will be finished, it may have many more components, but these are outside the scope of the user.

Let us have a look at each of the three components mentioned in the system and let us explain why each is part of the system. We begin with KEYBIND_SPEC, which is the most obvious component. It provides procedures PROC key: Char -> , PROC startup: -> and the functions FUNC screen: -> Text and FUNC cursor: -> Nat # Nat which are precisely enough when using the editor directly for editing texts.

A less obvious way to use the editor design is to employ it as just one layer of a larger composite design where new features have been added to the editor. A simple example of this would be to have another keybinding e.g. by introducing a procedure PROC key_2: Char -> , say. In order to write this key_2 one needs both the operations provided by the component WITEFA (e.g. backward_character) and those provided by MOREDOP (e.g. delete_previous_character). One could also go one step further and add a layer providing for dynamic keybinding and programmability e.g. as available with MLisp in Emacs [4]. In the latter case one may decide to ignore the KEYBIND component entirely. So there are in fact two categories of users. The first category consists of the users who simply connect key to their physical keyboard and then start typing. The users of the second category are real software developers constructing another 'higher level' design d_inter, say which is put on top of the editor design d_editor – which could be done with the operator c from Chapter 3. This explains why we included WITEFA and MOREDOP in addition to KEYBIND as part of the system.

This provides us with the following condition which is part of the postcondition of the software development process:

$$\text{top}(d_{editor}) = d_t$$

where the equality on designs is to be considered modulo the relative order of components.
5.3 The Bottom of the Editor Design

5.3.1 Introduction

Before we undertake a software development process, we must first introduce our primitive components. In the Sections 5.3.2 and 5.3.3 we introduce instances and attributes, which are the basic ingredients of an attribute-oriented approach, by which we mean that it is easy to add attributes to objects. This approach has been worked out by Jonkers in [3]. In the Sections 5.3.4 and 5.3.5 we introduce the data types of tables and blocks. We cast the specifications of these primitive components into the form of a bottom design. This will be done in Section 5.3.6

5.3.2 Specifying Instances

LET INST_SPEC :=
EXPORT
  SORT Inst.
  FUNC nil : -> Inst,
  PROC create: -> Inst
FROM
CLASS

SORT Inst VAR
FUNC nil: -> Inst

AXIOM
{INST1} INIT => nil! AND FORALL a:Inst ( a = nil )

PROC create: -> Inst MOD Inst

AXIOM
{INST2} < create > TRUE;
{INST3} [ LET a:Inst; a := create ]
  a! AND (PREV NOT a!) AND
  FORALL b:Inst ( (PREV NOT b!) => b = a )

END;
5.3.3 Specifying Attributes

\[
\text{LET ATTR_SPEC :=}
\]
\[
\text{LAMBDAX : CLASS SORT Inst FREE END OF}
\]
\[
\text{LAMBDAY : CLASS SORT Item FREE END OF}
\]

\[
\text{EXPORT}
\]
\[
\text{SORT Inst,}
\]
\[
\text{SORT Item,}
\]
\[
\text{FUNC attr : Inst \rightarrow Item,}
\]
\[
\text{PROC set_attr : Inst \# Item \rightarrow}
\]

\[
\text{FROM}
\]
\[
\text{IMPORT X INTO}
\]
\[
\text{IMPORT Y INTO}
\]
\[
\text{CLASS}
\]
\[
\text{FUNC attr : Inst \rightarrow Item VAR}
\]
\[
\text{PROC set_attr : Inst \# Item \rightarrow WOD attr}
\]

\[
\text{AXIOM FORALL } i: \text{Inst}, v: \text{Item} (\text{ATTR1} \land \text{ATTR2}) \rightarrow \text{TRUE;}
\]
\[
\text{ATTR1} = \text{ATTR2} \land
\]
\[
\text{FORALL } j: \text{Inst}, w: \text{Item} (\text{NOT } j = i \rightarrow
\]
\[
\text{attr}(j) = w \leftrightarrow \text{PREV attr}(j) = w)
\]

\[
\text{END;}
\]

5.3.4 Specifying Tables

In the description of the editor (Chapter 4) there is a variable map from strings to marked texts. Somehow this is going to be reflected in the implementation of the editor and therefore we introduce a specification of tables. These tables can be used for efficiently dealing with a mapping from implementable strings (sort 'String') to the representation of marked texts. Recall that we have already the class description MAP_SPEC which provides the sort Map of finite mappings with functions empty: \rightarrow Map for the empty mapping and add: Map \# Item1 \# Item2 \rightarrow Map for 'overwriting' addition. Furthermore there are operations rem: Map \# Item1 \rightarrow Map for removal, app: Map \# Item1 \rightarrow Item2 for map application, dom: Map \rightarrow Set1 for domain and ran: Map \rightarrow Set2 for range. These mappings are
described as an algebraic data type — just as Nat, say. All objects of sort Map exist already in the initial state and none of the operations may have side-effects. This means that formally it is not allowed to use certain conventional data reification techniques when implementing these mappings.

Therefore we introduce another sort of so-called tables which will be specified with implementability in mind. We must warn the reader here that this will give rise to several pages of technicalities. In the description TABLE_SPEC given below, a table will be modeled as a modifiable mapping by which we mean that associated with every table there is a variable map — in the sense of the sort Map. In this way the maps from MAP_SPEC will play a role in the formal specification, serving as auxiliaries for the description of something different, viz. tables. The tables themselves correspond with a variable sort Table. Initially precisely one table exists and optionally more tables can be created dynamically when needed. We employ the attribute-oriented approach mentioned in Section 5.3.1. Tables are introduced as a kind of instances and the variable maps associated with them are introduced as a kind of attributes.

The specification TABLE_SPEC is parameterised with respect to the domain sort and the range sort of the tables. The simplest way of describing this would be to adopt a definition beginning with something like LET TABLE_SPEC := LAMBDA X. ITEM1 OF LAMBDA Y. ITEM2 OF ... etc. We adopt ITEM2 indeed, but we do not adopt ITEM1 because of two complications which we want to take into account.

The first complication is necessary when conventional data reification techniques are to be allowed. Typical data reification techniques for tables are linked lists, sorted lists, open hashing, closed hashing, binary search trees and balanced trees. Most of these techniques can not be applied when the sort Item1 comes without any relations or operations — the only exception being the linked list technique. In order to sort a list we need a binary relation, less say, on Item1. The same holds for binary search trees and balanced trees. Both open hashing and closed hashing require that the Item1 values can be converted to natural numbers or integers. Of course we need not choose among these techniques here, but we must make suitable preparations not to exclude almost all options. Therefore we require that the sort Item1 comes with a binary predicate less. For this kind of applications, we have SL0 in our library of standard class descriptions — abbreviating Strict Linear Order.

There is a second complication which leads us to not adopting SL0, but a slightly different version of it called \textquoteleft SL0 \textquoteleft. This is needed because we want to allow for using a sort with an equivalence predicate eq such as \textquoteleft String\textquoteleft.
for the domain-sort of the tables. E.g. consider two 'String' objects \( s_1 \) and \( s_2 \) with \( s_1 \neq s_2 \) but \( \text{eq}(s_1, s_2) \); then a table-lookup using \( s_1 \) should yield the same result as a table-lookup using \( s_2 \). Furthermore we allow for a representation invariant \( \text{item1}_\text{inv} \) associated with the domain sort. The specification of 'SLO' takes SLO as a starting point and the formal relation between 'Item1' and Item1 is given by an abstraction function \( f \). When comparing 'SLO' with SLO we see that essentially, the former allows for a wider class of implementations of the concept 'strict linear order'; but strictly formally speaking, it is the other way around. The point is that the role of the unquoted names is not the same in 'SLO' and SLO.

The formal specification below begins with 'SLO' which imports a renamed version of SLO. Recall from our standard library that SLO provides a sort Item with a binary relation \( r \) which is axiomatically stated to be a strict linear ordering – the kind of relation often denoted by \( < \). Therefore \text{RENAME} SORT Item TO Item1 IN SLO provides a sort Item1 with a binary relation \( r: \) \text{Item1} \# \text{Item1} which is again a strict linear ordering. The \text{CLASS} ... END part of 'SLO' is an extension of SLO introducing the sort 'Item1' with binary predicates \( \text{eq} \) and \( \text{less} \). Furthermore it introduces a predicate \text{item1}_\text{inv} \) and an abstraction function \( f \). There is just one axiom which serves for stating the precise nature of the \( \text{eq} \) and \( \text{less} \) predicates: the function \( f \) is required to behave homomorphically such that \( = \) and \( r \) are the images of \( \text{eq} \) and \( \text{less} \) respectively. The axiom is relativised by \text{item1}_\text{inv}. Note that 'SLO' exports everything it contains, including Item1 and \( r \). As explained above 'SLO' serves as parameter restriction. Similarly Item2 which only requires the presence of a sort Item2 serves as a parameter restriction for a second formal parameter.

\begin{verbatim}
LET 'SLO1' :=
IMPORT
RENAME
    SORT Item TO Item1
    IN SLO
    INTO
    CLASS

    SORT 'Item1'          FREE
    PRED eq               : 'Item1' \# 'Item1' FREE
    PRED less              : 'Item1' \# 'Item1' FREE
    PRED item1_inv:       FREE
    FUNC f                : 'Item1' -> Item1 FREE
\end{verbatim}
AXIOM item1_inv =>
  FORALL i: 'Item1', j: 'Item1'
  ( eq(i, j) => f(i) = f(j);
   less(i, j) => r(f(i), f(j)) )
END;

LET TABLE_SPEC :=

LAMBEADA X: 'SLD1' OF
LAMBEADA Y: 'ITEM' OF

This completes the introduction of the formal parameters of TABLE_SPEC and
now we turn our attention to its body. There will be local definitions intro-
ducing TABLE_INST_SPEC, MAP_FROM_SLD1_TO_ITEM2 and TABLE_MAP_SPEC
which serve for introducing tables as renamed instances and for associating
variable maps with these tables. Within the CLASS ... END part of the
specification we shall define the table operations among which the proce-
dures new: -> Table and add: Table # 'Item1' # Item2 -> . We must
be prepared to allow techniques which require a representation invariant.
E.g. when a sorted list is used to represent tables, the invariant might state
that the list is cyclic and sorted; when a binary tree is used, the invariant
might state that all nodes in a left-hand side subtree have Item1 values which
are less than the Item1 values in the corresponding right-hand side subtree.
The actual choice is for the implementer and here we just introduce a pre-
dicate table_inv for which we state axiomatically that the table procedures
preserve it as an invariant. For the definition of the operations such as add
this also implies that we have to introduce a case-analysis. The purpose of
the case-analysis is to describe what can happen when the invariant does not
hold.

TABLE_SPEC has an explicit export list serving as a compact overview of
the sorts and operations provided. The sorts Map, Item1, the function map:
Table -> Map and the function app: Map # Item1 -> Item2 are needed for
reasoning purposes only. The remaining exported sorts and operations are
considered executable. All table operations are defined as PRDC allowing for
a maximum of implementation freedom – although maybe this is not needed
for certain of them, such as is_in_dom.
EXPORT

SORT Map,
SORT Item1,
FUNK map: Table -> Map,
FUNK app: Map # Item1 -> Item2,
SORT Bool,
SORT Table,
SORT Item2,
SORT 'Item1',
PRED table_inv: ,
PROC new : Table # 'Item1' # Item2 -> Table,
PROC add : Table # 'Item1' # Item2 -> ,
PROC rem : Table # 'Item1' -> ,
PROC app : Table # 'Item1' -> Item2,
PROC is_in_domain: 'Item1' # Table -> Bool

FROM

Now the local definitions follow. The first one defines TABLE_INST_SPEC which provides the sort Table with FUNC nil: -> Table and PROC create: -> Table. The second definition is based on MAP_SPEC which is a parametrised class description. Recall from our standard library that it has two parameters with parameter restrictions ITEM1 and ITEM2 requiring the presence of sorts Item1 and Item2 respectively. It provides the algebraic data type of maps, i.e. all maps that can be constructed by the constructor operations empty and add. Before we can employ this MAP_SPEC it must be instantiated which we do in this case by taking X and Y as actual parameters. Since we are within the scope of the abstractions LAMBDA X: 'SLO1' OF LAMBDA Y:ITEM2 OF ... we easily verify that X exports a sort Item1 and that Y exports a sort Item2.

LET TABLE_INST_SPEC :=
  RENAME
  SORT Inst TO Table
  IN INST_SPEC:

LET MAP_FROM_SLO1_TO_ITEM2 :=
  APPLY APPLY
  MAP_SPEC
  TO X TO Y;

The third local definition serves for associating maps with tables. We employ
5.3. THE BOTTOM OF THE EDITOR DESIGN

ATTR_SPEC from Section 5.3.3 which is parameterised with respect to the sorts of the instances and attributes. In this case the role of instances is played by Table whereas the role of attributes is played by Map. Furthermore we must get rid of the instance-and-attribute-oriented names and replace them by table-and-map-oriented names. As usual the renaming is done in the parameterised description and as a result we get a parameterised description whose parameter restrictions are renamed versions of CLASS SORT Inst FREE END and ITEM respectively. When suitably renamed, these require the presence of a sort Table and a sort Map respectively. This shows that we can take TABLE_INST_SPEC and MAP_FROM_SLO1_TO_ITEM2 as actual parameters. In this way we get an 'attribute' function map: Table -> Map and an 'assignment' procedure set_map: Table # Map ->.

LET TABLE_MAP_SPEC :=
  APPLY APPLY
  RENAME
  SORT Inst TO Table,
  SORT Item TO Map,
  FUNC attr : Inst -> Item TO map,
  PROC set_attr: Inst # Item -> TO set_map
  IN ATTR_SPEC
  TO TABLE_INST_SPEC TO MAP_FROM_SLO1_TO_ITEM2;

We import the formal parameters X and Y and the locally defined class descriptions. The auxiliary operation p corresponds with an arbitrary invocation of one of the table operations.

IMPORT X INTO
IMPORT Y INTO
IMPORT BOOL_SPEC INTO
IMPORT MAP_FROM_SLO1_TO_ITEM2 INTO
IMPORT TABLE_INST_SPEC INTO
IMPORT TABLE_MAP_SPEC INTO
CLASS

FRED table_inv: VAR

AXIOM {INVARINCE}
INIT AND item_inv => table_inv:
item_inv AND table_inv => p table_inv

PROC p: ->
DEF {FLUSH new
  | add(SOME t:Table(), SOME i:'Item1', SOME j:Item2())


Now the actual definitions of the table operations can be given. The procedure \texttt{new} yields a fresh table, denoted as \texttt{t}, which is obtained by \texttt{t := create}. Furthermore this fresh table gets the empty map associated with it and this is established by the expression \texttt{set_map(t, empty)} where we use the procedure \texttt{set_map} from \texttt{TABLE_MAP_SPEC}.

\begin{verbatim}
PROC new: -> Table
DEF LET t:Table; t := create;
    set_map(t,empty);
    t

PROC add: Table # 'Item1' # Item2 ->
PAR t:Table,i:'Item1',j:Item2
DEF ( NOT table_inv ?;
    MOD table_inv USE set_map END
    | table_inv ?;
    set_map(t,add(map(t),i,j))
)

PROC rem: Table # 'Item1' ->
PAR t:Table,i:'Item1'
DEF ( NOT table_inv ?;
    MOD table_inv USE set_map END
    | table_inv ?;
    set_map(t,rem(map(t),f(i)))
)
\end{verbatim}
5.3. THE BOTTOM OF THE EDITOR DESIGN

PROC app: Table # 'Item1' -> Item2
PAR t:Table,i:'Item1'
DEF ( NOT table_inv ?;
   SOME i:Item2 ()
   | table_inv ?;
   app(map(t),f(i))
)

PROC is_in_dom: 'Item1' # Table -> Bool
PAR i:'Item1',t:Table
DEF ( NOT table_inv ?;
   SOME b:Bool ()
   | table_inv ?;
   ( app(map(t),f(i))! ?: true
   | NOT app(map(t),f(i))! ?: false
   )

END:

This concludes the specification of tables.

5.3.5 Specifying Blocks

In order to store texts within our editor, we need a facility that manages an unbounded amount of storage and that on request releases finite portions of storage in the form of a kind of blocks of storage locations. Below we give a specification of such a facility. To keep things simple, we did not include possibilities for returning portions of storage that are no longer needed. The function cont applied to a block b and a natural number n < size(b) yields the contents of the n-th location in this block. The procedure store can be applied to a block b, a natural number n < size(b) and an item i. Its effect is to store the value i in the n-th location in this block. The numbers 0...size(b) – 1 serve as addresses.

In the class description BLOCK_SPEC below we have four axioms describing the initial state and the procedures alloc, grow and store respectively. The first axiom states that initially there are no blocks yet. The second axiom describes both the termination of alloc(n) and its postcondition which defines the size of the newly 'allocated' block and which states that precisely one new block is created. The third axiom describes the termination of grow(b,n) and its postcondition. Formally this axiom must be relativised by the pre-
miss size(b) which of course holds in the states reachable from the initial state by using alloc and grow. The last axiom gives the termination and the postcondition of store(b.n.i). Again this axiom is relativised and the premiss is n < size(b).

LET BLOCK_SPEC :=
LAMBDA X:ITEM OF
EXPORT

SORT Block,
SORT Nat,
SORT Item,
FUNC size : Block -> Nat,
FUNC cont : Block # Nat -> Item,
PROC store : Block # Nat # Item ->,
PROC alloc : Nat -> Block,
PROC grow : Block # Nat ->

FROM
IMPORT X INTO
IMPORT NAT_SPEC INTO
CLASS

SORT Block VAR
FUNC size : Block -> Nat VAR
PROC alloc : Nat -> Block MOD Block
PROC grow : Block # Nat -> MOD size

AXIOM INIT => NOT EXISTS a:Block ()

AXIOM FORALL n:Nat ( < alloc(n) > TRUE;
   [ LET b:Block: b := alloc(n) ]
   size(b) = n AND
   (PREV NOT b!) AND FORALL c:Block
   ( (PREV NOT c!) => c = b ) )

AXIOM FORALL b:Block. n:Nat ( size(b)!) =>
   ( < grow(b,n) > TRUE;
   [ grow(b,n) ]
   ( size(b) = add(n.PREV size(b));
   FORALL c:Block ( NOT b = c AND (PREV size(c)!)) =>
   ( size(c) = PREV size(c) ) ) ) )

FUNC cont: Block # Nat -> Item VAR
5.3. THE BOTTOM OF THE EDITOR DESIGN

PROC store : Block # Nat # Item -> MOD cont

AXIOM FORALL b:Block, n:Nat, i:Item
( len(n, size(b)) => ( < store(b, n, i) > TRUE;
[ store(b, n, i) ]
( cont(b, n) = i;
FORALL c:Block. m:Nat. j:Item
( len(m, size(c)) =>
( NOT n = m OR NOT b = c =>
cont(c, m) = j <=> PREV cont(c, m) = j ) ) ) )

END;

5.3.6 The Bottom of the Editor Design

We expect the underlying machine on which we have to build our editor to provide us with implementations of the following COLD-K class descriptions.

- Data-types immediately available from a typical programming language such as Pascal or C. Let us assume that we have general purpose data types BOOL_SPEC, NAT_SPEC, CHAR_SPEC, INST_SPEC, ATTR_SPEC.
- Implementable sequences described by 'SEQ_SPEC', modifiable maps, described by TABLE_SPEC, and so-called blocks described by BLOCK_SPEC.
- A display device. Let us assume that this is given by the class description DISPLAY_SPEC. Also we assume a simple file system: FILE_SPEC.

The primitive components from the second group are considered not available in the programming language at hand. Therefore we can expect that at some point in time we might implement some of them ourselves. On the other hand, these components are of a general-purpose nature and they do not reflect that we are aiming at an editor. These give rise to the following primitive components:

'SEQ' implementing 'SEQ_SPEC',
TABLE implementing TABLE_SPEC,
BLOCK implementing BLOCK_SPEC.

Now it is time to cast the specifications of all primitive components into the shape of a design \( d_b \). Recall from Chapter 3 that the bottom of a design \( d \) is the design with the empty system and in which only those components are retained that have no glass-box description - i.e. implementation module. The bottom of a design \( d \) is denoted as \( \text{bot}(d) \). We must cast the specifi-
cations of the primitive components into a design $d_s$ which is considered as the bottom of a design $d_{\text{editor}}$, which at its turn must be constructed during the subsequent development process. The design $d_s$ is shown below. We assume that at the position of the dots in this design we have a number of LET abbreviations, which introduce the names BOOL_SPEC, NAT_SPEC, CHAR_SPEC etc.

**DESIGN**

...  
COMP BOOL : BOOL_SPEC;
COMP NAT : NAT_SPEC;
COMP CHAR : CHAR_SPEC;
COMP INST : INST_SPEC;
COMP ATTR : ATTR_SPEC;
COMP 'SEQ' : 'SEQ_SPEC';
COMP TABLE : TABLE_SPEC;
COMP BLOCK : BLOCK_SPEC;
COMP DISPLAY : DISPLAY_SPEC;
COMP FILE : FILE_SPEC

**SYSTEM NONE**

This provides us with the following condition which is part of the postcondition of the software development process:

$$\text{bot}(d_{\text{editor}}) = d_s$$

In Appendix A we shall consider a design $d_{\text{basic}}$ whose system can be plugged into $d_s$ and hence also in $d_{\text{editor}}$. Of course the contents of this appendix are supposed to be outside the scope of our design $d_{\text{editor}}$. In particular, the principle of black-box validation does not allow the use of knowledge of implementation details from $d_{\text{basic}}$.

### 5.4 Summary of the Top-down Approach

At this point we have not started the actual construction of our software system yet and instead of that we have fixed the boundaries of the design $d_{\text{editor}}$ to be created. This was done in Sections 5.2 and 5.3 where we have chosen the user-view $d_t$ and the machine-view $d_s$ respectively. Let us assume that we can find a system-user who would like to use the system components specified by $d_t$. Let us also assume that somehow we can establish the availability of
5.4. SUMMARY OF THE TOP-DOWN APPROACH

a machine providing the primitive components specified by $d_i$. Under these assumptions the validation conditions mentioned in Chapter 3 are fulfilled.

We shall describe the principles of the top-down approach informally now. For the formal treatment we refer to Chapter 3. The top-down approach is based on the so-called top-down invariant which is defined as the conjunction of the following conditions.

1. $\text{top}(d_{editor}) = d_i$ where the equality on designs is to be considered modulo the relative order of components. Intuitively this means that the user-view on the editor design is kept fixed.

2. $d_{editor}$ is black-box correct. Roughly speaking, this corresponds with the situation where all black-box descriptions are true in the sense that the corresponding glass-box descriptions (if present) are implementations indeed. Formally this is described best by referring to the implementation relation $\subseteq$.

3. All components of $d_{editor}$ are directly or indirectly used in the system of $d_{editor}$ except possibly those occurring also in $d_i$, i.e. except possibly those which happen to be available anyway.

A suitable initial state for the development process can be established simply by assigning $d_i$ to $d_{editor}$. Let us check that this makes the top-down invariant hold indeed. First, the top of this design $d_i$ equals $d_i$, which is a property of top designs. Secondly the design $d_i$ is black-box correct since its components do not have a glass-box description yet and hence are correct by definition. Finally, all components occur in the system. In our particular case these components are WITEFA, MOREDOP, KEYBIND. This shows that assigning $d_i$ to $d_{editor}$ makes the top-down invariant hold.

After this assignment a top-down development process can really begin and this is the subject of Section 5.5. First we add the primitive components of $d_i$ to the editor design. Next, we must select some primitive component and make sure that it gets implemented according to the principle of black-box correctness. In our particular case we shall select KEYBIND as the first component to be implemented and since its specification is already in algorithmic style this is easy. Next we treat MOREDOP which is equally easy and after that there is only one component to be selected left which is WITEFA. To implement WITEFA is more complicated and it requires the introduction of several new primitive components. After WITEFA has been implemented we are done with the system components and this stage will be reached at the end of Section 5.5. Although at that stage all system components have been implemented, the design is still not finished yet because there are several newly postulated primitive components which were needed for WITEFA. In Section 5.6 these are implemented one after the other. Again this requires
the introduction of new primitive components and in this way the top-down
development process proceeds until only those primitive components are left
that occur in the bottom design $d_4$. This stage will be reached at the end of
Section 5.6.

At that stage we can finish the development process. The condition bot($d_{editor})
= d_4$ serves as the termination condition of the development process. Recall
that the clause bot($d_{editor}$) refers to the so-called bottom of the design $d_{editor}$
and that intuitively this bottom is a machine-view, containing information
about the primitive components. The top-down development process can
be viewed as the execution of a design program of the form given below,
where $td_{.step}$ satisfies the assumption that it does not violate the top-down
invariant.

\[
\text{technique } (d_0, d_1) \\
def \ d_{editor} := d_1; \\
\text{while not bot}(d_{editor}) = d_0 \text{ do } d_{editor} := td_{.step}(d_{editor}); \text{ od; } \\
\text{end } d_{editor}
\]

The precise syntax and semantics of such design programs is given in Chapter
3. Here we restrict ourselves to a short informal explanation. The header
\text{technique } (d_0, d_1) \text{ indicates that this is a design program where } d_0 \text{ and } d_1
\text{ act as parameters. The definition of the design program follows after the}
\text{keyword def and it consists of an assignment statement to the variable design}
\text{d_{editor} and a loop of iterated application of td_{.step}. The final d_{editor} serves as}
\text{the result of this design program.}

5.5 Implementing the System Components

5.5.1 Introduction

We begin with $d_4$, but we add immediately the primitive components from
$d_3$ to it. This yields the design consisting of the keyword \text{DESIGN} and the LET
abbreviations given before followed by the following components:

\text{COMP BOOL : BOOL_SPEC;}
\text{COMP NAT : NAT_SPEC;}
\text{COMP CHAR : CHAR_SPEC;}
\text{COMP INST : INST_SPEC;}
\text{COMP ATTR : ATTR_SPEC;}
\text{COMP 'SEQ' : 'SEQ_SPEC';}
and having as a system WITEFA, MOREDOP, KEYBIND. The black-box descriptions (the xxx_SPECS) of this design will remain unchanged during the development process and so does its system. However, some of these components will get a glass-box description later and as a reminder for this, they have been marked now by (g). The symbol (g) should be read as: "this is going to be replaced later". Since there are no glass-box descriptions yet, the above initial design is black-box correct.

5.5.2 Implementing KEYBIND

In this section we give the implementation of the component KEYBIND which contains a procedure key, taking characters as its argument and invoking operations provided by WITEFA and MOREDOP. Furthermore it exports the initialisation procedure startup - which we can assume to be provided by MOREDOP since MOREDOP_SPEC exports startup. It also exports screen and cursor which we can assume to be provided by WITEFA since WITEFA_SPEC imports APP_DOM_SPEC which at its turn imports DISPLAY_SPEC.

The $\text{KEYBIND\_IMPL}$ given below is almost the same as $\text{KEYBIND\_SPEC}$. The subtle difference is that we use component names in the import clauses rather than abbreviation names. Note that earlier $\text{KEYBIND\_SPEC}$ was used as a black-box description. Therefore it was necessary to import $\text{NAT\_SPEC}$, $\text{BOOL\_SPEC}$ etc. because importing $\text{NAT}$ or $\text{BOOL}$ would violate the principle of direct specification. Recall from Chapter 3 definition 3.2.13, that a design is directly specified if it is wellformed and no black-box description contains the name of a component. Now we are in a different position. We are going to construct a glass-box description $\text{KEYBIND\_IMPL}$ and now we are allowed to refer to component names. In order to get an executable implementation, we must in fact do so.

The procedure key is described by one large case-statement and we consider this as already executable. The procedure key calls the procedures from WITEFA and MOREDOP: insert_character, set_mark, ..., delete_
previous_character. Most procedures called by key come from WITEFA, but the following procedures come from MOREDOP: return, search_forward, insert_file, write_named_file, delete_to_killbuffer, yank_from_killbuffer, escape and delete_previous_character.

LET KEYBIND_IMPL :=
EXPORT
  SORT Nat,
  SORT Char,
  SORT Text,
  PROC startup: -> ,
  PROC key : Char -> ,
  FUNC screen : -> Text,
  FUNC cursor : -> Nat # Nat
FROM
IMPORT Nat INTO
IMPORT Char INTO
IMPORT WITEFA INTO
IMPORT MOREDOP INTO
CLASS

PROC key : Char ->
  FAR c:Char
  DEF ( printable(c) ; insert_character(c)
    | ord(c) = 0 {'0'} ?; set_mark
    | ord(c) = 1 {'A'} ?; beginning_of_line
    | ord(c) = 2 {'B'} ?; backward_character
    | ord(c) = 4 {'D'} ?; delete_next_character
    | ord(c) = 5 {'E'} ?; end_of_line
    | ord(c) = 6 {'F'} ?; forward_character
    | ord(c) = 13 {'N'} ?; return
    | ord(c) = 14 {'M'} ?; next_line
    | ord(c) = 16 {'P'} ?; previous_line
    | ord(c) = 19 {'S'} ?; search_forward
    | ord(c) = 20 {'T'} ?; insert_file
    | ord(c) = 21 {'U'} ?; write_named_file
    | ord(c) = 23 {'W'} ?; delete_to_killbuffer
    | ord(c) = 24 {'X'} ?; beginning_of_buffer
    | ord(c) = 25 {'Y'} ?; yank_from_killbuffer
    | ord(c) = 26 {'Z'} ?; end_of_buffer
    | ord(c) = 27 {'ESC'} ?; escape
    | ord(c) = 127(DEL) ?; delete_previous_character
  )

END;
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

Now we are in a position where the primitive component {4} COMP KEYBIND: KEYBIND_SPEC can be replaced by a component having both a black-box description and a glass-box description, viz. by COMP KEYBIND: KEYBIND_SPEC := KEYBIND_IMPL.

% COMP KEYBIND : KEYBIND_SPEC := KEYBIND_IMPL ;
% this is to replace an earlier primitive component.

We must add some explanation to this because we had to solve a technical complication regarding the presentation of the (dynamic) development process in this (static) chapter. When we were in the specification phase (Chapter 4) this was no problem because at each stage we only added formal texts after the texts obtained already. We organise our text in such a way that there is a simple mechanical operation of separating the formal texts from the informal introductions and annotations. Then the concatenation of the keyword DESIGN, the LET abbreviations from Chapter 4, Sections 5.3.2 to 5.3.5 and the remaining formal texts from this chapter (Section 5.5.1 to 5.6.13) will constitute one design which can be syntax- and type-checked. This one design encompasses all the formal texts employed in various stages of the top-down development process. The technical complication is that one component may occur initially just as primitive whereas in a later stage it is replaced by another component, having both a black-box description and a glass-box description. Of course we can not have a well-formed design where one component-name occurs twice and therefore we include such ‘second defining occurrences’ of components in the design, but we only include them as comment. Just as above, these situations will be marked by the sentence “% this is to replace an earlier primitive component.”

In fact, we have now carried out our first top-down design-transformation step d_code := td_step(d_code); as described in Section 5.4. The resulting design reads the same as the one given in Section 5.5.1 but for two modifications: first, the “LET KEYBIND_IMPL := … ;”-text is added at an appropriate place, and second, the line “COMP KEYBIND : KEYBIND_SPEC ;” is replaced by “COMP KEYBIND : KEYBIND_SPEC := KEYBIND_IMPL ;”.

5.5.3 Implementing MOREDOP

In this section we give the implementation of the component MOREDOP which deals with a few additional operations which are added on top of WITEFA. The description given below is the same as MOREDOP_SPEC, except for the fact that we use component names instead of abbreviation-names.

LET MOREDOP_IMPL :=
IMPORT NAT INTO
IMPORT BOOL INTO
IMPORT CHAR INTO
IMPORT WHITE INTO
CLASS

PROC mini: -> 'String'
DEF cons('m',cons('i',cons('n',cons('i',empty))))

PROC main: -> 'String'
DEF cons('m',cons('a',cons('i',cons('m',empty))))

PROC kill: -> 'String'
DEF cons('k',cons('i',cons('l',cons('l',empty))))

PROC startup: ->
DEF init(mini):
  switch_to_buffer(kill):
  switch_to_buffer(main)

PROC escape: ->
DEF ( eq(current_buffer_name,mini) ?: switch_to_buffer(main)
   | NOT eq(current_buffer_name,mini) ?: switch_to_buffer(mini)
   )

PROC return: ->
DEF ( eq(current_buffer_name,mini) ?: switch_to_buffer(main)
   | NOT eq(current_buffer_name,mini) ?: newline
   )

PROC delete_to_killbuffer: ->
DEF copy_region_to_buffer(kill):
  erase_region

PROC yank_from_killbuffer: ->
DEF yank_buffer(kill)

PROC search_forward: ->
DEF search_forward(buffer_to_string(mini))

PROC insert_file: ->
DEF insert_file(buffer_to_string(mini))

PROC write_named_file: ->
DEF write_named_file(buffer_to_string(mini))
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

PROC delete_previous_character: ->
    DEF ( bolp = true ?; SKIP
    | bolp = false ?;
        backward_character;
        delete_next_character
    )
END;

Again we indicate the completion of the second top-down design transformation step by simply writing the following two lines:

% COMP MOREDOP : MOREDOP_SPEC := MOREDOP_IMPL;
% this is to replace an earlier primitive component.

We implemented two components and we were lucky in the sense that we could do so in a rather trivial way. This was possible because the black-box descriptions of these components were already in the right form. The next component to be dealt with is WITEFA and in view of its complexity and the axiomatic style of its description, we can expect that the construction of its implementation will be less trivial. Indeed, almost of the entire rest of this chapter concerns the implementation of WITEFA and the implementation of components that we shall postulate and that are directly or indirectly used by WITEFA.

5.5.4 Transforming the State space

We must choose how to represent marked texts. This choice is crucial for the efficiency of the resulting editor, both in terms of execution time and memory usage. Making this choice is also an important point in the software development process, since many procedures to be developed later will be based on this choice.

Before presenting our choice, we present a few informal efficiency considerations. We assume that there is no a-priori bound on the size of the texts that can be edited: neither the number of lines nor the number of characters within one line should be bounded. Of course a real program execution environment imposes limits upon the amount of available memory, but this is not formally described by the specifications of our primitive components and we do not worry about that. Instead of that we want that no 'a priori' bounds get built-in into our design. So choosing a fixed-size two-dimensional array as a representation with the rows of the array representing lines would
be wrong in this respect. Better candidate representations are those based upon the use of sequences. For example, we might represent a text by one or more sequences of line-representations, where a line-representation at its turn is a sequence of characters. This certainly is a workable approach, but we do not choose it for the following reason. The problem is that the sequences themselves need some kind of representation, such as linked lists. The straightforward representation of sequences by linked list would use some form of cells, where each cell contains a 'next cell' address. This means that for each character stored in the editor, there is an overhead of one address. Even if we assume that a character and an address require the same amount of storage, we waste 50% of the available storage. Of course we could try to improve upon the representation of the sequences but then the approach might lose some of its elegance.

After these considerations we shall present our choice, but we would like to add immediately that there is no claim whatsoever that this would be the best choice. The amount of possible choices here is enormous and the question of which is the best one has many aspects. We just took one which seems reasonable.

We use for each text one block, which is a large extensible one-dimensional array, as already introduced in Section 5.3.5. Within this block, the text is stored in a way that is related to the string representation of text, viz. by using some separator between the lines. For efficiency reasons we shall not make the block grow in steps of one location at a time, but instead we shall use larger increments. From this it follows that often there is some free space in the block. We decide that this free space should be contiguous rather than scattered over the block and that it will take the shape of a kind of gap which may be at any position within the block. Of course we must also somehow keep track of the mark and the dot, which simply become natural numbers indicating a position in the block. A similar representation has been used also in the implementation of the EMACS editor described in [4].

In order to get elegant algorithms taking care for the window-invariant UI, we also want to keep track of the dot, viewed as a co-ordinate pair. Since we have operations such as exchange dot and mark and end of buffer, it is a consequence that we must also keep track of the mark viewed as a co-ordinate pair and of the reach of the entire text. Apart from storing a marked text as a block-with-positions, we explicitly store the reach of the text before dot, the reach of the text before mark and the reach of the entire text.

So we store the text, the dot and the mark as block-with-positions and we
redundantly store their homomorphic images under the reach operation as well. This means that many edit procedures have to do their work twice: first of all they operate on the block-with-positions data and then secondly they perform the same operation on reaches. For the operations on block-with-positions data we have to be careful with respect to efficiency, but fortunately the operations on reaches such as add, split, cut, and paste pose less efficiency problems. Actually we just take their definition, which can be considered executable already.

We start the formalisation of our choice now. As discussed before, we need some sort of objects that can represent MText objects. For this purpose we introduce a sort Buf, whose objects will be called buffers. Buf is nothing but a renamed version of Inst with several attributes. Each buffer has a block associated with it. Several Nat attributes are needed as 'pointers' to certain positions in the block.

* block: Buf → Block  VAR
* dot : Buf → Nat  VAR
* mark : Buf → Nat  VAR
* gap1 : Buf → Nat  VAR
* gap2 : Buf → Nat  VAR

Finally we need a number of pair-wise attributes. For example, we need an attribute dot: Buf → Nat<sup>2</sup> which keeps track of the dot viewed as a co-ordinate pair.

* dot : Buf → Nat<sup>2</sup>  VAR
* mark : Buf → Nat<sup>2</sup>  VAR
* reach: Buf → Nat<sup>2</sup>  VAR

Sometimes we use a terminology based on a spatial view of buffers. E.g. we shall speak about moving leftwards or going downwards when we mean to decrease the value of a variable that indicates a position.

### 5.5.5 Specifying Pair-wise Attributes

For dealing with co-ordinate pairs it is convenient to have pair-wise attributes.

```
LET ATTR2_SPEC :=
LAMBDA X : CLASS SORT Inst FREE END OF
LAMBDA Y : CLASS SORT Item1 FREE END OF
```
LAMBDA Z : CLASS SORT Item2 FREE END OF

EXPORT
    SORT Inst,
    SORT Item1,
    SORT Item2,
    FUNC attr : Inst -> Item1 # Item2,
    PRDC set_attr: Inst # Item1 # Item2 ->
FROM
IMPORT X INTO
IMPORT Y INTO
IMPORT Z INTO
CLASS

FUNC attr : Inst -> Item1 # Item2 VAR
PRDC set_attr: Inst # Item1 # Item2 -> MOD attr

AXIOM FORALL i:Inst,v:Item1,w:Item2 (  
    (ATTR1) < set_attr(i,v,w) > TRUE,
    (ATTR2) [ set_attr(i,v,w) ]

attr(i) = (v,w) AND
FORALL j:Inst,x:Item1, y:Item2 ( NOT j = i =>
( attr(j) = (x,y) <=> PREV attr(j) = (x,y) ) ) )

END;

5.5.6 ATTR2: a Postulated Component

Since we have already a specification ATTR2_SPEC, we can directly postulate the following component:

{0} COMP ATTR2 : ATTR2_SPEC;

We best employ the meta-operator COPY when using ATTR2. The effect of this operator is to make a textual copy of its argument. This will guarantee that no undesired aliasing takes place because each copy of ATTR2 can be viewed as a fresh programming variable. This is due to the so-called origin mechanism of COLD-K. For a discussion of the COPY operator we refer to [3] and for an explanation of the origin mechanism we refer to [1]. Here we only sketch how the use of COPY in combination with the origin mechanism works on a simplified example. Let us — by way of example — assume that ATTR0 is defined by LET ATTR0 := CLASS PRED attr: VAR END which
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

means that it describes a two-valued programming variable. Now to get two
instances of such variables we can write LET V := COPY(ATTR); LET W :=
COPY(ATTR); which is by definition of COPY nothing but LET V := CLASS
PRED attr: VAR END; LET W := CLASS PRED attr: VAR END:. Since the
latter text introducing V and W contains two defining occurrences of attr,
these have different origins; we can imagine a language implementation at-
taching these origins and then we get e.g. attr1 and attr2, respectively which
means that they act as two different programming variables. When syntax-
and type-checking we replace COPY(ATTR2) by ATTR2. Similar remarks apply
for ATTR.

5.5.7 Transforming the State space (continued)

Now we are ready to introduce a class description BUF describing the sort
Buf and the associated attributes. We give the formal definition below.

LET BUF_INST :=
RENAME
  SORT Inst TO Buf
IN INST;

LET BUF :=

% local definitions follow:

LET CHAR_BLOCK :=
  APPLY RENAME
  SORT Item TO Char
IN BLOCK TO CHAR;

LET BUF_BLOCK :=
  APPLY APPLY RENAME
    SORT Inst        TO Buf,
    SORT Item       TO Block,
    FUNC attr : Inst -> Item TO block,
    PROC set_attr: Inst # Item -> TO set_block
IN COPY(ATTR) TO BUF_INST TO CHAR_BLOCK;

LET BUF_DOT :=
  APPLY APPLY RENAME
    SORT Inst        TO Buf,
    SORT Item       TO Nat,
    FUNC attr : Inst -> Item TO dot,
PROC set_attr: Inst # Item -> TO set_dot
IN COPY(ATTR) TO BUF_INST TO NAT;

LET BUF_MARK :=
APPLY APPLY RENAME
  SORT Inst TO Buf,
  SORT Item TO Nat,
  FUNC attr : Inst -> Item TO mark,
  PROC set_attr: Inst # Item -> TO set_mark
IN COPY(ATTR) TO BUF_INST TO NAT;

LET BUF_GAP1 :=
APPLY APPLY RENAME
  SORT Inst TO Buf,
  SORT Item TO Nat,
  FUNC attr : Inst -> Item TO gap1,
  PROC set_attr: Inst # Item -> TO set_gap1
IN COPY(ATTR) TO BUF_INST TO NAT;

LET BUF_GAP2 :=
APPLY APPLY RENAME
  SORT Inst TO Buf,
  SORT Item TO Nat,
  FUNC attr : Inst -> Item TO gap2,
  PROC set_attr: Inst # Item -> TO set_gap2
IN COPY(ATTR) TO BUF_INST TO NAT;

% Now the pair-wise attributes follow:

LET BUF_DOT_2 :=
APPLY APPLY APPLY RENAME
  SORT Inst TO Buf,
  SORT Item1 TO Nat,
  SORT Item2 TO Nat,
  FUNC attr : Inst # Item1 # Item2 TO dot,
  PROC set_attr: Inst # Item1 # Item2 -> TO set_dot
IN COPY(ATTR2) TO BUF_INST TO NAT TO NAT;

LET BUF_MARK_2 :=
APPLY APPLY APPLY RENAME
  SORT Inst TO Buf,
  SORT Item1 TO Nat,
  SORT Item2 TO Nat,
  FUNC attr : Inst # Item1 # Item2 TO mark,
  PROC set_attr: Inst # Item1 # Item2 -> TO set_mark
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

\[
\text{IN COPY(ATTR2) TO BUF_INST TO NAT TO NAT.}
\]

\[
\text{LET BUF REACH 2 :=}
\]
\[
\text{APPLY APPLY APPLY RENAME}
\]
\[
\text{SORT Inst} \quad \text{TO Buf,}
\]
\[
\text{SORT Item1} \quad \text{TO Nat,}
\]
\[
\text{SORT Item2} \quad \text{TO Nat,}
\]
\[
\text{FUNC attr : Inst} \rightarrow \text{Item1 # Item2 TO reach,}
\]
\[
\text{PROC set_attr : Inst # Item1 # Item2 \rightarrow} \quad \text{TO set_reach}
\]
\[
\text{IN COPY(ATTR2) TO BUF_INST TO NAT TO NAT;}
\]

% end of local definitions

\[
\text{IMPORT NAT INTO}
\]
\[
\text{IMPORT CHAR_BLOCK INTO}
\]
\[
\text{IMPORT BUF_INST INTO}
\]
\[
\text{IMPORT BUF_BLOCK INTO}
\]
\[
\text{IMPORT BUF_DOT INTO}
\]
\[
\text{IMPORT BUF_MARK INTO}
\]
\[
\text{IMPORT BUF_GAP1 INTO}
\]
\[
\text{IMPORT BUF_GAP2 INTO}
\]
\[
\text{IMPORT BUF_DOT 2 INTO}
\]
\[
\text{IMPORT BUF_MARK 2 INTO}
\]
\[
\text{BUF REACH 2;}
\]

As already indicated, the basic idea behind this representation is as follows. An object Buf has a block associated with it, which contains all characters of the text represented by \( b \). Within this block there is a so-called gap. The gap of \( b \) corresponds with the positions gap1(\( b \)) (inclusive) to gap2(\( b \)) (exclusive). The gap contains the free space in the block and when the dot is near the gap, it is possible to do insertions easily and efficiently. The characters in the block correspond with the string representation, but the positions in the gap are not used. The following picture sketches a buffer with its gap.

![Buffer with gap](image)

**Fig 5.1.** Buffer with gap.

Recall the following variables spanning the state space of WITEFA, introduced
in Chapter 4.

- \textbf{FUNC} mtexts : \rightarrow \text{Map} \hspace{0.5cm} \text{VAR}
- \textbf{FUNC} current: \rightarrow \text{String} \hspace{0.5cm} \text{VAR}

These are going to be replaced by other 'new' variables. We want to introduce a table from strings (sort 'String') to buffers. Therefore we want to instantiate TABLE with an implementation of 'STRING_SPEC' and with BUF thereby getting the possibility to create tables. Actually we need only one table, but that does not matter. We introduce new variables, which serve for taking over the role of the original ones; these are the following:

- \textbf{FUNC} table : \rightarrow \text{Table} \hspace{0.5cm} \text{VAR}
- \textbf{FUNC} current: \rightarrow \text{'String'} \hspace{0.5cm} \text{VAR}

For purposes such as searching in a text or copying texts, we introduce a simple programming variable of sort \texttt{Nat}. We call it mover, since it is not a relatively static pointer, such as mark or dot, but it is supposed to move during the execution of a search operation, or when it is used as a loop-counter. We also need a pair-wise version of mover.

- mover: \rightarrow \texttt{Nat} \hspace{0.5cm} \text{VAR}
- mover: \rightarrow \texttt{Nat}^2 \hspace{0.5cm} \text{VAR}

Finally it will turn out to be convenient if we have one more simple programming variable of sort \texttt{Nat}. We intend to use it as a loop-counter in operations such as \texttt{yank_buffer} and \texttt{copy_region_to_buffer}. There is no need for a pair-wise version of counter.

- \textbf{FUNC} counter: \rightarrow \texttt{Nat} \hspace{0.5cm} \text{VAR}

As soon as we try to provide these new variables, we discover the need for two new components: \texttt{SVAR} providing simple programming variables and \texttt{'STRING'} providing an implementation of the sort 'String'. Therefore we have an intermezzo introducing two new components.

\section*{5.5.8 Specifying Simple Programming Variables}

We need a class description specifying a simple programming variable. With a \textit{simple programming variable} we mean a variable nullary function of type \texttt{\rightarrow Item}. Of course there must be an update procedure. We do not include some form of variable creation.
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

LET SVAR_SPEC :=

LAMBDAX : Item
EXPORT

SORT Item,
FUNC val: Item, -> Item,
PROC upd: Item ->

FROM
IMPORT X INTO
CLASS

FUNC val: Item -> Item VAR
PROC upd: Item -> MOD val

AXIOM FORALL i: Item
( < upd(i) > TRUE;
[ upd(i) ] val = i )

END;

5.5.9 SVAR: a Postulated Component

We postulate a new component SVAR with black-box description SVAR_SPEC.
Just as for ATTR2 and ATTR, we best employ the meta-operator COPY for using
SVAR. When syntax- and type-checking we replace this COPY(SVAR) by SVAR.

(a) COMP SVAR : SVAR_SPEC;

5.5.10 'STRING': another Postulated Component

Since we have already a specification 'STRING_SPEC', we can directly pos-
tulate the following component:

(a) COMP 'STRING' : 'STRING_SPEC';

5.5.11 Transforming the State space (continued)

After this intermezzo we can give the formal definition of the new variables
which are added to the state space of MITEFA. First of all we instantiate
TABLE.
LET \( \text{TABLE\_STRING\_BUF} := \) 
APPLY APPLY RENAME 
\[
\text{SORT Map TO Buf\_Map}, \\
\text{SORT Item1 TO String}, \\
\text{SORT 'Item1' TO 'String'}, \\
\text{PRED item1\_inv: TO string\_inv}, \\
\text{SORT Item2 TO Buf}
\]
IN TABLE TO 'STRING' TO BUF;

This instantiation of \( \text{TABLE} \) provides us with operations such as PROC new: \( \rightarrow \) Table, PROC add:Table \# 'String' \# Buf \( \rightarrow \) and PROC app: Table \# 'String' \( \rightarrow \) Buf.

Now we introduce our new variables which is achieved by renaming and instantiating \( \text{SVAR} \). This use of \( \text{SVAR} \) can be viewed as the COLD-K way of declaring variables.

Since these variables will serve as representations of (marked) texts, we collect them in a class description called \( \text{TEXT\_VARS} \). We import the application domain specific notational framework of \( \text{APP\_DOM\_SPEC} \). We also import the data type of maps from strings to marked texts, but this need not be executable; we only use it for reasoning purposes.

LET \( \text{TEXT\_VARS} := \) 

IMPORT Buf INTO 
IMPORT Char INTO 
IMPORT Bool INTO 
IMPORT 'STRING' INTO 
IMPORT APP\_DOM\_SPEC INTO

IMPORT APPLY APPLY RENAME 
\[
\text{SORT Item1 TO String}, \\
\text{SORT Item2 TO MText}
\]
IN MAP\_SPEC TO STRING\_SPEC TO MTEXT\_SPEC INTO

IMPORT APPLY RENAME 
\[
\text{SORT Item TO String}, \\
\text{SORT Set TO Set1}
\]
IN SET\_SPEC TO STRING\_SPEC INTO

IMPORT APPLY RENAME 
\[
\text{SORT Item TO Table}, \\
\text{FUNC val: \( \rightarrow \) Item TO table}, \\
\text{PROC upd: Item \( \rightarrow \) TO upd\_table}
\]
IN COPY(SVAR) TO TABLE\_STRING\_BUF INTO
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

IMPORT APPLY RENAME
  SORT Item TO 'String'.
  FUNC val: -> Item TO current.
  PROC upd: Item -> TO upd4_current
IN COPY(SVAR) TO 'STRING' INTO

IMPORT APPLY RENAME
  SORT Item TO Nat,
  FUNC val: -> Item TO mover.
  PROC upd: Item -> TO set_mover
IN COPY(SVAR) TO NAT INTO

IMPORT APPLY RENAME
  SORT Item TO Nat,
  FUNC val: -> Item TO mover1,
  PROC upd: Item -> TO set_mover1
IN COPY(SVAR) TO NAT INTO

IMPORT APPLY RENAME
  SORT Item TO Nat,
  FUNC val: -> Item TO mover2,
  PROC upd: Item -> TO set_mover2
IN COPY(SVAR) TO NAT INTO

IMPORT

CLASS

  FUNC mover: -> Nat # Nat
  DEF (mover1,mover2)

  PROC set_mover: Nat # Nat ->
  PAR i:Nat,j:Nat
  DEF set_mover1(i);
  set_mover2(j)

END INTO

IMPORT APPLY RENAME
  SORT Item TO Nat,
  FUNC val: -> Item TO counter,
  PROC upd: Item -> TO set_counter
IN COPY(SVAR) TO NAT INTO

IMPORT TABLE_STRING_BUF INTO
CLASS

Note that we have two alternative ways of dealing with the pair-wise version of the mover. The first approach is to write directly in terms of mover1 and mover2 and alternatively we could use mover: → Nat3.

Our next task is to describe precisely the relation between the old variables and the newly introduced ones. Therefore we aim at the definition of an abstraction function f: Buf → MText. We introduce the abstraction in two phases, using the algebra of strings as an intermediate level. This is natural since the linear structure of blocks is closely related to the structure of strings. We shall have several abstraction functions which are called f — by overloading. The functions f taking a Block argument take all characters into account, including those in the gap. The functions f taking a Buf argument skip the characters in the gap.

\[
\text{FUNC } f: \text{ Block } \# \text{ Nat } \# \text{ Nat } \rightarrow \text{ String}
\]
\[
\text{PAR } b: \text{ Block}, n: \text{ Nat}, m: \text{ Nat}
\]
\[
\text{% the string obtained from cont(b,m..cont(b,n-1)}
\]
\[
\text{DEF } ( m = n \quad ?; \text{ empty}
\]
\[
\quad \text{ NOT } m = n \quad ?; \text{ cons(cont(b,m),f(b,succ(m),n))}
\]
\[
\quad )
\]
\[
\text{PRED in_gap: Buf } \# \text{ Nat}
\]
\[
\text{PAR } b: \text{ Buf}, n: \text{ Nat}
\]
\[
\quad \% n is a position in the gap of b
\]
\[
\text{DEF } \text{ geq}(n, \text{ gap1}(b)) \land \text{ leq}(n, \text{ gap2}(b))
\]

The next function f is meant for arguments b,m and n such that \( m \leq n \) where m and n are not in the gap of b. This f is defined by a case analysis where we distinguish two cases. Assume the above restriction on b,m,n. The first case is characterised by \text{ leq}(n, \text{ gap1}(b)) \lor \text{ geq}(m, \text{ gap2}(b)) \) and thus either \( m \leq n \leq \text{ gap1}(b) \) (both parameters are to the left of the gap) or \( \text{ gap2}(b) \leq m \leq n \) (both parameters are to the right of the gap). The second case is characterised by \text{ NOT( leq}(n, \text{ gap1}(b)) \lor \text{ geq}(m, \text{ gap2}(b)) \)) and thus \( m \leq \text{ gap1}(b) \leq \text{ gap2}(b) \leq n \) which means that the positions \( m, \ldots, n-1 \) encompass the gap.

\[
\text{FUNC } f: \text{ Buf } \# \text{ Nat } \# \text{ Nat } \rightarrow \text{ String}
\]
\[
\text{PAR } b: \text{ Buf}, n: \text{ Nat}, m: \text{ Nat}
\]
\[
\% \text{ for leq}(m,n) \text{ and not in_gap}(b,m) \text{ and not in_gap}(b,n):
\]
\[
\% \text{ string from positions } m \ldots n-1 \text{ except for those in the gap}
\]
\[
\text{DEF } ( ( \text{ leq}(n, \text{ gap1}(b)) \lor \text{ geq}(n, \text{ gap2}(b)) \) \quad ?:
\]
\[
\quad f(\text{block}(b),m,n)
\]
\[
\quad \text{ NOT( leq}(n, \text{ gap1}(b)) \lor \text{ geq}(m, \text{ gap2}(b)) \) ?;
\]
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

LET s1:String; s1 := f(block(b), m, gap1(b));
LET s2:String; s2 := f(block(b), gap2(b), n);
    cat(s1, s2)
)

FUNC f: Buf # Nat -> String
PAR b: Buf, n: Nat
% string from positions 0..n-1 except for those in gap
DEF f(b, o, n)

FUNC f: Buf -> String
PAR b: Buf
% string obtained from the entire block, skipping the gap
DEF f(b, size(block(b)))

These functions f enable us to view a part of a buffer or an entire buffer as a string. As a next step we shall push the overloading somewhat further and we introduce another collection of functions f which enable us to view a part of a buffer or an entire buffer as a text.

FUNC f: Block # Nat # Nat -> Text
PAR b: Block, m: Nat, n: Nat
DEF text(f(b, m, n))

FUNC f: Buf # Nat # Nat -> Text
PAR b: Buf, m: Nat, n: Nat
DEF text(f(b, m, n))

FUNC f: Buf # Nat -> Text
PAR b: Buf, n: Nat
DEF text(f(b, n))

In the defining expression of the following function we need a type-cast because otherwise the Text expression text(f(b)) would become ambiguous (assuming b: Buf). The intended interpretation is that f(b) is a string; the other interpretation is that f refers to a function f: Buf -> MText to be introduced later, which would correspond with f(b) being a MText. The guideline of where to use these type-casts and where not to is very simple: use no type-casts except where a real ambiguity arises. When it turns out that many type-casts are needed, one may conclude that the usefulness of overloading has reached its limit.

FUNC f: Buf -> Text
PAR b: Buf
DEF text(f(b)); String)
FUNC \( f : \text{Buf} \rightarrow \text{MText} \)
PAR \( b : \text{Buf} \)
DEF LET do: Nat \# Nat; do := reach(f(b,(dot(b))\#Nat));
    LET ma: Nat \# Nat; ma := reach(f(b,(mark(b))\#Nat));
    ntext(f(b),copa(do),copa(ma))

If we would just aim at the definition of the last function \( f : \text{Buf} \rightarrow \text{MText} \),
then some of the other functions could easily be eliminated. However we prefer this step-by-step
approach, because some of the functions which now seem to be auxiliaries, might turn out useful
in their own right later, e.g. for formulating loop-invariants. The last function \( f \) is our abstraction
function showing how \( \text{Buf} \) objects represent marked texts.

We must extend the abstraction described by \( f : \text{Buf} \rightarrow \text{MText} \) to maps. Recall that \( \text{Buf.Map} \) is the sort of maps from strings to buffers. \( \text{Map} \) is the sort of maps from strings to marked texts.

FUNC \( f : \text{Buf.Map} \rightarrow \text{Map} \)
PAR \( b : \text{Buf.Map} \)
DEF SOME \( n : \text{Map} \)
    ( FORALL \( s : \text{String} \), \( m : \text{MText} \)
      ( app(s,s) = \( m \) \leftrightarrow f(app(bs,s)) = \( m \) ) )

We define the variables required by the specification (Chapter 4) in terms of newly introduced variables. Recall that by means of the imports of \( \text{TEXT_VARS} \) we have a variable function \( \text{current} \) which is of type \( \text{String} \); it is used below to get the String expression \( f(\text{current}) \).

FUNC \( \text{current} : \rightarrow \text{String} \)
DEF \( f(\text{current}) \)

FUNC \( \text{ntexts} : \rightarrow \text{Map} \)
DEF \( f(\text{map}(\text{table})) \)

Using these newly defined variables, we can easily define the functions \( \text{text} \) and \( \text{dot} \) which are formally required here because they occur (as auxiliaries) in the specification of Chapter 4.

FUNC \( \text{text} : \rightarrow \text{Text} \)
DEF \( \text{text}(\text{app(\text{ntexts},\text{current}))} \)

FUNC \( \text{dot} : \rightarrow \text{Nat} \# \text{Nat} \)
DEF \( \text{dot}(\text{app(\text{ntexts},\text{current}))} \)

Now we turn our attention to formulating a suitable invariant assertion. The assertions \( \text{table.inv} \) and \( \text{string.inv} \) do not require special attention since they will hold automatically. Still we must formally have them as conjuncts
of our invariant. This is condition (1) below. The value of the 'String' variable current must be an entry in table. This is condition (2) below. Furthermore we must regulate the relative positions of dot, mark, gap1, gap2 and block-size. This is condition (3) below. Finally the co-ordinate pair representations of dot and mark should correspond with their natural number representations and the reach attribute should contain the reach of the text represented by the buffer. This is condition (4) below.

All these conditions are collected in a predicate called TI'. We can view it as a transformed version of the text-invariant TI from our specification. At the same time it is the representation invariant for the chosen representation of marked texts. We repeat TI1, TI2 etc. here as well because formally the principle of signature incision requires us to have them 'somewhere' in the implementation of WITEFA.

PRED TI1: % as in Section 4.5.6
PRED TI2: % as in Section 4.5.6
PRED TI: % as in Section 4.5.6
PROC witefa_op: -> % as in Section 4.6.5

PRED TI':
DEF (1) string_inv:
    table_inv;
    {2} app(map(table).current);!
    {3} forall a:String( app(map(table),a) ! =>
        ( let b:Buf; b := app(map(table),a);
          leq(dot(b),size(block(b))); 
          leq(mark(b),size(block(b))); 
          leq(gap1(b),size(block(b))); 
          leq(gap2(b),size(block(b))); 
          leq(gap1(b),gap2(b));
          not in_gap(b, dot(b));
          not in_gap(b, mark(b)))));
    {4} forall a:String( app(map(table),a) ! =>
        ( let b:Buf; b := app(map(table),a);
          (dot(b)) : Nat # Nat = reach(f(b, dot(b)):Nat));
          (mark(b)) : Nat # Nat = reach(f(b, mark(b)):Nat));
          (reach(b)) : Nat # Nat = reach(f(b)) ) )

It should be possible to show that

TI' ⊨ TI.

Recall from Chapter 4 that TI ⊨ TI1 ∧ TI2 and that TI1 requires that for
every string $s$ in the domain of $\text{mtext}$s with corresponding marked text $\text{mt}$ the following holds:

\[
\text{intext}(\text{text}(\text{mt}), d) \land \text{intext}(\text{text}(\text{mt}), m) \land \text{ok}(\text{text}(\text{mt}))
\]

where $d$ and $m$ denote the dot and the mark of $\text{mt}$ respectively.

$\text{T12}$ requires that $\text{app}(\text{mtext}s, \text{current})!$. We shall show that $\text{T1'} \Rightarrow \text{T1}$ now.

Assume $\text{T1'}$ and let $s$ be an arbitrary string in the domain of $\text{mtext}s$. Then $\text{app}(\text{mtext}s, s) = f(\text{app}(\text{map}(\text{table}), s))$. If we denote the buffer $\text{app}(\text{map}(\text{table}), s)$ by $\text{b}$, then $\text{T1'}$ gives us the following facts about $\text{b}$.

- $\text{dot}(\text{b}) \leq \text{size}(\text{block}(\text{b}))$
- $\text{mark}(\text{b}) \leq \text{size}(\text{block}(\text{b}))$
- $\text{gap}(\text{b}) \leq \text{size}(\text{block}(\text{b}))$
- $\text{gap2}(\text{b}) \leq \text{size}(\text{block}(\text{b}))$
- $\text{gap}(\text{b}) \leq \text{gap2}(\text{b})$
- $\text{dot}(\text{b}) \not\in \text{gap}(\text{b})$
- $\text{mark}(\text{b}) \not\in \text{gap}(\text{b})$

where we used some obvious shorthand. Therefore $f(b, \text{dot}(b) : \text{Nat})$ is a prefix text of $f(b)$ so its reach is an existing position in $f(b)$. This shows $\text{intext}(\text{text}(\text{mt}), d)$ where $d$ is the dot of $\text{mt}$. The same can be done for mark. Furthermore notice that $f(b)$ is ok since it is defined by means of the conversion function $\text{text} : \text{String} \rightarrow \text{Text}$. This shows $\text{T1}$. Finally $\text{T12}$ follows from $\text{app}(\text{map}(\text{table}), \text{current})!$. Hence we have shown that $\text{T1'} \Rightarrow \text{T1}$.

We also add a few simple operations and predicates which are directly connected with the chosen representation. The names of the operations $\text{eobp}$, $\text{eolp}$ etc. abbreviate end-of-buffer-predicate, end-of-line-predicate etc. We write $(\text{C})\text{OR}$ for $\text{conditional OR}$ as a hint related to the executability of certain assertions.

We shall use some informal terminology but we show by means of a few examples how this can be formalised when needed. Let $\text{b}$ be a given buffer. Then the phrase "$i$ is a position at end-of-buffer" means $f(b,i) : \text{Text} = f(b)$. The phrase "i is a position at end-of-line" means $i = \text{sel}(\text{profile}(f(b)), i_1)$ where $(i_1, i_2) = \text{reach}(f(b, i))$. The phrase "$i$ is a position at beginning-of-line" means $i = 0$ where $(i_1, i_2) = \text{reach}(f(b, i))$. The phrase "$i$ is a position at beginning-of-buffer" means $\text{reach}(f(b,i)) = (0, 0)$.

\[
\text{FUNC right : Buf \# Nat } \rightarrow \text{Nat}
\]

\[
\text{PAR b:Buf,i:Nat}
\]
% next position (going rightwards), skipping gap if necessary
DEF ( succ(i) = gap1(b) \iff; gap2(b) \\
       \iff NOT succ(i) = gap1(b) \iff; succ(i) \\
       )

PRED eobp: Buf \# Nat
PAR b:Buf,i:Nat
% i is a position at end-of-buffer
DEF i = size(block(b))

PRED eolp: Buf \# Nat
PAR b:Buf,i:Nat
% i is a position at end-of-line
DEF eolp(b,i) \iff (c)OR cont(block(b),i) = \text{ctr}_j

PRED eobp: Buf
PAR b:Buf
% dot is at end-of-buffer
DEF eobp(b,\text{dot}(b))

PRED eolp: Buf
PAR b:Buf
% dot is at end-of-line
DEF eolp(b,\text{dot}(b))

FUNC left: Buf \# Nat -> Nat
% next position (going leftwards), skipping gap if necessary
PAR b:Buf,i:Nat
DEF ( i = gap2(b) \iff; pred(gap1(b)) \\
       \iff NOT i = gap2(b) \iff; pred(i) \\
       )

PRED bobp: Buf \# Nat
PAR b:Buf,i:Nat
% i is a position at beginning-of-buffer
DEF i = 0 OR gap1(b) = 0 AND i = gap2(b)

PRED bobp: Buf
PAR b:Buf
% dot is at beginning-of-buffer
DEF bobp(b,\text{dot}(b))

PRED bolp: Buf \# Nat
PAR b:Buf,i:Nat
% i is a position at beginning-of-line
DEF bebp(b,i) {C}HR cont(block(b),left(b,i)) = ctr_j

PRED belp: Buf
PAR b:Buf
% dot is at beginning-of-line
DEF belp(b, dot(b))

END; {of TEXT_VARS}

We must make one remark about the use of the attributes called mover. When we use both mover: Nat and mover: Nat², then we want for a given buffer b the following assertion to hold.

reach(f(b, mover: Nat)) = mover: Nat²,

and we shall refer to this situation by saying that the pair-wise version of mover precisely follows the mover.

We would like to start programming the WITEFA operations, but first we need another intermezzo. Just as in the specification phase (Chapter 4) we prefer to postpone dealing with the fact that the physical display device must mirror the contents of the current buffer. We do so by postulating a suitable component. This is the subject of the next two sections.

5.5.12 Specifying a Window-Invariant Package

We postulate a component which takes care for the window invariant WI from Chapter 4. It provides two procedures called mod_text_restore and mod_dot_restore. The first procedure is intended for being used in the initialisation phase of an editor or after a series of modifications affecting the current text and dot; then mod_text_restore establishes the window invariant WI. The second procedure is intended for being used after a series of modifications moving the dot alone, but still thereby possibly disturbing the window invariant WI.

Below we give a class description WI_PACKAGE_SPEC, specifying these two procedures. Clearly they must operate on the representation of the marked texts and therefore we can only expect them to work if the corresponding representation invariant TI' holds. Note that TEXT_VARS is the class description containing the definition of TI'. Although at first sight it seems natural to import TEXT_VARS into WI_PACKAGE_SPEC, this can not be done because of the condition 'directly specified' from Chapter 3. Instead, we employ a class description TEXT_VARS_SPEC. It can be verified that TEXT_VARS ⊆ TEXT_VARS_SPEC.
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

LET TEXT_VARS_SPEC :=
IMPORT APP_DOM_SPEC INTO
IMPORT APPLY APPLY RENAME
SORT Item1 TO String.
SORT Item2 TO MText
IN MAP_SPEC TO STRING_SPEC TO MTEXT_SPEC INTO
CLASS

SORT Table VAR
FUNC mtexts: -> Map VAR
FUNC current: -> String VAR
PROC upd: Table -> MOD mtexts
PRED TI*: DEP Table, mtexts, current

END;

In WI_PACKAGE_SPEC we have a local invariant wi_package_inv which is necessary in order to allow for additional variables which may be needed for the implementation of WI_PACKAGE. Such variables might be created dynamically and/or need initialisation; hence there must be an initialisation procedure which is init_wi_package. We repeat the definition of WI because it is employed in the axioms; p_sub, p_add and size at their turn are repeated because they occur in WI—also for reasons of signature inclusion.

LET WI_PACKAGE_SPEC :=
IMPORT TEXT_VARS_SPEC INTO
IMPORT DISPLAY_SPEC INTO
CLASS

FUNC p_sub: Nat # Nat # Nat # Nat -> Nat # Nat % as in Section 4.5.12
FUNC p_add: Nat # Nat # Nat # Nat -> Nat # Nat % as in Section 4.5.12
FUNC size: -> Nat # Nat % as in Section 4.5.12
PRED WI: % as in Section 4.5.12

PRED wi_package_inv: VAR

PROC init_wi_package: ->
MOD wi_package_inv

PROC mod_text_restore: ->
MOD wi_package_inv
USE displ_op

PROC mod_dot_restore: ->
MOD wi_package_inv
USE displ_op

PROC mod_dot: -> % auxiliary
DEF LET st': Map;
st' := sttexts;
USE upd:Table -> END;
TI' AND text(app(sttexts,current)) = text(app(st',current)) AND
(mark(app(sttexts,current)).Copa = mark(app(st',current)) AND
FORALL s:String
( NOT a = current => app(sttexts,s) = app(st',s) ) ?

AXIOM [ init_wi_package ] wi_package_inv;
wi_package_inv =>
[ mod_text_restore | mod_dot_restore ] wi_package_inv

AXIOM TI' AND wi_package_inv =>
[ mod_text_restore ] WI

AXIOM TI' AND wi_package_inv =>
WI => [ mod_dot ] [ mod_dot_restore ] WI

AXIOM (TERMINATION)

< init_wi_package > TRUE;
TI' AND wi_package_inv => < mod_text_restore > TRUE;
TI' AND wi_package_inv => < mod_dot_restore > TRUE

We can consider the conjunction of WI and wi_package_inv as a strengthened version of WI and in view of the similarity with the strengthening of TI to TI' we introduce the notation WI'.

PRED WI';
DEF WI AND wi_package_inv

END;

One might be tempted to think that wi_package_inv needs not be in the modification lists of mod_text_restore and of mod_dot_restore. For these procedures are supposed not to violate wi_package_inv and this is guaranteed indeed if wi_package_inv cannot be modified. However, this would be too strong, because it forbids an implementation to change wi_package_inv from false to true – just by luck.
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

5.5.13 WI_PACKAGE: a postulated component

We postulate a component having WI_PACKAGE_SPEC as its black-box description. This is an important component in the sense that it hides all problems related to the fact that our editor is display-oriented. Note that in particular mod_text_restore works as a kind of magic button: just activate it and then the display is completely taken care of. We take a certain risk by postulating this component because it is not entirely clear in advance that sufficiently efficient algorithms for mod_text_restore and mod_dot_restore will be found.

{0} COMP WI_PACKAGE : WI_PACKAGE_SPEC;

5.5.14 Programming the Editing Operations

Recall that the specification requires a classical invariant WI which should logically imply WI and TI. We shall propose such an invariant now.

LET WIIFA_INPL :=

IMPORT DISPLAY INTO
IMPORT TEXT_VARS INTO
IMPORT WI_PACKAGE INTO
IMPORT (APPLY FILE TO 'STRING') INTO CLASS

FRED WI;
DEF WI AND TI

Since we know already that WI' \Rightarrow WI and TI' \Rightarrow TI, the fact that WI logically implies WI and TI is immediate from the definition of WI. Now we have this invariant we can already write the init procedure and one of the witefa_op procedures, viz. switch_to_buffer, which serves as a kind of buffer initialisation. We begin with init. By means of the imports of TEXT_VARS we have ‘assignment’ procedures upd_table and up_current respectively. These are used in the body of init below where upd_table(now) updates the ‘COLD variable’ table, giving it the value new and where upd_current(s) updates the ‘COLD variable’ current, giving it the ‘String’ value s. The procedure create_buffer serves as an auxiliary for both init and switch_to_buffer. Arbitrarily we choose the value 1024 for the initial block-size of newly created buffers.

PROC init: 'String' ->
PAR s: 'String'
DEF upd_table(new):
    create_buffer(s);
    upd_current(s);
    init_wi_package;
    mod_text_restore

PROC create_buffer: 'String' ->
PAR s: 'String'
DEF LET b:Buf: b := create;
    add(table,s,b);
    set_block(b,alloc(1024));
    set_gap1(b,0);
    set_gap2(b,1024);
    set_dot(b,1024);
    set_mark(b,1024);
    % pair-wise:
    set_dot(b,0,0);
    set_mark(b,0,0);
    set_reach(b,0,0)

The specification (Chapter 4) requires that WTI is a classical invariant. Therefore, among other things we must verify INIT ⇒ ∀s: 'String' [ init(s) ] WTI. We shall briefly show this now and we begin with with TI'. This amounts to checking all conjuncts of TI', which are labelled {1}, {2} etc.

Conjunct {1} requires string_inv and table_inv. Now string_inv holds because 'STRING_SPEC' has been obtained from 'SEQ_SPEC' and in the corresponding instantiation it was defined that item_inv ↔ TRUE; therefore INIT ∧ item_inv ⇒ string_inv applies. We apply this reasoning once more: table_inv holds because TABLE_STRING_BUF has been obtained from TABLE and in the corresponding instantiation it was defined that item_inv ↔ string_inv; therefore INIT ∧ item_inv ⇒ table_inv applies. After having verified this we need not worry about string_inv and table_inv any more, for there are absolutely no operations that could make them false.

Conjunct {2} requires the definedness of the current buffer. This is established by upd_table(new), create_buffer(s) and upd_current(s). Conjuncts {3} and {4} regulate the various buffer attributes. In this case there is just one buffer and create_buffer makes its gap cover the entire block. It puts the dot and the mark at the end of the block which is (0,0) when viewed as a co-ordinate pair. This establishes {3} and {4}.

We must also verify that wi_package_inv holds, which precisely is the effect of init_wi_package.
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

Finally we must show that WI holds and this is established by the invocation of mod_text_restore. The procedure mod_text_restore only works if both TI' and wi_package_inv hold but this is the case indeed thanks to the fact that mod_text_restore comes after all other expressions of init. This concludes our verification of INIT ⇒ ∀ s: 'String' [ init(s) ] WI and we proceed with another editor operation.

PROC switch_to_buffer: 'String' ->
PAR s: 'String'
DEF upd_current(s);
   { is_in_dom(s, table) = true ?; SKIP
     | NOT is_in_dom(s, table) = true ?; create_buffer(s) }
   ;
   mod_text_restore

We must verify that switch_to_buffer preserves WI which is WI ⇒ ∀ s: 'String' [ switch_to_buffer(s) ] WI.

Conjunct (1) of TI' holds since we assumed WI as a preconidition. (2) holds because either s is already an entry in table or s is added to table. In the former case it holds by the definition of is_in_dom; in the latter case create_buffer guarantees that the buffer corresponding to s represents a defined text. For (3) and (4) we reason differently for the old buffers (where we must refer to the precondition WI) and for the newly created buffer (where we reason just as for init(s) before).

Finally the invocation of mod_text_restore preserves wi_package_inv and it establishes WI. This shows that switch_to_buffer preserves WI.

Apart from verifying the preservation of WI, we must also verify the satisfaction of the pre- and postcondition style axioms from the specification (Chapter 4). By way of example we show this for switch_to_buffer(s). Its postcondition requires current = f(s) which is made true by the expression upd_current(s). The remainder of its postcondition is based on a case-analysis. If s is an existing entry in table already, then mtexts = PREV mtexts must hold, which is the effect of SKIP indeed. If s is a new entry, then the marked text corresponding with s should be

mtext(zero, cpoa(0, 0), cpoa(0, 0))

which should have been added to PREV mtexts. We easily verify that this is the marked text represented by the buffer created and initialized by create_buffer. This shows that switch_to_buffer satisfies its pre- and postcondition style axiom.
Later we shall program the remaining WITEFA operations. Just like
switch_to_buffer these must also have the properties of preserving WTI and
satisfying their pre- and postcondition style axioms. We shall not treat these
at the same level of detail as switch_to_buffer. We postpone a discussion
of the termination axioms until Section 5.5.15.

We proceed by implementing the operations for dot and mark control as
specified in Chapter 4 Section 4.5.6. To this end we first introduce another
predicate on buffers which is not part of T1', but which should be viewed
as an additional nice buffer property. This predicate is called ready and
roughly speaking, it holds for a buffer if the gap is located at the dot, so that
the buffer is ready for an insert operation at dot. The following buffer is not
ready:

\[ \text{block(b)} \]
\[ \text{dot} \]
\[ \text{gap1} \]
\[ \text{gap2} \]

\text{Fig 5.2. Buffer which is not 'ready'.}

But the buffer below \textit{is} ready:

\[ \text{block(b)} \]
\[ \text{gap1} \]
\[ \text{gap2} \]
\[ \text{dot} \]

\text{Fig 5.3. Buffer which is 'ready'.}

We do not add the requirement that all buffers are ready to T1' because this
would make dot-movements much more complicated and hence probably also
slower.

\text{PRED ready: Buf}
\text{PAR b:Buf}
\text{DEF dot(b) = gap2(b)}

So if a buffer b with non-empty gap is ready, then inserting a character c in
it, as necessary for the insert_character operation, becomes very simple:

\text{store(block(b), gap1(b), c);}
\text{set_gap1(b, gap1(b) + 1)}
where we omitted the necessary updating of the pair-wise mark, dot and
reach attributes. If we want to do an insert operation in a state in which
the buffer in question is not ready, then it should be made so. Therefore
we introduce a procedure make_ready operating on buffers and having use-
rights with respect to store, set_dot, set_mark, set_gap1 and set_gap2.
This procedure must have the following properties which of course need only
be guaranteed if \( TT' \) holds:

1. make_ready(b) only affects attributes of \( b \),
2. [ make_ready(b) ] ready(b),
3. [ make_ready(b) ] \( (f(b)):Text = PREV f(b), \)
   so essentially the represented marked text does not change.

We start the development of make_ready now. We must deal with two
cases depending on the relative position of the gap and the dot. There-
fore we use two auxiliaries called make_ready1 and make_ready2. The task
of make_ready1 is to move the gap downwards to let it meet the dot. The
task of make_ready2 is to move the gap upwards until the new gap2 equals
the dot. Let us give the definition of make_ready now and after that deal
with the auxiliaries.

PROC make_ready: Buf ->
PAR b:Buf
DEF ( dot(b) = gap2(b) ) ?; SKIP
   | less(dot(b),gap1(b)) ?; make_ready1(b)
   | grtr(dot(b),gap2(b)) ?; make_ready2(b)

For make_ready1 we use a loop and as a the loop-body we employ a procedure
gap_down. It must leave the text represented by the buffer unaffected.

\[ \forall b:Buf \ ( TT' \land gap1(b) > 0 \Rightarrow \]
\[ [ gap_down(b) ] \]
\[ ( (f(b)):\text{Text} = \text{PREV } f(b) \]
\[ \land dot(b) : \text{Nat} = \text{PREV } dot(b) \]
\[ \land mark(b) : \text{Nat} = \text{PREV } mark(b) \) \]

After the loop of make_ready1 has been executed, the new position of the
mark must be calculated. This calculation is done by a case-analysis on the
relative position of the mark. Depending on the position of the mark, it
may be necessary to move the mark over the gap, which means to increase
the value of mark by gap2(b) - gap1(b). After execution of the gap_down
loop, there are three possible situations: the first situation is characterised
by mark(b) < dot(b) and since the gap has reached dot(b) by moving
downwards, we are certain that the gap has not crossed mark(b). The second situation is characterised by \( \text{mark}(b) \geq \text{'old value of gap2(b)'} \) which means that \( \text{mark}(b) \) was already at the high end of the buffer compared with the gap and since the gap has moved downwards we are certain again that the gap has not crossed mark(b). The third situation is when neither \( \text{mark}(b) < \text{dot}(b) \) nor \( \text{mark}(b) \geq \text{'old value of gap2(b)'} \) holds, which is precisely when the assertion \( \text{dot}(b) \leq \text{mark}(b) < \text{gap1}(b) \) was valid before moving the gap. In this third situation the value of mark must be increased and after that the mark will have crossed the gap in the sense that \( \text{gap2}(b) \leq \text{mark}(b) \).

\[
\text{PROC make_ready1: Buf ->}
\text{PAR b:Buf}
\text{DEF LET old_gap2:Nat;}
\text{old_gap2 := gap2(b);} 
\text{( NOT dot(b) = gap1(b) ?;}
\text{gap_down(b))} 
\text{ *) dot(b) = gap1(b) ?;}
\text{( lesb(mark(b),(dot(b)):Nat) ?; SKIP}
\text{| geq(mark(b),old_gap2) ?; SKIP}
\text{| NOT (lesb(mark(b),(dot(b)):Nat) OR geq(mark(b),old_gap2)) ?;}
\text{set_mark(b,add(mark(b),sub(gap2(b),gap1(b))))})
\text{set_dot(b,gap2(b))}
\]

The task of make_ready2 is to move the gap upwards until the value of gap2 equals the dot. For make_ready2 we use a loop and in the loop-body we employ a procedure gap_up with the property:

\[
\forall b:Buf \ ( \text{'TI' \& \text{gap2}(b) < size(block(b))} \Rightarrow
\ [ \text{gap_up}(b) ]
\ ( f(b)):Text = PREV f(b)
\ & \text{dot}(b):Nat = PREV \text{dot}(b)
\ & \text{mark}(b):Nat = PREV \text{mark}(b) )
\]

Again we have a loop and a calculation of the mark. Depending on the position of the mark, it may be necessary to move the mark over the gap, which now means to decrease the value of mark by \( \text{gap2}(b) - \text{gap1}(b) \). Again we distinguish three situations where the third situation is precisely when the assertion \( \text{gap2}(b) \leq \text{mark}(b) < \text{dot}(b) \) was valid before moving the gap. In this third situation the value of mark must be decreased and after that the mark will have crossed the gap in the sense that \( \text{mark}(b) < \text{gap1}(b) \).
3.5. IMPLEMENTING THE SYSTEM COMPONENTS

PROC make_ready2: Buf ->
PAR b:Buf
DEF LET old_gap1:Nat;
old_gap1 := gap1(b);
(NOT dot(b) = gap2(b) ?;
gap_up(b)
) *; dot(b) = gap2(b) ?;
( leq(mark(b),old_gap1) ?; SKIP
| geq(mark(b),dot(b)) ?; SKIP
| NOT (leq(mark(b),old_gap1) OR geq(mark(b),dot(b)) ?;
set_mark(b,sub(mark(b),sub(gap2(b),gap1(b)))))
)

We give the definitions of gap_down and gap_up below.

PROC gap_down: Buf ->
PAR b:Buf
% move the gap one position downwards
DEF store(block(b),pred(gap2(b)),cont(block(b),pred(gap1(b))));
set_gap1(b,pred(gap1(b)));
set_gap2(b,pred(gap2(b))

PROC gap_up: Buf ->
PAR b:Buf
% move the gap one position upwards
DEF store(block(b),gap1(b),cont(block(b),gap2(b)));
set_gap1(b,succ(gap1(b)));
set_gap2(b,succ(gap2(b))

We add one remark about the algorithm for make_ready as developed above. As an alternative, it would have been possible to put the adaptation of dot and mark in gap_up and gap_down. This alternative would make the presentation of the algorithm somewhat smoother. On the other hand, the current algorithm is slightly more efficient, which hopefully is worthwhile when editing large texts.

This concludes the work on making buffers ready and we proceed with a few more editor operations, among which those operations which deal with dot movements. Actually this turns out to be quite a lot of work, which seems of a somewhat ad-hoc nature. We shall have to introduce many auxiliaries. In the definition of right_dot below we need a type-cast for the ambiguous expression dot(b); this is because later there will be another function right: Buf # Nat # Nat -> Nat # Nat — which will appear in connection with search_forward. No function left: Buf # Nat # Nat -> Nat # Nat is defined, however.
PROC right_dot: Buf ->
PAR b:Buf
% to be used only if dot is not at end-of-line
DEF set_dot(b,right(b,(dot(b)):Nat));
    LET d1:Nat,d2:Nat; d1,d2 := dot(b); set_dot(b,d1,succ(d2))

PROC forward_character: ->
DEF LET b:Buf; b := app(table,current);
    ( eolp(b) ?; SKIP
      | NOT eolp(b) ?;
       right_dot(b);
        mod_dot_restore )

PROC left_dot: Buf ->
PAR b:Buf
% to be used only if dot is not at beginning-of-line
DEF set_dot(b,left(b,dot(b)));
    LET d1:Nat,d2:Nat; d1,d2 := dot(b); set_dot(b,d1,pred(d2))

PROC backward_character: ->
DEF LET b:Buf; b := app(table,current);
    ( belp(b) ?; SKIP
      | NOT belp(b) ?;
       left_dot(b);
        mod_dot_restore )

PROC belp: -> Bool
DEF LET b:Buf; b := app(table,current);
    ( belp(b) ?; true
      | NOT belp(b) ?; false )

PROC eolp: -> Bool
DEF LET b:Buf; b := app(table,current);
    ( eolp(b) ?; true
      | NOT eolp(b) ?; false )

Next, we turn our attention to the procedure next_line and the first thing to do is introducing an auxiliary function, also called next_line. It serves for finding out whether it is possible to move the dot to the same horizontal position in the next line; if so, then it calculates this new position. The following picture shows the situation where the dot is neither in the first line
nor in the last line of the text.

![Diagram](image)

**Fig 5.4. Operation of next_line.**

This function `next_line` requires a little bit of searching and stepping through the block. There is some searching rightwards for finding the end of the current line and hence the beginning of the next line. Then some further stepping to the right is needed for getting to the position with the appropriate horizontal co-ordinate. Essentially this searching and stepping is programmed using recursion. This need not be very efficient, but at least the recursion depth is limited by the line length. The function `next_line` yields a pair consisting of a Boolean value and a natural number. This pair is `(true, the 'next-line position')`, if this exists; it is `(false,0)` otherwise. We must explain the term 'next-line position'. For a given buffer `b` we say that `n` is the 'next-line position' if

\[
    n_1 = d_1 + 1 \land n_2 = d_2
\]

where \((n_1, n_2) = \text{reach}(f(b, n))\)

and \((d_1, d_2) = \text{dot}(b)\).

Actually the value 0 is just arbitrary. We use some auxiliaries which will be defined immediately afterwards. In the definition of `next_line` below we employ a function `end_of_line` yielding two results which are denoted as `j` and `k`, although `k` is not needed here. Later we shall encounter another usage of `end_of_line` where its second result `(k)` is actually used.

```
FUNC next_line: Buf -> Bool # Nat
PAR  b:Buf
DEF  LET j:Nat, k:Nat;
     j,k := end_of_line(b, dot(b));
     ( eobp(b, j) ?; (false,0) | NOT eobp(b, j) ?; right_stepping(b, right(b, j), hpos(b)) )
```

We used several auxiliaries which will be defined now. The function `right_stepping` applied to a buffer `b` and natural numbers `j` and `k` yields a pair consisting of a Boolean value and a natural number. This pair is
(true, the position obtained by going \( h \) steps rightwards within the same line starting at position \( j \)), if possible; it is \((false, 0)\) otherwise. Again we used a phrase which is easily formalised; we say that \( n \) is the position obtained by going \( h \) steps rightwards within the same line starting at position \( j \) if

\[
n_1 = j_1 \land n_2 = j_2 + h
\]

where \((n_1, n_2) = reach(f(b, n))\)

and \((j_1, j_2) = reach(f(b, j))\).

The procedure right_stepping is defined as follows.

\[
\text{FUNC right_stepping: Buf \# Nat \# Nat -> Bool \# Nat}
\]

\[
\text{PAR b:Buf, j:Nat, h:Nat}
\]

\[
\text{DEF ( h = 0 ?; (true, j) }
\]

\[
| \text{ NOT h = 0 ?; ( eolp(b, j) ?;}
\]

\[
| \text{ (false, 0) }
\]

\[
| \text{ NOT eolp(b, j) ?;}
\]

\[
\text{ right_stepping(b, right(b, j), pred(h)) }
\]

\]

The function hpos simply yields the horizontal co-ordinate of the dot for a given buffer. The function end_of_line applied to a buffer \( b \) and a position \( i \) yields the next position – going rightwards – for which the 'end-of-line' predicate holds and the number of rightward steps needed to reach this position.

It is defined recursively and both result values play a role in the recursion, although this is not the motivation for end_of_line having two results.

\[
\text{FUNC hpos: Buf -> Nat}
\]

\[
\text{PAR b:Buf}
\]

\[
\text{DEF LET d1:Nat, d2:Nat; d1, d2 := dot(b);}
\]

\[
d2
\]

\[
\text{FUNC end_of_line: Buf \# Nat -> Nat \# Nat}
\]

\[
\text{PAR b:Buf, i:Nat}
\]

\[
\text{DEF ( eolp(b, i) ?; (1, 0) }
\]

\[
| \text{ NOT eolp(b, i) ?;}
\]

\[
| \text{ LET n:Nat, m:Nat; n, m := end_of_line(b, right(b, i));}
\]

\[
| \text{ (n, succ(m)) }
\]

\]

The following function is called previous_line and it is an auxiliary for the
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

procedure with the same name. Again we give a sketch.

![Diagram](image)

**Fig 5.5.** Operation of previous_line.

This function yields a pair consisting of a Boolean value and a natural number. This pair is (true, the 'previous-line position'), if this exists; it is (false, 0) otherwise.

```plaintext
FUNC previous_line: Buf -> Bool # Nat
PAR  b:Buf
DEF LET j: Nat; j := beginning_of_line(b, dot(b));
    (bobp(b, j) ?;
     (false, 0)
    | NOT bobp(b, j) ?;
     right_stepping(b, beginning_of_line(b, left(b, j)), hpos(b))
    )
```

We need just one more auxiliary. It serves for finding the next position – going leftwards – for which the beginning-of-line predicate holds.

```plaintext
FUNC beginning_of_line: Buf # Nat -> Nat
PAR  b:Buf, i: Nat
DEF ( bobp(b, i) ?: i
    | NOT bobp(b, i) ?: beginning_of_line(b, left(b, i))
)
```

We implement the remaining procedures for dot and mark control. In the definition of end_of_line below, we see another usage of the earlier end_of_line function where we do need its second result – which reveals the motivation for that end_of_line having two results.

```plaintext
PROC beginning_of_line: ->
DEF LET b:Buf; b := app(table, current);
    set_dot(b, beginning_of_line(b, dot(b)));
    LET d1: Nat, d2: Nat; d1, d2 := dot(b); set_dot(b, d1, 0);
    mod_dot_restore
```
CHAPTER 5. DESIGN OF A TEXT EDITOR

PROC end_of_line: ->
DEF LET b:Buf; b := app(table,current);
LET n:Nat, m:Nat; n.m := end_of_line(b, dot(b));
set_dot(b, n);
LET d1:Nat, d2:Nat; d1, d2 := dot(b);
set_dot(b, d1, add(d2, m));
mod_dot_restore

PROC next_line: ->
DEF LET b:Buf; b := app(table, current);
LET bb:Bool, i:Nat;
b.i := next_line(b);
| bb = false => SKIP
| NOT bb = false =>
  set_dot(b, i);
LET d1:Nat, d2:Nat; d1, d2 := dot(b);
set_dot(b, succ(d1), d2);
mod_dot_restore

PROC previous_line: ->
DEF LET b:Buf; b := app(table, current);
LET bb:Bool, i:Nat;
b.i := previous_line(b);
| bb = false => SKIP
| NOT bb = false =>
  set_dot(b, i);
LET d1:Nat, d2:Nat; d1, d2 := dot(b);
set_dot(b, pred(d1), d2);
mod_dot_restore

FUNC beginning_of_buffer: Buf -> Nat
PAR b:Buf
DEF ( gap1(b) = 0 => gap2(b)
| NOT gap1(b) = 0 => 0
)

PROC beginning_of_buffer: ->
DEF LET b:Buf; b := app(table, current);
set_dot(b, beginning_of_buffer(b));
set_dot(b, 0, 0);
mod_dot_restore

FUNC end_of_buffer: Buf -> Nat
PAR b:Buf
DEF size(block(b))
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

PROC end_of_buffer; ->
DEF LET b:Buf; b := app(table, current);
    set_dot(b,end_of_buffer(b));
    set_dot(b,reach(b));
    mod_dot_restore

PROC set_mark; ->
DEF LET b:Buf; b := app(table, current);
    set_mark(b,(dot(b)):Nat);
    set_mark(b,(dot(b))):Nat # Nat)
    {no screen updating needed}

PROC exchange_dot_and_mark; ->
DEF LET b:Buf; b := app(table, current);
    LET i:Nat; i := dot(b);
    LET j:Nat; j := mark(b);
    LET d1:Nat,d2:Nat; d1,d2 := dot(b);
    LET m1:Nat,m2:Nat; m1,m2 := mark(b);
    set_dot(b,i);
    set_dot(b,i);
    set_dot(b,m1,m2);
    set_mark(b,d1,d2);
    mod_dot_restore

Next comes the implementation of the operations for text modification as
specified in Chapter 4 Section 4.5.7. Again we introduce an additional nice
buffer property. We shall refer to the value of gap2(b) – gap1(b) as the
size of the gap. We shall devote some effort to solving the problem that the
size of the gap may become 0 which implies that there is no more space for
insertions. We introduce a simple predicate space which holds for a buffer
b in case the size of the gap is non-zero. Just as ready, this predicate space
is not part of TI’, although of course it is a desirable buffer property.

PRED space: Buf
PAR b:Buf
DEF gtr(gap2(b),gap1(b))

Inserting in a buffer that has no space requires that space should be created
and for this purpose we introduce a procedure make_space of type Buf #
Nat — and having use-rights with respect to store, set_gap1, set_gap2 and
grow. This procedure should have the following properties which must hold
for n > 0 under the assumption TI’ & ready(b):
1.\texttt{make\_space}(b,n) only affects attributes of \(b\),
2. \([\texttt{make\_space}(b,n) \land \text{space}(b) \land \text{ready}(b)] \rightarrow \text{Text} = \text{PREV Text} \rightarrow \text{STATE}(f(b))\).
3. \([\texttt{make\_space}(b,n)] (\text{f}(b)):\text{Text} = \text{PREV f(b)}\).

The second parameter of \texttt{make\_space} describes the space-increment. It serves for exploiting estimates of the required amount of space. We expect a typical implementation of \textsc{block} to show a reasonable performance if \texttt{grow} is invoked every now and then for a large increment; we expect it to be wasteful and disproportionately slow if it is invoked many times for very small increments. We arrange matters such that the procedure \texttt{make\_space} can also be used in cases where the buffer did not run out of space yet.

The introduction of new space in a buffer takes place at the high end of the buffer. We can view this as the introduction of a second gap, whose size is equal to the amount of new space. From the point of view that we want to respect \(\text{T1}\) this is one gap too many and hence this second gap has to move downwards to meet the other gap. The technique of moving this second gap is essentially the same as used in \texttt{make\_ready}. When the two gaps have met, which is when \texttt{mover} = \texttt{gap2}\((b)\), the invariant \(\text{T1}\)' and the readiness of the buffer can be restored. This is done by merging the two gaps which amounts to making \texttt{gap2}\((b)\) and \texttt{dot}(b) equal to the position immediately after the merged gaps.

\begin{verbatim}
PROC \texttt{make\_space}: Buf \# Nat \rightarrow
PAR b:Buf, n:Nat
DEF \texttt{set\_mover} (\text{size}(\text{block}(b)))
   \text{grow}(\text{block}(b),n);
   ( \text{NOT mover} = \text{gap2}(b) \rightarrow
     \text{second\_gap\_down}(b,n)
   ) \ast; mover = \text{gap2}(b) \rightarrow;
\text{set\_gap2}(b,\text{add}(\text{gap2}(b),n));
\text{set\_dot}(b,\text{gap2}(b));
( \text{NOT len}(\text{mark}(b),\text{gap1}(b)) \rightarrow \text{SKIP}
\rightarrow \text{NOT len}(\text{mark}(b),\text{gap1}(b)) \rightarrow \text{set\_mark}(b,\text{add}(\text{mark}(b),n))
                )
\end{verbatim}

The following picture sketches the situation.
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

![Diagram of merging two gaps]

Fig 5.6. Merging two gaps.

The procedure \texttt{second\_gap\_down} serves for moving the second gap one position downwards. It has two arguments where the first argument indicates a buffer and where the second argument indicates the size of the second gap.

\begin{verbatim}
PROC second_gap_down: Buf \# Nat ->
PAR b:Buf,n:Nat
DEF store(block(b),pred(add(mover,n)),cont(block(b),pred(mover)));
    set_mover(pred(mover))
\end{verbatim}

This concludes the work on making space in buffers. As discussed before, it is simple to insert a non-\texttt{ctrl-\_j} character in a ready buffer with space, leaving the readiness of the buffer invariant. In order to update the pair-wise attributes, we use the natural addition on reaches, which is the function \texttt{add: Nat^2 \rightarrow Nat^2} and furthermore we use other operations on reaches such as paste. The following procedure is meant for inserting a non-\texttt{ctrl-\_j} character in a ready buffer with space. When the dot comes \texttt{before} the mark, the new pair-wise mark attribute is calculated as \texttt{paste(mark(b),0,1,dot(b))}.

In such calculations we avoid using the abstraction functions \texttt{f} which are meant for reasoning purposes only – some of them would be very inefficient indeed.

\begin{verbatim}
PROC insert_character: Buf \# Char ->
PAR b:Buf,c:Char
% only if space(b)
DEF store(block(b),gap1(b),c);
    set_gap1(b,succ(gap1(b)));
    ( lss((mark(b)):Nat,dot(b)) ?;
        SKIP
    | NOT lss((mark(b)):Nat,dot(b)) ?;
        set_mark(b,paste(mark(b),0,1,dot(b)))
    );
    set_reach(b,paste(reach(b),0,1,dot(b)));
    set_dot(b,add(dot(b),0,1))
\end{verbatim}

Also the introduction of a new line in a ready buffer with space can be done
in a simple way, leaving the readiness of the buffer invariant.

PROC newline: Buf ->
PAR b:Buf
% only if space(b)
DEF store(block(b),gapl(b),ctr_j);
set_gapl(b,succ(gapl(b)));

  ( last((mark(b)):Nat,dot(b)) ?;
    SKIP
  | NOT less((mark(b)):Nat,dot(b)) ?;
    set_mark(b,paste(mark(b),1,0,dot(b)))
  )
set_reach(b,paste(reach(b),1,0,dot(b)));
set_dot(b,add(dot(b),1,0))

Now we can program some of the insertion procedures. The first one is insert_file and to keep things simple, we use the insert_character and newline procedures described above. If it is necessary to insert large files frequently, then it might be worthwhile to reconsider this algorithm and to remove the pasting from the main loop.

PROC insert_file: 'String' ->
PAR s:'String'
DEF LET b:Buf; b := app(table.current);
  make_ready(b);
  reset(s);
  ( NOT eof?; ( space(b) ?; SKIP
    | NOT space(b) ?; make_space(b,1024)
  )
  LET c:Char; c := read;
  ( c = ctr_j ?; newline(b)
    | NOT c = ctr_j ?; insert_character(b,c)
  )
) *; eof?;
mod_text_restore

PROC insert_character: Char ->
PAR c:Char
DEF LET b:Buf; b := app(table.current);
  make_ready(b);
  ( space(b) ?; SKIP
    | NOT space(b) ?; make_space(b,128)
  )
  insert_character(b,c):
After this, the newline procedure is easy.

PROC newline : ->
    DEF LET b:Buf; b := app(table, current);
    make_ready(b);
    ( space(b) ?; SKIP
    | BUT space(b) ?; make_space(b, 128) )
    newline(b);
    mod_text_restore

Now we turn our attention to the procedure yank_buffer. We shall use the term yank buffer, by which we mean the buffer corresponding with the 'String' argument of yank_buffer. All characters in the yank buffer must be copied into the current buffer, and as can be expected, a loop will take care for this. At first sight this loop seems rather obvious, but actually its construction requires some care in order not to exclude the case b = y, where b denotes the current buffer and y denotes the yank-buffer. If b = y then the current buffer is inserted into itself. In order to develop the loop, we shall indicate what will be left unaffected by the loop-body and we shall state the loop-invariant.

The loop-body does not affect the marked texts represented by the buffers in table. The invariant is that an initial fragment of the text in the yank buffer is present in the gap of the current buffer b. Both the counter and the mover variables indicate the size of this initial fragment albeit in a different way; more precisely:

\[ f(\text{block}(b), \text{gap1}(b), \text{counter}) = f(y, \text{mover}) \land \text{counter} \leq \text{gap2}(b) \]

Note that two functions f are involved. The first f takes the characters in the gap into account, whereas the second function f is defined to ignore the characters in the gap, which in the latter case is the gap of the yank buffer. Finally the ready and space properties of the current buffer are part of the invariant.

The initialisation for this loop is done by putting the mover at the first non-gap position of y and by making the counter equal to gap1(b). Furthermore of course b must be made ready. To make enough space, make_space is invoked and its second argument is the amount of free space needed for the contents of the yank buffer y. The latter amount equals the size of the block of y minus the size of its gap.
After the contents of the yank buffer has been copied into the gap of \( b \), the value of gap1, which is the lower bound of the gap can be increased in order to make the copied characters really part of the text represented by \( b \). Note that immediately after the contents of \( y \) has been copied, the reach of \( y \) has not changed yet, even not when \( b = y \). Finally the pair-wise mark, dot and reach attributes must be modified by pasting the reach of the yank buffer into them. These modifications of pair-wise attributes should be conducted as a kind of simultaneous assignment due to the possible aliasing in the sense that \( b = y \).

PROC yank_buffer: 'String' ->
PAR s:'String'
DEF LET b:Buf; b := app(table.current);
    LET y:Buf; y := app(table.s);
    make_ready(b);
    make_space(b, sub(size(block(y)), sub((gap2(y), gap1(y)))));
    set_mover({gap1(y) = 0 ?; gap2(y) | NOT gap1(y) = 0 ?; 0});
    set_counter(gap1(b));
    ( NOT mover = size(block(y)) ?;
        store(block(b), counter, cont(block(y), mover));
        set_mover(right(y, mover: Nat));
        set_counter(succ(counter))
    )*: mover = size(block(y)) ?;
    set_gap1(b, counter);

LET r:Nat # Nat; r := reach(y);
    ( los(mark(b), (dot(b)): Nat) ?;
        SKIP
    | NOT las(mark(b), (dot(b)): Nat) ?;
        set_mark(b, paste(mark(b), r, dot(b)))
    );
    set_reach(b, paste(reach(b), r, dot(b)));
    set_dot(b, add(dot(b), r));
mod_text_restore

Now we develop copy_region_to_buffer. Let \( b \) denote the current buffer. First it must be tested if \( \text{mark}(b) < \text{dot}(b) \) for if not, then no action is required. Otherwise, the next thing to be done is to check if the target buffer exists already. If it does not exist, then it should be created and appropriately initialised.
Let $c$ denote the target buffer. The next step is to enter the characters of the region of $b$ into the target buffer $c$. This entering starts at position 0 in $c$. It is possible to calculate the required amount of space in $c$ in advance. More precisely, if the region is larger than the block-size of $c$, then a space-increment of \( \text{dot}(b) - \text{mark}(b) - \text{size}(\text{block}(c)) \) is needed.

There is a simple loop for copying the text from the region to the target buffer $c$. The loop-body leaves the value of the following expression unaffected:

\[
\text{f}(\text{block}(c), 0, \text{counter}) + \text{f}(\text{block}(b), \text{mover}, \text{dot}(b))
\]

where the + denotes natural addition of texts. The first summand is the text already copied and the second summand is the text to be copied yet. Furthermore the loop-body only affects attributes of $c$. The assertion $\text{mark}(b) \leq \text{mover} \leq \text{dot}(b)$ serves as the loop-invariant. Notice again that some care is needed not to exclude the case $b = c$ for in that case we must invoke \text{mod_text_restore}.

```plaintext
PROC copy_region_to_buffer: 'String' ->
PAR s:'String'
DEF LET b:Buf; b := app(table, current);

( NOT less(mark(b), (dot(b)): Nat) ?;
  SKIP
 | less(mark(b), (dot(b)): Nat) ?;

  ( in_in_dom(s, table) = true ?; SKIP
   | NOT in_in_dom(s, table) = true ?; create_buffer(s) )
  );
LET c:Buf; c := app(table, s);

  ( leq(sub(dot(b), mark(b)), size(block(c))) ?; SKIP
   | NOT leq(sub(dot(b), mark(b)), size(block(b))) ?;
   grow(block(c), sub(sub(dot(b), mark(b)), size(block(c))))
  );
set_mover((mark(b)): Nat);
set_counter(0);
( NOT mover = (dot(b)): Nat ?;
  store(block(c), counter, cont(block(b), mover));
  set_mover(right(b, mover: Nat));
  set_counter(succ(counter))
 ) *;
set_mover = (dot(b)): Nat ?;
set_gap1(c, counter);
set_gap2(c, size(block(c)));
set_dot(c, 0);
```
set_mark(c, 0);

LET x: Nat # Nat, y: Nat # Nat;
 x, y := cut(reach(b), mark(b), dot(b));
 set_reach(c, y);
 set_dot(c, 0, 0);
 set_mark(c, 0, 0);

 ( b = c ?; mod_text_restore | NOT b = c ?; SKIP )
 }

Next, we shall proceed with a few operations which are concerned with deleting pieces of text.

PROC delete_next_character: =>
 DEF LET b: Buf; b := app(table, current);
 LET i: Nat, j: Nat; i, j := dot(b);

 ( (cont(block(b), dot(b)) = ctr_j OR
do(b) = size(block(b)) ?;
   SKIP
| NOT (cont(block(b), dot(b)) = ctr_j OR
   dot(b) = size(block(b)) ?;
   make_ready(b);
   ( mark(b) = (dot(b)): Nat ?; set_mark(b, succ(dot(b)))
   | NOT mark(b) = (dot(b)): Nat ?; SKIP
   );
   set_dot(b, succ(dot(b)));
   set_gap2(b, succ(gap2(b)));
   ( leq(mark(b), (dot(b))): Nat ) ?; SKIP
 | NOT leq(mark(b), (dot(b))): Nat ) ?;
   LET x: Nat # Nat, y: Nat # Nat;
   x, y := cut(mark(b), i, j, i, succ(j));
   set_mark(b, x)
 )

 LET p: Nat # Nat, q: Nat # Nat;
 p, q := cut(reach(b), i, j, i, succ(j));
 set_reach(b, p);

 mod_text_restore
 )
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

PROC erase_region: ->
DEF LET b:Buf; b := app(table.current);

  ( geq(mark(b),(dot(b)):Nat) ?;
    SKIP
  | NOT geq(mark(b),(dot(b)):Nat) ?;
    make_ready(b);
    set_gap1(b,mark(b));
    set_mark(b,(dot(b)):Nat);
  )

LET p:Nat # Nat, q:Nat # Nat;
p,q := cut(reach(b),mark(b),dot(b));
set_reach(b,p);
set_dot(b,(mark(b)):Nat # Nat);
mod_text_restore

)

PROC erase_buffer: 'String' ->
PAR a:'String'
DEF ( is_in_dom(a.table) = true ?; SKIP
  | NOT is_in_dom(a.table) = true ?; create_buffer(a) )

LET b:Buf; b := app(table,a);
set_gap1(b,0);
set_gap2(b,0);
set_dot(b,gap2(b));
set_mark(b,gap2(b));

set_dot(b,0,0);
set_mark(b,0,0);
set_reach(b,0,0);

  ( eq(a.current) ?; mod_text_restore
    | NOT eq(a.current) ?; SKIP
  )

Now we turn to the implementation of the operations for marked-text management as specified in Chapter 4 Section 4.5.8, (with an exception for switch_to_buffer which is dealt with already before).

PROC current_buffer_name: -> 'String'
DEF current
For the procedure `write_named_file` we employ a loop and the mover serves as a kind of loop-counter. The loop-body leaves the value of following expression unaffected:

\[
\text{file}(f(s)) + f(b, \text{mover, size}(\text{block}(b)))
\]

and also the fact that mover is not within the gap. The loop-body does not affect other files than the file indicated by `s`.

```plaintext
PROC write_named_file: 'String' ->
  VAR s:'String'
  DEF LET b:Buf; b := app(table, current);
      rewrite(s);
      set_mover((gap1(b) = 0 ?; gap2(b) | NOT gap1(b) = 0 ?; 0));
      ( NOT mover = size(block(b)) ?;
        write(cont(block(b), mover));
        set_mover(right(b, mover:Nat))
      ) *;
      mover = size(block(b)) ?
```

Now we turn to the implementation of the operations for searching as specified in Chapter 4 Section 4.5.9. In the implementation of `search_forward`, we employ a simple loop. We need an auxiliary predicate `match`. We write (C)OR for conditional OR and (C)AND for conditional AND as a hint related to the executability of this recursive predicate.

```plaintext
FREQ match: Buf # 'String' # Nat
  VAR b:Buf, s:'String', i:Nat
  DEF eq(s, empty) (C)OR
    { abs(i, size(block(b)))
      (C)AND hd(s) = cont(block(b), i)
      (C)AND match(b, tl(s), right(b, i))
    }
```

The intuition behind this predicate `match` is that `match(b, s, i)` holds if the text represented by the `len(s)` non-gap positions in `b` starting at `i` equals the text represented by `s`. Of course this definition of `match` corresponds only with the intuition if `i` is not in the gap of `b`. To put it more formally:

\[ i \notin \text{gap}(b) \Rightarrow \text{match}(b, s, i) \equiv \text{match}(f(b), f(s), \text{len}(f(b, i))) \]

where the second `match` is the one from the specification (Chapter 4).

```plaintext
FREQ match': Buf # 'String' # Nat
  VAR b:Buf, s:'String', i:Nat
  DEF i = size(block(b)) (C)OR match(b, s, i)
```

For searching a string `s` in a buffer `b`, the main assertion of the loop-invariant
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

is

\[ \forall i \in \text{Nat} \ ( \text{dot}(b) \leq i < \text{mover} \land i \notin \text{gap}(b) \Rightarrow \text{NOT match}'(b, i, s, t) ) \]

where we used some obvious shorthand. Furthermore we make sure that the pair-wise version of mover precisely follows mover: Nat. This is necessary for in case of a successful search the pair-wise dot attribute must be updated. We have the initialisation mover = dot(b) and the termination condition match'(b, s, mover).

PROC search_forward: 'String' ->
PAR s: 'String'
DEF LET b:Buf; b := app(table, current);
set_mover((dot(b))):Nat # Nat);
set_mover((dot(b))):Nat);

( NOT match'(b, s, mover) ?;
set_mover(right(b, mover):Nat # Nat));
set_mover(right(b, mover):Nat))

) *;
match'(b, s, mover) ?;

( geq(mover, size(block(b))) ?;
SKIP
| NOT geq(mover, size(block(b))) ?;
set_dot(b, mover:Nat # Nat);
set_dot(b, mover:Nat);

mod_dot_restore
)

where we used a simple auxiliary function right. This function right depends on mover: Nat.

FUNC right: Buf # Nat # Nat -> Nat # Nat
PAR b:Buf, i: Nat, j: Nat
DEF ( cont(block(b), mover) = ctr_j ?; (succ(i), 0)
| NOT cont(block(b), mover) = ctr_j ?; (i, succ(j))
)

We expect that improvements could be made with respect to the efficiency of this search-algorithm. Especially the use of the Knuth-Morris-Pratt search algorithm \[5\] could be considered.

Finally we turn to the implementation of the operations for string conversion as specified in Chapter 4 Section 4.5.10. We use recursion which is accept-
able if the procedure below is only used for obtaining relatively short strings such as words to be searched for or such as file names.

```plaintext
PROC buffer_to_string: 'String' -> 'String'
PAR s:'String'
DEF LET b:Buf; b := app(table, s);
    buffer_to_string(b, ( gap1(b) = 0 ?; gap2(b)
    | NOT gap1(b) = 0 ?; 0
    )
)

FUNC buffer_to_string: Buf # Nat -> 'String'
PAR b:Buf, i:Nat
DEF ( i = size(block(b)) ?;
    empty
    | NOT i = size(block(b)) ?;
    cons(cont(block(b),i),buffer_to_string(b,right(b,i)))
)

END;
```

5.5.15 Termination

In the specification (Chapter 4) there is a termination axiom for init, requiring

\[
\text{INIT} \Rightarrow \forall s: \text{String} \quad ( < \text{init}(s) > \text{TRUE} )
\]

and furthermore there is one large axiom dealing with the termination of all operations from the witefa_op group.

\[
\text{WTI} \Rightarrow ( \quad
    < \text{FLUSH bolp} > \text{TRUE};
    < \text{FLUSH eolp} > \text{TRUE};
    < \text{FLUSH forward_character} > \text{TRUE};
    < \text{FLUSH backward_character} > \text{TRUE};
    \text{etc.}
\)

We did not mention the satisfaction of our implementation with respect to these axioms yet and we shall by way of example discuss this for init now. We must unfold the definition of init(s) as presented in Section 5.5.14. This yields

```plaintext
upd_table(new);
create_buffer(s);
```
5.5. IMPLEMENTING THE SYSTEM COMPONENTS

upd_current(s);
init_wi_package;
mod_text_restore

To show that this sequential composition can terminate, we must consecu-
tively verify the termination of each of its sub-expressions separately. We
begin with the first sub-expression which is upd_table(new). We unfold the
specification of new as given in Section 5.3.4.

t := create;
set_map(t, empty);
t

The first step to be verified is t := create which follows from the assertion

\{INST2\} < create > TRUE

given in Section 5.3.2. The next sub-expression is set_map(t, empty) which
we must analyse, starting with t and empty. This t is defined because it is
the result of create whereas empty is a defined function from the class de-
scription MAP_SPEC which was specified algebraically. So set_map is invoked
with defined arguments and hence its termination is a consequence (noting
that set_map is a renamed version of set_attr) of the assertion

\{ATTR1\} < set_attr(i,v) > TRUE

given in Section 5.3.3 where we take t for i and empty for v. This shows that
the evaluation of new can terminate. Also upd_table(new) can terminate;
to see this, notice that upd_table is a renamed version of upd, so we can use
the axiom

< upd(i) > TRUE

of Section 5.5.8. In this way we can proceed until we have checked each
sub-expression of the sequential composition of init. We do not present the
rest here.

Similar verification steps as indicated above for init apply in fact to all
WITEFA operations. Some of these operations have been programmed with
repetition constructs and in these cases the classical technique of variant
functions applies. We do not present these verification steps in this chapter.

This concludes WITEFA_IMPL whence we can implement the last system com-
ponent WITEFA - just as we did earlier with KEYSIND and MOREDOP.
% COMP WITEFA : WITEFA_SPEC := WITEFA_IMPL;
% this is to replace an earlier primitive component.

At this point of the development, $d_{\text{obs}}$ equals (apart from lots of \texttt{LET-constructs}) the design sketched below.

```
<table>
<thead>
<tr>
<th>DESIGN</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP BOOL : BOOLEAN_SPEC;</td>
</tr>
<tr>
<td>COMP NAT : NAT_SPEC;</td>
</tr>
<tr>
<td>COMP CHAR : CHAR_SPEC;</td>
</tr>
<tr>
<td>COMP INST : INST_SPEC;</td>
</tr>
<tr>
<td>COMP ATTR : ATTR_SPEC;</td>
</tr>
<tr>
<td>COMP 'SEQ' : 'SEQ_SPEC';</td>
</tr>
<tr>
<td>COMP TABLE : TABLE_SPEC;</td>
</tr>
<tr>
<td>COMP BLOCK : BLOCK_SPEC;</td>
</tr>
<tr>
<td>COMP DISPLAY : DISPLAY_SPEC;</td>
</tr>
<tr>
<td>COMP FILE : FILE_SPEC; % from $d_5$</td>
</tr>
<tr>
<td>COMP ATTR2 : ATTR2_SPEC;</td>
</tr>
<tr>
<td>COMP SVAR : SVAR_SPEC;</td>
</tr>
<tr>
<td>COMP 'STRING' : 'STRING_SPEC';</td>
</tr>
<tr>
<td>COMP WI_PACKAGE : WI_PACKAGE_SPEC; % newly postulated</td>
</tr>
</tbody>
</table>

COMP WITEFA : WITEFA_SPEC := WITEFA_IMPL;
COMP MOREDOP : MOREDOP_SPEC := MOREDOP_IMPL;
COMP KEYBIND : KEYBIND_SPEC := KEYBIND_IMPL

SYSTEM WITEFA,MOREDOP,KEYBIND |
```

This is a typical point of the top-down development: all system components have been implemented, but this was only achieved by introducing a number of newly postulated primitive components, to wit: \texttt{ATTR2}, \texttt{SVAR}, \texttt{'STRING'} and \texttt{WI_PACKAGE}. Therefore the development process is by far not finished yet: we have to embark on the implementation of these primitive components now. It is characteristic for the top-down approach that no implementation effort has been spent on un-used components: each of the three implemented components was actually part of the system and hence was formally part of the \textit{visible external interface} of the design.
5.6 Implementing the Internal Components

During the implementation of the system components, several new components have been postulated: ATTR2, SVAR, 'STRING' and WI_PACKAGE. We call them internal components, in view of the fact that they are not part of the system of the design d_clear. In this section the top-down development process is continued; the internal components are implemented which leads again to the introduction of new components etc. This goes on until only the primitives of Section 5.3 are left.

5.6.1 Implementing WI_PACKAGE

We turn our attention to mod_text_restore and mod_dot_restore, but it will take about 10 pages of preparations before arriving at the actual definition of these procedures. We adopt the following two-phase approach for the execution of mod_text_restore and mod_dot_restore.

- First phase: decide if a new cursor or a new screen contents is needed. If so, then derive from the buffer named current the desired cursor value (let us call this the concept cursor) and store the desired screen contents in a two-dimensional array variable (the concept screen).
- Second phase: if necessary, transfer the contents of the concept screen to the actual screen of the physical display device and make the actual cursor equal to the concept cursor.

The main reason for adopting this two-phase approach is that it gives rise to a very desirable separation of concerns. Indeed, the task to be performed by mod_text_restore is rather complex: first of all there is a non-trivial structure-clash between the representation of the text in the buffer named current and the two-dimensional layout of the text to be visualised on the screen. Secondly the text must be subject to applications of the look, fill and printify operations, as prescribed by WI. Also mod_text_restore has to establish suitable values for cursor and origin. In addition to this we assume that we are faced with a physical display device with limited capabilities and whose communication link is a potential bottleneck in the execution speed of the editor. Although the screen has a two-dimensional layout, we assume that the transfer of text towards the screen is done by so-called serial communication. This amounts to the restriction that all text transfer goes by means of commands such as ce (clear to end-of-line), cm (cursor motion) and print (send one single character). Avoiding gross inefficiencies in the display-handling is a major complication of the task of mod_text_restore.
We must pay a price for this solution of achieving a separation of concerns. The price is the time to be spent on copying characters from the buffer named current into the two-dimensional array variable (the concept screen). Immediately after such copying has taken place, one or more of these characters is transferred again from the concept screen towards the display and clearly it would be a short-cut to take the characters directly from the buffer.

These considerations heavily depend on the nature of the computing machinery available. When there is enough computing power available whereas the connection with the display is a classical and relatively slow serial link we need not worry about the price of filling an array. This could be the case when a VT102 is connected to a VAX, say. But when the computing power is very restricted whereas the display is easily accessible, e.g. because it is directly built into the computer hardware, then we better adopt another solution. This might be the case when using an APPLE-II, say. Since the available computing machinery comes from a rapidly evolving technology, it is almost impossible to give a timeless solution.

The relevant information is passed from the first phase to the second phase by two variables. We present them in informal notation first.

- \texttt{cursor} : \texttt{Nat}^2 \texttt{VAR}
- \texttt{cscreen} : \texttt{array} \,[0..11 - 1,0..co - 1]\, of \texttt{Char} \texttt{VAR}

Before proceeding with \texttt{W1\_PACKAGE}, we insert an intermezzo which is about two-dimensional arrays.

### 5.6.2 Specifying Two-dimensional Arrays

To prepare for the updating of the screen, it is convenient to use a two-dimensional array. In this section we specify two-dimensional arrays. To keep things simple, indexing in an array always starts at zero.

\begin{verbatim}
LET DOMAIN1 :=
EXPORT
  SORT Nat.
  FUNC size1: -> Nat
FROM
IMPORT NAT\_SPEC INTO
CLASS
  FUNC size1: -> Nat
  Axiom size1
END;
\end{verbatim}
5.6. IMPLEMENTING THE INTERNAL COMPONENTS

LET DOMAIN2 :=
EXPORT
SORT Nat,
FUNC size2: -> Nat
FROM
IMPORT NAT_SPEC INTO
CLASS
FUNC size2: -> Nat
AXIOM size2!
END;

LET ARRAY2_SPEC :=
LAMBDAX: DOMAIN1 OF
LAMBDAY: DOMAIN2 OF
LAMBDAZ: ITEM OF
EXPORT
SORT Nat,
SORT Item,
SORT Array2,
FUNC size1: -> Nat,
FUNC size2: -> Nat,
PROC create: -> Array2,
FUNC val : Array2 # Nat # Nat -> Item,
PROC upd : Array2 # Nat # Nat # Item ->
FROM
IMPORT X INTO
IMPORT Y INTO
IMPORT Z INTO
IMPORT NAT_SPEC INTO
CLASS
SORT Array2 VAR
PROC create: -> Array2 MOD Array2

AXIOM
{ARRAY1} < create > TRUE;
{ARRAY2} INIT -> NOT EXISTS a: Array2 ();
{ARRAYS} [ LET a: Array2; a := create ]
a1 AND (PREV NOT a1) AND
FORALL b: Array2 ( (PREV NOT b1) => b = a )

FUNC val: Array2 # Nat # Nat -> Item VAR
**PROC upd**: Array2 # Nat # Nat # Item -> MOD val

**AXIOM**
\[
\forall a: Array2, m: Nat, n: Nat, i: Item \\
\{ \text{ARRAY4}\} < \text{upd}(a, m, n, i) > \text{TRUE}; \\
\{ \text{ARRAY5}\} [ \text{upd}(a, m, n, i) ] \\
\{ \text{val}(a, m, n) = i; \\
\exists b: Array2, k: Nat, l: Nat, j: Item \\
\{ \text{lsq}(k, \text{size}) \land \text{lsq}(l, \text{size}) \} \Rightarrow \\
\{ \text{NOT \text{NOT k = m OR NOT l = n OR NOT a = b \Rightarrow}} \\
\text{val}(b, k, l) = j \Leftrightarrow \text{PREV val}(b, k, l) = j \} \} \}
\]

**END**;

### 5.6.3 ARRAY2: a Postulated Component

We postulate the following component:

\{(t) COMP ARRAY2: ARRAY2_SPEC\};

### 5.6.4 Implementing WLPACKAGE (continued)

After this intermezzo we can introduce the necessary variables.

**LET ccursor** :=

**IMPORT APPLY RENAME**

**SORT Item** TO Nat,
**FUNC val:** -> Item TO ccursor1,
**PROC upd:** Item -> TO set_ccursor1

**IN COPY(SVAR) TO NAT INTO**

**IMPORT APPLY RENAME**

**SORT Item** TO Nat,
**FUNC val:** -> Item TO ccursor2,
**PROC upd:** Item -> TO set_ccursor2

**IN COPY(SVAR2) TO NAT INTO**

**CLASS**

**FUNC ccursor:** -> Nat # Nat

**DEF (ccursor1, ccursor2)**

**PROC set_ccursor:** Nat # Nat ->
PAR \( i \in \text{Nat}, j \in \text{Nat} \)
\[
\text{DEF set_cursor}(i);
\text{set_cursor}(j)
\]
\text{END};

We introduce a class description called \text{CSCREEN} which at its turn contains a local definition \text{CS\_ARRAY} which provides us with arrays of dimensions \( li \) and \( co \). It is interesting to have a look at the application expression below where a renamed version of \text{ARRAY2} is applied to its three actual parameters; these are \text{DISPLAY}, \text{DISPLAY} (again) and \text{CHAR}. Let us check that this application expression is correct: the renamed version of \text{ARRAY2} has three \text{LAMBDA} parameters, each with its associated parameter restriction. The first parameter restriction requires the presence of a sort \text{Nat} and a defined function \( li \); the second parameter restriction requires again \text{Nat} and a defined \( co \) whereas the third parameter restriction only mentions a sort \text{Char}. Now we verify that indeed, \text{DISPLAY} provides both \( li \) and \( co \) - whence it can appear twice. In fact \text{DISPLAY} provides much more than just \text{Nat}, \( li \) and \( co \), but that does not matter. Finally, of course \text{CHAR} provides \text{Char}. To put it somewhat more formally: \text{DISPLAY} \subseteq \text{renamed DOMAINS}' (\( i = 1,2 \)) where \( \subseteq \) refers to the formal implementation relation.

\text{LET CS\_SCREEN} :=

\text{LET CS\_ARRAY} :=
\text{APPLY \text{APPLY \text{APPLY \text{RENAME}}}}
\text{SORT Item TO Char,}
\text{FUNC size\_1: } \Rightarrow \text{Nat TO } li,
\text{FUNC size\_2: } \Rightarrow \text{Nat TO } co
\text{IN ARRAY2 TO DISPLAY TO DISPLAY TO CHAR};

\text{IMPORT CS\_ARRAY INTO}
\text{APPLY \text{RENAME}}
\text{SORT Item TO Array2,}
\text{FUNC val: } \Rightarrow \text{Item TO cscreen,}
\text{PROC upd: Item } \Rightarrow \text{TO upd\_cscreen}
\text{IN COPY(SVAR) TO CS\_ARRAY};

The purpose of using \text{SVAR} in connection with \text{ARRAY2} is to provide for precisely one array. This yields a specialisation of the somewhat more general \text{ARRAY2}, since \text{ARRAY2} allows for a multitude of arrays, all having the same dimensions. In general, these arrays can be created dynamically, e.g. by \text{LET a: Array2}; a := create but because we need only one array, we shall use create just once and keep the Array2 object returned by create in the
simple programming variable `FUNC cscreen: -> Array2`. So we have a two-dimensional variable array `cscreen` and we can index with `i, j : Nat` by writing `val(cscreen, i, j)` and an assignment is denoted as `upd(cscreen, i, j, c)`. This notation for indexing should be compared with the conventional `cscreen[i, j]` or `cscreen[i][j]` of Pascal and C respectively. The notation for assignment should be compared with the conventional `cscreen[i, j] := c` of Pascal or `cscreen[i][j] = c;` of C. Of course the module `CSCREEN` needs initialisation, which is `upd_cscreen(create: Array2)` (cf. the code of `init_wi_package` in Section 5.6.7).

We collect the variables for interfacing the two phases as sketched before in a class description called `CONCEPT_VARS`. We add to this an abstraction function `f: Array2 -> Text` which serves for reasoning purposes. In order to define this abstraction function, we need auxiliaries denoted as `f` and `g`; since this `g` is useful in its own right, we export it as well. In the definition of `CONCEPT_VARS` below we employ two auxiliary class descriptions `CSCREEN_SPEC` and `ABSTRACT_CSCREEN`. The latter class description actually contains the definitions of the functions `f` and `g`. The reader easily recognises that a generalisation-specialisation approach is used in connection with `ABSTRACT_CSCREEN`; the precise reason for this will become clear in the next section. At this point it is sufficient to to notice that in `CONCEPT_VARS` the generalisation-specialisation approach does no harm, since the associated `LAMBDA` can be eliminated easily. It can be verified that `CSCREEN \subseteq CSCREEN_SPEC`.

```
LET CSCREEN_SPEC :=

CLASS

SORT Nat
SORT Char
FUNC li: -> Nat
FUNC co: -> Nat
SORT Array2 VAR
FUNC cscreen: -> Array2 VAR
PROC create: -> Array2 \ MOD Array2
PROC upd_cscreen: Array2 -> MOD cscreen
FUNC val: Array2 \ Nat \ Nat -> Char VAR
PROC upd: Array2 \ Nat \ Nat \ Char -> MOD val

END;

LET ABSTRACT_CSCREEN :=
LAMBDA CS: CSCREEN_SPEC OF
EXPORT
```
SORT Nat.
SORT Line.
SORT Text.
SORT Array2.
FUNC f: Array2 -> Text.
FUNC g: Array2 # Nat -> Line

FROM

IMPORT NAT_SPEC INTO
IMPORT LINE_SPEC INTO
IMPORT TEXT_SPEC INTO
IMPORT CS INTO

CLASS

FUNC g: Array2 # Nat # Nat -> Line
PAR a: Array2, n: Nat, i: Nat
% the line from characters i..co-1 in the n-th line of a
DEF (i = co ?; empty
| NOT i = co ?; cons(val(a, n, i), g(a, n, succ(i)))
)

FUNC g: Array2 # Nat -> Line
PAR a: Array2, n: Nat
% the n-th line in a
DEF g(a, n, 0)

FUNC f: Array2 # Nat -> Text
PAR a: Array2, n: Nat
% the text from lines n..li-1 in a
DEF (n = li ?; niltext
| NOT n = li ?; cons(g(a, n), f(a, succ(n)))
)

FUNC f: Array2 -> Text
PAR a: Array2
% the text represented by a
DEF f(a, 0)

END;

LET CONCEPT_VARS :=
IMPORT CCURSOR INTO
IMPORT CSCREEN INTO
APPLY ABSTRACT_CSCREEN TO CSCREEN;
Note that CONCEPT_VARS is an abbreviation at component level, rather than local within WI_PACKAGE_IMPL. Later it will appear that we happen to have an opportunity for re-using CONCEPT_VARS.

Now we turn our attention to the first phase of the WI_PACKAGE procedures. We present some considerations which are still independent from the representation of marked texts. We start by having another look at the window invariant which was defined as a predicate WI and which was specified to be an observational invariant, i.e. it must be a consequence of the classical invariant WTI. We use + and − as shorthand for the pair-wise addition and subtraction operations on co-ordinate pairs. Let us write ψ as a shorthand for each of the printify functions. WI is given as:

\[
\text{LET origin: Nat}^2; \text{ origin := dot − cursor; }
\]
\[
\text{LET filled: Text; filled := fill(text, origin + size);}
\]
\[
\text{screen = ψ(look(filled, origin, origin + size))}
\]

We note that for a given dot and text, there is a certain degree of freedom in choosing cursor and origin. Although the clause origin := dot − cursor suggests that the cursor must be chosen first, and then after that the origin can be calculated, it can also be done the other way around. Actually we prefer the latter approach for our implementation. Therefore we introduce the origin by a quantifier. We rewrite origin = dot − cursor into cursor = dot − origin and in this way we get an alternative but equivalent definition of WI.

\[
\exists \text{ origin: Nat}^2
\]
\[
( \text{ cursor = dot − origin; }
\]
\[
\text{LET filled: Text; filled := fill(text, origin + size);}
\]
\[
\text{screen = ψ(look(filled, origin, origin + size)) }
\]

Our screen-update strategy will be based on the idea of not moving the origin unless this is really necessary. Therefore the editor must have a mechanism for recalling the previous origin value. More precisely, the editor will keep track of the previous origin value for each buffer separately. This is achieved by introducing a variable function origin: Buf → Nat^2.

If we want to argue that WI holds then we may do so by showing that the value of this variable origin for the current buffer makes the body of the quantified assertion within WI true. In this way we get another predicate which we shall name WI''.

\[
\text{cursor = dot − origin(b);}
\]
\[
\text{LET filled: Text; filled := fill(text, origin(b) + size);}
\]
5.6. IMPLEMENTING THE INTERNAL COMPONENTS

```
screen = \psi(\text{look}(\text{filled}, \text{origin}(b), \text{origin}(b) + \text{size}))
```

where \( b \) denotes the current buffer. Clearly \( WI'' \Rightarrow WI \). We also give the precise definition in COLD-K:

```
DEF WI'':
DEFLET b:Buf; b := \text{app}(\text{table}.current);
cursor = p_{\text{sub}}(\text{dot}(b), \text{origin}(b));
LET filled: Text; filled := \text{fill}(f(b), p_{\text{add}}(\text{origin}(b), \text{size}));
screen = \text{printify}(\text{look}(\text{filled}, \text{origin}(b), p_{\text{add}}(\text{origin}(b), \text{size})))
```

So now \( WI \) has been transformed into \( WI'' \) which essentially consists of two conditions of the following form:

- \( \text{cursor} = \text{dot}(b) - \text{origin}(b) \quad \ldots \quad c_1 \)
- \( \text{screen} = \psi(\text{look}(\text{filled}, \text{origin}(b), \text{origin}(b) + \text{size})) \quad \ldots \quad c_2 \)

We introduce the notations \( <_p \) and \( \leq_p \) for pair-wise comparison of co-ordinates with the meaning

(\( o_1, o_2 \) \( <_p \) \( o_1, o_2 \)) \( :\Leftrightarrow \quad o_1 < o_1 \land o_2 < o_2 \)

(\( o_1, o_2 \) \( \leq_p \) \( o_1, o_2 \)) \( :\Leftrightarrow \quad o_1 \leq o_1 \land o_2 \leq o_2 \)

We consider \( WI'' \) in conjunction with the display invariant which among other things states that \( \text{cursor} <_p \text{size} \). If we combine this with \( c_1 \), we obtain \( \text{dot}(b) <_p \text{origin}(b) + \text{size} \). Furthermore, when the subtraction \( \text{dot}(b) - \text{origin}(b) \) is to yield a defined value, we must have \( \text{origin}(b) \leq_p \text{dot}(b) \). In this way we get another condition, to be named \( c_3 \), with the property that \( c_1 \Leftrightarrow c_3 \). This \( c_3 \) depends on \( \text{dot}(b) \) and \( \text{origin}(b) \). The important observation regarding this \( c_3 \) is that once \( c_3 \) holds, it is always possible to make \( WI'' \) hold by 'assignments' to \( \text{screen} \) and \( \text{cursor} \).

\( c_3 :\Leftrightarrow \text{origin}(b) \leq_p \text{dot}(b) <_p \text{origin}(b) + \text{size} \)

Both \( \text{mod}.\text{dot}.\text{restore} \) and \( \text{mod}.\text{text}.\text{restore} \) have to deal with two possible situations: either \( c_3 \) holds, which means that the old \( \text{origin}(b) \) value is still usable, or \( c_3 \) does not hold, which means that a modification of \( \text{origin}(b) \) is inevitable.

Now we can give a first attempt in formulating our screen-update strategy. We must get two procedures which serve for establishing \( WI'' \) after a modification of the dot and/or the text. The first procedure serves for establishing \( WI'' \) after a modification of the dot. We expect that updating the screen will move the cursor, so we first adjust the screen and after that try to get the cursor right. The definition below still is far from complete, and an ex-
pression such as MOD origin(b); c3 ? is meant as shorthand for "modify origin(b) such that c3 holds".

```
% PROC mod_dot_restore: ->
% DEF ( c3 ?;
% 
%    MOD cursor : cursor = dot(b) - origin(b) {c1} ?;
% 
%    NOT c3 ?;
% 
%    MOD origin(b); c5 ?;
% 
%    MOD screen, cursor ; screen = printfy( ... ) (c2) ?;
% 
%    MOD cursor ; cursor = dot(b) - origin(b) (c1) ?
% )
```

So now, after dot has been modified, we can restore WI' by invoking mod_dot_restore.

The second procedure serves for establishing WI' after a modification of both the dot and the text or of just the text alone. In general, we expect that both the screen and the cursor must be updated, even in case the old origin is still usable. Note that it is reasonable to expect that the cursor needs updating, even if it can essentially stay the same, then it probably still will get messed-up by the screen updating.

```
% PROC mod_text_restore: ->
% DEF ( c3 ?;
% 
%    SKIP
% 
%    NOT c3 ?;
% 
%    MOD origin(b); c3 ?;
% 
% )

% MOD screen, cursor; screen = printfy( ... ) (c2) ?;
% 
% MOD cursor ; cursor = dot(b) - origin(b) (c1) ?
```

Before we program mod_dot_restore and mod_text_restore, we have an intermezzo for postulating a component dealing with display handling, by which we mean updating the cursor and the screen of the physical device for given desired cursor value and desired screen contents.

### 5.6.5 Specifying Display Handling

We introduce two procedures which take data from CONCEPT_VARS, or more precisely, which take data from the concept cursor and the concept screen; they transfer these data to the physical display device. The procedure update_cursor makes the cursor equal to the concept cursor without affecting the screen. The procedure update_screen makes the screen equal to the text represented by the concept screen, thereby possibly disturbing the
cursor. There is also an initialisation procedure and an invariant. The latter is called display_handling_inv.

We would like to add a remark about the module structure of the specification given below. DISPLAY_HANDLING_SPEC is constructed in terms of specification modules only (such as NAT_SPEC, DISPLAY_SPEC, and in fact also CURSOR_SPEC, CSSCREEN_SPEC) and no reference is made to component names (where we mean to those with COMP). To achieve this, we exploit the fact that ABSTRACT_CSCREEN is parameterised. This is the reason for the parameterisation: in the previous section ABSTRACT_CSCREEN was applied to CSSCREEN whereas here it is applied to CSSCREEN_SPEC. In the next section DISPLAY_HANDLING_SPEC will be used as a black-box description; the fact that it is constructed in terms of specification modules only, guarantees that the condition 'directly specified' of Chapter 3 is satisfied.

LET CURSOR_SPEC :=
CLASS

SORT Nat
FUNC cursor: -> Nat # Nat VAR
PROC set_cursor: Nat # Nat -> MOD cursor
END;

LET CONCEPT_VARS_SPEC :=
IMPORT CURSOR_SPEC INTO
IMPORT CSSCREEN_SPEC INTO
APPLY ABSTRACT_CSCREEN TO CSSCREEN_SPEC;

LET DISPLAY_HANDLING_SPEC :=
IMPORT NAT_SPEC INTO
IMPORT DISPLAY_SPEC INTO
IMPORT CONCEPT_VARS_SPEC INTO
CLASS

PRED display_handling_inv: VAR

PROC init_display_handling: ->
MOD display_handling_inv

PROC update_cursor: ->
MOD display_handling_inv USE displ_op

PROC update_screen: ->
MOD display_handling_inv USE displ_op
PRED pre:
DEF LET i:Nat, j:Nat; i,j := ccursor;
\text{ls}(i,j) AND \text{ls}(j,\text{co}) AND f(co\text{screen})

AXIOM [ init\_display\_handling ] display\_handling\_inv

AXIOM display\_handling\_inv AND pre =>
\begin{align*}
& [ \text{update\_cursor} ] \text{update\_screen} \} \text{display\_handling\_inv} \\
& [ \text{update\_cursor} ] \text{cursor} = \text{ccursor} \AND \text{screen} = \text{PREV} \text{screen}
\end{align*}

AXIOM display\_handling\_inv AND pre =>
\begin{align*}
& [ \text{update\_screen} ] \text{screen} = f(co\text{screen})
\end{align*}

AXIOM {TERMINATION}
\begin{align*}
& < \text{init\_display\_handling} > \text{TRUE}; \\
& \text{display\_handling\_inv AND pre} => < \text{update\_cursor} > \text{TRUE}; \\
& \text{display\_handling\_inv AND pre} => < \text{update\_screen} > \text{TRUE}
\end{align*}

END;

One might be tempted to think that display\_handling\_inv needs not to be in the modification lists of update\_cursor and of update\_screen. For these procedures are supposed not to violate display\_handling\_inv and this is guaranteed indeed if display\_handling\_inv cannot be modified. However, this would be too strong, because it forbids an implementation of the procedures to change display\_handling\_inv from false to true.

5.6.6 DISPLAY\_HANDLING: a Postulated Component

After having specified display handling, we formally introduce a component for it.

\begin{align*}
& \text{(4) COMP DISPLAY\_HANDLING : DISPLAY\_HANDLING\_SPEC;}
\end{align*}

5.6.7 Implementing WI\_PACKAGE (continued)

As argued before, our screen-update strategy requires a variable function origin: Buf \rightarrow \text{Nat}^2. We formally introduce this below.
5.6. IMPLEMENTING THE INTERNAL COMPONENTS

LET ORIGIN :=
   APPLY APPLY APPLY RENAME
   SORT Inst TO Buf,
   SORT Item1 TO Nat,
   SORT Item2 TO Nat,
   FUNC attr : Inst -> Item1 # Item2 TO origin,
   PROC set_attr : Inst # Item1 # Item2 -> TO set_origin
   IN COPY(ATR2) TO BUF_INST TO NAT TO NAT;

We shall need two simple programming variables of sort Nat. They will be used as loop-counters for processing two-dimensional arrays.

LET XX :=
   APPLY RENAME
   SORT Item TO Nat,
   FUNC val1: -> Item TO xx,
   PROC upd: Item -> TO upd_xx
   IN COPY(SVAR) TO NAT;

LET YY :=
   APPLY RENAME
   SORT Item TO Nat,
   FUNC val2: -> Item TO yy,
   PROC upd: Item -> TO upd_yy
   IN COPY(SVAR) TO NAT;

After these preparations we can present the import structure of the implementation of WI_PACKAGE.

We use TEXT_VARS of Section 5.5.11, DISPLAY_HANDLING of Section 5.6.6, ORIGIN of Section 5.6.7, XX, YY defined above and finally CONCEPT_VARS of Section 5.6.4.

LET WI_PACKAGE_IMPL :=

IMPORT TEXT_VARS INTO
IMPORT DISPLAY_HANDLING INTO
IMPORT ORIGIN INTO
IMPORT XX INTO
IMPORT YY INTO
IMPORT CONCEPT_VARS INTO

CLASS

At some point in the execution of mod_text_restore and mod_dot_restore it will be necessary to build-up the contents of the concept-screen. Clearly
this contents must be derived from the buffer named \texttt{current}. We shall develop a procedure \texttt{cscreen\_build} taking care for this. This procedure can be viewed as the heart of a display-oriented editor, because it must transform the ‘block-with-positions’ representation of a text into the desired two-dimensional, fixed-size screen-contents. Furthermore it may not be too inefficient because it will be invoked after every insert- or delete-command during execution of an editor. It will take a few pages of work before we shall have completed a detailed definition of this procedure.

We plan to make this \texttt{cscreen\_build} such that it modifies \texttt{cscreen} contents, \texttt{xx. yy.} mover, but nothing else and such that

\begin{verbatim}
cscreen! A c3 \& TI' \Rightarrow
    [ cscreen\_build ]
    f(cscreen)
    = \psi(block(fill(f,origin(b)+size),origin(b),origin(b)+size))
    where \( t = f(b, size(block(b))) \)
\end{verbatim}

and where \( b \) denotes \texttt{app(table(current))}. We shall use a loop for copying the relevant characters from the current buffer one by one into the \texttt{cscreen} array. We get a loop-invariant replacing the sub-term \texttt{size(block(b))} in the postcondition given above by \texttt{mover:Nat}. So this loop-invariant says that the concept-screen contains printified and ‘Procrustesed’ version of an initial fragment of the text in the current buffer. Furthermore we add the assertion that \texttt{mover:Nat} corresponds to \texttt{mover:Nat} to the loop-invariant.

We could initially establish this loop-invariant by making \( f(cscreen) \) equal to a text with blanks only and putting \texttt{mover} = 0. We adopt the idea of filling the \texttt{cscreen} array with blanks indeed, but we can improve with respect to the idea of putting \texttt{mover} = 0. The point is that everything \texttt{before} the origin becomes irrelevant due to the application of the \texttt{look} operator. Therefore we make the mover point initially to the position in the buffer which corresponds with the beginning of the line containing the origin. The initialisation for establishing this loop-invariant uses two procedures:

- \texttt{clear\_cscreen} which puts blanks at all positions of \texttt{cscreen},
- \texttt{mover\_to\_origin\_line} which puts the mover to the beginning of the line containing the origin.

Let us also discuss the termination condition for the loop of \texttt{cscreen\_build} already. Obviously the condition \texttt{mover \geq size(block(b))} would do the job, but again it is easy to do better than that. The point is that everything \texttt{after origin(b) + size} becomes irrelevant due to the application of the \texttt{look} operator. Therefore we adopt the following termination condition
$\text{mover } \geq \text{size(block($b$)) } \lor$
vertical co-ordinate of mover $\geq$ vertical co-ordinate of (origin($b$) $+$ size)

where $b$ denotes app(table(current)). The procedure clear_cscreen serves for putting blanks at all positions of cscreen. It uses $xx$ and $yy$ as loop-counters.

PROC clear_cscreen: $->$
DEF upd_xx(0):
  ( NOT $xx = li$ ?;
    upd_yy(0):
      ( NOT $yy = co$ ?;
        upd_cscreen,$xx,yy,blank$);
        upd_yy(succ(yy))
      ) $*$;
    $yy = co$ ?;
    $xx = succ(xx)$
  ) $*$;
  $xx = li$ ?

The procedure mover_to_origin_line should make the mover move towards the beginning of the line containing the origin. We use an auxiliary $v_{eq}$ to be used in formulating the stop-criterion of the loop of mover_to_origin_line:

PRED $v_{eq}$: Nat $\times$ Nat $\times$ Nat $\times$ Nat
PAR $m1$, $m2$, $o1$, $o2$, Nat
DEF $m1 = o1$

Now we program mover_to_origin_line and we begin with an efficiency consideration. We exploit that, for large texts, the origin is likely to be much closer to the position of the dot than to the begin of the text. Note that the dot, due to our invariant $M''$ (implying c3), never precedes the origin. Therefore we always have to carry out a backward scan, starting from the dot. For this scan we shall employ several 'assignments', where set_mover(loft($b$, mover)) changes mover:Nat whereas set_mover1(pred(mover)) affects mover:Nat $\times$ Nat. Note that we have mover:Nat which is the same as the vertical coordinate of mover:Nat $\times$ Nat. For the introduction of these movers we refer to TEXT_VARS in Section 5.5.11.

For mover_to_origin_line we employ a loop, whose invariant includes the assertion that the vertical co-ordinate of mover:Nat $^2$ (i.e. mover1) corresponds to mover:Nat. Formally this is mover1 = $m$, where ($m1$, $m2$) = reach(f($b$, mover:Nat)) and where $b$ is the relevant buffer. The horizontal co-ordinate mover2 is irrelevant during execution of the loop and afterwards it will be made equal to 0. Furthermore the assertion that the mover does
not precede the origin is part of the invariant as well. More precisely, we have the assertion $o_1 \leq m_1$ where $(o_1, o_2) = \text{origin}(b)$ and where $(m_1, m_2) = \text{reach}(f(b, \text{mover}; \text{Nat}))$. When the beginning of the line containing the origin has not been reached yet, it is easy to maintain the invariant by letting mover: \text{Nat} go one position leftwards and adapting mover1 correspondingly. For adapting mover1, two situations arise: either the leftward step crossed a line-boundary line, in which case mover1 must be decremented, or the leftward step took place within the same line, in which case mover1 must keep its value.

PRED \text{mover-to-origin-line}: \text{Buf} ->
PAR b:Buf
DEF set_mover((dot(b)):\text{Nat});
    set_mover((dot(b)):\text{Nat} \# \text{Nat});
    ( NOT ( v_eq(mover,origin(b)) AND dcolp(b,mover) ));
    set_mover(left(b,mover));
    ( ecolp(b,mover) \# set_mover1(pred(mover)))
\| NOT ecolp(b,mover) \# SKIP
)
) *; ( v_eq(mover,origin(b)) AND bcolp(b,mover) ));
set_mover2(0)

We introduce some more simple auxiliaries for comparing co-ordinate pairs.

PRED v_geq: \text{Nat} \# \text{Nat} \# \text{Nat} \# \text{Nat}
PAR m1:\text{Nat},m2:\text{Nat},o1:\text{Nat},o2:\text{Nat}
DEF geq(m1,o1)

PRED h_geq: \text{Nat} \# \text{Nat} \# \text{Nat} \# \text{Nat}
PAR m1:\text{Nat},m2:\text{Nat},o1:\text{Nat},o2:\text{Nat}
DEF geq(m2,o2)

PRED h_less: \text{Nat} \# \text{Nat} \# \text{Nat} \# \text{Nat}
PAR m1:\text{Nat},m2:\text{Nat},o1:\text{Nat},o2:\text{Nat}
DEF less(m2,o2)

To keep the actual definition of \text{screen-build} readable, we need two more auxiliaries. The first is a predicate called \text{built} and it describes the termination condition as discussed before.

PRED \text{built}: \text{Buf}
PAR b:Buf
DEF geq(mover,size(block(b))) OR v_geq(mover,p_add(origin(b),size))

FUNC size: -> \text{Nat} \# \text{Nat}
DEF li, co

The predicate in_window serves for finding out if the mover is at a position which contributes to the term
ψ(look(fill(t, origin(b) + size), origin(b), origin(b) + size)) from the postcondition of cscreen_build. In fact in_window only tests the horizontal co-ordinate of mover, which is safe if we know that the vertical co-ordinate of mover is within the limits given by origin(b) and origin(b) + size. For this reason we must formally add one more conjunct to the loop-invariant of cscreen_build, viz.

vertical co-ordinate of origin(b) ≤ vertical co-ordinate of mover

We easily define in_window using h_geq and h_leq.

PRED in_window: Buf
PAR b: Buf
DEF h_geq(mover, origin(b)) AND h_leq(mover, p.add(origin(b), size))

This concludes the preparations for cscreen_build. Its outline is

"initialisation"
( NOT built(b) ?;
  ( NOT in_window(b) ?; SKIP
   | in_window(b) ?; "copy char unless it is ctr.j"
   );
  "increment mover"
) *; built(b) ?

Finally we fill all the details.

PROC cscreen_build: Buf ->
PAR b: Buf
DEF clear_cscreen;
  mover_to_origin_line(b);
( NOT built(b) ?;
  ( NOT in_window(b) ?; SKIP
     | in_window(b) ?;
       LET c:Char; c := cont(block(b), mover);
       ( c = ctr.j ?;
         SKIP
         | NOT c = ctr.j ?;
           upd(cscreen.p_sub(mover, origin(b))).printify(c)
       )
  )
);
(cont(block(b),mover) = ctr_j ?;
    set_mover1(succ(mover1));
    set_mover2(0)
    | NOT cont(block(b),mover) = ctr_j ?;
    set_mover2(succ(mover2))
);
    set_mover(right(b,mover:Nat))
) =; built(b) ?

We introduce predicates p_1ss and p_leq for the pair-wise comparison of co-ordinates \( \leq_p \) and \( \leq_p \) and the operations p_add and p_sub for pair-wise addition and subtraction. We also introduce the condition c3 as discussed before in Section 5.6.4.

\[
\begin{align*}
\text{PRED } &\text{ p_1ss: } \text{Nat} \times \text{Nat} \times \text{Nat} \times \text{Nat} \\
\text{PAR } &o1: \text{Nat}, o2: \text{Nat}, s1: \text{Nat}, s2: \text{Nat} \\
\text{DEF } &\text{less}(o1,s1) \text{ AND less}(o2,s2) \\
\text{PRED } &\text{ p_leq: } \text{Nat} \times \text{Nat} \times \text{Nat} \times \text{Nat} \\
\text{PAR } &o1: \text{Nat}, o2: \text{Nat}, s1: \text{Nat}, s2: \text{Nat} \\
\text{DEF } &\text{leq}(o1,s1) \text{ AND leq}(o2,s2) \\
\text{FUNC } &\text{ p_sub: } \text{Nat} \times \text{Nat} \times \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \times \text{Nat} \\
\text{PAR } &d1: \text{Nat}, d2: \text{Nat}, o1: \text{Nat}, o2: \text{Nat} \\
\text{DEF } &\text{sub}(d1,o1), \text{sub}(d2,o2) \\
\text{FUNC } &\text{ p_add: } \text{Nat} \times \text{Nat} \times \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \times \text{Nat} \\
\text{PAR } &o1: \text{Nat}, o2: \text{Nat}, s1: \text{Nat}, s2: \text{Nat} \\
\text{DEF } &\text{add}(o1,s1), \text{add}(o2,s2) \\
\text{PRED } &\text{c3: } \text{Buf} \\
\text{PAR } &b: \text{Buf} \\
\text{DEF } &\text{p_leq(origin(b),dot(b)) AND p_1ss(dot(b),p_add(origin(b),size))} \\
\end{align*}
\]

The following detailed version of mod_dot_restore uses the old origin value, if possible. If this is not possible, then it applies the following strategy both for the vertical and horizontal co-ordinate of the origin: *first try zero*, but if zero is not acceptable, then choose the origin co-ordinate such that the cursor gets centered on the screen. In order to implement this idea of centering we need two auxiliaries:

\[
\begin{align*}
\text{FUNC } &\text{halfl_i: } \\ &\rightarrow \text{Nat} \\
\text{DEF } &\text{div}(li_i,2)
\end{align*}
\]
5.6. IMPLEMENTING THE INTERNAL COMPONENTS

FUNC half_co: -> Nat
DEF dlv(co,2)

PROC mod_dot_restore: ->
DEF LET b:Buf; b := app(table.current);
    ( c3(b) ?; SKIP
    | NOT c3(b) ?;
        LET d1:Nat,d2:Nat; d1,d2 := dot(b);
        set_origin(b, ( lss(d1,li) ?; 0
        | NOT lss(d1,li) ?; sub(d1,half_li) )
        ,
        ( lss(d2,co) ?; 0
        | NOT lss(d2,co) ?; sub(d2,half_co) )
    );
    cscreen_build(b);
    update_screen
    );
    set_cursor(p_sub(dot(b),origin(b)));
    update_cursor

This mod_dot_restore has the property to restore WI in the sense that TI \land
wi_package_inv \Rightarrow WI \Rightarrow [ mod_dot ] [ mod_dot_restore ] WI. Just for
completeness we also show what it means to 'modify dot' at the representation
level.

PROC mod_dot: ->
DEF LET t:Text;
    t := f(app(table.current));
    USE set_dot: Buf # Nat ->. set_dot: Buf # Nat # Nat ->,
        set_mover: Nat ->, set_mover: Nat # Nat ->,
        set_gap1, set_gap2
    END;
    f(app(table.current)) = t \land TI' ?

Along the same lines we obtain a detailed version of mod_text_restore.

PROC mod_text_restore: ->
DEF LET b:Buf; b := app(table.current);
    ( c3(b) ?;
    SKIP
    | NOT c3(b) ?;
        LET d1:Nat,d2:Nat; d1,d2 := dot(b);
        set_origin(b, ( lss(d1,li) ?; 0
        | NOT lss(d1,li) ?; sub(d1,half_li) )
    );
CHAPTER 5. DESIGN OF A TEXT EDITOR

)
( less(d2,co) ?; 0
| NOT less(d2,co) ?; sub(d2,half_co)
)
)
cscreen_build(b);
update_screen;
set_cursor(p_sub(dot(b),origin(b)));
update_cursor

To complete WI_PACKAGE_IMPL, we must give the definitions of wi_package_inv
and the initialisation procedure init_wi_package, and repeat the definitions
of WI and WI'.

PRED wi_package_inv:
DEF cscreen! AND display_handling_inv

PRED init_wi_package:
DEF upd_cscreen(create:Array2);
init_display_handling

PRED WI: % as in Section 4.5.12
PRED WI': % as in Section 5.6.12

END;

% COMP WI_PACKAGE : WI_PACKAGE_SPEC := WI_PACKAGE_IMPL;
% this is to replace an earlier primitive component.

5.6.8 Implementing DISPLAY_HANDLING

Now it is time to develop algorithms for the procedures update_cursor and
update_screen. Updating the cursor poses no problem, but updating the
screen is not trivial at all. The development of update_screen will be guided
by efficiency considerations.

We assume that the sending of command-messages to the display is relatively
slow and that it is a possible bottleneck in the execution speed of the editor.
Of course this assumption depends on the precise nature of the communica-
tion mechanism between the computer which executes the editor program
and the display device. If we consider a VT102 type terminal and a 1200
Baud serial communication link our assumption is quite right.

The simplest algorithm, of rewriting the entire screen every time when
update_screen is called, is considered prohibitively expensive in terms of the number of display command-messages issued. Therefore it seems worthwhile trying to economize on the use of these messages. The key observation is that during a typical edit session, most keystroke-command invocations require in fact only a modification of a small fragment of the contents of the screen. In such situations the editor only needs to direct the cursor towards the place where the modifications must take place and then overwrite the old screen contents there.

In order to do so, the editor needs keep track of the contents of the actual screen and of the actual cursor position. For this purpose we introduce a so-called shadow-administration of the screen and the cursor. This administration takes the shape of a two-dimensional array of characters for the shadow-screen and a programming variable of typeNat for the shadow-cursor. In view of the similarity with the variables of CONCEPT_VARS, it seems appropriate to do a re-use at textual level.

\[
\text{LET SHADOW_VARS :=}
\]

\[
\text{RENAME}
\]

\[
\text{FUNC ccursor1 : Nat TO ccursor1,}
\]

\[
\text{FUNC ccursor2 : Nat TO ccursor2,}
\]

\[
\text{FUNC ccursor : Nat # Nat TO ccursor,}
\]

\[
\text{PROC set_ccursor1 : Nat TO set_ccursor1,}
\]

\[
\text{PROC set_ccursor2 : Nat TO set_ccursor2,}
\]

\[
\text{PROC set_ccursor : Nat # Nat TO set_ccursor,}
\]

\[
\text{FUNC cscreen : Array2 TO cscreen,}
\]

\[
\text{PROC upd_cscreen : Array2 TO upd_cscreen}
\]

\[
\text{IN COPY(CONCEPT_VARS);}
\]

We shall need two simple programming variables of sort Nat. They will be used as loop-counters for processing two-dimensional arrays.

\[
\text{LET II :=}
\]

\[
\text{APPLY RENAME}
\]

\[
\text{SORT Item TO Nat,}
\]

\[
\text{FUNC val: Item TO ii,}
\]

\[
\text{PROC upd: Item TO upd_ii}
\]

\[
\text{IN COPY(SVAR) TO NAT;}
\]

\[
\text{LET JJ :=}
\]

\[
\text{APPLY RENAME}
\]

\[
\text{SORT Item TO Nat,}
\]

\[
\text{FUNC val: Item TO jj,}
\]

\[
\text{PROC upd: Item TO upd_jj}
\]

\[
\text{IN COPY(SVAR) TO NAT;}
\]
The definition of `display_handling_inv` is nothing but a formalisation of the idea of maintaining a shadow-administration. We also immediately give the corresponding initialisation procedure `init_display_handling` for establishing this invariant. The initialisation uses a procedure `clear_sscreen` which is invoked by `init_display_handling`. This procedure `clear_sscreen` serves for putting blanks at all positions of `sscreen`. It uses `ii` and `jj` as loop-counters and it is very similar to the procedure `clear_sscreen` as introduced in Section 5.6.7. In `init_display_handling` we use the procedure `cl` from `DISPLAY` to clear the entire screen.

```
LET DISPLAY_HANDLING_IMPL :=

IMPORT Nat     INTO
IMPORT Char    INTO
IMPORT DISPLAY INTO
IMPORT TEXT_OPS2_SPEC INTO
IMPORT CONCEPT_VARS INTO
IMPORT SHADOW_VARS INTO
IMPORT II      INTO
IMPORT JJ      INTO

CLASS

PRED pre: % as in Section 5.6.5

PRED display_handling_inv:
DEF cursor = scursor AND screen = f(sscreen)

PROC init_display_handling: ->
DEF cl;
    upd_sscreen(create:Array2);
    clear_sscreen;
    set_scursor(0,0)

PROC clear_sscreen: ->
DEF upd_i(0);
    ( NOT ii = li ?;
    upd_jj(0);
        ( NOT jj = co ?;
        upd_sscreen(ii,jj,blank);
        upd_jj(succ(jj))
     ) *; jj = co ?;
    upd_i(succ(ii))
    ) *; ii = li ?
```
5.6. IMPLEMENTING THE INTERNAL COMPONENTS

The procedure update_cursor is easy. We use the procedure cm from DISPLAY for moving the cursor.

PROC update_cursor: ->
  DEF ( scursor = ccursor ?; SKIP
    | NOT scursor = ccursor ?;
      cm(ccursor);
      set_scursor(ccursor)
  )

The procedure update_screen is more complicated. The complication comes from the fact that we want to avoid rewriting the entire screen when certain parts of it are in fact still right. We decompose this procedure in a kind of top-down approach (at procedure level rather than at component level). The procedure update_screen contains a loop with \( n \) serving as a loop-counter. Its invariant is given by

\[
\text{display_handling_inv} \land n \leq n1 \land \\
\forall i: \text{Nat} \ ( i < n1 \implies ( \text{sel}\text{screen}(i) = \text{g}\text{screen}(i) ) )
\]

So the relation between screen and its shadow-administration sscreen is always maintained and so is the relation between cursor and scursor. The correspondence between the real screen and the concept screen (cscreen) is enforced line after line.

PROC update_screen: ->
  DEF upd_ni(0):
    ( NOT ni = n1 ?;
      update_line;
      upd_ni(succ(ni))
    ) ; ni = n1 ?

where the procedure update_line is to be detailed below. This update_line should enforce the relation

\[
\text{sel}\text{screen}(ni) = \text{g}\text{cscreen}(ni) = \text{g}\text{sscreen}(ni)
\]

i.e. it should make sure that the \( ni \)-th line arrives both on the screen and in the shadow-administration. Furthermore scursor must be kept up-to-date and neither \( ni \) nor the contents of the other lines may be affected. We use a loop with \( jj \) serving as a loop-counter. Its invariant is given by

\[
\text{display_handling_inv} \land jj \leq co \land \\
\forall j: \text{Nat} \ ( j < jj \implies \\
  ( \text{sel}\text{sel}\text{screen}(ni),j) = \text{val}\text{cscreen}(ni,j) ) )
\]
to which we should add that neither ii nor the contents of the other lines may be affected.

PROC update_line: ->
DEF upd_jj(jj);  
   ( NOT jj = co ?; 
      update_character; 
      upd_jj(succ(jj)) 
   ) *; jj = co ?

where the procedure update_character is to be detailed below. This update_character should enforce the relation

sel(sel(screen,ii),jj) = val(cscreen,ii,jj) = val(sscreen,ii,jj)

whereby it should keep scursor up-to-date and where all positions with co-ordinates (i,j) for \( i \neq ii \lor j < jj \) remain unchanged. We do not require that only position (ii,jj) is affected. This is because we issue ce commands instead of writing blanks, as will be explained below.

We briefly explain update_character. First it is tested if the contents of the screen at position (ii,jj) happens to have the desired value already, for in that case no action is required. We expect it to have the desired value often, but if it has not, then the next step is to make sure that the cursor gets at position (ii,jj).

When a blank must be written, a “clear to end-of-line” command is issued and the corresponding modifications in the shadow-administration are made. Note that when this happens, everything on the screen with co-ordinates in \( \{(i,j) \mid i = ii \land j \geq jj\} \) might change. The motivation for using “clear to end-of-line” commands is is that we hope to gain efficiency because we expect many lines in cscreen to have trailing blanks.

When a non-blank character must be written, the shadow-administration must be kept up-to-date again. We have to be careful with the updating of scursor, because when a character is written, the horizontal co-ordinate of the cursor is incremented by one, except when it is already at co – 1. In the latter case the horizontal co-ordinate keeps its old value.

PROC update_character: ->
DEF ( val(sscreen,ii,jj) = val(cscreen,ii,jj) ?; SKIP 
     | NOT val(sscreen,ii,jj) = val(cscreen,ii,jj) ?; 
     % if the cursor is not at (ii,jj) then put it there:
5.6. IMPLEMENTING THE INTERNAL COMPONENTS

( scursor = (ii, jj) ; SKIP
| NOT scursor = (ii, jj) ;
| cm(ii, jj);
| set_scursor(ii, jj)
)

( val(cscreen, ii, jj) = blank ;
| ce;
| sce
| NOT val(cscreen, ii, jj) = blank ;
| print(val(cscreen, ii, jj));
| upd(cscreen, ii, jj, val(cscreen, ii, jj));
| jj = pred(co) ;
| set_scursor(ii, jj)
| NOT jj = pred(co) ;
| set_scursor(ii, succ(jj))
)

We used an auxiliary sce to be given below. It serves for modifying scscreen corresponding to the effect of a "clear to end-of-line" (ce) operation on the real screen. It has been programmed using recursion, which has the advantage that we do not need another programming variable. We could also use this sce to simplify clear_sscreen, but this would disturb the analogy between clear_sscreen and clear_sscreen; of course we could redesign clear_sscreen, but we prefer to avoid such unnecessary backtracking in the development process. After all, the true reason for introducing sce lies here, where we need it as a 'shadow'-version of ce.

PRDC sce: ->
DEF sce(scursor)

PRDC sce: Nat # Nat ->
PAR i: Nat, j: Nat
DEF ( j = co ; SKIP
| NOT j = co ;
| upd(cscreen, i, j, blank);
| sce(i, succ(j))
)

END;

% COMP DISPLAY_HANDLING:DISPLAY_HANDLING_SPEC:-DISPLAY_HANDLING_TMPL;
% this is to replace an earlier primitive component.

We conclude this section with a remark. Although we tried to do better than the simplest algorithm of rewriting the entire screen every time when
update_screen is called, the resulting solution certainly is not optimal yet. One reason for this lies in the relatively restricted set of terminal capabilities which we assumed by adopting DISPLAY. Another reason is that we consider the search for highly sophisticated screen updating-algorithms outside the scope of this case study.

5.6.9 Implementing ATTR2

It is not hard to give an implementation of ATTR2_SPEC using ATTR.

LET ATTR2_IMPL :=
LAMBDA X : CLASS SORT Inst FREE END OF
LAMBDA Y : CLASS SORT Item1 FREE END OF
LAMBDA Z : CLASS SORT Item2 FREE END OF

EXPORT
  SORT Inst,
  SORT Item1,
  SORT Item2,
  FUNC attr : Inst -> Item1 # Item2,
  PROC set_attr: Inst # Item1 # Item2 ->
FROM
IMPORT X INTO
IMPORT Y INTO
IMPORT Z INTO

IMPORT
  APPLY APPLY REPLACE SORT Item TO Item1 IN COPY(ATTR) TO X TO Y INTO

IMPORT
  APPLY APPLY REPLACE SORT Item TO Item2 IN COPY(ATTR) TO X TO Z INTO

CLASS

  FUNC attr: Inst -> Item1 # Item2
  PAR i:Inst
  DEF (attr(i),attr(i))

  PROC set_attr: Inst # Item1 # Item2 ->
  PAR i:Inst,v:Item1,w:Item2
  DEF set_attr(i,v):
    set_attr(i,w)
5.6. IMPLEMENTING THE INTERNAL COMPONENTS

END;

% COMP ATTR2 : ATTR2_SPEC := COPY(ATTR2_IMPL);
% this is to replace an earlier primitive component.

5.6.10 Implementing SVAR

We give an implementation of SVAR_SPEC where the procedure upd has been transformed into an algorithmic definition. We need not import any other components. The definitions below are considered executable because it is very obvious how they could be mapped onto classical imperative programming-language constructs. We could translate FUNC val: -> Item VAR into a declaration var val:Item; in Pascal or just Item val; in C. An assignment upd(1) would be translated into val := i in Pascal or val = i; in C.

LET SVAR_IMPL :=
LAM X : ITEM OF

IMPORT X INTO
CLASS

FUNC val: -> Item VAR

PROC upd: Item ->
PAR i:Item
DEF MOD val END: val = i ?

AXIOM FORALL i:Item ( < upd(i) > TRUE )

END;

% COMP SVAR : SVAR_SPEC := COPY(SVAR_IMPL);
% this is to replace an earlier postulated component.

5.6.11 Implementing 'STRING'

We give an implementation using 'SEQ'.

LET 'STRING_IMPL' :=

IMPORT CHAR INTO
and the only thing we must program explicitly here is the operation \textit{less}. The definition for \textit{less} given below is derived from its specification (\textit{less}: 'String' \# 'String') and the definition of \textit{less}: String \# String given in Chapter 4. The main goal of the transformation is to avoid evaluating sub-expressions such as \texttt{hd(s)} and \texttt{tl(s)} when \texttt{s} is empty.

\begin{verbatim}
CLASS

   PRED less: 'String' \# 'String'
   PAR a:'String', t:'String'
   DEF eq(a,empty) (C)AND NOT eq(t,empty)
       (C)OR ( NOT eq(a,empty) (C)AND NOT eq(t,empty)
           (C)AND ( less(ord(hd(a)),ord(hd(t))) (C)OR
           hd(a) = hd(t) (C)AND less(tl(a),tl(t)) ) )

END;

% COMP 'STRING' : 'STRING_SPEC' := 'STRING_IMPL';
% this is to replace an earlier primitive component.
\end{verbatim}

5.6.12 Implementing ARRAY2

We can give an implementation using \texttt{BLOCK}.

\begin{verbatim}
LET ARRAY2_IMPL :=
\end{verbatim}
5.6. IMPLEMENTING THE INTERNAL COMPONENTS

\[
\begin{align*}
\text{LET} & \quad \text{LAMBDA} \; X: \text{DOMAIN} \; \text{OF} \\
\text{LET} & \quad \text{LAMBDA} \; Y: \text{DOMAIN} \; \text{OF} \\
\text{LET} & \quad \text{LAMBDA} \; Z: \text{ITEM} \; \text{OF} \\
\text{IMPORT} & \quad X \; \text{INTO} \\
\text{IMPORT} & \quad Y \; \text{INTO} \\
\text{IMPORT} & \quad Z \; \text{INTO} \\
\text{IMPORT} & \quad \text{NAT} \; \text{INTO} \\
\text{IMPORT} & \quad \text{APPLY} \; \text{RENAME} \\
\text{SORT} & \quad \text{Block} \; \text{TU} \; \text{Array2} \\
\text{IN} & \quad \text{COPY} \; \text{(BLOCK)} \; \text{TO} \; \text{Z} \; \text{INTO} \\
\text{CLASS} \\
\text{PROC} & \quad \text{create}: \to \; \text{Array2} \\
\text{DEF} & \quad \text{alloc} \; \text{(mul} \; \text{(size1, size2)}\text{)} \\
\text{FUNC} & \quad \text{val}: \; \text{Array2} \; \# \; \text{Nat} \; \# \; \text{Nat} \; \to \; \text{Item} \\
\text{PAR} & \quad a: \text{Array2}, i: \text{Nat}, j: \text{Nat} \\
\text{DEF} & \quad \text{cont} \; \text{(a, add} \; \text{(mul} \; \text{(i, size1)}\text{), j)}\text{)} \\
\text{PROC} & \quad \text{upd}: \; \text{Array2} \; \# \; \text{Nat} \; \# \; \text{Nat} \; \# \; \text{Item} \; \to \to \\
\text{PAR} & \quad a: \text{Array2}, i: \text{Nat}, j: \text{Nat}, x: \text{Item} \\
\text{DEF} & \quad \text{store} \; \text{(a, add} \; \text{(mul} \; \text{(i, size1)}\text{), j), x)}\text{)} \\
\text{END} \\
\end{align*}
\]

\[
\% \quad \text{COMP \ ARRAY2 : ARRAY2\_SPEC := \text{COPY} \; \text{(ARRAY2\_IMPL)};} \\
\% \quad \text{this is to replace an earlier primitive component.}
\]

After this last design-transformation step \(d_{editor} := \text{td}\_\text{step}(d_{editor})\); we have arrived at the situation that \(\text{bot}(d_{editor}) = d_{s}\), i.e. no further steps are required and the top-down development process is completed.

5.6.13 Arriving at an Editor Design

This concludes our top-down development process and we formally state the system of the design \(d_{editor}\).

\[
\text{SYSTEM \ WITEFA, MOREDOP, KEYBIND}
\]

A rough sketch of the design obtained in this way is given in Appendix D. In a separate document [12], a number of diagrams are given which display
the structure of the resulting design in detail.

5.7 Related work

Partsch [7] describes the specification and transformation process of a line-oriented editor using a sugared version of CIP-L. It is interesting to compare this with the specification and implementation described before.

A text is considered as a sequence of lines, just as in our formalisation. In the formal model of the editor, the text being edited is called the "current text file" and it is modelled as a triple \((t_1, l, t_2)\), where \(t_1\) is the text before the current line, \(l\) is the current line and \(t_2\) is the text after the current line.

This conceptual model is in fact not changed during the subsequent transformation process. This should be contrasted with our design, where there is more difference between the conceptual model of the text being edited (viz. an ok sequence of lines with two co-ordinate pairs) and the chosen representation (viz. an array with a gap and a bunch of pointers and co-ordinate pairs). We feel that, with our choice, we were in a better position for studying issues of data refraction, modularisation and information hiding.

It is interesting to compare our approach to data refraction with Jones' approach presented in [11]. As an example we take 'SEQ_SPEC' and 'SEQ_IMPL' from Chapter 4 and Appendix A respectively. The specification introduces the sort 'Seq' and an abstraction function \(f: \text{Seq} \rightarrow \text{Seq}\). Jones proposes similar functions, but calls them retrieve functions (usually denoted as \(\text{reta}\)). In Jones' approach there are standardised proof obligations associated with each abstraction function. The main difference with our approach lies in the formal status of the proof obligations; in 'SEQ_SPEC' the proof obligations are written down explicitly as a part of the specification. These are the axioms labelled 'ABSTRACTION', e.g. containing the assertion \(\text{item inv AND seq inv} \Rightarrow \forall a, b: \text{Seq}. i, j: \text{Item} ( [ \text{LET} u: \text{Seq}; u := \text{cons}(1, a) \rightarrow f(u) = \text{cons}(f(i), f(s)) ]).\) In Jones' approach one would have similar assertions being part of the proof obligation. When these obligations are fulfilled, i.e. when the proof is given, this shows in Jones' terminology that cons: 'Item' \# 'Seq' \rightarrow 'Seq' models cons: 'Item' \# 'Seq' \rightarrow 'Seq', where strictly, this statement is with respect to the given abstraction function \(f: \text{Seq} \rightarrow \text{Seq}\). In our approach, the principle of black-box correctness gives rise to the statement \(\Gamma \vdash \text{SEQ_IMPL} \subseteq \text{SEQ_SPEC} \) where \(\Gamma\) provides information about BOOL, NAT, INST, ATTR etc. Showing this statement amounts to proving the axioms in 'SEQ_SPEC' from the definitions in 'SEQ_IMPL'. So essentially the same specification techniques and proof obligations are used, but the difference lies in the formal
status of the abstraction functions and the proof obligations.

Let us also compare our work with the EMACS editor [4]. We maintained a kind of compatibility with EMACS at the level of procedure names. Also the chosen representation of texts is somewhat similar to the representation chosen in EMACS. Without doubt, from the viewpoint of its amount of features and also of its efficiency the EMACS editor is superior to the editor constructed in this chapter. However, no attempts have been made to give a formal specification of it and as a consequence many features are of an ad hoc nature. In this context we like to point out that our approach has by no means been pushed to its limits by this case study, and that certainly extensions and improvements to the current design can be made. The documentation of this EMACS takes the shape of comments in the C program text; but here we touch the weak spot of the EMACS design. For example, its so-called ultrahot screen management package contains an explicit warning which suggests that the reader should not even try to understand this module, let alone make a modification in it. This is quite unacceptable from a methodological point of view and we feel that in this respect we did a better job.

5.8 Looking Back

In this section, we shall in retrospect summarise the main lines of the work presented in this chapter. For an evaluation and conclusions we refer to the next section (5.9). One of the main purposes of constructing the design presented in this chapter was to illustrate the notions of component, blackbox description and design as described in Chapter 2 and the correctness preserving transformations studied in Chapter 3. Furthermore we wanted to show how the language COLD-K can be used as a tool for developing complex systems. Therefore, let us explicitly point out some of the interesting points encountered during this case study.

In Section 5.2 we established the top \( d_t \) of the editor design which was easy in view of the preparatory work of Chapter 4. This lead to the following conjunct of the post-condition of the development process:

\[
\text{top}(d_{editor}) = d_t
\]

In Section 5.3 we established the bottom of the editor design. We had to enter a kind of specification phase in order to describe the available primitives. This lead to another conjunct of the post-condition of the development process:

\[
\text{bot}(d_{editor}) = d_b
\]
In Section 5.5 we began a top-down development process, starting with the system components KEYBIND, MOREDOP and WITEFA. When implementing KEYBIND and MOREDOP we were lucky in the sense that we could do so in a rather trivial way. This was possible because the black-box descriptions of these components were already in the right form.

The next component to be dealt with was WITEFA and due to its complexity and the axiomatic style of its description, it was far less trivial to implement. Another complication was the fact that we aimed at a high efficiency in terms of memory usage. Implementation of WITEFA began with choosing a suitable representation of marked texts. This was a crucial step in the development process and it was guided by efficiency considerations, both in terms of execution time and memory usage. Making this representation choice can be viewed as a data reification step. Programming of the various editing operations of the window-and-text facility had to be postponed in order to introduce the basic machinery of abstraction functions and representation invariants which was needed for this data reification.

The knowledge of the algebraic systems related to the concept of text as investigated in Chapter 4 turned out useful. We decided to store texts as block-with-positions which was introduced as a variant of the string representation of texts. We decided to the store dot and mark in two ways. First of all they were stored as natural numbers and secondly their homomorphic images under the reach operation were stored as well – redundantly. The collection of algebraic systems studied in Chapter 4 immediately provided a collection of alternative ideas for this representation: we could have adopted the string representation or we could have added profiles redundantly.

In section 5.6 the top-down development process was continued. The newly postulated components were implemented which lead again to the introduction of new components etc. This went on until at the end of Section 5.6 only the primitives given in Section 5.3 were left.

Let us discuss the well-known advantages and disadvantages of top-down development in the context of this Section 5.6. The main advantage is the possibility of having a separation of concerns. When implementing WITEFA we need not worry about the problems of satisfying the window-invariant \( \psi \), except for the simple fact that at the end of every buffer modification we inserted a mod_dot_restore or mod_text_restore invocation. The fact that at that phase of the development process there was no implementation of mod_dot_restore or mod_text_restore available made it impossible to use hidden implementation details. This is an illustration of Chapter 3 remark 3.4.9(ii). The disadvantage of top-down development is that one has to take a certain risk by postulating components; for \( \psi_1 \text{PACKAGE} \) it was not entirely
clear in advance that sufficiently efficient algorithms for `mod_text_restore` and `mod_dot_restore` could be found.

The next component to be implemented was `MT_PACKAGE` and again we found a workable decomposition of its functionality by adopting a two-phase approach for the execution of its procedures. The interface was given by the data structures of `CONCEPT_VARS` which includes a two-dimensional array. We had to postulate a component `DISPLAY_HANDLING` thereby getting the possibility of not worrying (yet) about the actual screen-updating. Later, when implementing `DISPLAY_HANDLING` we were able to demonstrate how the postcondition specification and the efficiency requirements guided us quite elegantly to an algorithm which performs much better than the simplest algorithm, of rewriting the entire screen every time.

Furthermore we had to implement some data type components such as `ATTR2`, `SVAR` etc. Finally at the end of Section 5.6 we arrived at the situation

\[ \text{bot}(d_{editor}) = d_s \text{ and } \text{top}(d_{editor}) = d_t \]

where furthermore the black-box correctness of \( d_{editor} \) could be assumed.

Our approach with respect to the degree of formalisation was based on the so-called rigorous approach; the black-box descriptions and the glass-box descriptions are in a formal language, but there are no formal proof objects as such. This means that there is no absolute certainty about the black-box correctness of \( d_{editor} \) but the degree of formalisation allows for a fast and convincing analysis of the correctness of a certain component when the suspicion arises that there is a weak spot.

In the Appendix A we give a design \( d_{basic} \) such that the composition of \( d_{editor} \) and \( d_{basic} \) is black-box valid. This means that we can construct the design

\[ d_{editor} \circ d_{basic} \]

whose black-box correctness follows from the black-box correctness of \( d_{editor} \) and \( d_{basic} \) and the black-box validation of their composition by Chapter 3 lemma 3.3.14 (ii). Very much in the same way one could conceive another design, \( d_{app-layer} \) say, where we could add more sophisticated features to the editor, such as dynamic keybinding and programmability. Such a layer would yield another example of the \( \circ \) operator when we would construct \( d_{app-layer} \circ d_{editor} \). We had to restrict ourselves, and we undertook the construction of \( d_{basic} \) but we spent no effort on \( d_{app-layer} \).

The design \( d_{editor} \circ d_{basic} \) was the starting point for an activity of system integration and code generation. This was performed manually, which was
doable, but which in fact also can be automated. In Appendix C we give
the C program resulting from the composition of the editor design and the
design of Appendix A. In Appendix C we briefly discuss some of the technical
points which arose during this manual process of code generation.

5.9 Evaluation

After all the effort spent on the editor case-study, it is time to evaluate the
entire approach and to draw some conclusions.

It is interesting to compare the relative sizes of the specification part of the
editor (Chapter 4) and the implementation part (this chapter), which do not
differ very much. At first sight this might seem strange, for one might expect
specifications to be abstract and hence compact, whereas implementations
tend to be complicated by efficiency considerations and executability con-
straints. So, we seem to have the situation \(|\text{spec}| \approx |\text{impl}|\) whereas one
might expect \(|\text{spec}| \ll |\text{impl}|\). In order to explain this, let us analyse the
situation in more detail. First of all, the chapter on the specification part
of the editor (Chapter 4) is more than just the specification of one editor:
it also contains a general-purpose library of standard modules (BOOL_SPEC,
NAT_SPEC, CHAR_SPEC etc.), a formalisation of the application domain which
is text editing and also models of the interfaces of the editor with its envi-
noment, viz. the display and the file-system. If we consider these things as
a vocabulary which is not a part of the actual editor-specification, we get a
much more restricted notion of 'specification'. Assuming the latter notion
of specification, we have that \(|\text{spec}| \approx \frac{1}{2} |(\text{Chapter 4})|\). But it would not be
fair to conclude that thus the specification is much smaller than the imple-
mentation indeed, because when we have a closer look to the structure of
this chapter, we see that it contains much more than just 'implementations'.
First of all we have again the phenomenon that the interfaces of the im-
plementations are made explicit (INST_SPEC, ATTR_SPEC, BLOCK_SPEC etc.).
Secondly this chapter contains much more than one monolithic implementa-
tion, but a large part of it consists of intermediate specifications, i.e. black-
box descriptions of internal components (ATTR2_SPEC, WI_PACKAGE_SPEC,
DISPLAY_HANDLING_SPEC etc.). It is also worth noting that formally all texts
from Chapter 4 are part of the final composite design \(d_{\text{editor}} \circ d_{\text{basic}}\). Certain
operations from the specification are already executable and are used in that
way indeed (e.g. cut and paste operations on reaches). To give some fig-
ures: \(|d_{\text{editor}}| \approx 4000\) lines whereas the C translation of the final composite
design \(|C(d_{\text{editor}} \circ d_{\text{basic}})| \approx 2000\) lines. Many parts of the COLD texts which
are translated yield, roughly speaking, one line C text for each line of COLD
text. In terms of 'number of lines', $d_{\text{side}}$ does not contribute much to the total composite design. To conclude our analysis, we see that $|\text{spec}| = |\text{impl}|$ is a rather imprecise statement, due to the presence of general-purpose descriptions and intermediate-level descriptions. Our designs contain much more than just a top-level specification of the product and its implementation: they contain much additional descriptions, making various notions and interfaces explicit, which is necessary for reasoning about program-correctness and for making the resulting product maintainable.

Now we turn our attention to the evaluation of the usefulness of COLD-K for the development of a complex system – as this editor is. The fact that COLD-K is a wide-spectrum language turned out very useful: we used both axiomatic descriptions (e.g. in the library) and algorithmic definitions (cut and paste operations on texts, implementation of the editor operations etc.). Furthermore we used both static descriptions (e.g. in the library and for all operations on texts) and state-based descriptions (e.g. for the display, the file-system, the editor operations, blocks, buffers etc.). In this way we were able to choose among several styles of descriptions, depending on the particular problem at hand, and also depending on matters of naturalness and even of taste. This can be considered as an advantage of COLD-K over more restricted formalisms such as algebraic specification languages.

We were able to mimic both the algorithmic constructs and the data-types of languages such as Pascal and C very well, although of course a code-generator would have been of help. The task of translating COLD-K texts to C did not take much time (5% of the total time spent on this case study, say) but there are additional advantages of having an automatic code-generator, viz. the standardisation of the translation and the possibility to do prototyping. There are no fundamental technical obstacles for the construction of such a code-generator for a reasonable subset of COLD-K (or some other COLD version). This should be viewed as an advantage, which in the future could lead to a further increase in attractiveness of the approach where the theory of Chapter 2 and Chapter 3 is instantiated by COLD-K. A weak point of the current manual approach is that maintaining the consistency of the COLD-K text and the C text must be done manually as well. When making design modifications or when correcting design errors (yes, we made some), a very error-prone situation arises.

Let us have a look at the syntax of COLD-K, which at some points is not optimal with respect to user-friendliness. For example instead of $\text{add}(p, q)$ one would like to write $p + q$ and also one would like to have short notations for the frequently occurring renaming and application expressions of parameterised data types, writing $\text{SEQ}[\text{Nat}]$ instead of $\text{APPLY} \text{RENAME SORT Item TO Nat IN SEQ TO NAT}$. Language versions of COLD are on their way where
this will be remedied but at the time this case study was done, no such version was available yet. Anyway it was worthwhile to undertake one or more serious applications with COLD-K before embarking on the construction of yet another language version. Furthermore, as this case study shows, the fact that certain syntactic sugar is missing is not essential for the applicability of the language. It might lead to writing 50% more text, say, and some clumsy notations but that is no serious obstacle for applicability, of course. The availability of powerful abstraction mechanisms and structuring mechanisms is much more essential.

As a next topic we shall review the resulting editor, viewed as a product plus its documentation. The editor design takes reasonable efficiency considerations into account, based on the assumptions that display communication is expensive and that any a-priori restriction to the size of the texts would be unacceptable. Furthermore we assumed that the ratio \[ \frac{|text stored|}{|memory used|} \] should be close to 1 rather than \( \frac{1}{2} \), and this has been achieved by using dynamic memory allocation primitives and the technique of having a movable gap in each buffer. Also the implementations of strings and tables are quite efficient; in fact the tables even could have been simpler for the restricted use made of them when employing the editor via KEYBIND (there are only 3 names). If we would undertake the construction of a higher-level design \( d_{subroutines} \), the complications of our table implementation could turn out worthwhile. There are also certain operations which have a relatively poor performance, but in these cases it is also easy to see why this is the case and how to improve it when needed. An example of this is the insert file operation which takes 43 seconds to insert a 100K text-file on a SUN 3/50. As it turns out, this operation spends 70% of its time to paste (0,1) reaches in the pair-wise mark and reaches attributes. But there is an obvious black-box correctness preserving glass-box modification (bbc-preserving gb-mod in the terminology of Chapter 2) to remedy this: modify WRITE_TA to make insert_file operate on the buffer directly instead of via insert_character and use two auxiliary counters of sort Nat to keep track of the reach of the inserted text; in this way the new mark and the new reach can be found by just one application of paste for each.

Let us have a look at the efficiency of some typical editor operations: to insert a character at the end of a short (i.e. \(< 50 \) characters, say) line takes 58 ms, which is fast enough to respond to manual typing. To insert a character at the beginning of a typical text line (\( = 40 \) characters, say) takes about 100 ms; the 100 ms is when using a VT100 emulation on a SUN whereas this time is three or more times higher when using a 1200 Bd terminal connection. In the latter case the terminal connection has become the bottleneck. To perform a write_named_file operation takes 2.5 seconds for
5.9. EVALUATION

a 100K text-file. We ‘profiled’ the editor during a short typical edit session where a 70 lines document was updated by addition of a few lines of text and several other small modifications. As it turned out, the editor spent 87% of time on mod_text_restore and 1.5% on mod_dot_restore. These mod_...._restore's invoke cscreen_build and update_screen and the editor spent 61% and 25% of its time on the latter two procedures respectively. The execution speed of most typical editor operations depends neither on the number of lines in the text being edited nor on the position of the dot – obvious exceptions being search_forward, insert_file and write_named_file. So when inserting characters, it does not matter whether the dot is at the beginning of a short text or in the middle of a 100K text. This advantage is due to our choice of the buffer data structures and the move_to_origin_line algorithm. The resulting editor has been used on several occasions and it turns out to be usable, though of course it is not the fastest and most sophisticated editor currently available. We conclude that the editor is quite usable from an efficiency point of view, but this does not mean that it is a true product which is ready for sales and distribution – this was not the purpose of the case study after all. Especially when considering the moderate assumptions on the display-capabilities and the fact that we took no special efforts to push the execution speed to its limits, we can from an efficiency viewpoint regard the editor experiment as relatively successful.

We must also review the documentation of the editor, which consists of Chapter 4 and this chapter. It provides for a formal description of all functional aspects of the editor design as well as for informal explanations. The strong point of our approach is that all interfaces with the environment of the editor have been made explicit and that there are specifications at several levels. This documentation (Chapter 4 and this chapter) is quite voluminous and maybe it is hard work for a reader to get completely through it, but it is important to realise that these chapters contain a complete description of the editor design, including all details of its interfaces and with very accurate specifications of all data-types involved; furthermore we dealt with various exceptional cases which tend to complicate the design, such as the case of the cursor reaching the end of a line or the bottom of the screen, the case of trying to move the dot rightwards when at the end of a text line etc. Altogether, this editor is a relatively large and complex software system. It can be expected that due to its completeness and its component structure, the editor design will be ‘robust’ for various forms of design evolution and efficiency improvements. The theory of correctness-preserving transformations on designs from Chapter 2 and Chapter 3 is applicable here: in particular, since the principle of black-box correctness has been adopted, many efficiency improvements can take the shape of black-box correctness preserving
glass-box modifications — meaning that there is a 'locality principle' which can yield a significant reduction of the verification task. It can be expected that this editor provides an excellent starting point for further optimisations and extensions. By way of example we mention two such extensions: to add multi-window features to the editor and to add a layer of dynamic keybinding and programmability.

We shall now explicitly point out two places in the entire documentation which can be considered as successful and elegant. These are the following:

- the formalisation of 'text', which lead to a rich collection of algebraic operations on texts and to the 'discovery' of elegant algebraic laws and of several important homomorphic mappings,
- the description of the buffer data-structure which we could describe completely formally together with its invariant properties (\( \text{gap1}(b) \leq \text{gap2}(b), \text{dot}(b) \not\in \text{gap}(b) \text{ etc.} \)) and several more subtle buffer properties such as \text{ready} and \text{space}.

They show how formal descriptions and efficiency considerations can go hand in hand. Note how the \text{ready} and \text{space} properties play a central role in the efficiency of dot-movements and text-modifications.

We must admit that the current documentation includes neither a user-manual nor diagrams showing some hierarchical decomposition of the editor; but this should not be viewed as a weak point of our approach. On the contrary, the formal specification of the editor provides an excellent starting point for writing a user manual — which is something different from the design documentation. The fact that the entire design is available as a formal text, makes it possible to have certain types of diagrams generated automatically, and indeed this has been done using van den Bos en van Ommering's graphical language POLAR. The reader is referred to [12] for the actual diagrams — which are quite helpful. These diagrams show how the various bits and pieces are put together, by means of a compact presentation-in-the-large of both \text{d-editor} and \text{d-mark}. This should be compared with the situation often encountered in current industrial practice, where the diagrams come instead of essential design documentation.

To conclude, let us once more explicitly recall the main achievement of the editor case study which is that it provides us with a large and realistic example of the notion of design developed in Chapter 2 and several applications of the theory concerning design transformations from Chapter 3.
Bibliography


[7] H. Partsch, From informal requirements to a running program: a case study in algebraic specification and transformational programming. Internal Report 87-7, Department of Informatics, Faculty of Science, University of Nijmegen.


Appendix A

A Lower Design Layer

A.1.1 Introduction

In this appendix we describe the development of a design $d_{basic}$ which provides for a few basic data types. In the context of the development of an editor, it can be viewed as a design-layer below the editor design. It provides for implementations of `SEQ` and `TABLE`. The 'interfaces' of $d_{basic}$ have been chosen such that its system fits into the primitive components of $d_{editor}$. In the terminology of Chapter 3 this can be stated more formally as: "the pair $(d_{editor}, d_{basic})$ is black-box valid".

A.1.2 The Top of the Lower Design Layer

The design $\text{top}(d_{basic})$, is shown below. At the position of the dots in this design we assume LET-constructs introducing the names BOOL_SPEC, NAT_SPEC, CHAR_SPEC etc. This top design contains all essential information for checking the (black-box) validation of $d_{basic}$ with respect to $d_{editor}$. Therefore, this top design should match the bottom design given in Section 5.3.6.

```
DESIGN

COMP BOOL     :  BOOL_SPEC;
COMP NAT      :  NAT_SPEC;
COMP CHAR     :  CHAR_SPEC;
COMP INST     :  INST_SPEC;
COMP ATTR     :  ATTR_SPEC;
COMP BLOCK    :  BLOCK_SPEC;
COMP DISPLAY  :  DISPLAY_SPEC;
```
A. A LOWER DESIGN LAYER

COMP FILE : FILE_SPEC;
COMP 'SEQ' : 'SEQ_SPEC';
COMP TABLE : TABLE_SPEC

SYSTEM BOOL, NAT, CHAR, INST, ATTR, 'SEQ', TABLE, BLOCK, DISPLAY, FILE

A.1.3 The Bottom of the Lower Design Layer

The design \(d_{\text{lower}}\), i.e. the bottom of our lower design layer is given below. At the position of the dots in the bottom design below we assume a number of \textit{LET} abbreviations, introducing the names BOOL_SPEC, NAT_SPEC, CHAR_SPEC etc. This bottom design also includes the components BLOCK, DISPLAY, and FILE which play no role in this design except for the fact that they are simply passed on to next higher design layer. This bottom does not include components 'SEQ', and TABLE because it is precisely the purpose of the design \(d_{\text{lower}}\) to provide implementations for these.

```
DESIGN
...
COMP BOOL : BOOL_SPEC;
COMP NAT : NAT_SPEC;
COMP CHAR : CHAR_SPEC;
COMP INST : INST_SPEC;
COMP ATTR : ATTR_SPEC;
COMP BLOCK : BLOCK_SPEC;
COMP DISPLAY : DISPLAY_SPEC;
COMP FILE : FILE_SPEC

SYSTEM NONE
```

A.1.4 Implementing the System Components

A.1.4.1 Introduction

We begin a simple development process and we start with the top design given in Section A.1.2. This yields the design consisting of the keyword DESIGN and a large number of \textit{LET} abbreviations including those for BOOL_SPEC, NAT_SPEC, CHAR_SPEC, INST_SPEC, ATTR_SPEC, BLOCK_SPEC, DISPLAY_SPEC,
FILE_SPEC, 'SEQ_SPEC', and TABLE_SPEC given before, followed by the following components:

```plaintext
COMP BOOL : BOOL_SPEC;
COMP NAT : NAT_SPEC;
COMP CHAR : CHAR_SPEC;
COMP INST : INST_SPEC;
COMP ATTR : ATTR_SPEC;

COMP BLOCK : BLOCK_SPEC;
COMP DISPLAY : DISPLAY_SPEC;
COMP FILE : FILE_SPEC;

(4) COMP 'SEQ' : 'SEQ_SPEC';
(4) COMP TABLE : TABLE_SPEC;
```

and having as a system BOOL, NAT, CHAR, INST, ATTR, 'SEQ', TABLE, BLOCK, DISPLAY, FILE. The black-box descriptions of this design remain unchanged from now on. Again the symbol (4) means “this is going to be replaced later”. Just as we did with 

\[ \text{deditor} \]

we have mixed the formal texts with informal descriptions such that we can present the various stages of the development process in an incremental fashion and such that there is a mechanical operation of extracting all formal texts to get one design which can be syntax- and type-checked. Trivially, the above initial design is black-box correct. For this simple design there will be no difference between a top-down and a bottom-up development process. Note that we just have to implement 'SEQ' and TABLE and these can be independent. When TABLE would use 'SEQ', or conversely, there would be a difference, but as it happens, this will not be the case.

A.1.4.2 Implementing Sequences

Recall the data type of implementable sequences whose specification was given already in Chapter 4. Below an implementation of these sequences is given. It is based on the idea of linked lists and we use an attribute-oriented approach. The application of the export operator below is not strictly required, but the reader may appreciate it here because the export signature serves as a summary of what is required by the specification.

LET 'SEQ_IMPL' :=

LAMBDA X : 'ITEM' OF
EXPORT
SORT Seq.
FRED seq_inv:
FUNC f : 'Seq' -> Seq.

SORT 'Seq',
SORT Nat,
SORT 'Item',
FUNC empty: -> 'Seq',
FPROC cons: 'Item' # 'Seq' -> 'Seq',
FUNC hd: 'Seq' -> 'Item',
FUNC tl: 'Seq' -> 'Seq',
FPROC eq: 'Seq' # 'Seq',
FUNC sel: 'Seq' # Nat -> 'Item',
FPROC cat: 'Seq' # 'Seq' -> 'Seq',
FPROC rev: 'Seq' -> 'Seq'

FROM

Next we shall introduce several local definitions. 'S_INST' introduces the
sort 'Seq' as a renamed version of Inst. SEQ_ITEM introduces sequences of
items. More precisely, it exports the sorts Item, Nat, Bag, Seq, the predicate
empty and the operations cons, hd, tl, len, sel, cat, rev and bag.

LET 'S_INST' :=
  RENAME
    SORT Inst TO 'Seq'
  IN INST;

LET SEQ_ITEM :=
  APPLY
    SEQ_SPEC
  TO 1;

'S_INST_ITEM' and 'S_INST_NEXT' serve for associating item and next
attributes to 'Seq' objects respectively. Let us analyse the structure of
'S_INST_ITEM' below in detail. It consists of a renamed version of ATTR
which is applied to two actual parameters, viz. 'S_INST' and X. This
renamed version of ATTR is a parameterised description which requires a
sort 'Seq' for its first parameter and a sort 'Item' for its second para-
meter; it exports the sorts 'Seq' and 'Item', the function item: 'Seq' ->
'Item' and the procedure set_item: 'Seq' # 'Item' -> . The definition
of 'S_INST_NEXT' is very similar to that of 'S_INST_ITEM'.

LET 'S_INST_ITEM' :=
5. DESIGN OF A TEXT EDITOR

APPLY APPLY
RENAME
SORT Inst TO 'Seq',
SORT Item TO 'Item',
FUNC attr : Inst -> Item TO item,
PROC set_attr: Inst # Item TO set_item
IN COPY(ATTR)
TO 'S_INST' TO X;

LET 'S_INST_NEXT' :=
APPLY APPLY
RENAME
SORT Inst TO 'Seq',
SORT Item TO 'Seq',
FUNC attr : Inst -> Item TO item,
PROC set_attr: Inst # Item TO set_item
IN COPY(ATTR)
TO 'S_INST' TO 'S_INST'.

% end of local definitions

IMPORT X INTO
IMPORT NAT INTO
IMPORT 'S_INST' INTO
IMPORT SEQ_ITEM INTO
IMPORT 'S_INST_ITEM' INTO
IMPORT 'S_INST_NEXT' INTO

CLASS

The invariant seq_inv presented below states that each 'Seq' object represents a defined sequence; in view of the given abstraction function f this implies that the graph of the next attribute is cycle-free.

PRED seq_inv:
DEF FORALL s:'Seq' ( f(s) != )

FUNC f: 'Seq' -> Seq
PAR n:'Seq'
DEF ( n = nil => empty
| NOT n = nil ?; cons(f(item(n)),f(next(n))) )

FUNC empty: -> 'Seq'
DEF nil
PROC cons: 'Item' @ 'Seq' -> 'Seq'
PAR c: 'Item', s: 'Seq'
DEF LET t: 'Seq'; t := create;
    set-item(t, c);
    set_next(t, s);
    t

FUNC hd: 'Seq' -> 'Item'
PAR o: 'Seq'
DEF item(o)

FUNC tl: 'Seq' -> 'Seq'
PAR s: 'Seq'
DEF next(s)

PRED eq: 'Seq' @ 'Seq'
PAR s: 'Seq', t: 'Seq'
DEF s = t (C)OR
    NOT (s = nil) AND NOT (t = nil)
    (C)AND eq(hd(s), hd(t)) (C)AND eq(tl(s), tl(t))

FUNC sel: 'Seq' @ Nat -> 'Item'
PAR s: 'Seq', n: Nat
DEF ( n = O ?; hd(s)
    | NOT n = O ?; sel(tl(s), pred(n)) )

PROC cat: 'Seq' @ 'Seq' -> 'Seq'
PAR s: 'Seq', t: 'Seq'
DEF ( s = nil ?; t
    | NOT s = nil ?; cons(hd(s), cat(tl(s), t)) )

% If both s and t are sequences, then revap(s, t)
% appends "the reverse of s" to t.

PROC revap: 'Seq' @ 'Seq' -> 'Seq'
PAR s: 'Seq', t: 'Seq'
DEF ( s = nil ?; t
    | NOT s = nil ?; revap(tl(s), cons(hd(s), t)) )

PROC rev: 'Seq' -> 'Seq'
PAR s: 'Seq'
DEF revap(s, empty)

END;

% COMP 'SEQ' : 'SEQ_SPEC' := 'SEQ_IMPL'
% this is to replace an earlier primitive component.

A.1.4.3 Implementing Tables

Below an implementation of tables is given. It is based on the use of binary trees. We replace the original Map attribute (function map: Table -> Map from TABLE_SPEC) by a Node attribute, where each Node object at its turn is attributed by 'Item1', Item2 attributes and two Node attributes. Our implementation should be compared with [6] Algorithm 4.52. We shall easily avoid Wirth's var parameters by using procedures with one or more result parameters. We considered Jonkers' four-step transformation technique [3] which would mean to (1) add new attributes and relate them to the old attributes, (2) add assignments to the new attributes, (3) use the new attributes and (4) to remove the original attributes. This would be a natural continuation of the attribute-oriented approach adopted already in the specification TABLE_SPEC. We did not employ this transformation technique however, because the last step of removing the original Map attribute could not be justified in terms of the formal implementation relation mentioned in Section 5.1 – although from a practical point of view it seems acceptable. Again the export operator below is not strictly needed.

LET TABLE_IMPL :=

LAMBDA X : 'SLO1' OF
LAMBDA Y : ITEM2 OF

EXPORT

SORT Map.
SORT Item1.
FUNC map: Table -> Map.
FUNC app: Map # Item1 -> Item2.

SORT Bool.
SORT Table.
SORT Item2.
SORT 'Item1'.
PREL table_inv.
A. A LOWER DESIGN LAYER

PROC new : Table # 'Item1' # 'Item2' -> Table;
PROC add : Table # 'Item1' # 'Item2' -> Table;
PROC rem : Table # 'Item1' -> Table;
PROC app : Table # 'Item1' # Table -> Table;
PROC is_in_dom : 'Item1' # Table -> Bool;

FROM

Now we get several local definitions. T_INST below introduces the sort Table.
MAP_FROM_SLO1_TO_ITEM2 introduces the sort Map of finite maps from Item1
to Item2. NODE_INST introduces the sort of Node of so-called nodes.
SET_OF_NODE introduces sets of these nodes — with associated set operations
ins, ins_in, union etc. We introduce SET_OF_NODE because when formulating
the representation invariant table_inv, we need to speak about the set of
nodes in a (sub)tree. NODE_SLO1, NODE_ITEM2, NODE_LEFT, and NODE_RIGHT
associate various attributes with these nodes and as a result, every node has
the following attributes: item1, item2, left and right. Finally TABLE_ROOT
associates a so-called root-node with every table.

LET T_INST :=
  RENAME
    SORT Inst TO Table
  IN INST;

LET MAP_FROM_SLO1_TO_ITEM2 :=
  APPLY APPLY
    MAP_SPEC
    TO X TO Y;

LET NODE_INST :=
  RENAME
    SORT Inst TO Node
  IN INST;

LET SET_OF_NODE :=
  APPLY
    RENAME
      SORT Item TO Node,
      SORT Set TO NodeSet
    IN SET_SPEC
    TO NODE_INST;

LET NODE_SLO1 :=
  APPLY APPLY
RENAME
SORT Inst     TO Node,
SORT Item     TO 'Item1',
FUNC attr : Inst -> Item TO item1,
PROC set_attr: Inst # Item -> TO set_item1
IN COPY(ATTR)
TO NODE_INST TO X;

LET NODE_ITEM2 :=
APPLY APPLY
RENAME
SORT Inst     TO Node,
SORT Item     TO Item2,
FUNC attr : Inst -> Item TO item2,
PROC set_attr: Inst # Item -> TO set_item2
IN COPY(ATTR)
TO NODE_INST TO Y;

LET NODE_LEFT :=
APPLY APPLY
RENAME
SORT Inst     TO Node,
SORT Item     TO Node,
FUNC attr : Inst -> Item TO left,
PROC set_attr: Inst # Item -> TO set_left
IN COPY(ATTR)
TO NODE_INST TO NODE_INST;

LET NODE_RIGHT :=
APPLY APPLY
RENAME
SORT Inst     TO Node,
SORT Item     TO Node,
FUNC attr : Inst -> Item TO right,
PROC set_attr: Inst # Item -> TO set_right
IN COPY(ATTR)
TO NODE_INST TO NODE_INST;

LET TABLE_ROOT :=
APPLY APPLY
RENAME
SORT Inst     TO Table,
SORT Item     TO Node,
FUNC attr : Inst -> Item TO root,
PROC set_attr: Inst # Item -> TO set_root
IN COPY(ATR)
TO T_INST TO NODE_INST;

% end of local definitions
IMPORT X INTO
IMPORT Y INTO
IMPORT BOOL INTO
IMPORT T_INST INTO
IMPORT MAP_FROM_SLO1_TO_ITEM2 INTO
IMPORT NODE_INST INTO
IMPORT SET_OF_NODE INTO
IMPORT NODE_SLO1 INTO
IMPORT NODE_ITEM2 INTO
IMPORT NODE_LEFT INTO
IMPORT NODE_RIGHT INTO
IMPORT TABLE_ROOT INTO
CLASS

FUNC nodes: Table -> Nodeset
PAR t:Table
DEF nodes(root(t))

FUNC nodes: Node -> Nodeset
PAR n:Node
DEF ( n = nil ?; empty
| NOT m = nil ?; ins(m,union(nodes(left(n)),nodes(right(n))))
)

PRED table_inv: DEF
{1} FORALL t:Table,u:Table
( NOT t = u => intersect(nodes(t),nodes(u)) = empty )

{2} FORALL n:Node,m:Node
( NOT m = nil =>
  ( is_in(m,nodes(left(n))) => less(item1(m),item1(n));
  is_in(m,nodes(right(n))) => less(item1(n),item1(m)) ) )

FUNC map: Table -> Map
PAR t:Table
DEF SOME p:Map
( FORALL i:Item1, j:Item2
  ( app(p,i) = j <=>
    EXISTS n:Node
    ( nodes(root(n)) = nodes(t) ) )
  )


( in_in(n, nodes(root(t)));
  f(item1(n)) = i;
  item2(n) = j ) ) )

Now we can implement the procedure new which was specified by LET  t:Table;
  t := create; set_map(t, empty); t. We preserve the sequential structure of
this definition, but the assignment set_map(t, empty) is replaced by an
assignment of nil to the root-node attribute. Because we defined the map
attribute in terms of the root-node attribute, this replacement is justified when
we can show that assigning nil to the root-node makes the map attribute
empty indeed. The map of a table is the map of its root-node and because
nodes(nil) is the empty set, we get (by definition of map: Node -> Map) a
map p with the property \( \forall i, j \left( \text{app}(p, f(i)) = j \Leftrightarrow \exists n : \text{Node} \left( n \in \emptyset \land \ldots \right) \right) \). This is the empty map.

PROC new: -> Table
DEF LET t:Table; t := create;
  set_root(t, nil);
  t

The remaining procedures are dealt with in a very similar way. Some of
these were specified by a case-analysis on the invariant table_inv, such that
here we need not worry about what happens when the invariant does not
hold: an arbitrary modification of all attributes is allowed in that case. In
the other case, the specification prescribes that a certain assignment to the
map attribute must take place. E.g. for add(t, i, j) this is
set_map(t, add(map(t), f(i), j)). Again this is established by an assign-
ment to the root-node.

PROC add: Table # 'Item1' # 'Item2' ->
PAR t:Table, i:'Item1', j:'Item2
DEF set_root(t, add(root(t), i, j))

PROC add: Node # 'Item1' # 'Item2' -> Node
PAR n:Node, i:'Item1', j:'Item2
DEF ( n = nil7:
    LET m:Node; m := create;
    set_item1(m, i);
    set_item2(m, j);
    set_left(m, nil);
    set_right(m, nil);
    m
  ) NOT n = nil AND eq(i, item1(n)) ?;
    set_item2(n, j);
A. A LOWER DESIGN LAYER

\[\text{PROC } \text{rem: Table } \# 'Item1' \to \text{ Table, i:'Item1'} \]
\[\text{DEF set_root(t,delete(root(t),i))}\]

\[\text{PROC delete: Node } \# 'Item1' \to \text{ Node}\]
\[\text{PAR n:Node, i:'Item1'} \]
\[\text{DEF ( n = nil ?; }\]
\[\text{ n = nil AND less(i, item(n)) ?; set_left(c,delete(left(n),i)); n} \]
\[\text{ n = nil AND less(item(n), i) ?; set_right(n,delete(right(n),i)); n} \]
\[\text{ n = nil AND eq(1, item(n)) ?;}\]
\[\text{ ( left(n) = nil ?; \text{right(n)} }\]
\[\text{ right(n) = nil ?; \text{left(n)} }\]
\[\text{ NOT left(n) = nil AND NOT right(n) = nil ?; LET m:Node, j:'Item1', k:Item2; m,j,k := del(left(n)); set_left(n,m); set_item1(n,j); set_item2(n,k); n} \]
\]

We used an auxiliary function del to delete the rightmost node in a tree, i.e. the one with the largest item value; its result consists of (1) the modified tree and (2) the item1, item2 values of the deleted node. Somewhat more formally, del(n) could be specified as follows LET i:'Item1', j:Item2; i,j := rightmost(n); set_map(n,rem(map(n),f(i))); n,i,j. The function rightmost is defined recursively below.
FNC rightmost: Node -> 'Item1' # Item2
PAR m:Node
DEF NOT n = nil ?:
  ( right(n) = nil ?; item1(n), item2(n)
  | NOT right(n) - nil ?; rightmost(right(n)) )

PROC del: Node -> Node # 'Item1' # Item2
PAR n:Node
DEF ( NOT right(n) = nil ?:
  LET m:Node,i:'Item1',j:Item2;
  m,i,j := del(right(n));
  set_right(n,m);
  n,i,j
  | right(n) = nil ?;
  left(n), item1(n), item2(n)
)

PROC app: Node # 'Item1' -> Item2
PAR n:Node,i:'Item1'
DEF ( eq(i,item1(n)) ?;
  item2(n)
  | less(i,item1(n)) ?;
  app(left(n),i)
  | less(item1(n),i) ?;
  app(right(n),i) )

PROC app: Table # 'Item1' -> Item2
PAR t:Table,i:'Item1'
DEF app(root(t),i)

PROC is_in_dom: 'Item1' # Node -> Bool
PAR i:'Item1',n:Node
DEF ( n = nil ?;
  false
  | NOT n = nil AND eq(i,item1(n)) ?;
  true
  | NOT n = nil AND less(i,item1(n)) ?;
  is_in_dom(i,left(n))
  | NOT n = nil AND less(item1(n),i) ?;
  is_in_dom(i,right(n))
)
A. A LOWER DESIGN LAYER

PROC is_in_dom: 'Item' # Table -> Bool
PAR i:'Item', t:Table
DEF is_in_dom(i, root(t))

END

The above implementation contains several functions and predicates which need not be executable but which are needed for reasoning purposes: both nodes and both map functions, rightmost and of course table_inv.

% COMP TABLE : TABLE_SPEC := TABLE_TMPL
% this is to replace an earlier primitive component.

The structure of the lower design layer \( d_{\text{low}} \) is given by its initial design (Section A.1.4.1) and the two replacements discussed in Sections A.1.4.2 and A.1.4.3. We conclude by mentioning its system.

SYSTEM BOOL, NAT, CHAR, INST, ATTR, 'SEQ', TABLE, BLOCK, DISPLAY, FILE
Appendix B

List of Symbols

In this appendix we give a list of the sorts, functions, predicates and procedures used. The list does not include symbols from the C program of the editor, nor does it include the symbols introduced already in Chapter 4. The symbols from Appendix A are also not mentioned unless they are introduced in Section 5.3. If a symbol occurs both in a black-box description and in a glass-box description, then it is mentioned only once and its place in the list is derived from its black-box description.

For each symbol the list contains a very short informal description. The list has been subdivided into a number of sub-lists. The first sub-list contains the symbols that are introduced in Section 5.3. The second sub-list contains the symbols that are introduced in Section 5.5. The third sub-list contains the symbols that are introduced in Section 5.6.

Symbols concerning the bottom of the editor design

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst.</td>
<td>instances (atoms)</td>
</tr>
<tr>
<td>nil:</td>
<td>instance constant</td>
</tr>
<tr>
<td>PROC create: → Inst</td>
<td>creation of a new instance</td>
</tr>
<tr>
<td>attr: Inst → Item,</td>
<td>attribute</td>
</tr>
<tr>
<td>PROC set_attr: Inst # Item →</td>
<td>attribute modification</td>
</tr>
<tr>
<td>Table.</td>
<td>modifiable tables</td>
</tr>
<tr>
<td>table_inv: ,</td>
<td>table invariant</td>
</tr>
<tr>
<td>map: Table → Map,</td>
<td>Map attribute of table</td>
</tr>
<tr>
<td>PROC set_attr: Table # Map →</td>
<td>modification of Map attribute</td>
</tr>
<tr>
<td>PROC p: →</td>
<td>arbitrary table operation</td>
</tr>
<tr>
<td>PROC new: → Table</td>
<td>empty table creation</td>
</tr>
</tbody>
</table>
B. LIST OF SYMBOLS

PROC add: Table # 'Item1' # Item2 →
PROC rem: Table # 'Item1' →
PROC app: Table # 'Item1' → Item2
PROC is_in_domain: 'Item1' # Table → Bool

Block,
size: Block → Nat,
cont: Block # Nat → Item,
PROC store: Block # Nat # Item →,
PROC alloc: Nat → Block,
PROC grow: Block # Nat →

attr: Inst → Item1 # Item2
PROC set_attr: Inst # Item1 # Item2 →

Symbols concerning system components

Buf
block: Buf → Block
PROC set_block: Buf # Block →
dot: Buf → Nat
PROC set_dot: Buf # Nat →
mark: Buf → Nat
PROC set_mark: Buf # Nat →
gap1: Buf → Nat
PROC set_gap1: Buf # Nat →
gap2: Buf → Nat
PROC set_gap2: Buf # Nat →
dot: Buf → Nat2
PROC set_dot: Buf # Nat2 →
mark: Buf → Nat2
PROC set_mark: Buf # Nat2 →
reach: Buf → Nat2
PROC set_reach: Buf # Nat2 →

val: → Item,
PROC upd: Item →

Buf_Map
table: → Table
PROC upd_table: Table →
current: → 'String'
PROC upd_current: 'String' →
mover: → Nat
PROC set_mover: Nat →

addition of an entry
removal of an entry
table look-up
'is in domain' test
memory blocks
number of locations
contents of a location
storing a value in a location
creation of a new block
increment of block size
pair-wise attribute
pair-wise modification

(value) buffers
block attribute
block attribute modification
dot attribute
dot attribute modification
mark attribute
mark attribute modification
lower bound of gap
modification of lower bound
upper bound of gap
modification of upper bound
pair-wise dot attribute
modification of pair-wise dot
mark attribute
modification of pair-wise mark
reach attribute
modification of reach attribute

value of programming variable assignment
maps from strings to buffers
name of current buffer assignment
general purpose 'loop-counter' assignment
mover1: -> Nat
PROC set_mover1: Nat -> mover1
mover2: -> Nat
PROC set_mover2: Nat -> mover2
mover: -> Nat
PROC set_mover: Nat -> mover
counter: -> Nat
PROC set_counter: Nat ->

f: Block # Nat^2 -> String
in_gap: Buf # Nat
f: Buf # Nat^2 -> String
f: Buf # Nat -> String
f: Buf -> String
f: Block # Nat^2 -> Text
f: Buf # Nat^2 -> Text
f: Buf # Nat -> Text
f: Buf -> Text
f: Buf -> MText
f: Buf_Map -> Map

right: Buf # Nat -> Nat
eobp: Buf # Nat
eolp: Buf # Nat
eobp: Buf
eolp: Buf
left: Buf # Nat -> Nat
bobp: Buf # Nat
bobp: Buf
bolp: Buf # Nat
bolp: Buf

WI':
wi_package_inv:
PROC init_wi_package: ->
PROC mod_text_restore: ->
PROC mod_dot_restore: ->
PROC mod_dot: ->

ready: Buf
PROC make_ready: Buf ->
PROC make_ready!: Buf ->

mover (vertical co-ordinate) assignment
mover (horizontal co-ordinate) assignment
the pair (mover1,mover2) pair-wise assignment
general purpose loop-counter assignment

abstraction function
test if position is in gap
abstraction function
abstraction function
abstraction function
abstraction function
abstraction function
abstraction function
abstraction function
abstraction function
abstraction function
stretched text-invariant

next position (rightwards)
end-of-buffer predicate
end-of-line predicate
dot is at end-of-buffer
dot is at end-of-line
next position (leftwards)
beginning-of-buffer predicate
dot is at beginning-of-buffer
beginning-of-line predicate
dot is at beginning-of-line

stretched window-invariant
invariant of wi_package
initialisation procedure
restore WI (text modified)
restore WI (dot modified)
 arbitrary dot modification

buffer is ready for insert
make buffer ready
auxiliary for make_ready
B. LIST OF SYMBOLS

PROC make_ready2: Buf →
PROC gap_down: Buf →
PROC gap_up: Buf →
PROC right_dot: Buf →
PROC left_dot: Buf →
next_line: Buf → Bool # Nat
right_stepping: Buf # Nat# → Bool # Nat
hpos: Buf → Nat
end_of_line: Buf # Nat → Nat#
previous_line: Buf → Bool # Nat
beginning_of_line: Buf # Nat → Nat
end_of_buffer: Buf → Nat

space: Buf
PROC make_space: Buf # Nat →
PROC second_gap_down: Buf # Nat →
PROC insert_character: Buf # Char →
PROC newline: Buf →
match: Buf # 'String' # Nat
match: Buf # 'String' # Nat
right: Buf # Nat# → Nat#
buffer_to_string: Buf # Nat → 'String'

Symbols concerning internal components

Array2,
PROC create: → Array2,
val : Array2 # Nat# → Item,
PROC upd : Array2 # Nat# # Item →
cursor1: → Nat
PROC set_cursor1: Nat →
cursor2: → Nat
PROC set_cursor2: Nat →
cursor: → Nat#
PROC set_cursor: Nat# →
cscreen: → Array2
PROC upd_cscreen: Array2 →
g: Array2 # Nat# → Line
g: Array2 # Nat → Line
f: Array2 # Nat → Text
f: Array2 → Text

display_handling_inv:
PROC init_display_handling: →

auxiliary for make_ready
move gap one position down
move gap one position up
move dot one position right
move dot one position left
auxiliary for next_line
going rightwards (if possible)
horizontal position
auxiliary for end_of_line
auxiliary for previous_line
auxiliary for beginning_of_line
auxiliary for end_of_buffer
test for free space
introduction of more free space
move free space down
auxiliary for insert_character
auxiliary for newline
search-string matching
match with sentinel
pair-wise rightward move
aux. buffer_to_string
two-dimensional arrays
array creation
indexing in an array
assignment
concept cursor (vertical)
assignment
concept cursor (horizontal)
assignment
concept cursor
assignment
concept screen
assignment
abstraction function
abstraction function
abstraction function
abstraction function
transformed window-invariant
display handling invariant
initialisation procedure
PROC update_cursor: →
PROC update_screen: →
origin: Buf → Nat # Nat
PROC set_origin: Buf # Nat # Nat →
xx: → Nat
PROC upd_xx: Nat →
yy: → Nat
PROC upd_yy: Nat →
PROC clear_cscreen: →
v.eq: Nat² # Nat²
PROC never_to_origin_line: Buf →
v.geq: Nat² # Nat²
h.geq: Nat² # Nat²
b_line: Nat² # Nat²
built: Buf
size: → Nat²
in_window: Buf
PROC cscreen_build: Buf →
p_lines: Nat² # Nat²
p_leq: Nat² # Nat²
p_nb: Nat² # Nat² → Nat²
p_add: Nat² # Nat² → Nat²
r3: Buf
half_li: → Nat
half_co: → Nat

ii: → Nat
PROC upd_ii: Nat →
jj: → Nat
PROC upd_jj: Nat →
scursor1: → Nat
scursor2: → Nat
scursor: → Nat²
PROC set_scursor1: Nat →
PROC set_scursor2: Nat →
PROC set_scursor: Nat² →
screen: → Array2
PROC upd_cscreen: Array2 →
PROC clear_cscreen: →
PROC update_line: →
PROC update_character: →
PROC sce: →
PROC sce: Nat² →

transfer of concept to display
transfer of concept to display
leftmost uppermost corner
assignment
simple programming variable
assignment
simple programming variable
assignment
fill concept screen with blanks
equality on vertical co-ordinate
put mover at line with origin
comparison (vertical)
comparison (horizontal)
comparison (horizontal)
build-up of cscreen done
the pair (i,j)

if position is in window
build-up concept screen
comparison of co-ordinate pairs
comparison of co-ordinate pairs
comparison of co-ordinate pairs
addition of co-ordinate pairs

necessary condition
li divided by two
cj divided by two

simple programming variable
assignment
simple programming variable
assignment
shadow cursor (vertical)
shadow cursor (horizontal)
shadow cursor
assignment
assignment
shadow administration screen
assignment
fill shadow screen with blanks
update line on screen
update character on screen

'shadow' clear to end of line
'shadow' clear to end of line
Appendix C

The C Program of the Editor

After the development processes of the editor and the lower design layer have been finished, the resulting glass-box descriptions have been translated into the C programming language [8] manually. Let us devote a few pages of explanation to this translation. First we explain the translation of the data structures. We assumed the data types of BOOL, NAT, and CHAR as built-in into C, and we dealt with them by a few simple macro definitions. E.g. for BOOL we only needed the following.

```c
#define Bool int
#define true 1
#define false 0
```

We employed a straightforward technique for dealing with the attribute-oriented approach based on Inst and Attr. This was done by collecting all attributes of a given object sort and declare that sort as a pointer to a structure. The latter structure contains one field for each attribute function. By introducing macro definitions – one for each attribute function – it becomes possible to use the same functional notation as in COLD-X again. We show this for the object sort Node (from Appendix A) having four attribute functions: Item1: Node → Item1, Item2: Node → Item2 etc.

```c
#define Node struct attr Node *
struct attr Node {
    Item1 attr_item1;
    Item2 attr_item2;
    Node attr_left;
    Node attr_right;
};
```
Next we explain the translation of the algorithms. Whenever possible we sticking to a straightforward line-by-line translation, although we did no attempt to devise a completely standardised translation scheme yet. We show a simple example and for that purpose we recall the procedure make_space from WITEFA.IMPL:

```c
PROC make_space: Buf # Nat ->
PAR b:Buf, n:Nat
DEF set_mover(size(block(b)));
grow(block(b),n);

( NOT mover = gap2(b) ?;
  second_gap_down(b,n)
 ) *; mover = gap2(b) ?;
set_gap2(b,add(gap2(b),n));
set_dot(b,gap2(b));

( lsw(mark(b),gap1(b)) ?; SKIP
| NOT ism(mark(b),gap1(b)) ?; set_mark(b,add(mark(b),n)) )
```

We show its C translation below. It refers to obvious macros NOT and SKIP. Recall that in C the symbol = denotes assignment whereas == denotes equality!

```c
void make_space(b,n)
Buf b;
Nat n:
{
  mover = (size(block(b)));
grow(block(b),n);
  while ( NOT (mover == gap2(b)) ) {
    second_gap_down(b,n);
  }
  gap2(b) = add(gap2(b),n);
dot(b) = gap2(b);
```
C. The C PROGRAM OF THE EDITOR

    if ( last(mark(b), gap(b)) ) {
        SKIP
    } else {
        mark(b) = add(mark(b), n);
    }

The above example is rather unproblematic. At other places we faced technical complications such as operations yielding composite results (e.g., next_line: Buf -> Bool # Nat and right_stepping: Buf # Nat -> Bool # Nat from WITEFA_IMPL). For the latter complication we employed two different solutions, depending on the question whether the operation is defined recursively — which is not the case for next_line but which is the case for right_stepping. For a non-recursive operation we made several copies of its direct translation, viz. one for each element of the result. For a recursive operation, this technique of making copies could lead to gross inefficiencies so we had to simulate a call-by-reference mechanism using the C operators * (indirection) and & (address-of). Other complications are related to (1) name clashes due to overloading, (2) using macros for efficiency reasons, (3) COLD-K renamings and (4) modularisation of the C program text. Although it might be somewhat more work to devise universal solutions for these problems, we easily solved them for the particular cases at hand.

We had to provide small C modules providing implementations of BLOCK, DISPLAY and FILE. These components have no COLD-K glass-box descriptions and the C modules should be viewed as reasonable approximations of BLOCK_SPEC, DISPLAY_SPEC and FILE_SPEC. For BLOCK we had to use the C library functions malloc and realloc [9]. For DISPLAY we adopted the command sequences of a VT102 terminal [10], which can also be emulated on SUN 3/80 or Philips P2000-C computer systems. For FILE we had to use the so-called standard I/O library as partially described in [8] and which becomes available by inclusion of the line

    #include <stdio.h>

For each instance of SVAB we declared a simple programming variable in C. We by-passed the implementation of ARRAY2, just using C arrays. We inserted a flush(stdout); instruction in the translation of write_named_file. We added a top-level initialisation and main-loop to connect the key procedure with the standard input. This is done by means of the C function called main which has a local variable c of sort Char and which has the program text given below. The user must type a control-c character to terminate an
edit session — whence the termination condition of the loop, \( \text{ord}(c) \neq 3 \).

```c
MOREDOF_startUp();
c = getchar();
while (ord(c) != 3) {
    key(c);
    c = getchar();
}
```

The translation process took several days, but the time needed to do this was only a small fraction of the total time spent on the editor (5%, say). In the C program texts, we inserted small comments related to the translation; we did not insert comments to explain the program as such, because after all we have the designs \( \text{d} \text{editor} \) and \( \text{d} \text{man} \) for that. Although the manual translation process introduced several small mistakes, we did not spend much time in debugging. Of course an automatic code-generator would be useful, but the current manual approach was already very satisfactory. The choice of the target-language (e.g. C versus Pascal) is a relatively unimportant detail of the editor case study and in fact the choice for C was only made when most of the design work was already done. We could have chosen Pascal as well, although in retrospect, we learned to appreciate several pragmatic issues related to C such as the \text{cpp} macro preprocessor and the powerful \text{malloc}/\text{realloc} memory-management functions. The decisive point in favor of C was its growing popularity in industrial contexts.

The complete C program texts are provided in [12]. Since the above discussion contains already an overview of most relevant technical issues, we do not include all C program texts here; it is always possible to look-up details in [12]. Compilation of these C program texts by \text{cc} -O yields an executable editor. Just by way of example, we include the C program text of one component and we have chosen to show the translation of \text{TABLE_IMPL}. 
C. The C PROGRAM OF THE EDITOR

/* table_impl.c */

#define guard(N) if (!(N)) { printf("FATAL ERROR (guard false)\n"); }  
#define Nat int

/* TABLE_IMPL */
'Item' --> Item1,  
eq: 'Item1' # 'Item1' --> eq_item1,  
add: Table # 'Item1' # 'Item2' --> add_table,  
add: Node # 'Item1' # 'Item2' --> add_node,
*/

#define Node struct attr_node *
struct attr_node {  
    Item1 attr_item1;  
    Item2 attr_item2;  
    Node attr_left;  
    Node attr_right;  
};
#define item1(N) ((N)->attr_item1)  
#define item2(N) ((N)->attr_item2)  
#define left(N) ((N)->attr_left)  
#define right(N) ((N)->attr_right)  
Node create_node() {  
    char *malloc(); 
    return((Node)malloc(sizeof(struct attr_node)));  
}

#define Table struct attr_table *
struct attr_table {  
    Node attr_root;  
};
#define root(N) ((N)->attr_root)  
Table create_table() {  
    char *malloc(); 
    return((Table)malloc(sizeof(struct attr_table)));  
}

Table new()
{
    Table t;
    t = create_table();  
    root(t) = nil;
return(t);
}

Node add_Node(n,i,j)
Node m;
Item1 i;
Item2 j;
{
    Node m;

    if (n == nil) {
        m = create_Node();
        item1(m) = i;
        item2(m) = j;
        left(n) = nil;
        right(n) = nil;
        return(m);
    }
    else if (eq_Item1(i,item1(n))) {
        item2(n) = j;
        return(n);
    }
    else if (less(i,item1(n))) {
        left(n) = add_Node(left(n),i,j);
        return(n);
    }
    else if (less(item1(n),i)) {
        right(n) = add_Node(right(n),i,j);
        return(n);
    }
}

void add_Table(t,i,j)
Table t;
Item1 i;
Item2 j;
{
    root(t) = add_Node(root(t),i,j);
}

de1(n,return1,return2,return3)
Node n;
Node *return1;
Item1 *return2;
Item2 *return3;
\{ 
   if (right(n) != nil) { 
      Node m; Item i; Item2 j; 
      del(right(n), &m, &i, &j); 
      right(n) = m; 
      *return1 = n; *return2 = i; *return3 = j; 
      return; 
   } 
   else if (right(n) == nil) { 
      *return1 = left(n); 
      *return2 = item1(n); 
      *return3 = item2(n); 
      return; 
   } 
\}

Node delete(n, i)
Node n;
Item i;
\{ 
   if (n == nil) { 
      return(n); 
   } 
   else if (less(i, item1(n))) { 
      left(n) = delete(left(n), i); 
      return(n); 
   } 
   else if (less(item1(n), i)) { 
      right(n) = delete(right(n), i); 
      return(n); 
   } 
   else if (eq_Item(i, item1(n))) { 
      if (left(n) == nil) { 
         return(right(n)); 
      } 
      else if (right(n) == nil) { 
         return(left(n)); 
      } 
      else if (left(n) != nil AND right(n) != nil) { 
         Node m; Item j; Item2 k; 
         del(left(n), &m, &j, &k); 
         left(n) = m; item1(n) = j; item2(n) = k; 
         return(n); 
      } 
   } 
\}
} } 

void rem(t,i) 
Table t; 
Item1 i; 
{ 
    root(t) = delete(root(t),i); 
} 

rightmost(n,return1,return2) 
Node n; 
Item1 *return1; 
Item2 *return2; 
{ 
    guard(n != nil); 
    if (right(n) == nil) { 
        *return1 = item1(n); *return2 = item2(n); 
        return; 
    } else if (right(n) != nil) { 
        rightmost(right(n),return1,return2); 
        return; 
    } 
} 

Item2 app_Node(n,i) 
Node n; 
Item1 i; 
{ 
    if (eq_Item1(i,item1(n))) { 
        return(item2(n)); 
    } 
    else if (less(i,item1(n))) { 
        return(app_Node(left(n),i)); 
    } 
    else if (less(item1(n),i)) { 
        return(app_Node(right(n),i)); 
    } 
} 

Item2 app(t,i) 
Table t;
C. The C PROGRAM OF THE EDITOR

Item1 i;
{ return(app_Node(root(t),i)) ; }

Bool is_in_dom(i,n)
Item1 i;
Node n;
{ if (n == nil) {
    return(false);
}
else if (eq_Item(i,item(n))) {
    return(true);
}
else if (less(i,item(n))) {
    return(is_in_dom(i, left(n)));
}
else if (less(item(n),i)) {
    return(is_in_dom(i, right(n)));
}
}

Bool is_in_dom(i,t)
Item1 i;
Table t;
{ return(is_in_dom(i, root(t))); }
Appendix D: Reference Chart

For each component of $d_{edit}$ its most important sorts and operations are given below.

**DESIGN**

COMP BOOL : BOOL_SPEC; ( Bool, true, false, not, and, or
COMP NAT : NAT_SPEC; ( Nat, zero, succ, isz, leq, add, sub, 0, 1, etc.
COMP CHAR : CHAR_SPEC; ( Char, ord, chr, 'a', 'b', etc.
COMP INST : INST_SPEC; ( Inst, add, create
COMP ATTR : ATTR_SPEC; ( attr, set attr
COMP 'SEQ' : 'SEQ_SPEC'; ( 'Seq', empty, cons, hd, tl, eq, sel, car, rev
COMP TABLE : TABLE_SPEC; ( Table, new, add, rem, app, isin, dom
COMP BLOCK : BLOCK_SPEC; ( Block, size, cons, elem, allow, grow
COMP DISPLAY : DISPLAY_SPEC; ( li, co, cr, ni, bc, ce, cl, nd, up, con, print
COMP FILE : FILE_SPEC; ( valid, file, pos, rewrite, reset, read, write, oof
COMP ARRAY2 : ARRAY2_SPEC := ARRAY2_IMPL;
   ( Array2, create, val, upd
COMP 'STRING' : 'STRING_SPEC' := 'STRING_IMPL';
   ( 'String', empty, cons, hd, tl, eq, sel, car, rev, len
COMP SVAR : SVAR_SPEC := SVAR_IMPL;
   ( val, upd
COMP ATTR2 : ATTR2_SPEC := ATTR2_IMPL;
   ( attr, set attr
COMP DISPLAY_HANDLING : DISPLAY_HANDLING_SPEC := DISPLAY_HANDLING_IMPL;
   ( init, display, handling, update, cursor, update, screen,
COMP W1_PACKAGE : W1_PACKAGE_SPEC := W1_PACKAGE_IMPL;
   ( init, w1, packages, mod, text, restore, mod, dot, restore
COMP WITEFA : WITEFA_SPEC := WITEFA_IMPL;
   ( bolp, colp, forward, character, backward, character, next, line, previous, line, beginning, of, line, end, of, line, beginning, of, buffer, end, of, buffer, set, mark, exchange, dot, and, mark, insert, file, insert, character, newline, yank, buffer, delete, next, character, erase, region, erase, buffer, copy, region, to, buffer, current, buffer, name, write, named, file, switch, to, buffer, search, forward, buffer, to, string, buffer, to, string
COMP MOREDOP : MOREDOP_SPEC := MOREDOP_IMPL;
   ( mini, main, kill, startup, loops, return, delete, to, kill, buffer, yank, from, kill, buffer, search, forward, insert, file, write, named, file, delete, previous, character
COMP KEYBIND : KEYBIND_SPEC := KEYBIND_IMPL
   ( key

SYSTEM WITEFA, MOREDOP, KEYBIND
Curriculum vitae

Samenvatting

Het onderzoek dat in het proefschrift beschreven wordt, heeft formalisering van ontwerpmethoden voor complexe systemen als onderwerp. Het bestaat uit twee delen: een eerste deel met een theoretiisch karakter en een tweede deel waarin een aantal van de resultaten van het eerste deel aan een complex ontwerpprobleem worden getoetst.

Centraal in het eerste deel staat het begrip ‘ontwerp van een systeem’. In het algemeen is een systeem opgebouwd uit modules, en bestaan er relaties tussen die modules onderling (meestal in de vorm van een hiërarchische opbouw) en tussen modules en specificaties daarvan – een relatie die wij implementatie-relatie noemen. Daarbij staan wij binnen één ontwerp toe dat modules en hun specificaties naast elkaar bestaan met een verschillende graad van detaillering. Het begrip ontwerp (design) blijkt gefundeerd te kunnen worden op een speciale versie van λ-calculus, de λσ-calculus, die in het proefschrift wordt ontwikkeld. Op basis daarvan is het tevens mogelijk diverse methodologisch inzichtelijke correctheidsbegrippen voor ontwerpen te introduceren. Deze begrippen, die betrekking hebben op ‘information hiding’, worden op deze wijze van een wiskundig-logische grondslag voorzien. Aangetoond wordt dat er verbanden bestaan tussen correctheidsbegrippen voor ontwerpen enerzijds en reductiestrategieën voor λσ-calculus anderzijds.

Uitgaande van het formele ontwerpbegrip kunnen correctheidsbehoudende transformaties op ontwerpen gedefinieerd worden. Een belangrijke rol speelt het feit dat ontwerpen in het algemeen samengesteld worden uit componenten en dat, onder zekere voorwaarden, correctheid van het geheel uit die van de delen kan worden afgeleid. Diverse ontwerpmethodieën, waaronder de welbekende ‘top-down’ methode, blijken strikt formeel gekarakteriseerd te kunnen worden met de in het proefschrift geïntroduceerde begrippen. Met behulp van deze aanpak kan een zeer duidelijk onderscheid gemaakt worden tussen de statische en de dynamische aspecten van het proces van software-ontwikkeling, hetgeen bijdraagt tot een beter inzicht in het proces van software-ontwikkeling.

Het tweede gedeelte is een case-study. Het omvat de specificatie en een daaruit met de top-down methode ontwikkelde realistische implementatie van een tekst-verwerker. Beide zijn uitgevoerd met behulp van de ontwerptaal COLD die mede op de resultaten uit het eerste deel is gebaseerd. Bij de implementatie van de tekst-verwerker worden enerzijds de abstracte specificaties getransformeerd tot algoritmen en worden anderzijds de data-typen vervijld door het kiezen van geschikte representaties via meerdere nivo's van
verfijning. Deze studie laat zien dat de gebruikte technieken leiden tot een systematische aanpak, een goed gestructureerd en tevens efficiënt product en een redelijk toegankelijke documentatie. Tevens toont ze aan dat er nog mogelijkheden te over zijn voor verdere uitbouwing van theorie, gereedschappen en door toepassing verkregen ervaringen.
STELLINGEN

behorende bij het proefschrift

A FORMALISATION OF DESIGN METHODS
A $\lambda$-calculus Approach to System Design
with an Application to Text Editing

van

L.M.G. Feijs

Eindhoven,
23 februari 1990.
1. Het idee van Nami-Schneiderman diagrammen, n.l. dat doostsche doodge tekeningen gebruikt kunnen worden om met ‘flow-of-control’ operaties opgebouwde programma’s weer te geven, kan mutatis mutandis van nut zijn om met ‘module-composition’ operaties opgebouwde modules als tekeningen weer te geven.


2. De met INGRES geassocieerde vraagtaal QUEL kan geëxporteerd worden met technieken uit de denotationele semantiek, waarbij elke deel-betekenisfunctie een extra argument heeft, te weten een bedeling van tuples aan tuple-enheden. In het bijzonder, voor een relatie geïntroduceerd met create A (naam = string, adres = string) en een tuple-variabele geïntroduceerd met range of l is A, kan de semantiek van een vraag als

\[ \text{[retrieve} (t, \text{naam}) \text{where not t.adres} = \text{‘geldrop’}] \]

op compositionele wijze beschreven worden door

\[ \{ \forall t \in \text{string} | \exists_{\text{string}} \text{dem}(\Gamma) = \{ t \} \land \] 
\[ ([t, \text{naam}])(\Gamma) = \text{true} \land \] 
\[ \text{not t.adres} = \text{‘geldrop’} \} \land \forall_{\text{range}(\Gamma)} \Gamma(t) \subseteq A \]

waarbij \( \forall \) de verzameling van tuple-enheden voorstelt en waarbij de beide deel-betekenisfuncties \( \lbrack \rbrack \) op voor de hand liggende wijze gedefinieerd zijn.


3. De gedachte dat typecorrectheid van expressies in programmeertalen en specifikatietalen iets te maken heeft met dimensiecontrole zoals die bij natuurkundige vergelijkingen gebruikelijk is, leeft wel op latere wijze maar wordt slechts zelden uitgewerkt. Nootens is dit zeer wel mogelijk.

Ze zullen in een programmamechanisme, d.g. in de machines geïntegreerd kan worden, types L en T gebruikt worden voor lengte en iPod respectievelijk en L/T voor elektriciteit en warmte. Ze zullen in ieder geval worden dat ze reëel opereren + : L × L → L, + : T × T → T, / : L × T → L/T en × : L/T × T → L zijn, maar bijvoorbeeld geen × : L × L → L.
4. Een ‘Spartaans’ maar krachtig formalisme verdient vaak de voorkeur boven formalismen waarvan de schijnbare uitdrukkingskracht streelt op een ad hoc collectie van elk op zichzelf aantrekkelijke handelingen.


6. De volgende gereedschappen vormen een minimaal maar tevens zeer nuttige collectie ter ondersteuning van een als ‘breed-spectrumaal’ aangeduid formalisme zoals OOLD: controleprogramma’s voor syntactische correctheid, een modulebibliotheek, een codegenerator voor een onprobleematische deeltaal alsmede gereedschap voor het maken en hanteren van grafische representaties.

7. Voor de ontwerper van elektronische schakelingen is er in de loop der jaren een omgangstaal opgebouwd met veel woorden die doelmatig zijn door hun compactheid of in de zin dat ze niet op storinge wijze overbelast zijn. Voor de ‘software engineering’ is een passende omgangstaal ook gewenst maar nog slechts in mindere mate opgebouwd.

8. Bron-coderingstechnieken gebaseerd op een twee-componentestrategie waarbij een signaal $S(t)$ wordt gesplitst in twee signalen $S_1(t)$ en $S_2(t)$ zodat $S_1(t) + S_2(t) = S(t)$, dragen steeds het risico met zich mee dat de splitsing niet leidt tot twee geheel onafhankelijke signalen en dat dus de som van de bit-rates gebruikt voor de cording van $S_1$ en $S_2$ groter is dan strikt noodzakelijk voor $S$.

9. Zolang bij het ontwerpen van computer beeldschermen nog uitgegaan wordt van beeldherhalingsfrequenties die voor het menselijk oog waarneembaar zijn (knipperend beeld), verdient het aanbeveling om gebruik te maken van heldere tekens op een zwarte achtergrond.

10. Het in de context van technische systemen vaak gebruikte begrip 'upward-compatibility' heeft zonder nadere toelichting geen precieze betekenis. Een uitstekend voorbeeld is te verkrijgen door te stellen dat er een upward-compatibility relatie bestaat tussen de systemen van DUPLO en LEGO bouwblokken en dan details van het hiermede opgeroepen beeld te toetsen aan de werkelijkheid.


11. De constatering dat een van de voornaamste toepassingen van het spel 'LIFE' betrekking heeft op het genereren van een bewegend patroon om beeldschermen van werkstations voor inbranden te behoeden, is hieruit vanuit de optiek van wetenschapstemodologie.