Formalising Interface Specifications

Louis van Gool
This research was financially supported by the Dutch Technology Foundation STW (EES.5141).

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2006-19

Cover design by Paul Verspaget

Copyright © 2006 Louis van Gool
Formalising Interface Specifications

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 27 november 2006 om 16.00 uur

door

Louis Clemens Martinus van Gool

geboren te Venlo
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. J.C.M. Baeten
en
prof.dr. A. de Bruin

Copromotor:
dr. R. Kuiper
To properly balance the finishing of your Ph.D. thesis with your full-time job and social life appears to be not that easy. Because you always have less and need more time than you think, the final date keeps on being postponed. This cannot go on forever though. There are members of a doctorate committee waiting for that one final version, who you keep bothering with new partial versions. There are also family, friends and colleagues waiting for that party, who you keep telling that it is *almost* finished. Then there is the author of the thesis, who finally wants to spend a holiday on the beach instead of one behind paper and a computer screen. One should also be aware of the fact that science is *not* waiting for you to finish and research might be severely outdated before copies of your thesis are printed. However, more than two years late, here it finally is.

There are several people without whom this thesis would have never been written. First of all, I want to thank Jos Baeten and Ruurd Kuiper for accepting me as a Ph.D. student. After my master’s, there could not have been a better opportunity for me to do the kind of research that I liked to do. This is for a large part thanks to the freedom that Ruurd gave me. I thank him for that and also for the fact that he knew when to restrict this freedom.

One person that was the source of many of the ideas that are formalised in this thesis, is ‘the father of ISpec’ Hans Jonkers. In the many discussions that we had and the research that initiated and followed from these discussions, I really learned to appreciate the careful thought that went into the conception of ISpec. I thank Hans for his patience during our many hours of discussion where I often tried to make things simpler than they were.

For many theoretical ideas in this thesis, it seems impossible to trace them back to the person that should be accredited for them. All I can do is name the person that taught me most of these ideas, being the supervisor of my master’s thesis, Jaap van der Woude. Several of these ideas were undoubtfully his own original ones. Much of the formal theory in this thesis is no more than an elaboration of things that he taught me, but that have, to the best of my knowledge, never been written down in articles. I also thank Jaap for pointing me to this Ph.D. position.

Also the Systematic Object-Oriented Programming (SOOP) group of the Technische

v
Universiteit Eindhoven, in which I participated, was the source of several ideas. I thank all members of that group for the interesting sessions.

Creation of tool support for ISpec was one of the primary goals of our project. I like to thank all people that worked on this: Ella Roubtsova, Sergei Roubtsov, Erik Luit and the Callisto team. I especially thank Ella for the fact that, without her, I would probably not have published a single article during my Ph.D. period.

I thank all members of the core committee, Jos Beaten, Arie de Bruin, Ruurd Kuiper, Roland Backhouse and Hans Jonkers for their more-than-two-year patience and careful reading of an ever-growing number of pages. I also thank Mark van den Brand, Loet Feijs and Wim Hesselink for joining the doctorate committee.

Then there are colleagues that enable you to go to your work with pleasure. I thank all colleagues of the (Eindhoven) Embedded Systems Institute (ESI) and formal methods (FM) group for this. I especially thank Harm van Beek for the ‘deep’ discussions during lunch and for enhancing my table-tennis skills. For the same reasons, I also thank Koos van Gogh who did his master’s within our project.

Finally I want to thank my family and friends that, during my four-year Ph.D. period, enabled me to not spend time on thinking about my work outside working hours. I’m convinced that without them, I would not have been able to avoid the infamous ‘AIO-dip’. I thank Werner and Barry for being my paranymphs.
# Contents

**Preface** v  
**Contents** vii  

1 Introduction 1  
1.1 Motivation ........................................... 1  
1.2 Overview .............................................. 1  

2 ISpec 3  
2.1 Suites .................................................. 3  
2.2 Syntax ................................................... 4  
  2.2.1 Suite diagrams ........................................ 4  
  2.2.2 Invariants ........................................... 7  
  2.2.3 Methods .............................................. 9  
  2.2.4 Bases ................................................. 12  
2.3 Semantic model ........................................ 17  
  2.3.1 Traces ................................................ 17  
  2.3.2 Roles, interfaces and methods ......................... 17  
  2.3.3 Parameters and results ............................... 18  
  2.3.4 Global state ......................................... 18  
  2.3.5 This .................................................. 19  
2.4 Semantics ............................................... 20  
  2.4.1 Suite diagrams ...................................... 20  
  2.4.2 Invariants .......................................... 21  
  2.4.3 Methods ............................................. 23  
  2.4.4 Bases ................................................. 25  
2.5 Future research ....................................... 27  
  2.5.1 Required interfaces .................................. 27  
  2.5.2 Parallelism ......................................... 27  
  2.5.3 Object existence .................................... 29  
  2.5.4 Renaming ............................................ 29  
  2.5.5 Black-box semantics ................................. 29
2.5.6 Activation specifications ............................................. 29
2.5.7 Keeping parent roles ............................................... 29
2.6 Conclusions ........................................................... 30

3 Relations ................................................................. 31
3.1 Basics .................................................................. 31
  3.1.1 Predicate calculus ............................................... 31
  3.1.2 Omission of brackets ............................................ 32
  3.1.3 Omission of quantifications ................................. 32
  3.1.4 Basic elements ............................................... 32
  3.1.5 Collections .................................................... 32
  3.1.6 Connections .................................................. 33
  3.1.7 Relations .................................................... 33
  3.1.8 Emptiness and fullness ................................. 34
  3.1.9 Functionality and totality ............................. 35
  3.1.10 Application .................................................. 36
  3.1.11 Tuples ...................................................... 36
  3.1.12 Arguments .................................................. 36
  3.1.13 Fix notation .................................................. 36
  3.1.14 Sections .................................................... 37
  3.1.15 Application relation ........................................ 37
  3.1.16 Predicates .................................................. 37
  3.1.17 Binary predicates ........................................... 38
  3.1.18 Equality .................................................... 38
  3.1.19 Sets ......................................................... 38
  3.1.20 Element-of and owner-of ................................... 38
  3.1.21 Types ........................................................ 39
  3.1.22 Enumeration .................................................. 41
  3.1.23 Comprehension .............................................. 41
  3.1.24 Non-strict evaluation ..................................... 43
  3.1.25 Feijen notation ............................................. 43
  3.1.26 Axiom of choice ............................................. 44
  3.1.27 Point-wise against point-free .......................... 45
3.2 Collection operators .................................................. 45
  3.2.1 Inclusion ..................................................... 46
  3.2.2 Equality ...................................................... 46
  3.2.3 Binary intersection and union ......................... 47
  3.2.4 Then and when ............................................... 47
  3.2.5 Removed-from and without ............................. 47
  3.2.6 Same and other ............................................. 48
  3.2.7 Empty and full collection ................................ 48
  3.2.8 Complement .................................................. 48
  3.2.9 Arbitrary intersection and union ....................... 49
  3.2.10 Notational considerations ............................... 49
3.3 Relation operators .................................................... 50
  3.3.1 Sequential composition .................................... 50
  3.3.2 Over and under ............................................... 50
### 3.3.3 Identity relation ................................. 50
### 3.3.4 Domain and range ............................... 51
### 3.3.5 Converse ......................................... 51
### 3.3.6 Iteration .......................................... 51
### 3.3.7 Power ............................................. 52
### 3.3.8 Spick and spack .................................. 52
### 3.3.9 Constant function ............................... 52
### 3.3.10 Doubling function .............................. 53
### 3.3.11 Everywhere and somewhere ...................... 53
### 3.4 Properties .......................................... 54
   ### 3.4.1 Emptiness and fullness ......................... 54
   ### 3.4.2 Uniqueness and singleness ..................... 55
   ### 3.4.3 Functionality, injectivity, totality and surjectivity ...... 55
   ### 3.4.4 Sets, conditions and squares ................... 55
   ### 3.4.5 Reflexivity, transitivity and symmetry .......... 56
   ### 3.4.6 Monotonicity ................................... 57
   ### 3.4.7 Closedness ..................................... 57
   ### 3.4.8 Disjointness and conjointness ................. 58
   ### 3.4.9 Domain equalness ................................ 58
   ### 3.4.10 Idempotency, commutativity and associativity .... 58
   ### 3.4.11 Zeros and units ................................ 59
   ### 3.4.12 Distribution ................................... 59
   ### 3.4.13 Commutation ................................... 59
   ### 3.4.14 Abidence ...................................... 60
   ### 3.4.15 Comparability, chains and continuity .......... 60
### 3.5 Galois connections .................................. 61
   ### 3.5.1 Definition ..................................... 61
   ### 3.5.2 More cancellation laws ......................... 61
   ### 3.5.3 Uniqueness and existence of adjoints .......... 62
   ### 3.5.4 Exploiting Galois connections ................... 62
### 3.6 Fixed points ........................................ 63
### 3.7 Relators ............................................ 65
### 3.8 Some theorems ...................................... 67
   ### 3.8.1 Shunting ....................................... 67
   ### 3.8.2 Complementing things ......................... 68
   ### 3.8.3 Distribution of sequential composition over intersection ... 68
   ### 3.8.4 Extreme point-freeness ......................... 70
### 3.9 Conclusions ........................................ 72

### 4 Types ................................................ 73
   ### 4.1 Cylindric typing ................................ 73
   ### 4.2 Type operators .................................... 75
      ### 4.2.1 Cylindric-type operator ....................... 75
      ### 4.2.2 Functionality and totality ................... 75
      ### 4.2.3 Restrictedness ................................ 76
      ### 4.2.4 Combining type operators .................... 77
      ### 4.2.5 Domain ....................................... 77
4.2.6 Bidirectional-type operator .................................. 77
4.2.7 Set-preservingness .............................................. 77
4.2.8 Extensibility ..................................................... 78
4.2.9 Type diagrams ................................................... 79
4.2.10 Twelve type operators ........................................ 79

4.3 Type-generalising some other concepts ............................. 82
4.3.1 Emptiness and fullness ......................................... 82
4.3.2 Reflexivity ....................................................... 82
4.3.3 Shunting .......................................................... 82

4.4 Properties of the cylindric-type operator ......................... 83
4.4.1 Inclusion .......................................................... 83
4.4.2 Equality .......................................................... 83
4.4.3 Intersection ...................................................... 84
4.4.4 Union ............................................................. 84
4.4.5 Sequential composition ....................................... 85
4.4.6 Identity relation ................................................ 85
4.4.7 Converse ........................................................ 88

4.5 Conclusions .......................................................... 88

5 Products ............................................................... 91
5.1 Cylindric product ................................................. 91
5.2 Constrained cylindric products ................................... 94
5.3 Constrained cylindric packs ....................................... 96
5.4 Properties of the binary constrained cylindric products ...... 100
5.4.1 Cancellation ..................................................... 100
5.4.2 Inclusion ........................................................ 101
5.4.3 Equality ........................................................ 101
5.4.4 Intersection .................................................... 101
5.4.5 Union ............................................................. 102
5.4.6 Sequential composition ...................................... 103
5.4.7 Identity relation ............................................... 108
5.4.8 Converse ........................................................ 109

5.5 Arbitrary products ................................................ 110
5.6 Conjoint sum ........................................................ 111
5.7 Properties of the constrained cylindric packs ................... 112
5.8 Loose product ....................................................... 113
5.9 Conclusions .......................................................... 116

6 Expressions ............................................................. 117
6.1 Definition ............................................................ 117
6.1.1 Context .......................................................... 119
6.1.2 Constants ......................................................... 119
6.1.3 Strict unary operators ....................................... 119
6.1.4 Loose binary operators ...................................... 120
6.1.5 Loose unary operators ...................................... 121
6.2 Non-strict evaluation .............................................. 122
10.4.3 Field selection ............................................. 244
10.4.4 Field-modify .............................................. 244
10.4.5 Parameters ................................................ 245
10.4.6 Result ..................................................... 245
10.4.7 Example ................................................... 245
10.5 Conclusions .................................................. 246

11 Suites 247
11.1 Some more operators ......................................... 247
11.1.1 Over and under ........................................... 247
11.1.2 Arbitrary intersection ..................................... 249
11.1.3 Aspect operator ........................................... 250
11.2 Formal ISpec language ....................................... 251
11.2.1 Suites ..................................................... 253
11.2.2 Roles ...................................................... 254
11.2.3 Interfaces ................................................ 254
11.2.4 Methods .................................................. 254
11.2.5 Posts ....................................................... 255
11.2.6 Action clauses ............................................ 255
11.2.7 Post-conditions .......................................... 255
11.2.8 Result types ............................................... 255
11.2.9 Pres ......................................................... 256
11.2.10 Pre-conditions ........................................... 256
11.2.11 Parameters types ...................................... 256
11.2.12 Role invariants .......................................... 257
11.2.13 Action invariants ........................................ 257
11.2.14 Post invariants ......................................... 257
11.2.15 State invariants ......................................... 258
11.2.16 Attribute types ......................................... 258
11.2.17 Primitive types ......................................... 258
11.2.18 Sequential composition ................................. 259
11.2.19 Modify .................................................... 259
11.2.20 Call ....................................................... 260
11.3 Plug pattern .................................................. 262
11.3.1 Signature .................................................. 262
11.3.2 Hierarchy .................................................. 262
11.3.3 Invariants ................................................ 263
11.3.4 Methods ................................................... 264
11.3.5 A trace of insertInto .................................... 268
11.4 Observer pattern ............................................. 273
11.4.1 Signature .................................................. 273
11.4.2 Hierarchy .................................................. 274
11.4.3 Invariants ................................................ 274
11.4.4 Methods ................................................... 275
11.4.5 A trace of insertInto .................................... 277
11.4.6 Property ................................................... 281
11.5 Conclusions .................................................. 282
Make things as simple as possible, but not simpler.

Albert Einstein
Chapter 1

Introduction

At Philips Research the specification approach ISpec has been developed for the specification of interfaces of component-based systems. The main idea behind ISpec is that specifications are a kind of design patterns [20], called “suites” in ISpec. To describe suites, ISpec uses several academic notions like invariants, pre-conditions, post-conditions and abstract operational descriptions called “action clauses”. In this thesis, several fundamental concepts of ISpec are formally investigated and a formal language for suites is defined.

1.1 Motivation

Why should we formalise an approach that is already successful? From an academic point of view, trying to formalise ISpec is interesting because it is an approach that is actually used in practice, forcing one to think about real-life concepts like ‘object’, ‘role’, ‘interface’ and ‘design pattern’. A mathematical foundation also helps to better understand ISpec and can act as a solid basis for the construction of tool support. Relatively simple tool support that is based on a solid basis can already be very helpful in improving consistency of specifications. Tool support for ISpec is currently under development.

1.2 Overview

Chapter 2 starts with an informal description of ISpec to sketch the context of the formalisation that follows. The main mathematical formalism that is used for the formalisation, is the calculus of relations. In chapter 3 we introduce the basis of this calculus. Chapter 4 systematically investigates different ways of typing, introducing a rich collection of type operators. A key role is played by the cylindric-type operator whose algebraic properties are investigated in detail. Using the cylindric-type
operator, we define and investigate in chapter 5 a new construct called the cylindric product and several constrained forms of this construct, including the product and sum of allegory theory. These products are used in the construction of a model for expressions that evaluate in several different ways: strictly, non-strictly and even non-deterministically. This is the subject of chapter 6. In chapter 7 we apply the theory developed thus far by showing how several fundamental concepts like refinement, correctness, mixing of declarative and operational descriptions, structural hierarchy, components, aspects and invariants can be formalised. In that chapter we observe that recursion needs some extra attention. This subject is treated in chapter 8. There we observe that the abstraction level of relations is too high to capture semantically visible calls. Chapter 9 addresses this issue by the introduction of protocols. In all above-mentioned chapters we abstracted from the particular shape of states. Chapter 10 shows how states can be structured such that common variable-concepts of object-oriented programming are captured. In chapter 11 we show how the concepts developed thus far relate to ISpec by defining a formal language for suites. Chapter 12 concludes the thesis.
Chapter 2

ISpec

In this chapter we give an overview of ISpec’s main concepts.

2.1 Suites

The core observation from which ISpec originated, is that interaction patterns (called suites in ISpec) are a better unit of specification than individual classes. An interaction pattern describes the interaction between objects by means of roles that the objects can play. A role represents a certain task that an object has in an interaction and is comparable to the notion of abstract class in object-oriented programming. An object can play multiple roles and multiple objects can play the same role, all this at one point in time.

A classical example of an interaction pattern is the observer pattern that describes the interaction between subjects and observers. Each subject has a number of observers and each observer observes at most one subject. A subject has a state of which all its observers have a copy. Proper interaction between the subject and its observers should ensure that its observers’ copies are updated when necessary. Although it is possible to describe the class of subjects and the class of observers in isolation and see it as a coincidental choice of a programmer that both classes interact, it is more natural to describe the two classes as part of one coherent interaction pattern.

Although we agree that building systems by means of interaction patterns is a powerful approach, this thesis is not about the motivations behind this approach. For this we refer the reader to [34]. The intention of this thesis is to provide a formal ground for the main concepts that play a role in such an approach and in ISpec in particular. As proof of the pudding, we present in chapter 11 a completely formal specification of the observer pattern (restricted to at most one observer per subject to simplify things a little).
2.2 Syntax

In [35] ISpec's syntax for suites is defined by means of an object-oriented class model, called the ISpec metamodel. The classes in a metamodel are called metaclasses. In [35] the ISpec metamodel is described by means of several UML-like class diagrams. The following subsections introduce a slightly adjusted version of the ISpec metamodel step by step.

2.2.1 Suite diagrams

We start with the syntax of suite diagrams. It is described by the following UML class diagram:

![UML class diagram of suite diagrams]

We first describe what the above UML class diagram expresses and then explain the notation in general. A NamedItem has exactly one Name as name. A Suite is a NamedItem and has SuiteMembers as members. A SuiteMember is also a NamedItem and is member of exactly one Suite. Roles, Interfaces, Methods, Attributes and Types are all SuiteMembers. Each Interface is providedInterface of exactly one Role and requiredInterface of at least one Role. Each Method is method of exactly one Interface. An Attribute is attribute of exactly one Role and is a TypedItem. A TypedItem has exactly one Type as type and a Type is type of exactly one TypedItem. ValueTypes and ObjectTypes are Types.

Each "[X]" in the diagram represents the metaclass with name X (metaclasses have unique names). An instance of a metaclass X is simply called “an X”. A "[Y] −→ [Z]" means that metaclass Y is a subclass of metaclass Z: all instances of metaclass Y are also instances of metaclass Z, or “a Y is a Z”. A "[Y] _n_ −→ [Z]" means that each Y has n Zs as x and each Z is x of m Ys, where "0..*" means “zero or more”, “0..1” means “zero or one”, “1..*” means “at least one” and “1” means “exactly one”. If a metaclass name X is written in italics, the metaclass is abstract and otherwise it is concrete. A concrete metaclass can have direct instances (an abstract metaclass cannot). A direct instance of a (concrete) metaclass is an instance of that metaclass and an instance of any metaclass is a direct instance of
exactly one (concrete) metaclass.

To keep text readable, we from now on omit “as $x$” if we think that it is clear from the context which $x$ is meant. We write for example “an Interface has Methods” instead of “an Interface has Methods as methods”. We also talk about “the X n” instead of “the X that has Name n (as name)”, writing for example “the Suite ObserverPattern” instead of “the Suite that has the Name ObserverPattern as name”.

The fact that each Type is type of exactly one TypedItem may look strange at first sight. Why could two items not have the same type? An instance of the metaclass Type is not a type however, but a placeholder for a Name that represents a type. This is the reason why we talk about a “Type” instead of a “type”. Two ValueTypes could for example both have “bool” as Name. Both ValueTypes then represent the type of the booleans.

An example of a suite diagram is the diagram of what we shall call the plug pattern:

The Suite PlugPattern has three Roles: Hole, Plug and Main. The Interface IFill is a providedInterface of Role Hole (represented by the “−”) and a requiredInterface of Role Plug (represented by the “⇒”). The Interface IInsert is a providedInterface of Role Plug and a requiredInterface of Role Main. The use of dotted lines for Main is explained in section 2.5.6. Interface IFill has Methods fillWith and empty and Interface IInsert has Methods insertInto and remove. Role Hole has Attributes hasPlug and plug and Role Plug has Attributes inHole and hole. The Attributes hasPlug and inHole both have a ValueType bool, Attribute plug has an ObjectType Plug and Attribute hole has an ObjectType Hole (underlining is used to distinguish ValueTypes from ObjectTypes).

It could be tempting to think that the two occurrences of bool in the above picture refer to one and the same instance of ValueType. This cannot be the case however because, as explained earlier, a Type is type of exactly one TypedItem. In general we postulate that different occurrences in a suite-diagram picture represent different instances of metaclasses, with the exception of Names. This enables us to draw pictures where two different instances of the same metaclass have the same name. This is different from the way we interpret metamodel pictures. There we do assume
that two rectangles with the same name represent the same metaclass (as mentioned before, metaclasses have unique names). This enables one to scatter the picture of the metamodel over multiple pages.

Using the tree structure that the \( \rightarrow \)s of the metamodel constitute, we can construct a BNF-like representation of the syntax (\( X^* \) represents the set of all lists of \( X \))s):

\[
\begin{align*}
\text{Suite} & ::= \text{suite} \quad \text{Name} \\
& \quad \text{Role}^* \\
\text{Role} & ::= \text{role} \quad \text{Name} \\
& \quad \text{Interface}^* \\
& \quad \text{Attribute}^* \\
\text{Interface} & ::= \text{interface} \quad \text{Name} \\
& \quad \text{Method}^* \\
\text{Method} & ::= \text{method} \quad \text{Name} \\
\text{Attribute} & ::= \text{attribute} \quad \text{Name} \\
& \quad \text{Type} \\
\text{Type} & ::= \text{valueType} \quad \text{Name} \\
& \quad \mid \text{objectType} \quad \text{Name}
\end{align*}
\]

The representation of \( \text{Name} \)s (\( \text{Name} \)) is left unspecified.

The \( \text{Interface}s \) of the BNF-like representation of the syntax correspond to the \text{providedInterfaces} of the metamodel. The \text{requiredInterfaces} are missing. These are discussed in section 2.5.1.

In terms of the above syntax, the plug-pattern example can be written as
PlugPattern = suite PlugPattern
<Hole, Plug, Main>

Hole = role Hole
<HIFill>
<HHasPlug, HPlug>

Plug = role Plug
<PInsert>
<PInHole, PHole>

Main = role Main
<> <>

HIFill = interface IFill
<HFillWith, HEmpty>

PInsert = interface IInsert
<PInsertInto, PRemove>

HFillWith = method fillWith

HEmpty = method empty

PInsertInto = method insertInto

PRemove = method remove

HHasPlug = attribute hasPlug
valueType bool

HPlug = attribute plug
objectType Plug

PInHole = attribute inHole
valueType bool

PHole = attribute hole
objectType Hole

2.2.2 Invariants

In ISpec several kinds of invariants exist, although not described in [35]. We restrict to the most commonly found invariants: the ones that are attached to roles, called role invariants. We distinguish three kinds of role invariants: action invariants, post invariants and state invariants. The following diagram shows how the
The excludedInterfaceNames of a RoleInvariant are not an ISpec concept. Their meaning is explained in section 2.4.2.

In our BNF-like notation, the extension of the metamodel with invariants is reflected by the following adjustment:

\[
\begin{align*}
\text{Role} & ::= \text{role} \\
& \quad \text{Name} \quad \text{Interface}^* \\
& \quad \quad \text{Attribute}^* \\
& \quad \quad (\text{RoleInvariant Name}^*)^* \\
\text{RoleInvariant} & ::= \text{actionInvariant} \quad \ldots \\
& \quad | \quad \text{postInvariant} \quad \ldots \\
& \quad | \quad \text{stateInvariant} \quad \ldots \\
\end{align*}
\]

The language that is used to describe RoleInvariants is left open in ISpec. We use a language that should be understandable for most people that know object-oriented programming. For our plug-pattern suite we define some StateInvariants:

\[
\begin{align*}
\text{Hole} & = \text{role} \quad \text{Hole} \quad <\text{HIFill}> \\
& \quad <\text{HHasPlug}, \text{HPlug}> \\
& \quad <\text{HInv1}, \text{HInv2} >> \\
\text{Plug} & = \text{role} \quad \text{Plug} \\
& \quad <\text{PInInsert}> \\
& \quad <\text{PInHole}, \text{PHole}> \\
& \quad <\text{PInv1}, \text{PInv2} >> \\
\text{Main} & = \text{role} \quad \text{Main} \\
& \quad <> \\
& \quad <> \\
& \quad <> \\
\end{align*}
\]
\[ HSInv1 = \text{stateInvariant} \ hasPlug \Rightarrow \text{plug.inHole} \]
\[ HSInv2 = \text{stateInvariant} \ hasPlug \Rightarrow \text{plug.hole} = \text{this} \]
\[ PSInv1 = \text{stateInvariant} \ inHole \Rightarrow \text{hole.hasPlug} \]
\[ PSInv2 = \text{stateInvariant} \ inHole \Rightarrow \text{hole.plug} = \text{this} \]

The StateInvariant \( HSInv2 \) expresses for example that if a hole has a plug in it, the plug’s attribute \( \text{hole} \) should point to this hole. Details about the meaning of RoleInvariants are given in section 2.4.2.

### 2.2.3 Methods

In ISpec a Method has several Parameters, each of which has a Name and a Type. It also has a Result with a Type and it has several Effects. Each Effect has a PreCondition, an ActionClause and a PostCondition. The following class diagram shows how the metamodel is extended to enable this more detailed description of Methods:

```
or in our BNF-like notation:
```
Method ::= method Name
Parameter* Result Effect*

Parameter ::= parameter Name Type

Result ::= result Type

Effect ::= effect PreCondition ActionClause PostCondition

PreCondition ::= preCondition ... 

ActionClause ::= actionClause ...

PostCondition ::= postCondition ...

The language that is used to specify PreConditions, ActionClauses and PostConditions is left open in ISpec. For the plug-pattern suite we define the Methods fillWith, empty, insertInto and remove by

\[
HF\text{FillWith} = \text{method } \text{fillWith} <HF\text{FiParameter}> \HF\text{FiResult} <HF\text{FiEffect}>
\]

\[
HF\text{FiParameter} = \text{parameter } p \text{ objectType Plug}
\]

\[
HF\text{FiResult} = \text{result } \text{valueType void}
\]

\[
HF\text{FiEffect} = \text{effect } HF\text{FiPreCondition} HF\text{FiActionClause} HF\text{FiPostCondition}
\]

\[
HF\text{FiPreCondition} = \text{preCondition } \neg\text{hasPlug} \land p.\text{inHole} \land p.\text{hole} = \text{this}
\]

\[
HF\text{FiActionClause} = \text{actionClause } \text{modify}\{\text{hasPlug, plug}\}
\]

\[
HF\text{FiPostCondition} = \text{postCondition } \text{hasPlug} \land \text{plug} = p
\]
<table>
<thead>
<tr>
<th>HE\text{Empty}</th>
<th>= method</th>
<th>empty</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>HE\text{EmResult}</td>
<td>= result</td>
<td>valueType void</td>
</tr>
</tbody>
</table>
| HE\text{EmEffect}  | = effect    | HE\text{EmPreCondition}  
|                     |            | HE\text{EmActionClause}  
|                     |            | HE\text{EmPostCondition}  |
| HE\text{EmPreCondition} | = preCondition | hasPlug |
| HE\text{EmActionClause} | = actionClause | modify\{hasPlug.plug\} |
| HE\text{EmPostCondition} | = postCondition | \neg hasPlug |

| P\text{InsertInto} | = method    | insertInto  |
|                     |            | \langle P\text{InParameter} \rangle  
|                     |            | P\text{InResult}  
|                     |            | \langle P\text{InEffect} \rangle  |
| P\text{InParameter} | = parameter | h  |
|                     |            | objectType Hole |
| P\text{InResult}   | = result    | valueType void |
| P\text{InEffect}   | = effect    | P\text{InPreCondition}  
|                     |            | P\text{InActionClause}  
|                     |            | P\text{InPostCondition}  |
| P\text{InPreCondition} | = preCondition | \neg inHole  
|                     |            | \wedge \neg(h.hasPlug) |
| P\text{InActionClause} | = actionClause | modify\{inHole.hole\}  
|                     |            | ; hole.fillWith(this) |
| P\text{InPostCondition} | = postCondition | inHole  
|                     |            | \wedge hole = h |

---

11
The Method $\text{PInsertInto}$ expresses for example that if parameter $h$ is a Hole that (see the PreCondition) has no plug in it and the Plug on which method insertInto is called, is not inserted into a Hole, then this Plug is inserted into Hole $h$ (see the ActionClause and PostCondition). Details about the meaning of Methods are given in section 2.4.3.

### 2.2.4 Bases

In ISpec it is possible to base a suite (called child in this context) on other suites (called parents). The idea behind it is that the child suite combines and specialises the behaviour of its parent suites. Connections between Roles of the child Suite and those of its parent Suites define where the behaviour of the parent suites occurs in the child suite. The following class diagram shows how the metamodel is extended to support bases:

![Class Diagram](image)

or in terms of our BNF-like notation:
Suite ::= suite
   Name
   Base* 
   Role*

Base ::= base
   Name
   RoleConnection*

RoleConnection ::= roleConnection
   Name
   Name

A main idea of ISpec is that a suite can be built by basing it on (combining and specialising) other suites that can again be built in this manner. The following picture, that we took from [34], illustrates this idea:

In case the rectangle of a suite $s$ is completely included in the rectangle of another suite $s'$, then $s'$ is (directly or indirectly) based on $s$. The inheritance arrows (the arrows with a triangular head) represent the RoleConnections. The outer rectangle represents the one suite that is the result of all this combining and specialising.

When examining this picture, several questions arise like: “What does it mean that a role $r$ inherits from two roles $r_1$ and $r_2$ of one suite $s$?”, “Is this the same or different from $r$ inheriting from $r_1$ and $r_2$ that belong to different occurrences of $s$?” and “What happens with the roles of a parent suite that are not inherited from?”. Unfortunately we have to leave the exact answers to these questions for future research (see also sections 2.5.4 and 2.5.7). We only model the simple case where for each Base of a Suite there exists exactly one RoleConnection for each parent role.

An example of a suite that is based on another suite is the observer pattern. The observer pattern is based on the plug pattern:
Using our BNF-like notation, we can fill in the details:
PlugPattern = suite PlugPattern
<>
<Hole, Plug, Main>

ObserverPattern = suite ObserverPattern
<Base>
<Subject, Observer>

Base = base PlugPattern
(Subject2Hole, Observer2Plug)

Subject2Hole = roleConnection Subject
Hole

Observer2Plug = roleConnection Observer
Plug

Subject = role Subject
<SIFill>
<SOrig>
<SSInv <>>

Observer = role Observer
<OIIUpdate>
<OCopy>
<OSInv <IUpdate>>

SIFill = interface IFill
<SFillWith>

OIIUpdate = interface IUpdate
<OIIUpdate>

SOrig = attribute orig
valueType void

OCopy = attribute copy
valueType void

SSInv = stateInvariant hasPlug ⇒ plug.copy = orig

OSInv = stateInvariant inHole ⇒ copy = hole.orig
The plug pattern is extended with an interface IUpdate and attributes orig and copy. The type of these attributes is not restricted (the ValueType void represents this fact). Method update of interface IUpdate is used to update an Observer’s copy of the state of the Subject it is inserted into. Method fillWith is adapted such that if an Observer is inserted into a Subject, update is called to ensure that the Observer’s state is updated.

In the above specification we borrowed several parts of the plug pattern. The definition of Method fillWith (re)uses HFiParameter, HFiResult, HFiPreCondition and HFiPostCondition. We call this syntactic inheritance. More on this in section 2.4.4.
2.3 Semantic model

In this section we describe the semantic model for suites.

2.3.1 Traces

During the study of object-oriented design patterns like the observer pattern, we became aware of the fact that a relational model is not sufficient to cope with an essential aspect of these kinds of design patterns: **communication** (see also [10]). ISpec addresses communication by means of its action clauses, in which it is possible to write calls. To capture communication, a semantic model for suites should be richer than a relational model. Where a relational model only captures the ‘outermost’ calls and returns of a system, we use a model that can also capture calls and returns that occur ‘in between’. This thesis only discusses ‘single threaded’ systems. Paralllelism is left for future research (see section 2.5.2). We define the model of a suite to be a set of (finite) **traces** where, at a high level of abstraction, a trace consists of a **call**, a (finite) **trace sequence** and a **return**:

\[
\text{Trace} ::= \text{trace} \quad \text{Call} \\
\quad \text{Trace}^* \\
\quad \text{Return}
\]

\[
\text{Call} ::= \text{call} \quad \ldots
\]

\[
\text{Return} ::= \text{return} \quad \ldots
\]

Now that we have chosen which points ‘in time’ can be observed, we have to define what information is observed at these points (the “…”s). This is addressed in the following subsections.

2.3.2 Roles, interfaces and methods

To distinguish the different methods that can be called, a trace should contain an identification of the called method. In our case a method is identified by the combination of a role name, an interface name and a method name:

\[
\text{Trace} ::= \text{trace} \quad \text{Call} \\
\quad \text{Trace}^* \\
\quad \text{Return}
\]

\[
\text{Call} ::= \text{call} \quad \text{Name} \\
\quad \text{Name} \\
\quad \text{Name} \\
\quad \ldots
\]

\[
\text{Return} ::= \text{return} \quad \ldots
\]
A “call \( r \cdot i \cdot m \ldots \)” means that method \( m \) of the interface \( i \) provided by role \( r \) is called.

### 2.3.3 Parameters and results

In ISpec parameters can be sent along with a call and a result along with a return. We describe the values of the parameters by means of a total function from parameter names to values (represented by “\( Value \leftrightarrow Name \)”):

\[
\begin{align*}
Trace & ::= trace \quad Call \\
& \quad Trace^* \\
& \quad Return \\
Call & ::= call \quad Name \\
& \quad Name \\
& \quad Name \\
& \quad Value \leftrightarrow Name \\
& \quad \ldots \\
Return & ::= return \quad Value \\
& \quad \ldots 
\end{align*}
\]

We leave the set \( Value \) unspecified.

Notice that we postulate that a parameter value is given for each possible parameter name and that a return always has a result. The main idea is to read “has no value” as “has no specific value”. If in a specification, a call to a method does not mention a parameter \( p \), it means that the semantics contains a trace for every possible value of \( p \) at that call. This is a natural view in the context of specialisation and makes it for example straightforward to add parameters when specialising a call.

### 2.3.4 Global state

In an object-oriented system there are of course also things that are called objects. An object consists of a number of attributes each of which has a value. The collection of all objects constitutes the global state in which methods execute. For each call and return in a trace, a snapshot of the global state is included. For the outermost call and return of a trace, these snapshots are called the initial global state and final global state respectively. We model a global state as a total function from (object name, attribute name)-pairs, called fields, to values. The field \((o, a)\) is referred to as “attribute \( a \) of object \( o \)”.
This model does not capture information about the existence of objects. We assume that all objects ‘always exist’. Creation and deletion of objects is an essential feature of ISpec that we do not model in this thesis and leave for future research (see section 2.5.3).

2.3.5 This

In object-oriented languages (non-static) methods are always called ‘on a specific object’. This object then becomes the ‘currently active object’, usually called “this” or “self”. We make the name of the ‘this’ object part of the call:
2.4 Semantics

In this section we informally describe the relation between the syntax of suites and the semantic model of the previous section.

2.4.1 Suite diagrams

A suite diagram determines the structure of a suite. This structure consists of the name of the suite, the names of its roles, for each of these roles, the names of its attributes and interfaces and for each of these interfaces, the names of its methods. The suite diagram of the plug pattern

defines for example the following suite structure:

\[
\begin{array}{c}
\text{PlugPattern} \\
\text{Hole} \\
\text{plug : Plug} \\
\text{hasPlug : bool} \\
\text{empty} \\
\text{Plug} \\
\text{inHole : bool} \\
\text{hole : Hole} \\
\text{fillWith} \\
\text{insertInto} \\
\text{Main}
\end{array}
\]

One might expect that we use this structure to restrict the set of traces that is the semantics of the suite. This is however not the approach that we take. If we do not exclude traces with calls that fall outside the structure of a suite, the introduction of new roles, interfaces and methods in a specialisation becomes straightforward as shown in section 7.5. This cylindric view is one of the main topics of this thesis.

For the definition of a semantics for suites it is helpful to assume that all interfaces have different names. We can then identify an interface by means of one name instead of the combination of a role name and an interface name. This fact is exploited in section 11.2.19.

The signature of a suite consists of the names of its attributes and interfaces. The plug pattern has for example the following signature:

\[
\langle\{\text{hasPlug.plug.inHole.hole}\},\{\text{IFill, IInsert}\}\rangle
\]
The signature of a suite determines the attributes and interfaces the suite ‘talks about’. The signature is needed for the meaning of statements that occur in action clauses and action invariants. Take for example method `empty` of the plug pattern:

\[
\begin{align*}
HEmpty &= \text{method} \quad \text{empty} \\
&\quad <> \\
&\quad HEmResult \\
&\quad <HEmEffect>
\end{align*}
\]

\[
\begin{align*}
HEmResult &= \text{result} \\
&\quad \text{valueType void}
\end{align*}
\]

\[
\begin{align*}
HEmEffect &= \text{effect} \\
&\quad HEmPreCondition \\
&\quad HEmActionClause \\
&\quad HEmPostCondition
\end{align*}
\]

\[
\begin{align*}
HEmPreCondition &= \text{preCondition} \\
&\quad \text{hasPlug}
\end{align*}
\]

\[
\begin{align*}
HEmActionClause &= \text{actionClause} \\
&\quad \text{modify(hasPlug, plug)}
\end{align*}
\]

\[
\begin{align*}
HEmPostCondition &= \text{postCondition} \\
&\quad \neg \text{hasPlug}
\end{align*}
\]

The statement of the action clause (`modify(hasPlug, plug)`) states that, with respect to the signature, only the attributes `hasPlug` and `plug` of the ‘this’ object possibly change and no calls are performed. What this exactly means is that, from the attributes `hasPlug, plug, inHole` and `hole` of any object, only the attributes `hasPlug` and `plug` of the ‘this’ object possibly change. Furthermore, no calls are performed on the interfaces `IFill` and `IInsert`. The specification does not tell anything about attributes and interfaces outside the signature. Anything may happen to them. This means that if we construct a specialisation of this action clause, we may change these attributes and call methods of these interfaces in any possible manner.

Next to the structure of a suite, a suite diagram also specifies types for attributes: if an Attribute named `a` of a Role named `r` has a Type representing some type `T`, then in every trace of the suite, attribute `a` of the ‘this’ object has type `T` for every call and return of every method of every interface of role `r`.

Associating a type `t` with an attribute `a` has the same meaning as adding a state invariant that states that attribute `a` of the ‘this’ object has type `t`. The semantics of state invariants is discussed in the following subsection.

### 2.4.2 Invariants

As mentioned in section 2.2.2, three kinds of invariants are considered in this thesis: action invariants, post invariants and state invariants. Each of these invariants is a special kind of role invariant. In general, a role invariant is something that holds for
all methods of the role to which the invariant is attached. This includes methods
that are added in specialisations.

A role invariant is always formulated in the context of a ‘this’ object. For a role
invariant C and an object o, we say that “C holds for o” or “o adheres to C” if C
holds where o is the ‘this’ object.

Action invariants and post invariants can be seen as action clauses and post-
conditions respectively that are added to all methods, except for the fact that,
whereas an action clause and post-condition only need to hold if their corresponding
pre-condition holds, action invariants and post invariants always hold, irrespective
of pre-conditions. The semantics of pre-conditions, action clauses and post-conditions
is discussed in section 2.4.3. In the remainder of this section we discuss the specific
restrictions that state invariants impose.

A state invariant talks about a single state. Similar to the attribute-type case, if
a Role named r has a StateInvariant representing some constraint C, then in every
trace of the semantics, C holds for every call and return of every method (of every
interface) of role r.

We do not explicitly model the association of objects with roles. The fact that an
object plays a certain role is a derivable property from (the attributes of) that
object: an object plays a certain role if it adheres to the role’s attribute(type)s
and state invariants. A consequence of our semantics of state invariants is that
for each call and return of a method of a role r, the ‘this’ object plays role r by
definition. This approach of not explicitly modeling which roles an object plays,
results in a significantly simpler semantic model.

Sometimes it is necessary that a method of a role r is called, but the object on
which it is called, does not adhere to a particular state invariant of role r. For
these cases we made it possible to exclude certain interfaces from a role invariant.
Excluding an interface from an invariant means that the methods of that interface
do not necessarily adhere to the invariant. An example where this is needed, is the
observer pattern of section 2.2.4. After a call to method insertInto on an object
o, method update needs to re-enter o to update attribute copy of o for the purpose
of re-establishing state invariants SSInv and OSInv. However, when o is re-entered,
state invariant OSInv does not hold in general. This is the reason why interface
IUpdate is excluded from invariant OSInv.

In ISpec the interpretation of state invariants is different from ours. We postulate
that a state invariant holds at the beginning (call) and end (return) of each method
of a role. In ISpec the interpretation is that if a state invariant holds at the
beginning of a method then it holds at the end of that method. In section 7.7 this
topic is discussed in detail.

This is not the only point where we deviate from ISpec. In ISpec state invariants
are not allowed to constrain the behaviour of a suite more than already done
by pre-conditions, action clauses and post-conditions. The reason for this is prag-
matic. Programmers often forget adhering to the state invariants of a role. The pre-conditions and post-conditions of the methods of a role are therefore strengthened such that each state invariant of the role is implied by them. In practice this does not seem to cause much overhead. With proper tool support however it can be ensured that programmers do not forget invariants. Invariant aspects of a role should in our opinion then be localised in role invariants. Details can again be found in section 7.7.

2.4.3 Methods

A Method consists of Parameters, a Result and a number of Effects where each Effect consists of a PreCondition, an ActionClause and a PostCondition. With respect to the semantic model that was presented in section 2.3, Parameters and PreConditions talk about first calls of traces, Results talks about last returns of traces, PostConditions relate first calls and last returns of traces and ActionClauses can in principle talk about all calls and returns of traces.

Similar to the correspondence between attribute types and state invariants, associating a parameter with a type has the same meaning as stating in the pre-condition that the parameter is an element of that type. The same goes for results: it does not matter whether we use the natural numbers as result type or use integers and add to the post-condition that the result is at least zero. The semantics of a method specification is that for each of its effects, if the parameters have the specified types and the pre-condition of the effect holds, then the action clause and post-condition of the effect hold and the result is of the specified type.

Our intended semantics of a method specification cannot be defined in a decent denotational manner for the syntactic structure that was defined in section 2.2.3:

\[
M([p_1], \ldots, [p_n], [r], E.([pre], [action], [post]))
\]
It is however not possible to get the intended semantics in a decent manner using a formula of this form. To allow for a decent denotational semantics, the syntax of methods is for the formal language of chapter 11 restructured as follows:

![Diagram of method structure]

To cope with the fact that a method can have multiple Effects, we allow multiple Methods with the same name. When a method with name \( m \) is called, the behaviours of all Methods with name \( m \) are realised.

Action clauses are a kind of abstract programs. In this thesis we discuss three fundamental constructs for building action clauses: **modify statements**, **call statements** and **sequential composition** of statements. An example where all three constructs are used, is method \( \text{insertInto} \) of the plug pattern (see section 2.2.3).

**Modify statements**

A **modify statement** specifies fields of the global state that are possibly modified. Which fields are possibly modified, is indicated by a set of **field expressions**. The modify statement \( \text{modify}\{a, o.b\} \) means for example that, with respect to the signature, only attribute \( a \) of the ‘this’ object and attribute \( b \) of the object referred to by attribute \( o \) of the ‘this’ object are possibly modified and no calls are performed. As mentioned in section 2.4.1, anything may happen with attributes and interfaces outside the signature.

In object-oriented programs, attributes of objects other than ‘this’ are usually only modified by means of method calls on those objects. This helps to prevent falsification of state invariants for those other objects. In an implementation of \( \text{modify}\{a, o.b\} \), attribute \( b \) of the object referred to by attribute \( o \) of the ‘this’ object should then be modified by means of a call to a method of an additionally introduced interface.

**Call statements**

A call statement has several ingredients. First of all there is an expression that specifies on which object a method is called. Next to this, there is an identification of the method that is called. A list of expressions specifies the values of the parameters that are passed. Finally, the possibility exists to specify a variable where the result of the call is stored.

Also for calls, the signature of a suite is of importance. Before and after a call, anything may happen with attributes and interfaces outside the signature.
Sequential composition of statements

The sequential composition of two statements is a generalisation of traditional relational composition. The generalisation ensures that trace sequences are appropriately pasted together.

Details about these three constructs can be found in chapter 9.

2.4.4 Bases

In section 2.2.4 we used the plug pattern as a base for the observer pattern. This means that the observer pattern specialises the plug pattern in some way. In this section we discuss what it exactly means for a suite to specialise another one.

A simplification we make, is that we do not model renaming. Renaming is left for future research (see section 2.5.4). For the observer pattern we therefore simply use the names Hole and Plug instead of Subject and Observer respectively.

Without renaming, the most straightforward definition of a suite $s'$ being a specialisation of a suite $s$ is that the set of traces of $s'$ is a subset of the set of traces of $s$. Denoting the set of traces of a suite $s$ by $\mu s$ (the meaning of $s$):

$$\mu s' \subseteq \mu s$$

When we base a suite $s'$ on a suite $s$, our intention is not to define $s'$ completely from scratch. We only want to define an extension $e$ that describes how $s$ is specialised. In case of the observer pattern, the extension describes two new attributes (orig and copy), one new interface ($IUpdate$) and a new definition for method $fillable$.

In terms of our BNF-like notation, the extension $Extension$ would be defined by ($ESubject$ and $EObserver$ only differ from $Subject$ and $Observer$ of section 2.2.4 with respect to their role names, which reflects the fact that we do not model renaming):

\[
\begin{align*}
Extension & = \text{suite} \quad EExtension \\
& \quad \langle ESubject, EObserver \rangle \\
ESubject & = \text{role} \quad Hole \\
& \quad \langle SIFill \rangle \\
& \quad \langle SOrig \rangle \\
& \quad \langle SSInv \rangle \\
EObserver & = \text{role} \quad Plug \\
& \quad \langle OIUpdate \rangle \\
& \quad \langle OCopy \rangle \\
& \quad \langle OSInv \langle IUpdate \rangle \rangle 
\end{align*}
\]

It now seems natural to define the meaning of ObserverPattern as the intersection
of the sets of traces $\mu_{\text{PlugPattern}}$ and $\mu_{\text{Extension}}$. The interpretation in ISpec is subtly different however, which is related to the notion of signature. If we take for example method empty of the plug pattern, its action clause states that \emph{with respect to the signature}, only the attributes hasPlug and plug of the ‘this’ object possibly change and no calls are performed. This means that if we do intersection at the semantic level, method empty is interpreted in the signature of the plug pattern and not in the signature of the observer pattern. We call this \textbf{semantic inheritance} of method empty. In ISpec however, if no new definition is given for a method, its specification is reinterpreted in the new signature. We call this \textbf{syntactic inheritance}. The specification of method empty should thus be reinterpreted in the signature of the observer pattern. More formally, the meaning of ObserverPattern is equal to

$$\mu_{\text{PlugPattern}} \cap \mu_{\text{Extension}}'$$

where $\text{Extension}'$ is defined by

\begin{align*}
\text{Extension}' &= \text{suite} \quad \text{Extension} \\
&= \langle > \langle E\text{Subject}', E\text{Observer}' \rangle \\
E\text{Subject}' &= \text{role} \quad \text{Hole} \\
&= \langle E\text{IFill}' \rangle \langle S\text{Orig} \rangle \langle S\text{SIInv} \rangle \\
E\text{Observer}' &= \text{role} \quad \text{Plug} \\
&= \langle P\text{Insert}, O\text{IUpdate} \rangle \langle O\text{Copy} \rangle \langle O\text{SIInv} \langle I\text{Update} \rangle \rangle \\
E\text{IFill}' &= \text{interface} \quad \text{IFill} \\
&= \langle S\text{FillWith}, H\text{Empty} \rangle
\end{align*}

The difference between $\text{Extension}'$ and $\text{Extension}$ is that $\text{Extension}'$ contains a syntactic copy of interface $I\text{Insert}$ and method empty of the plug pattern, where $\text{Extension}$ does not.

Now that we know how inheritance of methods works, what happens exactly if we give a new definition for a method, like we did for $\text{fillWith}$ in the observer pattern? In principle, what happens, is determined by the fact that we take the intersection of the sets of traces. A consequence is that if we used “true” as post-condition in the new definition of $\text{fillWith}$, the post-condition of the plug pattern’s $\text{fillWith}$ would still hold. In ISpec however the behaviour of a method for which a new definition is given, is solely determined by this new definition. This is called \textbf{overriding}. ISpec does not allow arbitrary overriding, but combines overriding with \textbf{behavioural subtyping} \cite{38}. Behavioural subtyping means that when we override some methods, the behaviour of the resulting (child) suite should be a
specialisation of its parent suites. Behavioural subtyping is usually associated with the following rules: pre-conditions and parameter types should not be strengthened and attribute types, result types, post-conditions, action clauses and invariants should not be weakened. Our semantics for these concepts was chosen in such a way that these rules indeed guarantee behavioural subtyping.

In ISpec there is another requirement on specialisation. If someone has written a partial implementation of the plug pattern, it should be possible to combine this partial implementation with a partial implementation of the observer-pattern extension into a new (possibly again partial) implementation of the observer pattern. In the context of simple set inclusion (also called partial refinement), this case already has some complications as shown in section 8.5.7. Additional complications arise in the context of total refinement. Then we need to prove that a suite that specialises another suite, is a conservative extension of that suite. This guarantees that an implementation of the extended part of a suite ‘is able to cooperate’ with the already implemented part of the suite, no matter how that part was implemented. Details can be found in section 8.6.4.

In ISpec the observer pattern would, next to the roles Subject and Observer, also still contain the roles Plug and Hole. This is discussed in section 2.5.7.

2.5 Future research

In this section we discuss some topics that we left for future research.

2.5.1 Required interfaces

Next to provided interfaces, ISpec also specifies required interfaces. Actually “requires” should be read as “is allowed to use”. A rule in ISpec is that if a suite $s'$ specialises a suite $s$ and $s$ contains an interface $i$ that is required by roles $r_0, \ldots, r_n$, then each role $r'$ in $s'$ that wants to use $i$, should be connected to one of $r_0, \ldots, r_n$ by means of an inheritance arrow. A way to represent the fact that $i$ is a required interface of roles $r_0, \ldots, r_n$ in our semantics, is to exclude all traces where a role other than one of $r_0, \ldots, r_n$ performs a call on interface $i$. Adding the notion of required interfaces increases the expressive power of the formalism. Its general purpose is to ensure that roles that are not explicitly part of an interaction pattern, cannot cause unwanted interference.

2.5.2 Parallellism

As mentioned in section 2.3.1, we only consider single-threaded systems. In [45], section 4, an attempt is made to model interleaved parallellism. The main idea is that every call is placed on a (virtual) stack. Where in a single-threaded system, only the top-most call of the stack may return, in a multi-threaded system, every
call in the stack may return. The call depth of a return is a natural number that indicates how deep in the stack the call that returns, lies. Traces are then represented as sequences of an equal number of calls and returns where the call depth of each return is smaller than the number of calls preceding the return minus the number of returns preceding it. If the call depth of every return equals zero, the model is isomorphic to the single-threaded model of this thesis. Although this call depth may appear to be a nice trick, the model does not really seem appropriate as a semantic model for ISpec with parallelism. Replacing the total orders (sequences) of our traces with partial orders, provides an interesting alternative. Where traces in the single-threaded model look like (🍅 represents a trace, 🍓 a call and 🍔 a return):

the use of partial orders instead of total orders would allow traces like:

In this model, multiple threads executing in parallel should access disjoint parts of the global state to avoid interference. For many systems this is probably not a feasible assumption and a more complicated model is needed.
2.5.3 Object existence

In this thesis we do not model object creation and deletion. In section 2.3.4 we phrased this as: “Each object always exists.”. To model object existence, a straightforward generalisation of our semantic model would be to omit the totality requirement on the function that models the global state. For several constructs, like the call statement, it then has to be decided what happens if they are executed on a non-existent object.

2.5.4 Renaming

As mentioned in section 2.4.4, we do not formalise the concept of renaming. Although it appears a rather straightforward concept, several questions arise, like what restrictions to impose on a renaming relation and how exactly renaming should be defined. In [22] renaming is investigated in more detail.

2.5.5 Black-box semantics

Attributes are in ISpec only considered a means to specify what really matters: traces of calls and returns. ISpec therefore distinguishes two kinds of semantics: **white-box semantics**, containing state information, and **black-box semantics** where no state information is present. The black-box semantics of a suite is obtained by projecting away the global-state information of the suite’s white-box semantics. In this thesis we only consider white-box semantics. Going from white-box to black-box semantics probably has profound consequences on specialisation and combination of suites.

2.5.6 Activation specifications

In ISpec the specification of a suite is complemented with an activation specification that corresponds to what is usually called “the main” in object-oriented languages. An activation specification can for example specify that a certain method is called every millisecond. We do not model activation specifications in this thesis. This is the reason why we used dotted lines for **Main**.

2.5.7 Keeping parent roles

In ISpec, in case a (child) suite is based on another (parent) suite, the child suite does not only contain the specialised versions of the roles of the parent suite, but also the non-specialised versions of these roles. If we keep non-specialised roles in a child suite, the notion of specialisation becomes part of a suite instead of only a matter between suites. It is then desirable that a method call does not contain a (statically fixed) role name, but that each object is explicitly associated with a set
of roles that determines the semantics of a method call on that object. In object-oriented programming this is referred to as dynamic binding, although there each object is associated with only one role (class) instead of a set of roles. We do not explicitly model dynamic binding in this thesis. However, in section 9.9.3 we do show that our model incorporates an implicit form of dynamic binding.

2.6 Conclusions

In this chapter we gave an informal description of the language of ISpec and its semantics. Chapter 11 defines a completely formal subset of the ISpec language and shows how the plug pattern and observer pattern can be formalised in this language. The formalisation of the ISpec language is far from complete as future-research section 2.5 shows, but already many interesting topics arise. In the upcoming chapters, the theory that we use for the formalisation is introduced in an incremental manner. Interesting topics are first (and sometimes only) investigated in the simplest models in which the topics can be treated.
Chapter 3

Relations

The primary mathematical formalism that is used in this thesis is the calculus of (binary) relations. In this chapter we describe the foundations of this calculus. Much of the theory in this chapter is well-known in the area of relational algebra. The purpose of this chapter is to fix the notation that we use and the exact meaning of this notation, in order to avoid any misinterpretations.

What, to the best of our knowledge, is new and might therefore deserve some extra attention, is the generalisation from functions to relations of sectioning (section 3.1.14) and the application relation (section 3.1.15), the spick and spack operators (section 3.3.8), the uniqueness and singleness properties (section 3.4.2) and the conjointness property (section 3.4.8). All these play an important role in our discussion of products in chapter 5.

3.1 Basics

We start with the introduction of some basic theory.

3.1.1 Predicate calculus

We assume that the reader is familiar with predicate calculus. The propositional operators that we use, are the conjunction $\land$, the disjunction $\lor$, the implication $\Rightarrow$, the consequence $\Leftarrow$, the equivalence $\equiv$, the booleans $false$ and $true$ and the negation $\neg$. For universal quantification and existential quantification we use the notation

$$\forall \langle x \mid p.x \mid q.x \rangle$$

$$\exists \langle x \mid p.x \mid q.x \rangle$$
The “p.x” tells which values the dummy x attains and the “q.x” tells what x satisfies. The following formulas are respectively equivalent to the above ones:

\[ \forall \langle x \mid p.x \Rightarrow q.x \rangle \]
\[ \exists \langle x \mid p.x \land q.x \rangle \]

### 3.1.2 Omission of brackets

We often omit brackets to make formulas more readable. We do not postulate any priority rules, but rely on common sense and often add some white space to show what is meant. We write for example \(a \land b \lor c\) if we mean \((a \land b) \lor c\) and \(a \land b \lor c\) if we mean \(a \land (b \lor c)\). In some cases, brackets can be removed without the need for extra white space. Because \((a \Rightarrow b) \Leftarrow c\) equals \(a \Rightarrow (b \Leftarrow c)\), we are allowed to write \(a \Rightarrow b \Leftarrow c\).

### 3.1.3 Omission of quantifications

If we write a formula like

\[ y^x \geq x+1 \Leftarrow x \geq 1 \land y \geq 2 \]

we actually mean that it holds for all real numbers x and y:

\[ \forall \langle x, y \mid x, y \in \mathbb{R} \mid y^x \geq x+1 \Leftarrow x \geq 1 \land y \geq 2 \rangle \]

We often assume that the reader is able to guess the free variables in a formula, the placement of universal quantifications over these free variables and the type restrictions on these free variables.

### 3.1.4 Basic elements

Some basic elements we distinguish are the booleans \texttt{false} and \texttt{true} and the default tags \texttt{0}, \texttt{1}, \texttt{2}, \ldots. For giving examples, it is also useful to have identifiers. We represent these elements in typewriter font: \texttt{x}, \texttt{piano}, \ldots. Also the real numbers \(0, \pi, \ldots\), a part of which are the integer numbers \(0, -1, \ldots\), a part of which are the natural numbers \(0, 1, \ldots\), are useful elements for examples.

For these basic elements, we assume that equality is properly defined. We write \(y = z\) for the boolean that indicates whether or not y and z are equal.

### 3.1.5 Collections

A collection is what would usually be called a “set”. We write
\( x \in C \)  

(\textit{x is-contained-by} \( C \)) for the boolean that indicates whether or not collection \( C \) contains element \( x \).  

The letters \( C \) and \( D \) usually denote collections.  

Equality between collections \( C \) and \( D \) is defined by
\[
C = D \equiv \forall \langle x \mid x \in C \equiv x \in D \rangle
\]

We use the word “collection” instead of “set” because we want to reserve the word “set” for a special kind of relations (see section 3.1.19).  

An important difference between our collections and the sets of basic set theory is that we do have a universal collection that appears to contain all elements. A type system prevents however that the wrong kinds of elements can be put together in one collection. A collection that has as elements both \( \odot \) and a collection containing this element, is for example prevented by this type system. See section 3.1.21 for details.

### 3.1.6 Connections

A \textit{connection} is what would usually be called a “pair”. A connection is written as  
\[
(y, z)
\]
where element \( y \) is called the \textbf{output} and element \( z \) the \textbf{input} of the connection. The reason for these names is explained in the next subsection.

Equality between connections \((y_0, z_0)\) and \((y_1, z_1)\) is defined by
\[
(y_0, z_0) = (y_1, z_1) \equiv y_0 = y_1 \land z_0 = z_1
\]

### 3.1.7 Relations

A \textit{relation} is a collection of connections. Instead of
\[
(y, z) \in R
\]
we usually write
\[
y (R) z
\]
(\( R \) connects output \( y \) with input \( z \)).
The letters $P$, $Q$, $R$, $S$, $T$, $U$ and $V$ usually denote relations.

How equality between relations is defined, follows from the definition of equality for collections and connections:

$$R = T \equiv \forall(y, z \in y(R)z \equiv y(T)z)$$

From our definition of a relation, it is clear that we are not defining an abstract relational algebra that admits several interpretations. If we say that we define some relation by a certain formula, it means that the formula fixes for each $y$ and $z$ whether $y(R)z$ holds or not.

Calling $y$ the output and $z$ the input of a connection $(y, z)$, is the result of our preference to write $R.z$ for the application of a relation $R$ to an element $z$. The application is described in section 3.1.10.

There are three ways to picture relations that help in understanding the theory about them:

In the first picture, we write all output elements on the left and all input elements on the right. A line between an input element and an output element represents a connection. The graph thus created, is an undirected bipartite graph.

In the second picture, we only have one instance of each element and use arrows from inputs to outputs to represent connections, obtaining a directed graph.

The third picture represents a relation by a dot matrix where the input elements are placed on the bottom and the output elements on the left. A dot represents a connection.

How useful a certain representation of a relation is for one’s understanding, depends on the context in which the relation is used.

### 3.1.8 Emptiness and fullness

A relation $R$ is empty in an element $z$ if $R$ connects input $z$ with no output:
\neg \exists \langle y \mid y \ (R) \ z \rangle

and \textbf{full in} an element \( z \) if \( R \) connects input \( z \) with every output:

\forall \langle y \mid y \ (R) \ z \rangle

\subsection*{3.1.9 Functionality and totality}

A relation \( R \) is \textbf{functional in} an element \( z \) if it connects input \( z \) with at most one output:

\forall \langle x, y \mid x \ (R) \ z \wedge y \ (R) \ z \mid x = y \rangle

it is \textbf{total in} an element \( z \) if it connects input \( z \) with at least one output:

\exists \langle y \mid y \ (R) \ z \rangle

and \textbf{single-valued in} \( z \) if it connects input \( z \) with exactly one output, or equivalently, if it is functional and total in \( z \).

We leave it to the reader to verify that functionality of \( R \) in \( z \) can be written as

\exists \langle y \mid \forall (x \mid x \ (R) \ z \Rightarrow x = y) \rangle

totality of \( R \) in \( z \) as

\exists \langle y \mid \forall (x \mid x \ (R) \ z \Leftarrow x = y) \rangle

and single-valuedness of \( R \) in \( z \) as

\exists \langle y \mid \forall (x \mid x \ (R) \ z \equiv x = y) \rangle

A relation \( R \) is \textbf{functional} if it connects each input with at most one output:

\forall \langle x, y, z \mid x \ (R) \ z \wedge y \ (R) \ z \mid x = y \rangle

it is \textbf{total} if it connects each input with at least one output:

\forall \langle z \mid \exists \langle y \mid y \ (R) \ z \rangle \rangle

and \textbf{single-valued} if it connects each input with exactly one output:

\forall \langle z \mid \exists \langle y \mid \forall (x \mid x \ (R) \ z \equiv x = y) \rangle \rangle

A functional relation is called a \textbf{function}. The letters \( f, g, h \) and \( k \) usually denote functions.
Notice that a function is empty in $z$ exactly when it is not single-valued in $z$.

We purposely do not use “defined in” for “single-valued in” because “not defined in some input” means that it is left open which outputs are connected with that input. A reason could be that we do not know what the best choice is for such inputs.

3.1.10 Application

If a relation $R$ is single-valued in some element $z$ with output $y$, then the application of $R$ to $z$, denoted by $R.z$, is equal to $y$.

3.1.11 Tuples

An $n$-tuple is a function that is single-valued in exactly the elements $\langle 0 \rangle, \ldots, \langle n-1 \rangle$. The $n$-tuple for which the respective outputs connected to these elements are $x_0$, $\ldots$, $x_{n-1}$, is denoted by

$$\langle x_0, \ldots, x_{n-1} \rangle$$

Equality between $n$-tuples follows from their definition in terms of relations:

$$\langle y_0, \ldots, y_{n-1} \rangle = \langle z_0, \ldots, z_{n-1} \rangle \equiv y_0 = z_0 \land \ldots \land y_{n-1} = z_{n-1}$$

A 2-tuple is called a pair.

3.1.12 Arguments

For $n \geq 0$, an $n$-argument relation is a relation whose inputs are $n$-tuples. The elements in the tuples are called arguments.

3.1.13 Fix notation

For a relation that is denoted by a symbol $\circ\ldots\circ$ containing exactly $n$ “$\circ$”s ($n \geq 1$), we allow to write $z_0\circ\ldots\circ z_{n-1}$ instead of $\langle \circ\ldots\circ \rangle(z_0, \ldots, z_{n-1})$ in case $n \geq 2$ and instead of $\circ\ldots\circ z_0$ in case $n = 1$. We call this fix notation. For a symbol of the form $\circ$, it is called prefix notation, for a symbol of the form $\circ\circ$ postfix notation and for a symbol of the form $\circ\circ\circ$ infix notation. We often omit the “$\circ$”s of a symbol if it is clear which symbol is meant, talking for example about / instead of $\sqrt{\circ\circ\circ}$
3.1.14 Sections

From each $n$-argument relation, a relation is obtained by fixing some (but not all) arguments. The result is called a section of the original relation.

When a relation is denoted by a symbol with "\$\$"s, we use the convention to replace the "\$" of each argument that should be fixed, by the element it should be fixed to. For a relation $\langle [\_], \_ \rangle$, the sections $z_0 [\_]$ and $z_0 [\_] z_2$ are for example defined by

\[
y \left( z_0 [\_] \right) (z_1, z_2) \equiv y \left( [\_], z_1, z_2 \right)
y \left( z_0 [\_] z_2 \right) z_1 \equiv y \left( [\_], z_0, z_1, z_2 \right)
\]

Notice that, in accordance with fix notation, when only 1 argument is not fixed, we do not use a 1-tuple but ‘its’ element as input.

For readability, we write $x \circ \_$ also as $(x \circ)$ and $\_ \circ x$ as $(\circ x)$ if this notation is unambiguous (if we do not use the symbols $\circ \_$ and $\_ \circ$).

3.1.15 Application relation

The application $\_ \_ \_ \_$ is a 2-argument relation, defined by

\[
y \langle \_ \_ \_ \_ \rangle \langle R, z \rangle \equiv y \langle R \rangle z
\]

Notice that for an $R$ that is single-valued in $z$, $R.z$ (the meaning of which we gave in section 3.1.10) equals $\langle \_ \_ \_ \_ \rangle \langle R, z \rangle$, as should be the case according to fix notation (see section 3.1.13).

The application has for each relation $R$ a section $(\_ \_ \_ \_ \_ \_)$ and for each element $z$ a section $(\_ \_ \_ \_ \_ \_ \_ \_ \_)$ of $z$. If $R$ is single-valued in $z$, we have

\[
(R_.) z = R.z
(z_.) R = R.z
\]

The equation

\[
(R_.) = R
\]

even holds for arbitrary relations $R$.

3.1.16 Predicates

A relation is called a predicate if it is functional and only outputs booleans.

The letters $p$ and $q$ usually denote predicates.
3.1.17 Binary predicates

A binary predicate is a 2-argument predicate.

Each binary predicate \( \simeq \) has a corresponding negated binary predicate, denoted by a strike-through symbol \( \not\simeq \), and defined by

\[
b (\not\simeq) \langle y, z \rangle \equiv \neg b (\simeq) \langle y, z \rangle
\]

As mentioned in section 3.1.3, we do not always explicitly state the types of the free variables in a formula. The \( b \) in this equation ranges for example over the booleans, which we assume to be clear from the fact that \( \not\simeq \) is a binary predicate.

A binary predicate \( \simeq \) defines a relation \( \simeq \) by the equation

\[
y (\simeq) z \equiv \text{true} (\simeq) \langle y, z \rangle
\]

This allows us to use \( y (\simeq) z \) and \( y \simeq z \) interchangeably if \( \simeq \) is single-valued in \( \langle y, z \rangle \).

3.1.18 Equality

The is-equal-to \( = \) is a binary predicate that is single-valued in all pairs. The relation \( = \) that corresponds to \( = \) is called the equality relation.

3.1.19 Sets

A relation \( R \) is a set if it only connects equal elements:

\[
y (R) z \Rightarrow y = z
\]

The letters \( A \) and \( B \) usually denote sets.

Notice that a set is a function.

3.1.20 Element-of and owner-of

The binary predicate is-element-of \( \in \) is exactly single-valued in all pairs \( \langle x, A \rangle \) where \( A \) is a set. On such pairs it is defined by

\[
x \in A \equiv x (A) x
\]

The binary predicate is-owner-of \( \ni \) is exactly single-valued in all pairs \( \langle A, x \rangle \) where \( A \) is a set. On such pairs it is defined by
\[ A \ni x \equiv x(A)x \]

The corresponding relations ∈ and \ni are called the **element-of relation** and **owner-of relation** respectively.

We often use \( x_0, \ldots, x_n \in A \) as a shorthand for \( x_0 \in A \land \ldots \land x_n \in A \).

### 3.1.21 Types

Suppose we would define the set Russell by

\[ x \in \text{Russell} \equiv x \notin x \]

We then would have

\[ \text{Russell} \in \text{Russell} \equiv \text{Russell} \notin \text{Russell} \]

which means that true would equal false. We assume to have an **external type system** that rejects these kind of formulas by distinguishing several types of elements. It should support **implicit typing** (terms do not contain explicit type information), **polymorphism** (types can contain variables) and **dependent typing** (the type of the output of a relation can depend on the input or the type of the input of a relation can depend on the output. Furthermore, the type of an element in a tuple can depend on other elements in the tuple where cyclic dependencies are disallowed). We also assume that each type in this type system contains at least one element. Formalisation of this type system is left for future research. We hope common sense to prevent type errors in our case.

Next to external typing, we also do typing **within** the language to classify certain types of elements, using relations as **internal types**. If \( x(R)x \) holds for some relation \( R \), we say that \( R \) is a **type** of \( x \). We mainly use sets as internal types.

We now introduce some elementary internal types. We assume \( A(i) \) and \( B \) to be sets.

The set \( B \) consists of the booleans \textit{false} and \textit{true}, the set \( 2 \) consists of the default tags \( \ghto \) and \( \gghto \), the set \( I \) consists of all identifiers, the set \( R \) consists of all real numbers, the set \( Z \) consists of all integer numbers and the set \( N \) consists of all natural numbers:

\[
\begin{align*}
    x \in B & \equiv x = \text{false} \lor x = \text{true} \\
    x \in 2 & \equiv x = \ghto \lor x = \gghto \\
    x \in I & \equiv x = x \lor x = \text{piano} \lor \ldots \\
    x \in R & \equiv x = 0 \lor x = \pi \lor \ldots \\
    x \in Z & \equiv x = 0 \lor x = -1 \lor \ldots \\
    x \in N & \equiv x = 0 \lor x = 1 \lor \ldots
\end{align*}
\]
The set $I$, called the **universal set**, consists of all elements (that the external type system allows it to consist of):

$$x \in I \equiv true$$

The set $P_A$ consists of all collections with elements of type $A$:

$$C \in P_A \equiv \forall \langle x | x \in C \land x \in A \rangle$$

Although the formulas $x \in I$ and $x \in PI$ are both equivalent to $true$, the external type system assigns a more restricted type to variable $x$ in the last case, forcing $x$ to be a collection.

The set $A \star B$ consists of all connections with outputs of type $A$ and inputs of type $B$:

$$(y, z) \in A \star B \equiv y \in A \land z \in B$$

In case we write $x \in I \star I$, the external type system forces $x$ to be connection, meaning that $x = (y, z)$ for some $y$ and $z$. The formula $x \in I \star I$ is however again equivalent to $true$.

The set of relations $P(A \star B)$ is also written as $A \rightarrow B$. It consists of all relations whose outputs are elements of $A$ and whose inputs are elements of $B$:

$$R \in A \rightarrow B \equiv \forall \langle y, z | y \in A \land z \in B \rangle$$

The set $A \leftarrow B$ consists of all functions that are single-valued in exactly every element of $B$ and whose outputs are elements of $A$:

$$R \in A \leftarrow B \equiv R \in A \rightarrow B \land \forall \langle z | z \in B \land \exists \langle x | x \in A \land R \equiv x = y \rangle \rangle$$

The set $\wp A$ consists of all sets whose elements are elements of $A$:
\[ R \in \varphi A \]
\[ \equiv \forall \langle y, z \mid y (R) z \mid y (A) z \rangle \]

For \( n \geq 2 \), the set \( A_0 \times \ldots \times A_{n-1} \) consists of all \( n \)-tuples \( \langle x_0, \ldots, x_{n-1} \rangle \) where \( x_i \in A_i \):

\[ R \in A_0 \times \ldots \times A_{n-1} \]
\[ \equiv \exists \langle x_0, \ldots, x_{n-1} \mid R = \langle x_0, \ldots, x_{n-1} \rangle \mid x_0 \in A_0 \land \ldots \land x_{n-1} \in A_{n-1} \rangle \]

3.1.22 Enumeration

For \( n \geq 0 \), the collection enumeration \( [x_0, \ldots, x_{n-1}] \) is the collection that contains exactly the elements \( x_0, \ldots, x_{n-1} \):

\[ x \in [x_0, \ldots, x_{n-1}] \]
\[ \equiv x = x_0 \lor \ldots \lor x = x_{n-1} \]

A special case is the relation enumeration \( [(y_0, z_0), \ldots, (y_{n-1}, z_{n-1})] \) that contains exactly the connections between output \( y_i \) and input \( z_i \) for \( 0 \leq i < n \):

\[ y ([(y_0, z_0), \ldots, (y_{n-1}, z_{n-1})]) z \]
\[ \equiv (y = y_0 \land z_0 = z) \lor \ldots \lor (y = y_{n-1} \land z_{n-1} = z) \]

For \( n \geq 0 \), the set enumeration \( \{x_0, \ldots, x_{n-1}\} \) is the set that consists exactly of the elements \( x_0, \ldots, x_{n-1} \):

\[ \{x_0, \ldots, x_{n-1}\} = [(x_0, x_0), \ldots, (x_{n-1}, x_{n-1})] \]

For \( n \geq 0 \), the tuple \( \langle x_0, \ldots, x_{n-1} \rangle \) is the function that is exactly single-valued in each \( \odot \) for \( 0 \leq i < n \), with respective output \( x_i \):

\[ \langle x_0, \ldots, x_{n-1} \rangle = [(x_0, \odot), \ldots, (x_{n-1}, \odot)] \]

3.1.23 Comprehension

For a total predicate \( p \) and a relation \( f \) that is single-valued in every \( x \) that satisfies \( p.x \), the collection comprehension \( [f.x \mid p.x \mid x] \) is the collection that contains exactly the elements \( f.x \) for all \( x \) satisfying \( p.x \):
\[ w \in \{ f.x \mid p.x \mid x \} \]
\[
= \exists \langle x \mid p.x \mid w = f.x \rangle
\]

The collection comprehension is divided into three parts, separated by “\|”s. We call the left part the **range part**, the middle part the **domain part** and the right part the **dummy part**. The dummy part contains the name of the dummy that is used in the comprehension, the domain part tells which values the dummy attains and the range part tells for each of these values, which element is added to the collection.

A special case of the collection comprehension is the **relation comprehension**. For a total predicate \( p \) and relations \( g \) and \( h \) that are single-valued in every \( x \) that satisfies \( p.x \), the relation comprehension \( \{ (g.x, h.x) \mid p.x \mid x \} \) is the relation that consists exactly of the connections between output \( g.x \) and input \( h.x \) for all \( x \) satisfying \( p.x \):

\[
y (\{ (g.x, h.x) \mid p.x \mid x \}) z \]
\[
= \exists \langle x \mid p.x \mid y = g.x \land h.x = z \rangle
\]

For a total predicate \( p \) and a relation \( f \) that is single-valued in all \( x \) satisfying \( p.x \), the **set comprehension** \( \{ f.x \mid p.x \mid x \} \) is the set that consists exactly of the elements \( f.x \) for all \( x \) satisfying \( p.x \):

\[
\{ f.x \mid p.x \mid x \} = \{ (f.x, f.x) \mid p.x \mid x \}
\]

and the **function comprehension** \( \langle f.x \mid p.x \mid x \rangle \) is the function that is single-valued in exactly each \( x \) satisfying \( p.x \), with respective output \( f.x \):

\[
\langle f.x \mid p.x \mid x \rangle = \{ (f.x, x) \mid p.x \mid x \}
\]

For those cases where it is customary to put the dummy part on the left and the range part on the right, we define the **mirrored function comprehension** \( \langle x \mid p.x \mid f.x \rangle \) by

\[
\langle x \mid p.x \mid f.x \rangle = \langle f.x \mid p.x \mid x \rangle
\]

This notation is mainly used in combination with \( \forall \) and \( \exists \).

An omitted domain part means the same as “true” in the domain part.

We allow the use of a tuple of dummies instead of one dummy. We usually omit the brackets of a tuple in the dummy part. The predicate \( p \) then only needs to be single-valued in all \( n \)-tuples where \( n \) is the number of dummies.
3.1.24  Non-strict evaluation

The formula $R.z$ has a **single-valued semantics** if $R$ is single-valued in $z$. Formulas that do not have a single-valued semantics cannot be evaluated, so we do not have special elements like $\perp$ to represent the semantics of such formulas. A formula that has a subformula that is not single-valued, can usually also not be evaluated. There are a few exceptions however, like the comprehension notation where we allow a formula like

$$\{x/x \mid x \in \mathbb{R} \land x \neq 0 \mid x\}$$

Although the formula $x/x$ does not have a single-valued semantics for $x = 0$, the formula $\{x/x \mid x \in \mathbb{R} \land x \neq 0 \mid x\}$ can still be evaluated and is equal to $\{1\}$.

Operators that enable one to construct formulas with single-valued semantics from formulas with not necessarily single-valued semantics, are called **non-strict**. The conjunction $\land$, disjunction $\lor$, implication $\Rightarrow$ and consequence $\Leftarrow$ are examples of such operators. For a formula $E$ whose semantics is possibly not single-valued and possibly not even boolean-valued, we define

$$false \land E \equiv false \quad false \Rightarrow E \equiv true$$

$$E \land false \equiv false \quad E \Rightarrow true \equiv true$$

$$true \lor E \equiv true \quad true \Leftarrow E \equiv true$$

$$E \lor false \equiv true \quad E \Leftarrow false \equiv true$$

The division $\frac{x}{x}$ is not single-valued in $(0,0)$, so for $x = 0$, the formula $x/x$ cannot be evaluated and also the formula $x/x = 1$ cannot be evaluated. However, the formula $x = 0 \lor x/x = 1$ can be evaluated for $x = 0$ and is equal to $true$.

An example that shows the usefulness of the fact that $false \land E$ can even be evaluated if $E$ is not boolean-valued, is the formula $x \in B \land x$. For example, the formula $x \in B \land x$ is equal to $false$ (assuming that the external type system would allow $x$ to be equal to $piano$ in the first place).

In chapter 6 we show how formulas that evaluate non-strictly can be formalised.

3.1.25  Feijen notation

For binary predicates $\simeq_{\omega}$ we write the formula

$$x_0 \simeq_{0} x_1 \land x_1 \simeq_{1} x_2 \land \ldots \land x_n \simeq_{n} x_{n+1}$$

also as
We call this Feijen notation after Wim Feijen to whom this notation (in the context of proofs) is attributed [16]. In this thesis the notation is used to present proofs as well as (for $n = 0$) definitions and theorems. For proofs, each $x_i \simeq_{i} x_{i+1}$ is called a step within the proof. We often put a comment $\{\ldots\}$ to hint why a certain proof step holds.

We use a bullet “•” to indicate extra assumptions. The formula

$$y^x \geq x+1$$

$$\Leftarrow \{\bullet x \geq 1\}$$

$$y \geq 2$$

is for example equivalent to

$$(y^x \geq x+1 \Leftarrow y \geq 2) \Leftarrow x \geq 1$$

Interpreting this formula as a theorem that holds for all $x, y \in \mathbb{R}$, a proof of it could be as follows:

$$y^x$$

$$\geq \{\bullet x \geq 0 \land y \geq 2\}$$

$$2^x$$

$$\geq \{\bullet x \geq 1\}$$

$$x+1$$

Sometimes we omit the ‘new lines’ in this notation and write for example $x \leq y \leq z$. This is not always unambiguous (an example being $a \equiv b \equiv c$) and we only do this if we are convinced that it is clear what we mean.

### 3.1.26 Axiom of choice

The axiom of choice says that for predicates $p \in \mathbb{B} \hookrightarrow I$ and $q \in \mathbb{B} \hookrightarrow I \times I$, the fact that there exists a $z$ satisfying $q.(y,z)$ for each $y$ satisfying $p.y$, is equivalent to the fact that there exists a choice function $f$ that gives us such a $z$ for each $y$.petto
satisfying $p.y$:

$$\forall\langle y \mid p.y \mid \exists\langle z \mid q.(y,z)\rangle\rangle$$

$$\equiv$$

$$\exists\langle f \mid f \in I \leftrightarrow \{x \mid p.x \mid x\} \mid \forall\langle y \mid p.y \mid q.(y,f.y)\rangle\rangle$$

Take for example the formula

$$\forall\langle y \mid y \in \mathbb{N} \mid \exists\langle z \mid z \in \mathbb{N} \land y < z\rangle\rangle$$

The axiom of choice tells us that this formula is equivalent to

$$\exists\langle f \mid f \in I \leftrightarrow \mathbb{N} \mid \forall\langle y \mid y \in \mathbb{N} \mid f.y \in \mathbb{N} \land y < f.y\rangle\rangle$$

A choice function $f$ that shows that this formula holds, is

$$\langle x+1 \mid x \in \mathbb{N} \mid x\rangle$$

The axiom of choice is used in section 5.4.6 for what we consider the most intricate proof of this thesis.

### 3.1.27 Point-wise against point-free

The theory in this section deals with things at a point-wise level. In sections 3.2 and 3.3 we introduce operators that enable us to raise the abstraction level to point-freeness. Point-freeness means that we manipulate with collections/relations directly without recourse to variables that represent elements that are contained by them. Point-free formulas are more concise and therefore easier to read and manipulate. Although point-free formulas may sometimes at first be more difficult to interpret, they guide one to start thinking at a higher level of abstraction. Section 3.8.4 shows how point-freeness can be taken to an extreme.

A disadvantage of raising the abstraction level notionally, is an explosive growth in the number of operators and theorems. A quick browse through the index shows that the number of operators that are introduced in this thesis is huge. We tried to choose the symbols of operators in such a way that important structures that are present in the calculus, become apparent. We sometimes even only introduce operators for the mere purpose of revealing these structures. We hope that this helps in remembering operators and their meaning.

### 3.2 Collection operators

In this section we introduce some standard operators on collections.
3.2.1 Inclusion

The is-subcollection-of and is-supercollection-of \( \subseteq, \supseteq \) are defined by

\[
\begin{align*}
C \subseteq D &\equiv \forall \langle x \parallel x \in C \Rightarrow x \in D \rangle \\
C \supseteq D &\equiv \forall \langle x \parallel x \in C \Leftarrow x \in D \rangle
\end{align*}
\]

In general, the term inclusion is used to refer to these two binary predicates and their corresponding relations. The relations \( \subseteq \) and \( \supseteq \) are called the subcollection-of relation and supercollection-of relation respectively.

In case \( C \) and \( D \) are relations, we talk about subrelations and superrelations. In case they are sets, we talk about subsets and supersets.

Two theorems that are referred to as indirect inequality are

\[
\begin{align*}
C \subseteq D &\equiv \forall \langle X \parallel X \subseteq C \Rightarrow X \subseteq D \rangle \\
C \subseteq D &\equiv \forall \langle X \parallel C \subseteq X \Leftarrow D \subseteq X \rangle
\end{align*}
\]

For sets \( A \) and \( B \) we also have

\[
\begin{align*}
A \subseteq B &\equiv \forall \langle x \parallel x \in A \Rightarrow x \in B \rangle \\
A \supseteq B &\equiv \forall \langle x \parallel x \in A \Leftarrow x \in B \rangle
\end{align*}
\]

and the following two indirect-inequality theorems:

\[
\begin{align*}
A \subseteq B &\equiv \forall \langle X \mid X \in \mathcal{P}I \parallel X \subseteq A \Rightarrow X \subseteq B \rangle \\
A \subseteq B &\equiv \forall \langle X \mid A \subseteq X \Leftarrow B \subseteq X \rangle
\end{align*}
\]

3.2.2 Equality

As already mentioned in section 3.1.5, the is-equal-to \( = \in \mathcal{B} \leftarrow I \times I \) is between collections \( C \) and \( D \) defined by

\[
C = D \equiv \forall \langle x \parallel x \in C \equiv x \in D \rangle
\]

A point-free definition (as opposed to the above point-wise definition) is

\[
C = D \equiv C \subseteq D \land D \subseteq C
\]

Two point-free definitions that are called indirect equality are

\[
\begin{align*}
C = D &\equiv \forall \langle X \parallel X \subseteq C \equiv X \subseteq D \rangle \\
C = D &\equiv \forall \langle X \parallel C \subseteq X \equiv D \subseteq X \rangle
\end{align*}
\]

For sets \( A \) and \( B \) we also have
\[ A = B \equiv \forall (x \parallel x \in A \equiv x \in B) \]

and the following two indirect-equality theorems:

\[ A = B \equiv \forall (X \parallel X \in \wp \parallel X \subseteq A \equiv X \subseteq B) \]
\[ A = B \equiv \forall (X \parallel X \in \wp \parallel A \subseteq X \equiv B \subseteq X) \]

### 3.2.3 Binary intersection and union

The binary intersection and binary union \(\cap, \cup\) \(\in \wp \leftrightarrow \wp \times \wp\) are defined by

\[ x \in C \cap D \equiv x \in C \land x \in D \]
\[ x \in C \cup D \equiv x \in C \lor x \in D \]

or point free:

\[ X \subseteq C \cap D \equiv X \subseteq C \land X \subseteq D \]
\[ C \cup D \subseteq X \equiv C \subseteq X \land D \subseteq X \]

Conversely, if we assume that \(\cap\) and \(=\) are already defined, we can define \(\subseteq\) by

\[ C \subseteq D \equiv C \cap D = C \]

or if we assume that \(\cup\) and \(=\) are already defined, we can define \(\subseteq\) by

\[ C \subseteq D \equiv C \cup D = D \]

### 3.2.4 Then and when

The then and when \(\Rightarrow, \Leftarrow\) \(\in \wp \leftrightarrow \wp \times \wp\) are defined by

\[ x \in C \Rightarrow X \equiv x \in C \Rightarrow x \in X \]
\[ x \in X \Leftarrow D \equiv x \in X \Leftarrow x \in D \]

or point free:

\[ D \subseteq C \Rightarrow X \equiv C \cap D \subseteq X \]
\[ C \subseteq X \Leftarrow D \equiv C \cap D \subseteq X \]

### 3.2.5 Removed-from and without

The removed-from and without \(\setminus, \\not\subset\) \(\in \wp \leftrightarrow \wp \times \wp\) are defined by
\[ x \in C \setminus X \equiv x \notin C \land x \in X \]
\[ x \in X \setminus D \equiv x \in X \land x \notin D \]

or point free:
\[ C \setminus X \subseteq D \equiv X \subseteq C \cup D \]
\[ X \setminus D \subseteq C \equiv X \subseteq C \cup D \]

### 3.2.6 Same and other

The \textit{same} and \textit{other} \( \equiv \not\equiv \) \( \in \Pi \leftrightarrow \Pi \times \Pi \) are defined by
\[ x \in C \equiv D \equiv x \in C \equiv x \in D \]
\[ x \in C \not\equiv D \equiv x \in C \not\equiv x \in D \]

or in terms of previously introduced operators:
\[ C \equiv D = C \equiv D \cap C \equiv D \]
\[ C \not\equiv D = C \not\equiv D \cup \neg C \not\equiv D \]

### 3.2.7 Empty and full collection

The \textit{empty collection} and \textit{full collection} \( \emptyset, \Pi \in \Pi \) are defined by
\[ x \in \emptyset \equiv \text{false} \]
\[ x \in \Pi \equiv \text{true} \]

or point free:
\[ \emptyset \subseteq X \]
\[ X \subseteq \Pi \]

In case \( \emptyset \) and \( \Pi \) are relations, we talk about the \textit{empty relation} (or \textit{empty set}) and \textit{full relation} respectively.

### 3.2.8 Complement

The \textit{complement} \( \not\in \) \( \in \Pi \leftrightarrow \Pi \) is defined by
\[ x \in \not\in C \equiv x \notin C \]

Two point-free definitions are
\[ C \subseteq \not\in D \equiv C \cap D \subseteq \emptyset \]
\[ \not\in D \subseteq C \equiv \Pi \subseteq C \cup D \]
The complement can also be directly defined by one of the following four equations:

\[
\neg C = C \supset \emptyset \\
\neg C = \emptyset \supseteq C \\
\neg C = C \supset \Pi \\
\neg C = \Pi \setminus C
\]

Conversely, if we assume that \(\neg\) is already defined, we can define \(\supset, \subseteq, \setminus\) and \(\setminus\) by

\[
C \supset X = \neg C \cup X \\
X \subseteq D = X \cup \neg D \\
C \setminus X = \neg C \cap X \\
X \setminus D = X \cap \neg D
\]

### 3.2.9 Arbitrary intersection and union

The arbitrary intersection and arbitrary union \(\bigcap, \bigcup \in \mathcal{P} \leftarrow \varphi(\mathcal{P})\) are defined by

\[
x \in \bigcap W \equiv \forall (C \mid C \in W \mid x \in C) \\
x \in \bigcup W \equiv \exists (C \mid C \in W \mid x \in C)
\]

or point free:

\[
X \subseteq \bigcap W \equiv \forall (C \mid C \in W \mid X \subseteq C) \\
\bigcup W \subseteq X \equiv \forall (C \mid C \in W \mid C \subseteq X)
\]

### 3.2.10 Notational considerations

The similarity between the collection-operator symbols and boolean-operator symbols is not a coincidence of course. To show this similarity even better, some people use the symbols \(\Lambda\) and \(\lor\) instead of \(\forall\) and \(\exists\) respectively.

In the light of the notation for negated binary predicates (see section 3.1.17), a nice notation for the full collection would be \(\emptyset\). This notation would also be consistent with the \(Z\)-override relation \(\bigcirc\) that we introduce in section 7.6. We chose \(\Pi\) however because deviating too much from common notation makes formulas less readable for people from the field.

In most texts about relational algebra, the complement is usually denoted by the symbol \(\neg\). The reason why we use different symbols for negation (\(\neg\)) and complement (\(\neg\)) is type checking. The negation is only single-valued for elements of \(\mathbb{B}\) whereas the complement is single-valued in all elements of \(\mathcal{P}I\). Even if we identify booleans and collections (\(false = \emptyset\) and \(true = \Pi\)), we may still only write \(\neg X\) instead of \(\neg X\) if we know that \(X\) is an element of \(\{\emptyset, \Pi\}\). The same goes for other ‘sharp-edged’ against ‘round-edged’ operators.
3.3 Relation operators

Next to the operators of the previous section that deal with arbitrary collections, we also have several operators specifically for relations.

3.3.1 Sequential composition

The sequential composition \( S \circ P \) is defined by

\[
y(S \circ P) z \equiv \exists x \ y(S) x \land x(P) z
\]

If \( P \) is single-valued in \( z \) and \( S \) is single-valued in \( P.z \), we have

\[
(S \circ P).z = S.(P.z)
\]

3.3.2 Over and under

The over and under \( S \uparrow \downarrow R \) are defined by

\[
y(R/P) x \equiv \forall z \ y(R) z \leftarrow x(P) z
\]

\[
x(S \downarrow R) z \equiv \forall y \ y(S) x \Rightarrow y(R) z
\]

or point free:

\[
S \subseteq R/P \equiv S \circ P \subseteq R
\]
\[
P \subseteq S \downarrow R \equiv S \circ P \subseteq R
\]

Notice the difference between the symbols for the ‘over’ and ‘under’ (\( / \) and \( \backslash \)) and the ones for the ‘removed-from’ and ‘without’ (\( \setminus \) and \( \backslash \)).

3.3.3 Identity relation

The universal set \( I \) is also called the identity relation. The following definition shows that it is actually just another notation for the equality relation:

\[
y(I) z \equiv y = z
\]

Four point-free definitions are

\[
X \cdot I = X
\]
\[
I \cdot X = X
\]
\[
X / I = X
\]
\[
I \backslash X = X
\]
3.3.4 Domain and range

The domain and range \( \varphi \) of \( \pi \) \( (I) \) are defined by

\[
\begin{align*}
z \in R^\varphi & \equiv \exists y \parallel y(R) = z \\
y \in R^\vartheta & \equiv \exists z \parallel z(R) = y
\end{align*}
\]

A point-free definition is that for all sets \( A \) and \( B \)

\[
\begin{align*}
R^\varphi \subseteq B & \equiv R \subseteq \Pi^\varphi B \\
R^\vartheta \subseteq A & \equiv R \subseteq \Pi^\varvartheta A
\end{align*}
\]

If \( R^\varphi \subseteq B \) holds, we say that \( B \) is an upper domain of \( R \) and if \( R^\vartheta \subseteq A \) holds, we say that \( A \) is an upper range of \( R \).

Another point-free definition is that for all sets \( A \) and \( B \)

\[
\begin{align*}
B \subseteq R^\varphi & \equiv B \subseteq \Pi^\varphi R \\
A \subseteq R^\vartheta & \equiv A \subseteq R^\varvartheta
\end{align*}
\]

If \( B \subseteq R^\varphi \) holds, we say that \( B \) is a lower domain of \( R \) and if \( A \subseteq R^\vartheta \) holds, we say that \( A \) is a lower range of \( R \).

Notice that the first point-free definition already implies that \( R^\varphi \) and \( R^\vartheta \) are sets (take \( I \) for \( A \) and \( B \)). So the type constraint \( \varphi \in \Pi \pi \) \( (I) \) \( (I) \) would be strong enough to form together with these equations a complete definition.

3.3.5 Converse

The converse \( \Diamond \) of \( \pi \) \( (I) \) \( (I) \) is defined by

\[
\begin{align*}
z(R^\Diamond) y & \equiv y(R) z
\end{align*}
\]

The following two equations together also form a definition of \( \Diamond \):

\[
\begin{align*}
R^\Diamond \subseteq X & \equiv R \subseteq X^\Diamond \\
(S^\Diamond P^\Diamond) & = P^\Diamond S^\Diamond
\end{align*}
\]

3.3.6 Iteration

The iteration \( * \) of \( \pi \) \( (I) \) \( (I) \) is defined by

\[
\begin{align*}
y(R^*) z & \equiv \forall (X \parallel R \subseteq X \cap I \subseteq X \land X^\Diamond X \subseteq X \parallel y(X) z)
\end{align*}
\]

or completely point free:
\[ T \subseteq R^* \equiv \forall \langle X | R \subseteq X \land I \subseteq X \land X \cdot X \subseteq X \land T \subseteq X \rangle \]

Another point-free definition is given by the following two equations:

\[
\begin{align*}
I \subseteq R^* \land R^* \cdot R^* \subseteq R^* \\
I \subseteq X \land X \cdot X \subseteq X \Rightarrow R^* \subseteq X \equiv R \subseteq X
\end{align*}
\]

The following three theorems are called the \textbf{leapfrog} rules:

\[
\begin{align*}
R^* \cdot S \subseteq S \cdot T^* & \iff R \cdot S \subseteq S \cdot T \\
R^* \cdot S \supseteq S \cdot T^* & \iff R \cdot S \supseteq S \cdot T \\
R^* \cdot S = S \cdot T^* & \iff R \cdot S = S \cdot T
\end{align*}
\]

### 3.3.7  Power

For \( n \in \mathbb{N} \), the \( n \text{th} \) \textbf{power} \( n \in (I \rightarrow I) \leftarrow (I \rightarrow I) \) is defined by

\[
\begin{align*}
R^0 & = I \\
R^1 & = R \\
R^{i+j} & = R^i \cdot R^j
\end{align*}
\]

### 3.3.8  Spick and spack

The \textbf{spick} \( \blacklozenge \in \wp(I \times I) \) is defined by

\[
y \in R \blacklozenge x \equiv y \in R \cdot x
\]

and the \textbf{spack} \( \blacktriangleleft \in (I \rightarrow I) \leftarrow \wp I \times I \) is defined by

\[
y \in A \blacktriangleleft z \equiv y \in A \land z = x
\]

The set \( R \blacklozenge x \) consists of all elements that \( R \) connects to input \( x \) and \( A \blacktriangleleft x \) is the relation that consists exactly of the connections that connect input \( x \) to an element of \( A \).

The reason for names “spick” and “spack” is that they can be seen as simple operators that resemble the pick and pack that are introduced in chapter 5.

### 3.3.9  Constant function

The \textbf{constant function} \( K \in (I \leftarrow I) \leftarrow I \) is defined by

\[
(K,y).z = y
\]
For an element \( x \), \( K \cdot x \) is called the **constant-\( x \) function**.

### 3.3.10 Doubling function

The **doubling function** \( \Delta \in I \times I \leftrightarrow I \) is defined by

\[
\Delta \cdot x = \langle x, x \rangle
\]

### 3.3.11 Everywhere and somewhere

The **everywhere** and **somewhere** everywhere, somewhere \( \in P I \leftrightarrow P I \) are defined by (we borrowed the names for these operators from [15])

\[
x \in \text{everywhere}.C \equiv \forall \langle x \mid x \in C \rangle \quad \quad \text{everywhere}.R \equiv \Pi \setminus R / \Pi
\]

\[
x \in \text{somewhere}.C \equiv \exists \langle x \mid x \in C \rangle \quad \text{somewhere}.R \equiv \Pi \cdot R \cdot \Pi
\]

For a relation \( R \), they can also be defined by

\[
\text{everywhere}.R = \Pi | R | \Pi \quad \text{somewhere}.R = \Pi \cdot R \cdot \Pi
\]

A theorem that enables the construction of some ‘unconventional’ proofs is

\[
C \subseteq D \equiv \text{everywhere}.(C \Rightarrow D) = \Pi
\]

If we model the booleans by collections, defining \( false = \emptyset \) and \( true = \Pi \), we can simply write

\[
C \subseteq D \equiv \text{everywhere}.(C \Rightarrow D)
\]

An example of an ‘unconventional’ proof for

\[
\begin{align*}
C \subseteq \neg D \cup X & \equiv D \subseteq \neg C \cup X
\end{align*}
\]

is then
\[ C \subseteq \neg D \cup X \]
\[ \equiv \]
\[ \text{everywhere}. (C \Rightarrow \neg D \cup X) \]
\[ \equiv \]
\{definition of \Rightarrow in terms of \& and \cup\} \[ \text{everywhere}. (\neg C \cup \neg D \cup X) \]
\[ \equiv \]
\{commutativity of \cup, see section 3.4.10\} \[ \text{everywhere}. (\neg D \cup \neg C \cup X) \]
\[ \equiv \]
\{definition of \Rightarrow in terms of \& and \cup\} \[ \text{everywhere}. (D \Rightarrow \neg C \cup X) \]
\[ \equiv \]
\[ D \subseteq \neg C \cup X \]

### 3.4 Properties

In this section we introduce some properties that collections, and relations in particular, can satisfy.

#### 3.4.1 Emptiness and fullness

A collection \( C \) is **empty** if it contains no elements, **non-empty** if it does contain an element, **full** if it contains all elements and **non-full** if it does not contain all elements:

- \( C \) is empty \( \equiv \forall(x \mid x \notin C) \)
- \( C \) is non-empty \( \equiv \exists(x \mid x \in C) \)
- \( C \) is full \( \equiv \forall(x \mid x \in C) \)
- \( C \) is non-full \( \equiv \exists(x \mid x \notin C) \)

or point free:

- \( C \) is empty \( \equiv C = \emptyset \)
- \( C \) is non-empty \( \equiv C \neq \emptyset \)
- \( C \) is full \( \equiv C = \Pi \)
- \( C \) is non-full \( \equiv C \neq \Pi \)

Non-emptiness of a relation \( R \) can also be formulated as \( \Pi \cdot R \cdot \Pi = \Pi \) instead of \( R \neq \emptyset \) because we assumed to have no empty external types. We actually prefer \( \Pi \cdot R \cdot \Pi = \Pi \) to \( R \neq \emptyset \) because positive formulations are often more useful in proofs than negative ones.
3.4.2 Uniqueness and singleness

A relation has a **unique input** if each connection in the relation has the same input and a **unique output** if each connection in the relation has the same output:

- $R$ has a unique input $\equiv \exists \langle x \parallel \forall \langle y, z \parallel y (R) z \parallel z = x \rangle \rangle$
- $R$ has a unique output $\equiv \exists \langle x \parallel \forall \langle y, z \parallel y (R) z \parallel y = x \rangle \rangle$

or point free:

- $R$ has a unique input $\equiv R \cdot \Pi \cdot R \subseteq I$
- $R$ has a unique output $\equiv R \cdot \Pi \cdot R^\circ \subseteq I$

A relation has a **unique connection** if it has a unique input as well as a unique output. A non-empty relation has a **single input** if it has a unique input, it has a **single output** if it has a unique output and it has a **single connection** if it has a unique connection.

The notions of unique connection and single connection can be generalised to that of a **unique element** and a **single element** for collections:

- $C$ has a unique element $\equiv \exists \langle x \parallel \forall \langle w \parallel w \in C \Rightarrow w = x \rangle \rangle$
- $C$ has a single element $\equiv \exists \langle x \parallel \forall \langle w \parallel w \in C \equiv w = x \rangle \rangle$

3.4.3 Functionality, injectivity, totality and surjectivity

A relation $R$ is **functional** if it connects each input with at most one output, **injective** if it connects each output with at most one input, **total** if it connects each input with at least one output and **surjective** if it connects each output with at least one input:

- $R$ is functional $\equiv \forall \langle x, y, z \parallel x (R) z \land y (R) z \parallel x = y \rangle$
- $R$ is injective $\equiv \forall \langle x, y, z \parallel y (R) x \land y (R) z \parallel x = z \rangle$
- $R$ is total $\equiv \forall \langle z \parallel \exists \langle y \parallel y (R) z \rangle \rangle$
- $R$ is surjective $\equiv \forall \langle y \parallel \exists \langle z \parallel y (R) z \rangle \rangle$

or point free:

- $R$ is functional $\equiv R \cdot R^\circ \subseteq I$
- $R$ is injective $\equiv R^\circ \cdot R \subseteq I$
- $R$ is total $\equiv I \subseteq R^\circ \cdot R$
- $R$ is surjective $\equiv I \subseteq R \cdot R^\circ$

3.4.4 Sets, conditions and squares

As mentioned in section 3.1.19, a set is a relation that only connects equal elements:
\( R \) is a set \( \equiv \forall \langle y, z \mid y (R) \rangle z \mid y = z \)

This can be written point free as:

\( R \) is a set \( \equiv R \subseteq I \)

Sets model collections within the calculus of relations as subrelations of \( I \). There are also other ways to model collections as relations. A relation is an input condition if it connects each input in its domain with each output, an output condition if it connects each output in its range with each input and a square if it connects each input in its domain with each output in its range:

\[
R \text{ is an input condition } \equiv \forall \langle x, y, z \mid x (R) \rangle z \mid y (R) = z \\
R \text{ is an output condition } \equiv \forall \langle x, y, z \mid y (R) \rangle x \mid y (R) = z \\
R \text{ is a square } \equiv \forall \langle w, x, y, z \mid y (R) \rangle x \land w (R) \rangle z \mid y (R) = z
\]

or point free:

\[
R \text{ is an input condition } \equiv R = \Pi \circ R \\
R \text{ is an output condition } \equiv R = R \circ \Pi \\
R \text{ is a square } \equiv R = R \circ \Pi \circ R
\]

### 3.4.5 Reflexivity, transitivity and symmetry

A relation \( R \) is reflexive if it connects each element with itself, irreflexive if it connects no element with itself, transitive if it connects all elements that it connects ‘indirectly’, symmetric if each connection has a ‘back’ connection and anti-symmetric if no connection between different elements has a ‘back’ connection:

\[
R \text{ is reflexive } \equiv \forall \langle x \mid x (R) \rangle \\
R \text{ is irreflexive } \equiv \forall \langle x \mid \neg (x (R) \rangle \\
R \text{ is transitive } \equiv \forall \langle x, y, z \mid y (R) \rangle x \land x (R) \rangle z \mid y (R) = z \\
R \text{ is symmetric } \equiv \forall \langle y, z \mid z (R) \rangle y \mid y (R) \rangle z \mid y = z \\
R \text{ is anti-symmetric } \equiv \forall \langle y, z \mid z (R) \rangle y \land y (R) \rangle z \mid y = z
\]

or point free:

\[
R \text{ is reflexive } \equiv I \subseteq R \\
R \text{ is irreflexive } \equiv 1 \cap R = \varnothing \\
R \text{ is transitive } \equiv R \cap R \subseteq R \\
R \text{ is symmetric } \equiv R^* = R \\
R \text{ is anti-symmetric } \equiv R^* \cap R \subseteq I
\]

A relation that is reflexive and transitive is called a preorder and an anti-symmetric preorder is called a partial order.
3.4.6 Monotonicity

A total function \( F \in \mathcal{P} \mathcal{I} \) is **monotonic** if its output is larger for larger inputs and **anti-monotonic** if its output is smaller for larger inputs:

\[
F \text{ is monotonic } \equiv \forall \langle C, D \rangle \mid C \subseteq D \Rightarrow F.C \subseteq F.D
\]

\[
F \text{ is anti-monotonic } \equiv \forall \langle C, D \rangle \mid C \subseteq D \Rightarrow F.C \supseteq F.D
\]

If the implications in these formulas can be strengthened into equivalences, we call \( F \) **monomorph** and **anti-monomorph** respectively:

\[
F \text{ is monomorph } \equiv \forall \langle C, D \rangle \mid C \subseteq D \equiv F.C \subseteq F.D
\]

\[
F \text{ is anti-monomorph } \equiv \forall \langle C, D \rangle \mid C \subseteq D \equiv F.C \supseteq F.D
\]

An \( n \)-argument function \( F \in \mathcal{P} \mathcal{I} \) is monotonic in a certain argument if each section of \( F \) where all arguments except that certain one are fixed, is monotonic. The same goes for anti-monotonicity, monomorphity and anti-monomorphity. The function ‘over’ (\( \bigcup \)) is for example monotonic in its left argument and anti-monotonic in its right argument. Sometimes we are sloppy and call a function (like for example \( \bigcap \)) monotonic, although we actually mean that it is monotonic in its arguments.

For sets \( A \) and \( B \) that are equal to the universal set \( I \), the set of monotonic total functions \( \mathcal{P} A \equiv \mathcal{P} B \) is defined by

\[
R \in \mathcal{P} A \equiv \mathcal{P} B
\]

\[
\equiv \{ \cdot A = I \land B = I \}
\]

\[
R \in \mathcal{P} A \equiv \mathcal{P} B \land R \text{ is monotonic}
\]

The strange restriction on the sets \( A \) and \( B \) is there because we defined monotonicity only for total functions. Notice that we can for example fill in \( I \times I \) or \( \mathcal{P} I \) for \( A \) and \( B \).

3.4.7 Closedness

A set of collections \( W \) is **subcollection closed** if it owns all subcollections of each collection it owns and **supercollection closed** if it owns all supercollections of each collection it owns:

\[
W \text{ is subcollection closed } \equiv \forall \langle C, D \rangle \mid C \subseteq D \land D \in W \Rightarrow C \in W
\]

\[
W \text{ is supercollection closed } \equiv \forall \langle C, D \rangle \mid C \in W \land C \subseteq D \Rightarrow D \in W
\]
3.4.8 Disjointness and conjointness

Two collections $C$ and $D$ are disjoint if they do not share any elements and conjoint if together, they contain all elements:

\[
C \text{ and } D \text{ are disjoint} \equiv \forall \langle x | x \notin C \lor x \notin D \rangle \\
C \text{ and } D \text{ are conjoint} \equiv \forall \langle x | x \in C \lor x \in D \rangle
\]

or point free:

\[
C \text{ and } D \text{ are disjoint} \equiv C \cap D = \emptyset \\
C \text{ and } D \text{ are conjoint} \equiv C \cup D = I
\]

Two sets $A$ and $B$ are called set-conjoint if together they own all elements:

\[
A \text{ and } B \text{ are set-conjoint} \equiv A \cup B = I
\]

3.4.9 Domain equality

Two relations $R$ and $T$ are domain equal if they have equal domains:

\[
R \text{ and } T \text{ are domain equal} \equiv \forall \langle z | \exists \langle y | y \in (R) z \rangle \equiv \exists \langle y | y \in (T) z \rangle \rangle
\]

or point free:

\[
R \text{ and } T \text{ are domain equal} \equiv \Pi \cdot R = \Pi \cdot T
\]

or using the domain operator:

\[
R \text{ and } T \text{ are domain equal} \equiv R \rightarrow T
\]

We define the domain-equality relation $DEQ \in (I \rightarrow I) \rightarrow (I \rightarrow I)$ by

\[
R \cdot (DEQ) T \equiv R \text{ and } T \text{ are domain equal}
\]

3.4.10 Idempotency, commutativity and associativity

For a function $\odot \in A \leftrightarrow A \times A$, for some set $A$, idempotency, commutativity and associativity are defined as follows:

\[
\begin{align*}
\text{\textbullet} \odot \text{ is idempotent} & \equiv \forall \langle x | x \in A | x \circ x = x \rangle \\
\text{\textbullet} \odot \text{ is commutative} & \equiv \forall \langle x, y | x, y \in A | x \odot y = y \odot x \rangle \\
\text{\textbullet} \odot \text{ is associative} & \equiv \forall \langle x, y, z | x, y, z \in A | (x \odot y) \odot z = x \odot (y \odot z) \rangle
\end{align*}
\]

Associativity allows the removal of brackets as described in section 3.1.2 and is hardly ever mentioned explicitly in this thesis.
3.4.11 Zeros and units

For a function \( \circ \in A \leftrightarrow A \times A \) and an element \( x \in A \), for some set \( A \), we define the notions left-zero, right-zero, left-unit and right-unit by

- \( x \) is a left-zero of \( \circ \) \( \equiv \forall \langle y \mid y \in A \mid x \circ y = x \rangle \)
- \( x \) is a right-zero of \( \circ \) \( \equiv \forall \langle y \mid y \in A \mid y \circ x = x \rangle \)
- \( x \) is a left-unit of \( \circ \) \( \equiv \forall \langle y \mid y \in A \mid x \circ y = y \rangle \)
- \( x \) is a right-unit of \( \circ \) \( \equiv \forall \langle y \mid y \in A \mid y \circ x = y \rangle \)

A zero is an element that is a left-zero as well as a right-zero and a unit is an element that is a left-unit as well as a right-unit.

3.4.12 Distribution

For functions \( f \in A \leftrightarrow A \) and \( \circ \in A \leftrightarrow A \times A \), for some set \( A \), distribution of \( f \) over \( \circ \) is defined by

\[
f \text{ distributes over } \circ \equiv \forall \langle x, y \mid x, y \in A \mid f.(x \circ y) = f.x \circ f.y \rangle
\]

We also define distribution for 'arbitrary operators' like the arbitrary union and intersection. For \( f \in A \leftrightarrow A \) and \( \circ \in A \leftrightarrow \wp A \), for some set \( A \):

\[
f \text{ distributes over } \circ \equiv \forall \langle W \mid W \in \wp A \mid f.(\circ W) = \circ\{f.x \mid x \in W \mid x\} \rangle
\]

Although formally not correct, we also say that a binary operator \( \square \circ \) distributes over \( \circ \) if the sections \( (x \circ) \) and \( (\circ x) \) both distribute over \( \circ \).

3.4.13 Commutation

For functions \( g, h \in A \leftrightarrow A \), commutation of \( g \) with \( h \) is defined by

\[
g \text{ commutes with } h \equiv \forall \langle x \mid x \in A \mid g.(h.x) = h.(g.x) \rangle
\]

or point free:

\[g \circ h = h \circ g\]
3.4.14 Abidence

For functions \( \square \cdot \circ \in A \leftrightarrow A \times A \), for some set \( A \), abidence of \( \square \) with \( \circ \) is defined by

\[
\square \text{ abides with } \circ \\
= \\
\forall \langle x_0, x_1, y_0, y_1 \rangle \\
x_0, x_1, y_0, y_1 \in A \mid (x_0 \square x_1) \circ (y_0 \circ y_1) = (x_0 \circ y_0) \square (x_1 \circ y_1)
\]

The name “abide” stands for “above or beside” and is due to Richard Bird. The name was inspired by the following way of writing the law, due to Tony Hoare:

\[
\begin{align*}
& x_0 \square x_1 & x_0 & x_1 \\
& \circ & = & \circ \square \circ \\
& y_0 \circ y_1 & y_0 & y_1
\end{align*}
\]

3.4.15 Comparability, chains and continuity

Two collections are comparable if one is a subcollection of the other:

\[
C \text{ and } D \text{ are comparable } \equiv C \subseteq D \lor D \subseteq C
\]

A set of collections \( W \) is a chain if all collections in \( W \) are comparable with each other:

\[
W \text{ is a chain } \equiv \forall \langle C, D \mid C, D \in W \mid C \text{ and } D \text{ are comparable} \rangle
\]

A total function \( F \in P I \leftrightarrow PI \) is continuous if it distributes over the arbitrary union of all non-empty chains:

\[
F \text{ is continuous } \\
= \\
\forall \langle W \mid W \text{ is a chain } \land W \neq \emptyset \mid F.(\bigcup W) = \bigcup \{ F.X \mid X \in W \mid X \} \rangle
\]

For sets \( A \) and \( B \) that are equal to \( I \), the set of continuous total functions \( PA \leftrightarrow PB \) is defined by

\[
\begin{align*}
R \in PA & \leftrightarrow PB \\
= & \{ A = I \land B = I \} \\
R & \in PA \leftrightarrow PB \land R \text{ is continuous}
\end{align*}
\]

We leave it to the reader to prove that a continuous total function is monotonic (hint: the set \( \{ C, D \} \) with \( C \subseteq D \) is a chain).

Continuity comes into play in chapter 9 where calls in specifications can be made
visible in the semantics.

3.5 Galois connections

In this section we give a brief overview of the theory of Galois connections in the context of partial order \( \subseteq \). For more elaborate theory about Galois connections, we refer the reader to [1].

3.5.1 Definition

For \( F, G \in P \to P \), we call \((F, G)\) a Galois connection if

\[
F.X \subseteq Y \equiv X \subseteq G.Y
\]

The function \( F \) is called the lower adjoint of the Galois connection and the function \( G \) is called the upper adjoint.

An equivalent definition is

- \( F \) is monotonic
- \( G \) is monotonic
- \( G.(F.X) \supseteq X \)
- \( F.(G.Y) \subseteq Y \)

The lower two laws are referred to as cancellation.

3.5.2 More cancellation laws

A theorem that enables us to strengthen the cancellation laws into equalities is that if \((F, G)\) is a Galois connection then

\[
G \text{ is surjective } \equiv F \text{ is injective} = \forall\langle X \parallel X = G.(F.X)\rangle
\]

\[
F \text{ is surjective } \equiv G \text{ is injective} = \forall\langle Y \parallel F.(G.Y) = Y\rangle
\]

If we cannot use this theorem, we still have that for a Galois connection \((F, G)\)

\[
F.(G.(F.X)) = F.X \quad G.(F.(G.Y)) = G.Y
\]

All these theorems are also referred to as cancellation.
3.5.3 Uniqueness and existence of adjoints

The adjoints in a Galois connection are unique. So, if \((F_0, G_0)\) and \((F_1, G_1)\) are Galois connections then

\[ F_0 = F_1 \equiv G_0 = G_1 \]

The fundamental theorem of Galois connections gives us that a total function \(G\) is an upper adjoint in a Galois connection exactly when \(G\) distributes over \(\cap\) and that a total function \(F\) is a lower adjoint in a Galois connection exactly when \(F\) distributes over \(\cup\). For sets \(A\) and \(B\) that are equal to \(I\), we introduce the set \(PA \leftarrow \cap PB\) of total functions that distribute over \(\cap\) and the set \(PA \leftarrow \cup PB\) of total functions that distribute over \(\cup\):

\[ G \in PA \leftarrow \cap PB \equiv \{ A = I \land B = I \} \]

\[ G \in PA \leftarrow PB \land G \text{ distributes over } \cap \]

\[ F \in PA \leftarrow \cup PB \equiv \{ A = I \land B = I \} \]

\[ F \in PA \leftarrow PB \land F \text{ distributes over } \cup \]

Because adjoints are unique, we can talk about the upper adjoint of a total function that distributes over \(\cup\) and about the lower adjoint of a total function that distributes over \(\cap\). The function \(\sharp \) \(\in (PI \leftarrow \cap PI) \leftrightarrow (PI \leftarrow \cup PI)\) gives the upper adjoint of a total function that distributes over \(\cup\) and the function \(\flat \) \(\in (PI \leftarrow \cup PI) \leftrightarrow (PI \leftarrow \cap PI)\) gives the lower adjoint of a total function that distributes over \(\cap\). They can be defined by

\[ F^\sharp Y = \bigcup \{ X \mid F.X \subseteq Y \mid X \} \]

\[ G^\flat X = \bigcap \{ Y \mid X \subseteq G.Y \mid Y \} \]

3.5.4 Exploiting Galois connections

If we know that an operator is part of a Galois connection, we get many theorems for free. For the sequential composition we have for example:

\[ S \circ P \subseteq R \equiv S \subseteq R/P \]

\[ S \circ P \subseteq R \equiv P \subseteq S\backslash R \]

From these definitions we see that section \((\circ P)\) is the lower adjoint of a Galois connection with upper adjoint \((/ P)\) and that section \((S\circ)\) is the lower adjoint of a Galois connection with upper adjoint \((S\backslash)\). We thus get for free that \(\circ\) is monotonic in both arguments, that \(/\) is monotonic in its left argument and that \(\backslash\) is monotonic in its right argument:
\[
S_0 \subseteq S_1 \Rightarrow S_0 P \subseteq S_1 P \\
R_0 \subseteq R_1 \Rightarrow R_0/P \subseteq R_1/P \\
P_0 \subseteq P_1 \Rightarrow S \cdot P_0 \subseteq S \cdot P_1 \\
R_0 \subseteq R_1 \Rightarrow S \Delta R_0 \subseteq S \Delta R_1
\]

The cancellation laws that we get for free are
\[
(S \cdot P)/P \supseteq S \\
(R/P) \cdot P \subseteq R \\
S \setminus (S \cdot P) \supseteq P \\
S \cdot (S \setminus R) \subseteq R
\]

The distribution properties that we get for free are
\[
\bigcup W \cdot P = \bigcup \{S \cdot P \mid S \in W \} \\
S \cdot \bigcup W = \bigcup \{S \cdot P \mid P \in W \}
\]
\[
\bigcap W/P = \bigcap \{R/P \mid R \in W \} \\
S \setminus \bigcap W = \bigcap \{S \setminus R \mid R \in W \}
\]

Using the fact that \( \Pi = \bigcap \emptyset, \emptyset = \bigcup \emptyset, Q \cap T = \bigcap \{Q, T\} \) and \( Q \cup T = \bigcup \{Q, T\} \), we get
\[
\emptyset \cdot P = \emptyset \\
\Pi / P = \Pi \\
(Q \cup T) \cdot P = Q \cdot P \cup T \cdot P \\
S \cdot (Q \cup T) = S \cdot Q \cup S \cdot T \\
(Q \cap T) / P = Q / P \cap T / P \\
S \setminus (Q \cap T) = S \setminus Q \cap S \setminus T
\]

In chapter 5 the theory of Galois connections is exploited to prove several properties of products.

## 3.6 Fixed points

A fixed point \( X \) of a monotonic total function \( F \) is a collection that is ‘left unchanged’ by the function:

\[
F \cdot X = X
\]

A monotonic total function \( F \) has a least fixed point and a greatest fixed point. The least–fixed-point operator and greatest–fixed-point operator \( \mu F, \nu F \in P \) determine these fixed points. They are defined by

\[
\mu F = \bigcap \{X \mid F \cdot X = X \} \\

\nu F = \bigcup \{X \mid F \cdot X = X \}
\]
The least–fixed-point operator and greatest–fixed-point operator can both also be defined by the combination of two rules. One is called the **computation** rule. It says that $\mu F$ and $\nu F$ are fixed points of $F$:

$$F.\mu F = \mu F$$
$$F.\nu F = \nu F$$

The other is called the **induction** rule. This one says that $\mu F$ is the smallest and $\nu F$ is the largest of all fixed points:

$$F.X = X \Rightarrow \mu F \subseteq X$$
$$F.X = X \Rightarrow \nu F \supseteq X$$

It is a well-known fact that $\mu$ and $\nu$ can also be defined in terms of **prefix points** and **postfix points** respectively. A prefix point $X$ of a monotonic total function $F$ is a collection that satisfies $F.X \subseteq X$ and a postfix point $X$ is a collection that satisfies $F.X \supseteq X$. We can also define $\mu$ and $\nu$ equivalently by

$$\mu F = \bigcap\{X \mid F.X \subseteq X \mid X\}$$
$$\nu F = \bigcup\{X \mid F.X \supseteq X \mid X\}$$

or again by what we also call computation and induction:

$$F.\mu F \subseteq \mu F$$
$$F.\nu F \supseteq \nu F$$

$$F.X \subseteq X \Rightarrow \mu F \subseteq X$$
$$F.X \supseteq X \Rightarrow \nu F \supseteq X$$

The operators $\mu$ and $\nu$ satisfy several algebraic properties. One of them is that they are **monotonic**, that is, for $G, H \in \Pi \leftarrow \mu \Pi$ we have

$$\forall \langle X \mid G.X \subseteq H.X \rangle \Rightarrow \mu G \subseteq \mu H$$
$$\forall \langle X \mid G.X \subseteq H.X \rangle \Rightarrow \nu G \subseteq \nu H$$

We are actually not allowed to call this “monotonic” in our formalism, but we assume that the reader is able to tell the different uses of the word “monotonic” apart.

In case $F$ is a continuous total function, the least fixed point of $F$ is equal to the union of all $F^n.\emptyset$ for $n \in \mathbb{N}$:

$$\mu F = \bigcup\{F^n.\emptyset \mid n \in \mathbb{N} \mid n\} \Leftrightarrow F \in \Pi \leftarrow \mu \Pi$$

For a function $F$ that is a lower adjoint ($F \in \Pi \leftarrow \bigcup \Pi$), a function $G$ that is an upper adjoint ($G \in \Pi \leftarrow \bigcap \Pi$) and monotonic total functions $H$ and $K$ ($H, K \in \Pi \leftarrow \mu \Pi$), the following **fusion** theorems hold:
Least fixed points are usually associated with finite elements and greatest fixed points with infinite elements. We illustrate this using the natural numbers as an example.

If we assume the existence of an external type of possibly infinite natural numbers, where $0_\infty$ denotes its ‘zero’, $1_\infty$ denotes its ‘one’, $\infty_\infty$ denotes its ‘infinity’ and $+\infty$ denotes its ‘sum’, then for $f_\infty$, defined by

$$f_\infty.N = [n +_\infty 1_\infty | n \in N] \cup [0_\infty]$$

$\nu f_\infty$ contains $\infty_\infty$ whereas $\mu f_\infty$ does not:

$$\mu f_\infty = \Pi \setminus [\infty_\infty]$$

$$\nu f_\infty = \Pi$$

In case we would have used finite natural numbers as external type, both fixed points would be equal.

The theory of fixed points is used in chapter 8 to model inter-component communication.

3.7 Relators

For a set $A$, we defined $PA$ by

$$C \in PA \equiv \forall \langle x | x \in C \mid x \in A \rangle$$

and for sets $A$ and $B$, we defined $A*B$ by

$$\langle y,z \rangle \in A*B \equiv y \in A \land z \in B$$

There is a natural way to generalise these definitions to relations. We define the power relator $P_\infty \in (PI \rightarrow PI) \leftrightarrow (1 \rightarrow 1)$ by
\[
C (PR) D
\]
\[
\equiv \forall \langle y \mid y \in C \mid \exists \langle z \mid z \in D \mid y (R) z \rangle \rangle \land \\
\forall \langle z \mid z \in D \mid \exists \langle y \mid y \in C \mid y (R) z \rangle \rangle
\]

This definition is consistent with the one restricted to sets:

\[
C (PA) D \equiv C = D \land \forall \langle x \mid x \in C \rangle
\]

Furthermore, the following theorems can be proved for relations \( R \) and \( S \):

\[
R \subseteq S \quad \equiv \quad PR \subseteq PS
\]
\[
P(R \cap S) \subseteq PR \cap PS
\]
\[
P(R \cup S) \supseteq PR \cup PS
\]
\[
P(R \cdot S) = PR \cdot PS
\]
\[
P(I) = I
\]
\[
P(R^\sim) = (PR)^\sim
\]

Most of these theorems are straightforward to prove at the point-wise level, except for the fact that \( P(R \cdot S) \subseteq PR \cdot PS \). This theorem can be proved using the axiom of choice. A more algebraic treatment can be found in [11].

For \( \star \) we have a similar kind of generalisation. We define the connectional product \( \star \in \langle (I \star I) \rightsquigarrow (I \star I) \rangle \) by

\[
\langle y_0, y_1 \rangle \langle R_0 \star R_1 \rangle \langle z_0, z_1 \rangle \equiv y_0 (R_0) z_0 \land y_1 (R_1) z_1
\]

This definition is again consistent with the one restricted to sets:

\[
\langle y_0, y_1 \rangle \langle A_0 \star A_1 \rangle \langle z_0, z_1 \rangle \equiv (y_0, y_1) = (z_0, z_1) \land y_0 \in A_0 \land y_1 \in A_1
\]

Furthermore, the following theorems can be proved for relations \( R_i \) and \( S_i \):

\[
R_0 \subseteq S_0 \land R_1 \subseteq S_1 \Rightarrow R_0 \star R_1 \subseteq S_0 \star S_1
\]
\[
R_0 \cap S_0 \star R_1 \cap S_1 = R_0 \star R_1 \cap S_0 \star S_1
\]
\[
R_0 \cap S_0 \star R_1 \cup S_1 = R_0 \star R_1 \cup (S_0 \star R_1) \cup S_0 \star S_1
\]
\[
R_0 \cup S_0 \star R_1 \cup S_1 = R_0 \cup R_1 \star S_0 \cup S_1
\]
\[
I \star I = I
\]
\[
R_0 \star R_1 \Rightarrow R_0 \star R_1
\]
\[
R_0 \star R_1 \Rightarrow R_0 \star R_1
\]
\[
R_0 \star R_1 \Rightarrow R_0 \star R_1
\]

The proofs of these theorems are straightforward.

The generalisation of \( \star \) to relations is used in section 6.8 for the theory about what we call “connectional expressions”.

The power relator and connectional product are examples of relators. For more
information about relators we refer the reader to [11].

3.8 Some theorems

In this section we discuss some fundamental theorems.

3.8.1 Shunting

The following theorem is called shunting. For total functions $g$ and $h$ we have

$$S \subseteq g^{\circ} R \cdot h \equiv g \cdot S \cdot h^{\circ} \subseteq R$$

or equivalently (using the fact that $U \cdot S \cdot P \subseteq R \equiv S \subseteq U \backslash R \backslash P$ and indirect equality):

$$g^{\circ} R \cdot h = g \cdot U \backslash R \backslash h^{\circ}$$

We often use shunting with $g$ or $h$ instantiated to $I$. Specifically, for a total function $f$ we have

$$S \subseteq R \cdot f \equiv S \cdot f^{\circ} \subseteq R$$
$$S \subseteq f^{\circ} R \equiv f \cdot S \subseteq R$$

or equivalently:

$$R \cdot f = R \cdot f^{\circ}$$
$$f^{\circ} R = f \backslash R$$

Each of these four properties (universally quantified over $R$ and $S$) is in fact equivalent to $f$ being a total function. The proof of this fact is left to the reader (hint: use the point-free formulation of functionality and totality).

A theorem that we also refer to as “shunting”, is that for total functions $g$ and $h$

$$y (g^{\circ} R \cdot h) z \equiv g \cdot g \cdot (R) \cdot h \cdot z$$
$$y (g \cdot U \backslash R \backslash h^{\circ}) z \equiv g \cdot g \cdot (R) \cdot h \cdot z$$

This theorem is also used often with $g$, $h$ or $R$ instantiated to $I$.

The notion of shunting is fundamental in point-free relational calculus and it is used at several places in this thesis.
3.8.2 Complementing things

Especially from the dot-matrix representation of relations, it is clear that for each operator that we defined, there exists an operator that behaves exactly the same, ‘after taking the complement of everything’.

An example is the confrontation $\wedge \in (I\rightarrow I) \langle (I\rightarrow I) \times (I\rightarrow I) \ (\text{see } [15])$, defined by

$$y (S \uparrow P) z \equiv \forall(x \parallel y (S) x \vee x (P) z)$$

or in terms of $\circ$ and $\neg$:

$$S \uparrow P = \neg (\neg S \circ \neg P)$$

The confrontation is associative, has $\neg I$ as unit and distributes over $\cap$, which is a trivial consequence of the fact that $\circ$ is associative, has $I$ as unit and distributes over $\cup$. In general, the algebraic laws of

$$\langle \subseteq, \cap, \cup, \exists, \forall, \emptyset, \ø, \Pi, \neg, \cap, \cup, \circ, I, \neg, everywhere, somewhere \rangle$$

equal the algebraic laws of

$$\langle \supseteq, \cup, \cap, \emptyset, \ø, \Pi, \neg, \cup, \cap, \circ, I, \neg, somewhere, everywhere \rangle$$

3.8.3 Distribution of sequential composition over intersection

A well-known property of the relational calculus is that in general, sequential composition does not distribute over intersection. We do have that for a set $A$

$$(R \cap S) \cdot A = R \cdot A \cap S \cdot A$$

$$A \cdot (R \cap S) = A \cdot R \cap A \cdot S$$

More in general, for a function $f$ we have

$$(R \cap S) \cdot f = R \cdot f \cap S \cdot f$$

$$f^\circ (R \cap S) = f^\circ R \cap f^\circ S$$

Both these properties (universally quantified over $R$ and $S$) are even equivalent to the fact that $f$ is a function. In general for a relation $T$, we only have the following inclusion though:

$$(R \cap S) \cdot T \subseteq R \cdot T \cap S \cdot T$$

$$T^\circ (R \cap S) \subseteq T^\circ R \cap T^\circ S$$
This is called **half-distribution** of sequential composition over intersection.

There is a general theorem that gives an inclusion the other way around. This theorem is called the law of **Desargues** and is easy to prove at the point-wise level, although it involves many variables:

\[
(R_0 U_0 \cap R_1 U_1) \circ (U_0 \circ T \cap U_1 \circ T) \supseteq R_0 T_0 \cap R_1 T_1
\]

\[
\Rightarrow \quad U_0 U_1 \supseteq R_0 \circ R_1 \cap T_0 T_1
\]

The following instantiations of this theorem are known as the laws of **Dedekind**:

\[
(R \cap S T) T \supseteq R T \cap S
\]

\[
(R T \cap S) T \supseteq R \cap S T
\]

\[
T (R \cap T S) \supseteq T R \cap S
\]

\[
T (T \cap R S) \supseteq R T \cap S
\]

Using half-distribution and the law of Desargues, the following theorem is easy to prove:

\[
(R \cap R_1) \circ (T_0 \cap T_1) = (R_0 T) \cap (R_1 T)
\]

\[
\Rightarrow \quad \exists(U_0, U_1 \mid U_0 U_1 \supseteq R_0 \circ R_1 \cap T_0 T_1
\]

\[
R_0 = R_0 U_0 \land T_0 = U_0 \circ T_0 \land
\]

\[
R_1 = R_1 U_1 \land T_1 = U_1 \circ T_1
\]

Yet another two distribution theorems we want to mention:

\[
T (R \cap S) = T R \cap T S = R \cap S
\]

\[
(R \cap S) T = R T \cap S T = R \cap S
\]

Both are easy to prove using a **ping-pong** proof. For the first one, the proof is

\[
T \circ (R \cap S)
\]

\[
\subseteq \{ \text{half-distribution of } \circ \text{ over } \cap \}
\]

\[
T \cap T \cap S
\]

\[
\subseteq \{ \text{• } T \cap R \subseteq R \land T \cap S \subseteq S \}
\]

\[
R \cap S
\]

\[
= \{ \text{I is unit of } \circ \}
\]

\[
\text{I} \circ (R \cap S)
\]

\[
\subseteq \{ \text{• I } \subseteq T \}
\]

\[
T \circ (R \cap S)
\]

The proof of the second one is analogous.

The law of Desargues (which we obtained from [49]) and the last two distribution
theorems were used in previous versions of the proof in section 5.4.6. Although these versions were at a more point-free level and did not resort to the axiom of choice, they were not as general and more ad hoc than the current proof. We still included the above theorems however, because some readers might find them helpful for their understanding. The same goes for the next subsection.

3.8.4 Extreme point-freeness

Formulating equations in a point-free manner can be taken to an extreme, raising the level of abstraction even more. Most of the theorems in this section were used in previous versions of the proofs in chapters 4 and 5. These formulas helped us to increase the level of abstraction at which we could reason about the constructs that are presented in those chapters. Although the theorems of this section are not used anymore in the current version of this thesis, some readers might find them helpful for their understanding.

The following equations express the fact that I is a right-unit of $\circ$ and / and a left-unit of $\circ$ and $\setminus$:

$$ (\circ I) = I $$

$$ (/I) = I $$

$$ (I \circ) = I $$

$$ (I \setminus) = I $$

Three point-free ways to express associativity of sequential composition are:

$$ (R \circ) (\circ T) = (\circ T) (R \circ) $$

$$ (R \circ) (T \circ) = ((R \circ T) \circ) $$

$$ (\circ R) (\circ T) = (\circ (T \circ R)) $$

Monotonicity and anti-monotonicity of a total function $F$ can be formulated as

$$ F \text{ is monotonic } \equiv F \subseteq \circ \subseteq F $$

$$ F \text{ is anti-monotonic } \equiv F \circ \subseteq \subseteq \circ F $$

Subcollection closedness and supercollection closedness of a set of collections $W$ can be formulated as

$$ W \text{ is subcollection closed } \equiv \subseteq W \subseteq \circ \subseteq W $$

$$ W \text{ is supercollection closed } \equiv \subseteq W \subseteq \subseteq \circ W $$

The following theorem enables us to formulate monotonicity and closedness as a kind of ‘absorption’ laws:

$$ R \subseteq \subseteq \circ \subseteq R \equiv \subseteq \circ R \subseteq \subseteq = \subseteq \circ R $$

$$ \subseteq \subseteq R \subseteq \subseteq R \subseteq \equiv \subseteq \circ R \subseteq \subseteq = \circ \subseteq R $$
We prove the first one. The proof of the second one is analogous.

\[(⇒)\]

\[
\subseteq R \subseteq \subseteq
\]

\[
\subseteq \{ R \subseteq \subseteq \subseteq R \}
\]

\[
\subseteq \subseteq R
\]

\[
\subseteq \{ \subseteq \subseteq \subseteq \}
\]

\[
\subseteq \subseteq R
\]

\[
\subseteq \{ I \text{ is unit of } \}
\]

\[
\subseteq R \cdot I
\]

\[
\subseteq \{ I \subseteq \subseteq \}
\]

\[
\subseteq R \cdot \subseteq
\]

\[(⇐)\]

\[
R \subseteq \subseteq
\]

\[
\subseteq \{ I \text{ is unit of } \}
\]

\[
R \subseteq I
\]

\[
\subseteq \{ I \subseteq \subseteq \}
\]

\[
\subseteq R \cdot \subseteq
\]

\[
\subseteq \{ \subseteq R \cdot \subseteq \subseteq \}
\]

\[
\subseteq R
\]

To illustrate a nice ‘proof trick’, we prove that the sequential composition of two monotonic total functions is monotonic, using the formulation of monotonicity as an ‘absorption’ law:

\[
\subseteq F \cdot G \cdot \subseteq
\]

\[
= \{ \subseteq F = \subseteq F \subseteq \}
\]

\[
\subseteq F \cdot \subseteq G \cdot \subseteq
\]

\[
= \{ \subseteq G \subseteq \subseteq = \subseteq G \}
\]

\[
\subseteq F \cdot \subseteq G
\]

\[
= \{ \subseteq F \subseteq \subseteq = \subseteq F \}
\]

\[
\subseteq F \cdot G
\]

The shunting rules for a total function \( f \) can be formulated as

\[
(\cdot f) = (f^\sim)
\]

\[
(f^\sim) = (f\setminus)
\]

The fact that \((F, G)\) is a Galois connection can be formulated as
\[ F^\circ \subseteq \subseteq \subseteq G \]

where \( F^\circ \subseteq \) (or \( \subseteq G \)) is known as the **pair algebra** of the Galois connection (see [5]).

The alternative definition of a Galois connection \((F, G)\) can also be formulated as
\[
\begin{align*}
F^\circ \subseteq \subseteq \subseteq F \\
G^\circ \subseteq \subseteq \subseteq G \\
G^\circ F \subseteq \subseteq \subseteq G \\
F^\circ G \subseteq \subseteq G
\end{align*}
\]

and the extra cancellation laws as
\[
\begin{align*}
I \subseteq G^\circ G^\circ & \equiv F^\circ F^\circ \\
F^\circ F \subseteq I & \equiv G^\circ G \subseteq I \\
G^\circ F = I & \equiv F^\circ G = I
\end{align*}
\]

and
\[
\begin{align*}
F^\circ G^\circ F & = F \\
G^\circ F^\circ G & = G
\end{align*}
\]

If we define the **fixed-point-of relation** \( \text{FIXES} \in \Pi \leftarrow \leftarrow (\Pi \iff \leftarrow \leftarrow \Pi) \) by
\[
X (\text{FIXES}) \iff F.X = X
\]

then the definitions of the least–fixed-point operator and greatest–fixed-point operator can be formulated as
\[
\begin{align*}
\mu_\subseteq \subseteq \text{FIXES} \land \text{FIXES} \subseteq \supseteq \mu_\subseteq \\
\nu_\subseteq \subseteq \text{FIXES} \land \text{FIXES} \subseteq \subseteq \nu_\subseteq
\end{align*}
\]

### 3.9 Conclusions

The purpose of this chapter was to introduce the basic theory that is used in this thesis. We stuck to common notation where possible, but deviated where we thought this to be beneficial. An important ingredient of the theory is the external type system that we informally described in section 3.1.21. We see the formalisation of this type system as an important topic for future research.
Chapter 4

Types

Types are a means to restrict the elements that a variable can represent. In this chapter we introduce and investigate several types that can be used to restrict the values of variables that represent relations.

4.1 Cylindric typing

In functional type theories, the co/contra-variance rule is the primary rule for subtyping on function types. Denoting subtyping by $\leq$ and the type of functions from $B$ to $A$ by $A \leftarrow B$, the rule is as follows:

$$A' \leq A \land B' \leq B \implies A' \leftarrow B \leq A \leftarrow B'$$

The rule is usually formulated as “the output type may be strengthened and the input type may be weakened”, telling how to construct a subtype of a function type. The fact that functions of the subtype may be defined on a larger domain, corresponds to subtyping in object-oriented languages, where a subclass can extend a superclass by new methods, the collection of method names forming the domain on which the class is defined. We call this phenomenon extensibility, which is formalised in section 4.2.8.

In relational theories, the co/contra-variance rule is not very common. One of the main reasons is that relational theorists like to treat relations bidirectionally, meaning that no specific side of a relation is chosen as input side. The asymmetry between input and output in the co/contra-variance rule does not match this idea. A result is that typing of relations is usually done in a ‘symmetric’ way. A relation $S$ is typed by giving an upper range $A$ and an upper domain $B$ for $S$:

$$S : \subseteq A \land S : \subseteq B$$
We call this **bidirectional typing**. Some equivalent formulations are

\[
S \subseteq A \cdot \Pi \cdot B \\
S = A \cdot S \cdot B \\
S \subseteq A \cdot S \cdot B
\]

A way to write this down point wise is

\[
\forall \langle y, z \mid y (S) z \mid y \in A \land z \in B \rangle
\]

So we see that the type \( A \rightarrow B \) that was introduced in section 3.1.21, is intended for bidirectional typing. Extensibility can be obtained by a weaker form of typing, stating only that \( S \) outputs an element of \( A \) if the input is an element of \( B \):

\[
\forall \langle y, z \mid y (S) z \mid y \in A \iff z \in B \rangle
\]

We call this **cylindric typing**. In most functional type theories, a similar definition is used for “\( f : A \leftarrow B \)”. Notice that the symmetry between input and output is destroyed.

Some point-free formulations that are equivalent to the above formula are

\[
S \cdot B \subseteq A \cdot \Pi \\
S \cdot B = A \cdot S \cdot B \\
S \cdot B \subseteq A \cdot S
\]

For sets \( A \) and \( B \), we introduce the set \( A \rightarrow o B \) for cylindric typing:

\[
S \in A \rightarrow o B \\
\equiv \forall \langle y, z \mid y (S) z \mid y \in A \iff z \in B \rangle
\]

The following **cylindric-type rules** are easy to verify. For sets \( A, B \) and \( C \) we have

\[
\begin{align*}
R & \in A \rightarrow o B \iff R \subseteq S \land S \in A \rightarrow o B \\
R \cap S & \in A \rightarrow o B \iff R \in A \rightarrow o B \land S \in A \rightarrow o B \\
R \cup S & \in A \rightarrow o B \iff R \in A \rightarrow o B \land S \in A \rightarrow o B \\
R \cdot S & \in A \rightarrow o B \iff R \in A \rightarrow o B \land S \in A \rightarrow o B \\
R \setminus S & \in A \rightarrow o B \iff R \in A \rightarrow o B \land S \in A \rightarrow o B \\
R \div S & \in A \rightarrow o B \iff R \in A \rightarrow o C \land S \in C \rightarrow o B \\
1 & \in A \rightarrow o A \\
R^* & \in A \rightarrow o A \iff R \in A \rightarrow o A
\end{align*}
\]

In contrast to bidirectional typing, we do not have that \( S \subseteq A \rightarrow o B \) follows from \( S \subseteq B \rightarrow o A \) in general. This reflects the **unidirectionality** of cylindric typing.
4.2 Type operators

A **type operator** is a function with type

\[(I \rightarrow I) \rightarrow (I \rightarrow I) \leftrightarrow (I \rightarrow I) \times (I \rightarrow I)\]

We call the left argument of a type operator the **output type** and the right argument the **input type**.

4.2.1 Cylindric-type operator

The **cylindric-type operator** \(-\circ-\) is defined by

\[S (R \rightarrow P) Q \equiv S \circ P \subseteq R \circ Q\]

The correspondence between \(-\circ-\) and \(-\circ\) is reflected by the following equation:

\[S (A \rightarrow B) S \equiv S \in A \rightarrow B\]

An important property of \(-\circ-\) is that it is monotonic, or **covariant**, in its output type and anti-monotonic, or **contravariant**, in its input type:

\[R \subseteq T \land P \supseteq U \Rightarrow R \rightarrow P \subseteq T \rightarrow U\]

The proof of this fact is trivial. Contravariance in the input type has always been the standard in functional programming, but not in relational programming.

4.2.2 Functionality and totality

Recall the following two point-free formulations of functionality and totality:

\[S \circ S^* \subseteq I\]
\[I \subseteq S^* \circ S\]

These formulations can be generalised straightforwardly to variants where an input type and output type are included:

\[S \circ B \cdot S^* \subseteq A\]
\[B \subseteq S^* \circ A \cdot S\]

The first one means that \(S\) connects each input from \(B\) with at most one output and that this output is an element of \(A\). The second one means that \(S\) connects each input from \(B\) with at least one output from \(A\).

Just like we did with cylindric typing, we also generalise these two properties,
defining the type operators \(_\leftarrow\_\) and \(_\rightarrow\_\) by

\[
S(R \leftarrow P)Q \equiv S \cdot P \cdot Q' \subseteq R \\
S(R \rightarrow P)Q \equiv P \subseteq S^\ast R \cdot Q
\]

A relation \(S\) is **functional on** \(B\) if it is functional in every element of \(B\), or equivalently, \(S(I \leftarrow B)S\). A relation \(S\) is **total on** \(B\) if it is total in every element of \(B\), or equivalently, \(S(I \rightarrow B)S\). A relation \(S\) is **single-valued on** \(B\) if it is single-valued in every element of \(B\), or equivalently, if it is functional on \(B\) and total on \(B\).

### 4.2.3 Restrictedness

Saying that a relation \(S\) is total on \(B\), is the same as saying that \(B\) is a lower domain of \(S\):

\[S(I \rightarrow B)S \equiv S^\ast \supseteq B\]

We also have a type operator \(\rightarrow\rightarrow\) for upper domains, such that

\[S(I \rightarrow B)S \equiv S^\ast \subseteq B\]

This type operator \(\rightarrow\rightarrow\_\) is defined as follows:

\[S(R \rightarrow P)Q \equiv S \subseteq R \cdot Q \cdot P^\ast\]

If \(S(I \rightarrow B)S\) holds, we say that \(S\) is **restricted to** \(B\).

The type operator \(\rightarrow\rightarrow\) can actually be used for restricting the domain as well as the range:

\[S(A \rightarrow B)S \equiv S^\ast \subseteq A \land S^\ast \subseteq B\]

Another type operator that satisfies this property is \(\leftarrow\rightarrow\_\) defined by

\[S(R \leftarrow P)Q \equiv Q^\ast \subseteq P^\ast S^\ast R\]

For sets \(A\) and \(B\) we have

\[
S(A \rightarrow B)S \\
\equiv S \in A \rightarrow B \\
\equiv S(A \rightarrow B)S
\]

76
4.2.4 Combining type operators

By means of intersection, we can combine several type operators into one type operator. We write such a combined type operator as the combination of the symbols of the type operators it combines. The type operator \(_ \leftarrow \leftarrow \) is for example defined by

\[ R \leftarrow\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!!
We would also ‘desire’ however that for sets $A$, $B$ and $C$, $S((A \diamonds V B) \diamonds C)$ means that if $S$ inputs an element of type $C$, it outputs a relation of type $A \diamonds B$ and $S(A \diamonds (B \diamonds C))$ means that if $S$ inputs a relation of type $B \diamonds C$, it outputs an element of type $A$. The first case turns out to be true. A more general case is proved in section 4.4.6. The latter case does not hold in general however. This is easy to see if we fill in $I$ for $A$, $B$ and $C$. The ‘desired’ interpretation of $S(I \diamonds I) S$ would be that if $S$ inputs a relation of type $I \diamonds I$, it outputs an element of type $I$. This is equivalent to true. However,

$$S(I \diamonds I) S$$

$$\equiv \{ \text{definition of } \diamonds \}$$

$$S(I \diamonds I) \subseteq I - S$$

$$\equiv \{ I \diamonds I = \subseteq \text{ and } I \text{ is unit of } \}$$

$$S \subseteq \subseteq S$$

which is not equivalent to true for every $S$ (take for example $S = I$).

To solve this ‘problem’ that type formulas do not always mean what we ‘desire’ them to mean, we define the type operator $\cdots \square \cdots$ by

$$R \cdots P = I$$

We call this type operator the set-forcer. If we combine this type operator with $\cdots \diamonds \cdots$ in the way described in section 4.2.4, we obtain the type operator $\cdots \diamonds \cdots$. The formula $S \in A \diamonds (B \diamonds C)$ does have the ‘desired’ interpretation that $S$ outputs an element of type $A$ if it inputs a relation of type $B \diamonds C$. The fact that we get the ‘desired’ interpretation for type formulas that use set-forced type operators, is a trivial consequence of the fact that we designed each operator $\cdots \square \cdots$ such that $S(A \circ \square B) S$ had the ‘desired’ interpretation for sets $A$ and $B$, combined with the fact that $A \circ \square B$ is the set satisfying

$$S \in A \circ \square B \equiv S(A \circ \square B) S$$

As mentioned in section 4.2.6, $\cdots$ does not necessarily have to be set-forced because $A \cdots B$ is already a set for sets $A$ and $B$.

### 4.2.8 Extensibility

A set-preserving type operator $\circ \square$ is extensible if it does not impose any restrictions on a relation for input elements that are not element of the input type:

$$\circ \square \text{ is extensible}$$

$$\equiv \forall(A, B, S, X) S \in A \circ \square B \mid S \cup B \cup X \in (I \setminus B) \in A \circ \square B$$

78
The type operators $\rightarrow \circ$, $\leftarrow$, $\rightarrow$ and all their combinations (as defined in section 4.2.4) are extensible. The type operator $\leftarrow \circ$ is not extensible however.

### 4.2.9 Type diagrams

An insightful way to present type formulas, is by means of a **type diagram**. This is a graph where nodes represent sets and edges represent (set-preserving) type operators. We write the elements that are being typed *above* their corresponding type operator. The typing

$$ R \in A \rightarrow B \land S \in B \rightarrow C \land T \in D \rightarrow (C \land U \in A \rightarrow D $$

is for example represented by the type diagram

```
    C
   / \  \
  S   T
 /     \  \\  
B    A  D
```

Type diagrams express a little more than suggested by the above example. The external type associated with a node is fixed. The type diagram

```
    R
   / \  \
  I   S
 /     \  \\  
A    B
```

expresses not only that

$$ R \in A \leftarrow I \land S \in I \leftarrow B $$

but also that the inputs of $R$ have the same external type as the outputs of $S$. Without the type diagram, this would be apparent from the fact that we somewhere use a formula like $R \circ S$.

### 4.2.10 Twelve type operators

If we take a close look at the five non-combined type operators we have defined so far:
\[ S(R \rightarrow P) Q \equiv S \circ P \subseteq R \circ Q \]
\[ S(R \leftarrow P) Q \equiv S \circ P \circ Q' \subseteq R \]
\[ S(R \rightarrow P) Q \equiv P \subseteq S' \circ R \circ Q \]
\[ S(R \rightarrow P) Q \equiv S \subseteq R \circ Q' \circ P' \]
\[ S(R \rightarrow P) Q \equiv Q' \subseteq P' \circ S' \circ R \]

we see that each of them can be obtained from the others by ‘shunting’ a little. A table of twelve type operators can be obtained by ‘shunting’ a little more:

\[ S(R \leftarrow P) Q \equiv S \circ P \circ Q' \subseteq R \]
\[ S(R \rightarrow P) Q \equiv R' \circ S \subseteq Q \]
\[ S(R \rightarrow P) Q \equiv Q' \circ R' \subseteq P' \]
\[ S(R \rightarrow P) Q \equiv P' \circ Q' \subseteq S' \]
\[ S(R \rightarrow P) Q \equiv P \subseteq S' \circ R \]
\[ S(R \rightarrow P) Q \equiv S' \subseteq R \circ Q' \circ P' \]
\[ S(R \rightarrow P) Q \equiv R' \subseteq Q' \circ P' \circ S' \]
\[ S(R \rightarrow P) Q \equiv Q' \subseteq P' \circ S' \circ R \]

A property that these equations share, is that the external type of the inputs of \( S \) is equal to the external type of the outputs of \( P \), the external type of the inputs of \( P \) is equal to the external type of the inputs of \( Q \), the external type of the outputs of \( Q \) is equal to the external type of the inputs of \( R \) and the external type of the outputs of \( R \) is equal to the external type of the outputs of \( S \). The following type diagrams give an overview of the above definitions:
These type diagrams are not definitions of the type operators. For that we need **semi-commutative diagrams** as described in for example [11].

There exist various interesting correspondences between these twelve type operators, some of which are reflected by their symbols. For each group of four type operators, we have for example that the substitution

\[ S, R, P, Q := R^\sim, Q, S, P^\sim \]

gives the next type operator in the group. In particular, after two steps we get the ‘mirrored’ version of a type operator, for example:

\[ S (R \leftrightarrow P) Q \equiv Q^\sim (P^\sim \rightarrow R^\sim) S^\sim \]

For sets \( A \) and \( B \) we have

\[ S \in A \leftrightarrow B \equiv S^\sim \in B \rightarrow A \]

The shunting theorems give us that the definitions of \( \cdots \circ, \leftarrow, \rightarrow \cdots \) and \( \cdots \) are equivalent if \( S \) and \( Q \) are total functions. The first three of these four type operators play a main role in our theory. They allow us to express upper-range, functionality and totality in an extensible manner, as mentioned in section 4.2.8.
4.3 Type-generalising some other concepts

In the previous section we generalised functionality and totality such that an input type and an output type were included. In this section we generalise some other concepts in a similar manner.

4.3.1 Emptiness and fullness

A relation \( S \) is \textbf{empty on} \( B \) if it is empty in every element of \( B \), or equivalently \( S \upharpoonright B = \emptyset \). A relation \( S \) is \textbf{full on} \( B \) if it is full in every element of \( B \), or equivalently \( S \upharpoonright B = \Pi \upharpoonright B \).

4.3.2 Reflexivity

For a set \( A \), a relation \( S \) is \textbf{\( A \)-reflexive} if \( A \) is a subrelation of \( S \):

\[
A \subseteq S
\]

A relation that is \( A \)-reflexive and transitive is called an \textbf{\( A \)-preorder} and an antisymmetric \( A \)-preorder is called an \textbf{\( A \)-partial-order}.

4.3.3 Shunting

The shunting theorems that we introduced in section 3.8.1 can be generalised from total functions to relations that only need to be single-valued on a certain set. For \( g \in I \leftarrow S \) and \( h \in I \leftarrow S \) we have

\[
S \subseteq g^{-} \circ R \circ h \equiv g^{-} \circ S \circ h^{-} \subseteq R
\]

or equivalently:

\[
S \subseteq g^{-} \circ R \circ h \equiv S \subseteq g^{-} \circ R \circ h^{-}
\]

For \( g \in I \leftarrow \{ y \} \) and \( h \in I \leftarrow \{ z \} \) we have

\[
\begin{align*}
y \left( g^{-} \circ R \circ h \right) z & \equiv y \in g^{-} \land g.y \left( R \right) h.z \land z \in h^{-} \\
y \left( g^{-} \circ R \circ h^{-} \right) z & \equiv y \in g^{-} \Rightarrow g.y \left( R \right) h.z \leftarrow z \in h^{-}
\end{align*}
\]

In particular, for \( g \in I \leftarrow \{ y \} \) and \( h \in I \leftarrow \{ z \} \) we have

\[
\begin{align*}
y \left( g^{-} \circ R \circ h \right) z & \equiv g.y \left( R \right) h.z \\
y \left( g^{-} \circ R \circ h^{-} \right) z & \equiv g.y \left( R \right) h.z
\end{align*}
\]

and for sets \( A \) and \( B \) we have
\[ y (A \cdot R \cdot B) z \equiv y \in A \land y (R) z \land z \in B \]
\[ y (A \backslash R / B) z \equiv y \in A \Rightarrow y (R) z \Leftarrow z \in B \]

The proofs of these theorems are left to the reader.

### 4.4 Properties of the cylindric-type operator

In this section we investigate some algebraic properties of the cylindric-type operator.

#### 4.4.1 Inclusion

We already mentioned that \(\circ\) is monotonic in its output type. If we take a non-empty relation as input type, we can strengthen monotonicity to monomorphy:

\[ R \subseteq T \equiv \{ \bullet P \text{ is non-empty} \} \]

\[ R \circ P \subseteq T \circ P \]

**proof**

\[ R \subseteq T \]
\[ \Rightarrow \{ \text{monotonicity of } \circ P \} \]
\[ R \circ P \subseteq T \circ P \]
\[ \equiv \{ \text{definition of } \subseteq \text{ and } \circ \} \]
\[ \forall \langle S, Q \gtrless S \cdot P \subseteq R \cdot Q \Rightarrow S \cdot P \subseteq T \cdot Q \rangle \]
\[ \Rightarrow \{ \text{take } S = \{ y \} \cdot \Pi \text{ and } Q = \{ z \} \cdot \Pi \} \]
\[ \forall \langle y, z \gtrless \{ y \} \cdot \Pi \cdot P \subseteq R \cdot \{ z \} \cdot \Pi \Rightarrow \{ y \} \cdot \Pi \cdot P \subseteq T \cdot \{ z \} \cdot \Pi \rangle \]
\[ \equiv \{ \text{shunting, } \{ z \} \cdot \Pi \text{ is a total function and } (\{ z \} \cdot \Pi) \cdot \Pi = \Pi \cdot \{ z \} \} \]
\[ \forall \langle y, z \gtrless \{ y \} \cdot \Pi \cdot P \cdot \Pi \cdot \{ z \} \subseteq R \Rightarrow \{ y \} \cdot \Pi \cdot P \cdot \Pi \cdot \{ z \} \subseteq T \rangle \]
\[ \equiv \{ \bullet P \text{ is non-empty, then } \Pi \cdot P \cdot \Pi = \Pi \} \]
\[ \forall \langle y, z \gtrless \{ y \} \cdot \Pi \cdot \{ z \} \subseteq R \Rightarrow \{ y \} \cdot \Pi \cdot \{ z \} \subseteq T \rangle \]
\[ \equiv \{ \{ y \} \cdot \Pi \cdot \{ z \} \subseteq R \equiv y (R) z \} \]
\[ \forall \langle y, z \gtrless y (R) z \Rightarrow y (T) z \rangle \]
\[ \equiv \{ \text{definition of } \subseteq \} \]
\[ R \subseteq T \]

#### 4.4.2 Equality

Using the definition of equality in terms of inclusion, a trivial consequence of the inclusion theorem of the previous section is
\[ R = T \]
\[ \equiv \quad \{ \bullet P \text{ is non-empty} \} \]
\[ R \preceq P = T \preceq P \]

### 4.4.3 Intersection

If we restrict the inputs to functions, \((\preceq P)\) distributes over the intersection:

\[
\Pi_i (I \leftarrow I) \cap (R \cap T \preceq P) = \Pi_i (I \leftarrow I) \cap (R \preceq P \cap T \preceq P)
\]

**proof**

\[
S (R \cap T \preceq P) Q = S \circ P \subseteq (R \cap T) \circ Q
\]
\[
\equiv \quad \{ \text{definition of } \preceq \}\]
\[
S \circ P \subseteq (R \cap T) \circ Q
\]
\[
\equiv \quad \{ \text{distribution of } \circ \text{ over } \cap \text{ for functions, } \bullet Q \text{ is a function} \}
\[
S \circ P \subseteq R \circ Q \cap T \circ Q
\]
\[
\equiv \quad \{ \text{point-free definition of } \cap \}\]
\[
S \circ P \subseteq R \circ Q \land S \circ P \subseteq T \circ Q
\]
\[
\equiv \quad \{ \text{definition of } \preceq \}\]
\[
S (R \preceq P) Q \land S (T \preceq P) Q
\]
\[
\equiv \quad \{ \text{point-wise definition of } \cap \}\]
\[
S (R \preceq P \cap T \preceq P) Q
\]

### 4.4.4 Union

If we restrict the outputs to functions, then for all relations \(P\) with a unique connection, \((\preceq P)\) distributes over the union:

\[
(I \leftarrow I) \Pi \cap (R \cup T \preceq P) = (I \leftarrow I) \Pi \cap (R \preceq P \cup T \preceq P)
\]

**proof**
\[ S \left( R \cup T \circ P \right) Q \]
\[ \equiv \{ \text{definition of } \circ \} \]
\[ S \cdot P \subseteq (R \cup T) \circ Q \]
\[ \equiv \{ \text{distribution of } \circ \text{ over } \cup \} \]
\[ S \cdot P \subseteq R \cdot Q \cup T \cdot Q \]
\[ \equiv \{ S \cdot P \text{ has a unique connection if } * \text{ } S \text{ is functional and } * \text{ } P \text{ has a unique connection} \} \]
\[ S \cdot P \subseteq R \cdot Q \lor S \cdot P \subseteq T \cdot Q \]
\[ \equiv \{ \text{definition of } \circ \} \]
\[ S (R \circ P) Q \lor S (T \circ P) Q \]
\[ \equiv \{ \text{point-wise definition of } \cup \} \]
\[ S (R \circ P \cup T \circ P) Q \]

4.4.5 Sequential composition

For all sets \( P \), \((\circ \circ P)\) distributes over the sequential composition:

\[ R \cdot T \circ \circ P \]
\[ \equiv \{ * \text{ } P \text{ is a set} \} \]
\[ (R \circ \circ P) \circ (T \circ \circ P) \]

**proof**

\[ S (R \cdot T \circ \circ P) Q \]
\[ \equiv \{ \text{definition of } \circ \circ \} \]
\[ S \cdot P \subseteq R \cdot T \cdot Q \]
\[ \equiv \{ * \text{ } P \text{ is a set, then } (\Rightarrow): \text{ take } X = T \cdot Q, (\Leftarrow): S \cdot P \subseteq U \ \equiv \ S \cdot P \subseteq U \cdot P \} \]
\[ \exists (X \ | \ S \cdot P \subseteq R \cdot X \ \land \ X \cdot P \subseteq T \cdot Q) \]
\[ \equiv \{ \text{definition of } \circ \circ \} \]
\[ \exists (X \ | \ S (R \circ \circ P) X \ \land \ X \circ \circ (T \circ \circ P) Q) \]
\[ \equiv \{ \text{definition of } \circ \} \]
\[ S ((R \circ \circ P) \circ (T \circ \circ P)) Q \]

4.4.6 Identity relation

In section 4.2.7 we saw that for sets \( A \) and \( B \), \( A \circ B \) is not a set in general. In this section we search for properties that provide a little compensation. When we introduced \( \circ \circ \) in section 4.2.7, we remarked that although \( A \circ B \) is not a set, for a set \( C \), \( S ((A \circ B) \circ C) \) still had the desired interpretation:

\[ S ((A \circ B) \circ C) S \equiv S \in (A \circ B) \circ C \]
Stated otherwise:

$$(A \rightarrow\rightarrow B) \rightarrow\rightarrow C = (A \rightarrow\rightarrow B) \rightarrow\rightarrow C$$

The general case

$$(\ldots((A \rightarrow\rightarrow B_0) \rightarrow\rightarrow B_1)\ldots) \rightarrow\rightarrow B_n = \ldots((A \rightarrow\rightarrow B_0) \rightarrow\rightarrow B_1)\ldots) \rightarrow\rightarrow B_n$$

can be proved if we find a set of relations $W$ such that for all sets $B$

$$R \in \wp I \implies R \in W$$
$$R \in W \implies (R \rightarrow\rightarrow B) \in W$$
$$R \in W \implies R \rightarrow\rightarrow B = R \cap I \rightarrow\rightarrow B$$

We know that $W = \wp I$ does not satisfy the second property, so we need to find another $W$. We use the third property to try to find such a $W$. The `$\supseteq$' even holds without any restrictions on $R$, so we should use the `$\subseteq$' to find a proper restriction on $R$. The following calculation indicates a valid restriction:

$$S(R \rightarrow\rightarrow B) S$$
$$\equiv \{ \text{definition of } \rightarrow\rightarrow \}$$
$$S \cdot B \subseteq R \cdot S$$
$$\Rightarrow \{ S \subseteq \Pi \}$$
$$S \cdot B \subseteq R \cdot \Pi$$
$$\equiv \{ R \cdot \Pi = R \cdot \Pi \}$$
$$S \cdot B \subseteq R \cdot \Pi$$
$$\equiv \{ \bullet R = R \cap I \}$$
$$S \cdot B \subseteq (R \cap I) \cdot \Pi$$
$$\equiv \{ S \cdot B \subseteq A \cdot \Pi \equiv S \cdot B \subseteq A \cdot S \text{ for sets } A \text{ and } B \}$$
$$S \cdot B \subseteq (R \cap I) \cdot S$$
$$\equiv \{ \text{point-wise definition of } \rightarrow\rightarrow \}$$
$$S(R \cap I \rightarrow\rightarrow B) S$$

For the `$\bullet$' in this calculation we have:

$$R \cap I = R \cap I$$
$$\equiv \{ R \supseteq R \cap I \}$$
$$R \cap I \subseteq R \cap I$$
$$\equiv \{ R \subseteq I \}$$
$$R \subseteq R$$

86
So, if we choose \( W \) to be the set \( R_{no} \) (range owning), defined by

\[
R \in R_{no} \equiv R \subseteq R
\]

we can prove the third property. We now have to prove that \( W \) satisfies the first and the second property. The proof of the first property is trivial. For the proof of the second property we use the following point-wise formulation of the definition of \( R_{no} \), the proof of which is left to the reader:

\[
R \in R_{no} \equiv \forall (y, z \mid y (R) z \mid y (R) y)
\]

Using the point-free formulation of \( R \in R_{no} \) for its first occurrence in the second property and this point-wise formulation for its second occurrence, we thus have to prove that

\[
R \subseteq R \Rightarrow \forall (S, Q \mid S (R \leftarrow\rightarrow B) Q \leftarrow\rightarrow S (R \leftarrow\rightarrow B) S)
\]

For a set \( B \) we calculate

\[
S (R \leftarrow\rightarrow B) Q \equiv \{ \text{point-wise definition of } \leftarrow\rightarrow \}
\]

\[
S \cdot B \subseteq R \cdot Q \Rightarrow \{ Q \subseteq \Pi \}
\]

\[
S \cdot B \subseteq R \cdot \Pi
\]

\[
\equiv \{ R \cdot \Pi = R \cdot \Pi \}
\]

\[
S \cdot B \subseteq R \cdot \Pi
\]

\[
\equiv \{ S \cdot B \subseteq A \cdot \Pi \equiv S \cdot B \subseteq A \cdot S \text{ for sets } A \text{ and } B \}
\]

\[
S \cdot B \subseteq R \cdot S
\]

\[
\Rightarrow \{ \text{point-wise definition of } \leftarrow\rightarrow \}
\]

\[
S (R \leftarrow\rightarrow B) S
\]

So, the set \( R_{no} \) satisfies all three properties, thus proving that ‘as long as we use the cylindric-type operator in the output type, everything goes well’.

Summarising the results, we have that for a set \( P \), \( \leftarrow\rightarrow P \) preserves \( R_{no} \):

\[
R \in R_{no} \Rightarrow \{ \text{point-wise definition of } \leftarrow\rightarrow \}
\]

\[
(R \leftarrow\rightarrow P) \in R_{no}
\]

and is what we call diagonal preserving:
\[(R \cap I \circ P) \cap I\]
\[\{\bullet P \text{ is a set and } \bullet R \in Rno\}\]
\[(R \circ P) \cap I\]

4.4.7 Converse

For all sets \(P\), \((\circ P)\) commutes with the converse if we restrict to connections between functions with equal domains:

\[(1 \leftarrow I).DEQ.(1 \leftarrow I) \cap (R' \circ P)\]
\[= (1 \leftarrow I).DEQ.(1 \leftarrow I) \cap (R \circ P)\]

**proof**

\[S(R' \circ P)Q\]
\[\equiv \{\text{definition of } \circ\}\]
\[S \cdot P \subseteq R' \cdot Q\]
\[\equiv \{\text{shunting, } \bullet Q \in I \leftarrow I, \bullet S \subseteq Q\text{ and } \bullet P \text{ is a set}\}\]
\[S \cdot P \subseteq R' \subseteq Q\]
\[\equiv \{\text{shunting, } \bullet S \in I \leftarrow I, \bullet Q \subseteq S\text{ and } \bullet P \text{ is a set}\}\]
\[P \cdot Q' \subseteq S' \cdot R'\]
\[\equiv \{\bullet P \text{ is symmetric}\}\]
\[P' \cdot Q' \subseteq S' \cdot R'\]
\[\equiv \{(Y' \cdot X' = (X \cdot Y'))\}\]
\[(Q' \cdot P') \subseteq (R' \cdot S')\]
\[\equiv \{X' \subseteq Y' \equiv X \subseteq Y\}\]
\[Q' \cdot P \subseteq R' \cdot S\]
\[\equiv \{\text{definition of } \circ\}\]
\[Q(R \circ P)S\]
\[\equiv \{\text{point-wise definition of } \circ\}\]
\[S((R \circ P)' \circ Q)\]

4.5 Conclusions

In this chapter we introduced a rich collection of type operators that enable us to express some important properties of relations in a clear and concise manner. We paid special attention to the cylindric-type operator that generalises the traditional co/contra-variant typing of functional programming. The main reason for our interest in this type operator is that it is extensible. Extensibility is an important concept in the context of object-oriented programming. For our purposes, it is
more important than the fact that the cylindric type operator has some ‘algebraic conflicts’ with the identity (I \(\circ\) I \(\neq\) I) and the converse (\(R \in A \rightarrow B\) does not imply \(R' \in B \rightarrow A\) in general).
Chapter 5

Products

Products provide a means to combine information into packages that satisfy useful algebraic laws. In this chapter we introduce and investigate several different products. Products are used in chapter 6 for the construction of a model for expressions and in chapter 7 for the hierarchical structuring of procedures.

5.1 Cylindric product

In allegory theory [11], the product $R_0 \times R_1$ of two relations $R_0$ and $R_1$ is defined by postulating the existence of two projection functions $\text{exl}$ and $\text{exr}$ that satisfy the following properties (the set $I \times I$ is assumed to be already defined):

$\text{exl} \circ \text{exr} = \Pi$

$\text{exl} \circ \text{exl} \cap \text{exr} \circ \text{exr} = I \times I$

and stating that

$R_0 \times R_1 = \text{exl} \circ R_0 \cap \text{exr} \circ R_1$

The binary cartesian product $\times$ is a construct of this kind. The projection functions $\text{exl}$ and $\text{exr}$ are $\text{ex}_\emptyset$ and $\text{ex}_\emptyset$ respectively, defined by

$\text{ex}_i = (\cdot, i) \circ (I \leftarrow \emptyset)$

The set $I \leftarrow \emptyset$, that is equal to $1 \times 1$, constrains the input to pairs and the section $(\cdot, i)$ yields their output for input $i$. For $f \in I \leftarrow \emptyset$ and $i \in \emptyset$ we thus have

$\text{ex}_i.f = f.i$
We now show a reformulation of the allegoric definition of the binary cartesian product that gives rise to one of the main constructs of this thesis. We start with the lemma that is the core of this reformulation:

For \( g, h \in I \leftrightarrow \{i\} \) we have

\[
g (\langle i \rangle \cdot R \cdot \langle .i \rangle) h \equiv g (R \cdot \circ \{i\}) h
\]

**proof**

Assume \( g, h \in I \leftrightarrow \{i\} \).

\[
g (\langle i \rangle \cdot R \cdot \langle .i \rangle) h
\]

\[
\equiv \{\text{shunting}\}
\]

\[
g. i (R) h.i
\]

\[
\equiv \{\text{shunting}\}
\]

\[
i (g \cdot R \cdot h) i
\]

\[
\equiv \{i (R) i \equiv \{i\} \subseteq R\}
\]

\[
\{i\} \subseteq g \cdot R \cdot h
\]

\[
\equiv \{\text{shunting}\}
\]

\[
g \cdot \{i\} \subseteq R \cdot h
\]

\[
\equiv \{\text{definition of} \cdot \circ\}
\]

\[
g (\langle R \cdot \circ \{i\} \rangle) h
\]

Using this lemma, we can reformulate the allegoric definition of the cartesian product as follows:

\[
R_0 \times R_1
\]

\[
= \{\text{allegoric definition}\}
\]

\[
\text{ex}_{\circ} \cdot R_0 \cdot \text{ex}_{\circ} \cap \text{ex}_{\circ} \cdot R_1 \cdot \text{ex}_{\circ}
\]

\[
= \{\text{definition of} \text{ex}_{i}\}
\]

\[
\left( (I \leftrightarrow 2) \cdot (\circ) \right) \cdot R_0 \cdot (\circ) \circ (I \leftrightarrow 1) \cap
\left( (I \leftrightarrow 2) \cdot (\circ) \right) \cdot R_1 \cdot (\circ) \circ (I \leftrightarrow 1)
\]

\[
= \{\text{use the lemma,} \ 1 \leftrightarrow 1 \subseteq I \leftrightarrow \{i\} \text{ for} \ i \in \mathbb{2}\}
\]

\[
\left( (I \leftrightarrow 2) \cdot (R_0 \cdot \circ \{\circ\}) \circ (I \leftrightarrow 1) \cap
\left( (I \leftrightarrow 2) \cdot (R_1 \cdot \circ \{\circ\}) \circ (I \leftrightarrow 1)
\right)
\]

\[
= \{\text{distribution of} \cdot \circ \text{ over} \cap \text{ for sets}\}
\]

\[
(I \leftrightarrow 2) \cdot (R_0 \cdot \circ \{\circ\} \cap R_1 \cdot \circ \{\circ\}) \circ (I \leftrightarrow 1)
\]

We now see that the binary cartesian product is a constrained version of a construct that we shall call the **binary cylindric product**. The binary cylindric product \( \circ_{\circ} \in ((I \leftrightarrow 1) \circ (1 \leftrightarrow I)) \leftrightarrow (1 \leftrightarrow 1) \times (1 \leftrightarrow 1) \) is defined by

\[
R_0 \circ R_1 = (R_0 \cdot \circ \{\circ\}) \cap (R_1 \cdot \circ \{\circ\})
\]
The binary cartesian product can be expressed in terms of the cylindric product as follows:

\[ R_0 \times R_1 = (I \leftarrow \{\underline{1}\}) \cdot R_0 \otimes R_1 \cdot (I \leftarrow \{\underline{1}\}) \]

Another basic construct in allegory theory is the sum \( R_0 + R_1 \) of two relations \( R_0 \) and \( R_1 \). This construct is defined by postulating the existence of two injection functions \( \text{inl} \) and \( \text{inr} \) that satisfy the following properties (the set \( I + I \) is assumed to be already defined):

\[
\begin{align*}
\text{inl}^{-1} \circ \text{inr} & = \emptyset \\
\text{inl}^{-1} \circ \text{inl} & = I \\
\text{inr}^{-1} \circ \text{inr} & = I \\
\text{inl}^{-1} \circ \text{inl} \cup \text{inr}^{-1} \circ \text{inr} & = I + I
\end{align*}
\]

and stating that

\[ R_0 + R_1 = \text{inl}^{-1} \circ R_0 \circ \text{inl} \cup \text{inr}^{-1} \circ R_1 \circ \text{inr} \]

The binary cartesian sum \( + \) is a construct of this kind. An element \( x \) tagged with an \( i \), is modeled as \([x, i]\). Notice that \( I \leftarrow \{i\} \) is the set of all elements that are tagged with \( i \). The injection functions \( \text{inl} \) and \( \text{inr} \) are \( \text{in}_{\underline{1}} \) and \( \text{in}_{\underline{0}} \) respectively, defined by

\[ \text{in}_i = (I \leftarrow \{i\}) \circ (i)^{-1} \]

or point-wise:

\[ \text{in}_i \cdot x = [(x, i)] \]

We now show that the cartesian sum is also a constrained version of the cylindric product:

\[ R_0 + R_1 = ((I \leftarrow \{\underline{1}\}) \cup (I \leftarrow \{\underline{0}\})) \circ R_0 \otimes R_1 \circ ((I \leftarrow \{\underline{1}\}) \cup (I \leftarrow \{\underline{0}\})) \]

At first sight the proof may look complicated because of the big terms involved, but it is rather straightforward:
\[
((I \leftrightarrow \{ \circ \}) \cup (I \leftrightarrow \{ \circ \})) \circ R_0 \otimes R_1 \circ ((I \leftrightarrow \{ \circ \}) \cup (I \leftrightarrow \{ \circ \}))
\]

= \{\text{definition of } \otimes\}

\[
((I \leftrightarrow \{ \circ \}) \cup (I \leftrightarrow \{ \circ \})) \circ (R_0 \circ \circ \{ \circ \} \cap R_1 \circ \circ \{ \circ \}) \circ ((I \leftrightarrow \{ \circ \}) \cup (I \leftrightarrow \{ \circ \}))
\]

= \{\text{distribution of } \cap \text{ over } \cup \text{ for sets}\}

\[
((I \leftrightarrow \{ \circ \}) \cup (I \leftrightarrow \{ \circ \})) \circ (R_0 \circ \circ \{ \circ \}) \circ ((I \leftrightarrow \{ \circ \}) \cup (I \leftrightarrow \{ \circ \})) \cap
((I \leftrightarrow \{ \circ \}) \cup (I \leftrightarrow \{ \circ \})) \circ (R_1 \circ \circ \{ \circ \}) \circ ((I \leftrightarrow \{ \circ \}) \cup (I \leftrightarrow \{ \circ \}))
\]

= \{\text{distribution of } \cap \text{ over } \cup\}

\[
((I \leftrightarrow \{ \circ \}) \circ (R_0 \circ \circ \{ \circ \}) \circ (I \leftrightarrow \{ \circ \})) \cup
((I \leftrightarrow \{ \circ \}) \circ (R_1 \circ \circ \{ \circ \}) \circ (I \leftrightarrow \{ \circ \})) \cap
((I \leftrightarrow \{ \circ \}) \circ (R_0 \circ \circ \{ \circ \}) \circ (I \leftrightarrow \{ \circ \})) \cup
((I \leftrightarrow \{ \circ \}) \circ (R_1 \circ \circ \{ \circ \}) \circ (I \leftrightarrow \{ \circ \}))
\]

= \{\text{use the lemma from the beginning of this section, } I \leftrightarrow \{ i \} \subseteq I \leftrightarrow \{ i \}\}

\[
((I \leftrightarrow \{ \circ \}) \circ (R_0 \circ \circ \{ \circ \}) \circ (I \leftrightarrow \{ \circ \})) \cup
((I \leftrightarrow \{ \circ \}) \circ (R_1 \circ \circ \{ \circ \}) \circ (I \leftrightarrow \{ \circ \}))
\]

= \{\text{definition of } \circ \}

\[\text{in}_\circ R_0 \circ \circ \{ \circ \} \cup \text{in}_\circ R_1 \circ \circ \{ \circ \}\]

= \{\text{algebraic definition of } \circ \}

\[R_0 + R_1\]

So we see that both the cartesian product and the cartesian sum are constrained forms of the cylindric product. In the next section we introduce several other constrained forms.

### 5.2 Constrained cylindric products

The binary \( M \)-constrained cylindric product \(_\circ M \subseteq ((I \rightarrow I) \rightarrow (I \rightarrow I)) \leftrightarrow (I \rightarrow I) \times (I \rightarrow I)\) is defined by

\[R_0 \circ M R_1 = R_0 \circ R_1 \cap M\]

The constraints \( M \) we investigate, are combinations of the constraints \( UNI, \ FUN, MNP, \ DEQ, \ TOT_B \) and \( RES_B \) (for a set \( B \)), defined by
\[
S (UNI ) Q \equiv S \text{ and } Q \text{ both have a unique input}
\]
\[
S (FUN ) Q \equiv S \text{ and } Q \text{ are functional}
\]
\[
S (NMP ) Q \equiv S \text{ and } Q \text{ are non-empty}
\]
\[
S (DEQ ) Q \equiv S \text{ and } Q \text{ are domain equal}
\]
\[
S (TOT_B ) Q \equiv S \text{ and } Q \text{ are total on } B
\]
\[
S (RES_B ) Q \equiv S \text{ and } Q \text{ are restricted to } B
\]

Notice that all constraints except \(DEQ\) are squares (see section 3.4.4 for the definition of squares).

The cartesian product and sum are obtained as follows:

\[
R_0 \times R_1 = R_0 \otimes_{\text{FUN} \cap \text{TOT} \cap \text{RES}_1} R_1
\]
\[
R_0 + R_1 = R_0 \otimes_{\text{UNI} \cap \text{FUN} \cap \text{NMP} \cap \text{RES}_1} R_1
\]

This is a straightforward result of the fact that

\[
(\{\bullet\}) \Pi (I \leftarrow \{\bullet\}) = \text{FUN} \cap \text{TOT} \cap \text{RES}_1
\]

and

\[
((I \leftarrow \{\bullet\}) \cup (I \leftarrow \{\bullet\})) \Pi (\{\bullet\}) \cup (I \leftarrow \{\bullet\}) = \text{UNI} \cap \text{FUN} \cap \text{NMP} \cap \text{RES}_1
\]

Adding \(DEQ\) makes no difference in these two cases:

\[
R_0 \times R_1 = R_0 \otimes_{\text{FUN} \cap \text{DEQ} \cap \text{TOT} \cap \text{RES}_1} R_1
\]
\[
R_0 + R_1 = R_0 \otimes_{\text{UNI} \cap \text{FUN} \cap \text{NMP} \cap \text{DEQ} \cap \text{RES}_1} R_1
\]

For \(\times\) this holds because \(\text{TOT} \cap \text{RES}_1 \subseteq \text{DEQ}\).

For \(+\) it holds because

\[
S (UNI \cap \text{NMP} \cap \text{RES}_1) Q \land S (R_0 \otimes R_1) Q
\]

\[
\Rightarrow S \text{ and } Q \text{ are domain equal}
\]

The proof of this is left to the reader.

Because \(\text{TOT} \subseteq \text{NMP}\), we also have

\[
R_0 \times R_1 = R_0 \otimes_{\text{FUN} \cap \text{NMP} \cap \text{TOT} \cap \text{RES}_1} R_1
\]
\[
R_0 + R_1 = R_0 \otimes_{\text{FUN} \cap \text{NMP} \cap \text{DEQ} \cap \text{TOT} \cap \text{RES}_1} R_1
\]

95
Because we need the constraint from + more often, we introduce a shorthand for it, where we generalise $\text{RES}_2$ to $\text{RES}_B$ for an arbitrary set $B$:

$$T A G_B = \text{UNI} \cap \text{FUN} \cap \text{NMP} \cap \text{DEQ} \cap \text{RES}_B$$

or equivalently:

$$T A G_B = \bigcup \{ (I \rightarrow \{i\}) \circ \Pi \circ (I \leftarrow \{i\}) \mid i \in B \mid i \}$$

So,

$$R_0 + R_1 = R_0 \otimes_{T A G} R_1$$

Before we investigate the properties of the binary $M$-constrained cylindric product, we first introduce two related operators.

### 5.3 Constrained cylindric packs

In the previous chapter we proved several properties of $(\cdots P)$. For most of these properties we imposed a restriction on $P$, like "$P$ is a set" in section 4.4.7. The only $P$s that satisfy all these restrictions are sets with a single element, called **singleton sets**. Next to the restrictions on $P$, we also encountered some restrictions on the inputs and outputs, like "$(I \rightarrow \{i\}) \circ \text{DEQ} \circ (I \leftarrow \{i\})$" in section 4.4.7. These two facts suggest the introduction of an operator that we call the **$M$-constrained cylindric pack** $\odot_M \in ((I \rightarrow \{i\}) \rightarrow (I \leftarrow \{i\})) \leftarrow (I \rightarrow \{i\}) \times I$, defined by

$$R \odot_M i = (R \cdots (\{i\}) \cap M$$

Using this operator, $\odot_M$ can also be defined equivalently by

$$R_0 \odot_M R_1 = R_0 \odot_M \odot_M \cap R_1 \odot_M \odot_M$$

The cylindric pack puts a relation in a **package**, attaching a certain **tag** to it. The **pick** $\odot_M \in (I \rightarrow \{i\}) \leftarrow (I \leftarrow \{i\}) \times I$ retrieves a relation with a certain tag from a package. It is defined by

$$y \odot M i z \equiv [(y,i)] (X) [(z,i)]$$

or point free:

$$X \odot i = \text{in}_i^\ast \circ X \circ \text{in}_i$$

or equivalently by (see section 3.8.1 about shunting)

$$X \odot i = \text{in}_i \setminus X / \text{in}_i \ast$$

96
The pick satisfies some useful algebraic properties that follow from the fact that \((\triangleright \downarrow)\) is both a lower adjoint and an upper adjoint (see section 3.5 about Galois connections). If we define the operators \(\triangleright \uparrow\) and \(\Triangleright \downarrow\) by

\[
R \triangleright \uparrow = \text{in}_i \setminus R / \text{in}_i \\
R \Triangleright \downarrow = \text{in}_i \circ R \circ \text{in}_i
\]

then \((\triangleright \uparrow)\) is the upper and \((\triangleright \downarrow)\) the lower adjoint of \((\triangleright)\):

\[
X \triangleright \downarrow \subseteq R \equiv X \subseteq R \triangleright \uparrow
\]

This is a trivial result of the point-free definitions of \(\setminus\) and \(/\) and the above two point-free definitions of \(\triangleright\).

The theory of Galois connections now gives us for free that \((\triangleright \downarrow)\), \((\triangleright \uparrow)\) and \((\triangleright \downarrow)\) are all monotonic:

\[
X \subseteq Y \Rightarrow X \triangleright \downarrow \subseteq Y \triangleright \downarrow \\
R \subseteq T \Rightarrow R \triangleright \uparrow \subseteq T \triangleright \uparrow \\
R \subseteq T \Rightarrow R \Triangleright \downarrow \subseteq T \Triangleright \downarrow
\]

and that \((\triangleright \uparrow)\) distributes over \(\cap\), that \((\triangleright \downarrow)\) distributes over \(\cup\) and that \((\triangleright)\) distributes over both, specifically:

\[
\Pi \triangleright \downarrow i = \Pi \\
\emptyset \triangleright \downarrow i = \emptyset \\
U \cap V \triangleright \uparrow i = U \triangleright \uparrow i \cap V \triangleright \uparrow i \\
U \cup V \triangleright \uparrow i = U \triangleright \uparrow i \cup V \triangleright \uparrow i
\]

Furthermore, the following cancellation laws hold (the equalities hold because \((\triangleright)\) is surjective):

\[
R \triangleright \uparrow \triangleright \downarrow i = R \land X \subseteq X \triangleright \uparrow i \triangleright \downarrow i \\
R \Triangleright \downarrow \triangleright \uparrow i = R \land X \supseteq X \triangleright \uparrow i \triangleright \downarrow i
\]

We now show how \(\triangleright \uparrow\) and \(\triangleright \downarrow\) can be defined in terms of \(\dashv \circ\), to focus on their relationship with \(\otimes_M\):

\[
R \triangleright \uparrow i = \{ \text{in}_i \} \setminus (R \dashv \circ \{ i \}) / \{ \text{in}_i \} \\
R \Triangleright \downarrow i = \{ \text{in}_i \} \circ (R \dashv \circ \{ i \}) \circ \{ \text{in}_i \}
\]

Using the ‘core lemma’ of section 5.1, the proof of this is rather straightforward and left to the reader.

Making use of the fact that for sets \(A\) and \(B\)
we can write this equivalently as
\[ R \triangleright i = (R - \circ \{i\}) \subseteq \text{TAG}_{\{i\}} \]
\[ R \triangleright i = (R - \circ \{i\}) \cap \text{TAG}_{\{i\}} \]
A trivial consequence is
\[ R \triangleright M \subseteq R \triangleright i \]
\[ R \triangleright M \supseteq R \triangleright i \]
which gives us the following cancellation law:
\[ R \triangleright M \triangleright i \subseteq \text{TAG}_{\{i\}} \subseteq M \]
\[ R \triangleright M \triangleright i \supseteq \text{TAG}_{\{i\}} \subseteq M \]
We can use the following theorem in case the pack and pick use different tags:
\[ R \triangleright M j \triangleright i \subseteq \text{TAG}_{\{i\}} \subseteq M \land j \neq i \]
\[ R \triangleright M j \triangleright i \supseteq \text{TAG}_{\{i\}} \subseteq M \land j \neq i \]
\[ \begin{align*}
y \cdot (R \circ \mathcal{M}_j \triangleright i) \ z \\
\equiv & \quad \text{point-wise definition of } \triangleright \  \\
[(y, i)] \cdot (R \circ \mathcal{M}_j) \cdot [(z, i)] \\
\equiv & \quad \text{definition of } R \circ \mathcal{M}_j \  \\
[(y, i)] \cdot ((R \leftarrow \mathcal{M}_j) \cap M) \cdot [(z, i)] \\
\equiv & \quad \text{point-wise definition of } \cap \  \\
[(y, i)] \cdot (R \leftarrow \mathcal{M}_j) \cdot [(z, i)] \land [(y, i)] \cdot (M) \cdot [(z, i)] \\
\equiv & \quad \text{point-wise definition of } \leftarrow \mathcal{M}_j \  \\
[(y, i)] \cdot \{j\} \subseteq R \cdot [(z, i)] \land [(y, i)] \cdot (M) \cdot [(z, i)] \\
\equiv & \quad \{ \cdot j \neq i, \text{then } [(y, i)] \cdot \{j\} = \emptyset \} \  \\
[(y, i)] \cdot (M) \cdot [(z, i)] \\
\equiv & \quad \{ \cdot \text{TAG}_i \subseteq M \} \  \\
\text{true} \\
\equiv & \quad \text{point-wise definition of } \Pi \  \\
y \cdot (\Pi) \ z \\
\end{align*} \]

We also refer to this theorem as “cancellation”.

Because \((\leftarrow \mathcal{M}_j \mathcal{P})\) and \((\cap M)\) are monotonic, \((\mathcal{M}_j i)\) is also monotonic:

\[ R \subseteq T \Rightarrow R \circ \mathcal{M}_j i \subseteq T \circ \mathcal{M}_j i \]

If we add the restriction that \(\text{TAG}_{i(i)} \subseteq M\), we can strengthen it to monomorphy:

\[ R \subseteq T \]
\[ \equiv \{ \cdot \text{TAG}_{i(i)} \subseteq M \} \]
\[ R \circ \mathcal{M}_j i \subseteq T \circ \mathcal{M}_j i \]

**proof**

\[ R \subseteq T \]
\[ \Rightarrow \quad \{ \text{monotonicity of } (\mathcal{M}_j i) \} \]
\[ R \circ \mathcal{M}_j i \subseteq T \circ \mathcal{M}_j i \]
\[ \Rightarrow \quad \{ \text{monotonicity of } \triangleright \} \]
\[ R \circ \mathcal{M}_j i \triangleright i \subseteq T \circ \mathcal{M}_j i \triangleright i \]
\[ \equiv \quad \{ \text{cancellation, } \cdot \text{TAG}_{i(i)} \subseteq M \} \]
\[ R \subseteq T \]

A trivial consequence is

\[ R = T \]
\[ \equiv \{ \cdot \text{TAG}_{i(i)} \subseteq M \} \]
\[ R \circ \mathcal{M}_j i = T \circ \mathcal{M}_j i \]
5.4 Properties of the binary constrained cylindric products

In the previous section we proved some properties of the pick and the $M$-constrained cylindric pack. In this section we use these properties and the properties of the cylindric-type operator to prove properties of the binary $M$-constrained cylindric product.

5.4.1 Cancellation

The binary $M$-constrained cylindric product satisfies the following cancellation laws:

\[
R_0 = \{ \bullet TAG(\circ) \subseteq M \} \\
R_0 \ominus_M R_1 \supseteq \emptyset \\
R_1 = \{ \bullet TAG(\circ) \subseteq M \} \\
R_0 \ominus_M R_1 \supseteq \emptyset
\]

proof

We only prove one of them, the proof of the other is algebraically identical.

\[
R_0 \ominus_M R_1 \supseteq \emptyset \\
= \{ \text{definition of } \ominus_M \text{ in terms of } \ominus_M \} \\
= (R_0 \ominus_M \ominus \cap R_1 \ominus_M \ominus) \supseteq \emptyset \\
= \{ \text{distribution of } \ominus \text{ over } \cap \} \\
= R_0 \ominus M \ominus \cap R_1 \ominus M \ominus \supseteq \emptyset \\
= \{ \text{cancellation, } \bullet TAG(\circ) \subseteq M \} \\
R_0 \cap \Pi \\
= \{ \Pi \text{ is unit of } \cap \} \\
R_0
\]

If there exist functions $f_0$ and $f_1$ with appropriate domains, such that for all $R_0$ and $R_1$

\[
f_0(R_0 \ominus_M R_1) = R_0 \\
f_1(R_0 \ominus_M R_1) = R_1
\]

then we call $\ominus_M$ information preserving.

The cartesian sum is for example information preserving: take $f_0 = (\circ \ominus)$ and $f_1 = (\circ \ominus)$. The cartesian product is however not information preserving. For every $R$ we have

\[
R \times \emptyset = \emptyset
\]
and we cannot retrieve any information about $R$ from $\emptyset$.

### 5.4.2 Inclusion

The binary $M$-constrained cylindric product is monotonic in both arguments. The following theorem shows that we may strengthen monotonicity to monomorphism if $\text{TAG}_1 \subseteq M$ holds:

$$R_0 \subseteq T_0 \land R_1 \subseteq T_1$$

$$\equiv \{ \bullet \text{TAG}_1 \subseteq M \}$$

$$R_0 \otimes_M R_1 \subseteq T_0 \otimes_M T_1$$

**proof**

$$R_0 \subseteq T_0 \land R_1 \subseteq T_1$$

$$\Rightarrow \{ \text{monotonicity of } (\otimes_M) \}$$

$$R_0 \otimes_M R_1 \subseteq T_0 \otimes_M T_1$$

$$\Rightarrow \{ \text{monotonicity of } \cap \}$$

$$R_0 \otimes_M R_1 \cap R_1 \otimes_M R_1 \subseteq T_0 \otimes_M R_1 \cap T_1 \otimes_M R_1$$

$$\equiv \{ \text{definition of } \otimes_M \text{ in terms of } \otimes_M \}$$

$$R_0 \otimes_M R_1 \subseteq T_0 \otimes_M T_1$$

$$\Rightarrow \{ \text{monotonicity of } (\triangleright) \}$$

$$R_0 \otimes_M R_1 \triangleright \emptyset \subseteq T_0 \otimes_M T_1 \triangleright \emptyset \land$$

$$R_0 \otimes_M R_1 \triangleright \emptyset \subseteq T_0 \otimes_M T_1 \triangleright \emptyset$$

$$\equiv \{ \text{cancellation, } \bullet \text{TAG}_1 \subseteq M \}$$

$$R_0 \subseteq T_0 \land R_1 \subseteq T_1$$

### 5.4.3 Equality

A trivial consequence of the theorem of the previous section is

$$R_0 = T_0 \land R_1 = T_1$$

$$\equiv \{ \bullet \text{TAG}_1 \subseteq M \}$$

$$R_0 \otimes_M R_1 = T_0 \otimes_M T_1$$

### 5.4.4 Intersection

The binary $M$-constrained cylindric product abides with the intersection if $M$ restricts the inputs to functions:
\[ R_0 \cap T_0 \otimes M R_1 \cap T_1 \]
\[ = \{ \bullet M \subseteq I \leftrightarrow (I \leftarrow I) \} \]
\[ R_0 \otimes_M R_1 \cap T_0 \otimes_M T_1 \]

**proof**

\[ R_0 \cap T_0 \otimes_M R_1 \cap T_1 \]
\[ = \{ \text{definition of } \otimes_M \text{ and } \otimes \} \]
\[ (R_0 \cap T_0 \dashv \circ \{ 0 \}) \cap (R_1 \cap T_1 \dashv \circ \{ 1 \}) \cap M \]
\[ = \{ \text{distribution of } \dashv \circ \{ i \} \text{ over } \cap, \bullet M \subseteq I \leftrightarrow (I \leftarrow I) \} \]
\[ R_0 \dashv \circ \{ 0 \} \cap T_0 \dashv \circ \{ 0 \} \cap R_1 \dashv \circ \{ 1 \} \cap T_1 \dashv \circ \{ 1 \} \cap M \]
\[ = \{ \text{idempotency, commutativity and associativity of } \cap \} \]
\[ R_0 \dashv \circ \{ 0 \} \cap R_1 \dashv \circ \{ 1 \} \cap M \cap T_0 \dashv \circ \{ 0 \} \cap T_1 \dashv \circ \{ 1 \} \cap M \]
\[ = \{ \text{definition of } \otimes \text{ and } \otimes_M \} \]
\[ R_0 \otimes_M R_1 \cap T_0 \otimes_M T_1 \]

### 5.4.5 Union

The binary \( M \)-constrained cylindric product distributes over the union if \( M \) restricts the output to functions:

\[ R_0 \cup T_0 \otimes_M R_1 \cup T_1 \]
\[ = \{ \bullet M \subseteq (I \leftarrow I) \Pi \} \]
\[ R_0 \otimes_M R_1 \cup R_0 \otimes_M T_1 \cup T_0 \otimes_M R_1 \cup T_0 \otimes_M T_1 \]

**proof**

\[ R_0 \cup T_0 \otimes_M R_1 \cup T_1 \]
\[ = \{ \text{definition of } \otimes_M \text{ and } \otimes \} \]
\[ (R_0 \cup T_0 \dashv \circ \{ 0 \}) \cap (R_1 \cup T_1 \dashv \circ \{ 1 \}) \cap M \]
\[ = \{ \text{distribution of } \dashv \circ \{ i \} \text{ over } \cup, \bullet M \subseteq (I \leftarrow I) \Pi \} \]
\[ (R_0 \dashv \circ \{ 0 \} \cup T_0 \dashv \circ \{ 0 \}) \cap (R_1 \dashv \circ \{ 1 \} \cup T_1 \dashv \circ \{ 1 \}) \cap M \]
\[ = \{ \text{distribution of } \cap \text{ over } \cup \} \]
\[ ((R_0 \dashv \circ \{ 0 \} \cap R_1 \dashv \circ \{ 0 \}) \cup (R_0 \dashv \circ \{ 0 \} \cap T_1 \dashv \circ \{ 0 \}) \cup (T_0 \dashv \circ \{ 0 \} \cap R_1 \dashv \circ \{ 0 \}) \cup (T_0 \dashv \circ \{ 0 \} \cap T_1 \dashv \circ \{ 0 \}) \cap \}
\[ M \]
\[ = \{ \text{distribution of } \cap \text{ over } \cup \} \]
\[ (R_0 \dashv \circ \{ 0 \} \cap R_1 \dashv \circ \{ 0 \} \cap M) \cup (R_0 \dashv \circ \{ 0 \} \cap T_1 \dashv \circ \{ 0 \} \cap M) \cup (T_0 \dashv \circ \{ 0 \} \cap R_1 \dashv \circ \{ 0 \} \cap M) \cup (T_0 \dashv \circ \{ 0 \} \cap T_1 \dashv \circ \{ 0 \} \cap M) \]
\[ = \{ \text{definition of } \otimes \text{ and } \otimes_M \} \]
\[ R_0 \otimes_M R_1 \cup R_0 \otimes_M T_1 \cup T_0 \otimes_M R_1 \cup T_0 \otimes_M T_1 \]
5.4.6 Sequential composition

We now try to discover what restrictions on $M$ are needed such that the binary $M$-constrained cylindric product abides with the sequential composition:

\[
R_0 \cdot T_0 \otimes_M R_1 \cdot T_1 = \{ \bullet \text{ some condition on } M \} \\
R_0 \otimes_M R_1 \circ T_0 \otimes_M T_1
\]

A main insight was to use the axiom of choice (see section 3.1.26). We repeat it here for ease of reference. For $p \in \mathbb{B} \leftarrow 1$ and $q \in \mathbb{B} \leftarrow 1 \times 1$ we have

\[
\forall \langle y \mid p.y \mid \exists \langle z \mid q.\langle y, z \rangle \rangle \\
\equiv \exists \langle f \mid f \in I \leftarrow \{ x \mid p.x \mid x \} \mid \forall \langle y \mid p.y \mid q.\langle y, f.y \rangle \rangle
\]

The axiom of choice is the core of the following theorem that we also refer to as “the axiom of choice”:

\[
S \cdot \{ i \} \subseteq R \cdot U \\
\equiv \exists \langle f \mid f \in I \leftarrow S \cdot i \mid f \subseteq R \land S \cdot \{ i \} \subseteq f \cdot U \rangle
\]

Proof
\[ S \cdot \{ i \} \subseteq R \cdot U \]

\[ \forall y, j \; | \; y \; (S \cdot \{ i \}) \; j \; | \; y \; (R \cdot U) \; j \]

\[ \{ y \; (S \cdot \{ i \}) \; j \; | \; j = i \} \]

\[ \forall y \; | \; y \; (S \cdot \{ i \}) \; \exists z \; | \; y \; (R) \; z \; \land \; z \; (U) \; i \}

\[ \{ \text{definition of } \cdot \} \]

\[ \exists f \; | \; f \; \in I \leftrightarrow S \cdot i \; | \; \forall(y \; | \; y \; (S) \; i \; \land \; f \; y \; \land \; f \; y \; (U) \; i) \]

\[ \{ \text{shunting} \} \]

\[ \exists f \; | \; f \; \in I \leftrightarrow S \cdot i \; | \; \forall(y \; | \; y \; (S) \; i \; \land \; f \; y \; f \; (U) \; i) \]

\[ \{ \text{definition of } \cdot \} \]

\[ \exists f \; | \; f \; \in I \leftrightarrow S \cdot i \; | \; \forall(y \; | \; y \; (S) \; i \; \land \; f \; y \; (U) \; i) \]

\[ \{ \text{shunting} \} \]

\[ \exists f \; | \; f \; \in I \leftrightarrow S \cdot i \; | \; f \; \subseteq R \cdot f \; \land \; S \cdot \{ i \} \subseteq f \cdot U \]

We now write out some definitions of operators in the composition-abide theorem to see how far we get:

\[ S \; (R_0 \cdot T_0 \otimes M \; R_1 \cdot T_1) \; Q \]

\[ \{ \text{definition of } \otimes, \cap, \otimes, \text{and } \cdots \} \]

\[ S \; (M) \; Q \; \land \; S \cdot \{ i \} \subseteq R_0 \cdot T_0 \cdot Q \; \land \; S \cdot \{ i \} \subseteq R_1 \cdot T_1 \cdot Q \]

\[ \{ \bullet \text{ some condition on } M, \text{ see below} \} \]

\[ \exists(K \; | \; S \; (M) \; K \; \land \; K \; (M) \; Q) \]

\[ S \cdot \{ i \} \subseteq R_0 \cdot K \; \land \; S \cdot \{ i \} \subseteq R_1 \cdot K \; \land \]

\[ K \cdot \{ i \} \subseteq T_0 \cdot Q \; \land \; K \cdot \{ i \} \subseteq T_1 \cdot Q \]

\[ \{ \text{definition of } \cdots \} \]

\[ \exists(K \; | \; S \; (R_0 \otimes M \; R_1) \; K \; \land \; K \; (T_0 \otimes M \; T_1) \; Q) \]

\[ \{ \text{definition of } \cdot \} \]

\[ S \; (R_0 \otimes M \; R_1) \; T \otimes M \; T_1 \; Q \]

We have to find conditions on M that enable us to do the second step in this
calculation. We start with the \((\subseteq)\). We assume
\[
\begin{align*}
S & (M) K \land K (M) Q \\
S & \circ \{0\} \subseteq R_0 K \land S & \circ \{0\} \subseteq R_1 K \land \\
K & \circ \{0\} \subseteq T_0 Q \land K & \circ \{0\} \subseteq T_1 Q
\end{align*}
\]
and calculate:
\[
\begin{align*}
S & \circ \{0\} \subseteq R_0 \circ T_0 Q \\
\therefore \quad & \{K \circ \{0\} \subseteq T_0 Q\} \\
S & \circ \{0\} \subseteq R_0 \circ K S & \circ \{0\} \subseteq R_1 \circ K S & \circ \{0\} \subseteq R_1 \circ T_0 Q & \subseteq & \{A \land A = A\text{ for sets } A, \text{ reflexivity of } \subseteq\} \\
true
\end{align*}
\]
Same for \(S \circ \{1\} \subseteq R_1 \circ T_1 Q\). For \(S (M) Q\) we calculate:
\[
\begin{align*}
S & (M) Q \\
\therefore \quad & \{\bullet M \land M \subseteq M\} \\
S & (M,M) Q \\
\equiv \quad & \{\text{definition of } \circ\} \\
\exists & \langle X | S (M) X \land X (M) Q\rangle \\
\equiv \quad & \{\text{take } X = K\} \\
true
\end{align*}
\]
So for the \((\subseteq)\) we found \(M \cdot M \subseteq M\) as condition on \(M\). Now for the \((\Rightarrow)\):
\[
\begin{align*}
S & (M) Q \land S \circ \{0\} \subseteq R_0 \circ T_0 Q \land S & \circ \{0\} \subseteq R_1 \circ T_1 Q \\
\equiv \quad & \{\text{axiom of choice}\} \\
S & (M) Q \land \exists \langle f_0, f_1, f_0, f_1 \rangle \\
f_0 & \in \langle S \circ \{0\} \land f_0 \circ \subseteq R_0 \land S & \circ \{0\} \subseteq f_0 \circ T_0 \circ Q \land \\
f_1 & \in \langle S \circ \{0\} \land f_1 \circ \subseteq R_1 \land S & \circ \{0\} \subseteq f_1 \circ T_1 \circ Q\rangle \\
\Rightarrow \quad & \{\text{take } K = f_0 \circ S \circ \{0\} \cup f_1 \circ S \circ \{0\} \cup k \text{ for some } k \in \langle S \circ \{0\}\rangle, \\
\bullet \text{ this step and the assumptions needed, are treated below}\} \\
\exists & \langle K | S (M) K \land K (M) Q \rangle \\
S & \circ \{0\} \subseteq R_0 \circ K \land S & \circ \{0\} \subseteq R_1 \circ K \land \\
K & \circ \{0\} \subseteq T_0 \circ Q \land K & \circ \{0\} \subseteq T_1 \circ Q
\end{align*}
\]
For the latter of these two steps, we first remark that the fact that for every \(S\) there exists a \(k \in \langle S \circ \{0\}\rangle\), is obviously true. We now prove this latter step. We assume
\[ S(M) \land f_0 \in I \leftarrow S \circ \{0\} \land f_0 \triangleright R_0 \land S \circ \{0\} \subseteq f_0 \triangleright T_0 \wedge \]
\[ f_1 \in I \leftarrow S \circ \{1\} \land f_1 \triangleright R_1 \land S \circ \{1\} \subseteq f_1 \triangleright T_1 \wedge \]
\[ k \in I \leftarrow (S \setminus 2) \wedge \]
\[ K = f_0 \circ S \circ \{0\} \cup f_1 \circ S \circ \{1\} \cup k \]

and calculate
\[ S \circ \{0\} \subseteq R_0 \circ K \]
\[ \equiv \{X \cdot B \subseteq Y \equiv X \cdot B \subseteq Y \cdot B\} \text{ for sets } B \]
\[ S \circ \{0\} \subseteq R_0 \circ K \circ \{0\} \]
\[ \equiv \{K \circ \{0\} = f_0 \circ S \circ \{0\}\} \]
\[ S \circ \{0\} \subseteq R_0 \circ f_0 \circ S \circ \{0\} \]
\[ \equiv \{X \cdot B \subseteq Y \equiv X \cdot B \subseteq Y \cdot B\} \text{ for sets } B \]
\[ S \circ \{0\} \subseteq R_0 \circ f_0 \circ S \]
\[ \equiv \{f_0 \subseteq R_0\} \]
\[ S \circ \{0\} \subseteq f_0 \triangleright f_0 \circ S \]
\[ \equiv \{\text{shunting, } f_0 \in I \leftarrow S \circ \{0\}\} \]
\[ f_0 \circ S \circ \{0\} \subseteq f_0 \circ S \]
\[ \equiv \{X \cdot B \subseteq X\} \text{ for sets } B \]

true

Same for \( S \circ \{1\} \subseteq R_1 \circ K \). Furthermore
\[ K \circ \{0\} \subseteq T_0 \circ Q \]
\[ \equiv \{K \circ \{0\} = f_0 \circ S \circ \{0\}\} \]
\[ f_0 \circ S \circ \{0\} \subseteq T_0 \circ Q \]
\[ \equiv \{\text{shunting, } f_0 \in I \leftarrow S \circ \{0\}\} \]
\[ S \circ \{0\} \subseteq f_0 \triangleright T_0 \circ Q \]
\[ \equiv \{\text{assumption}\} \]

true

Same for \( K \circ \{0\} \subseteq T_1 \circ Q \).

The only thing that is left to prove is \( S(M) \land K(M) \land K(M) \). We need to find a condition on \( M \) that is independent of \( S, Q, f_0, f_1 \) and \( k \). The condition that we choose, is that for all \( S, Q, f_0, f_1, k \) and \( K \) (we use \( K \) actually just as a shorthand)

\[ S(M) \land f_0 \in I \leftarrow S \circ \{0\} \land f_0 \triangleright R_0 \land S \circ \{0\} \subseteq f_0 \triangleright T_0 \land \]
\[ f_1 \in I \leftarrow S \circ \{1\} \land f_1 \triangleright R_1 \land S \circ \{1\} \subseteq f_1 \triangleright T_1 \land \]
\[ k \in I \leftarrow (S \setminus 2) \land \]
\[ K = f_0 \circ S \circ \{0\} \cup f_1 \circ S \circ \{1\} \cup k \]
\[ \cdot \]

This condition on \( M \) together with the condition we needed for the other direction:

106
\( M \cdot M \subseteq M \)

guarantees abidence of \( \cdot \) with \( \otimes M \). We now show that for any set of constraints \( \mathcal{M} \) where

\[
\mathcal{M} \subseteq \{\text{UNI}, \text{FUN}, \text{NMP}, \text{DEQ}\} \cup \\
\{\text{TOT}_B \mid B \in \wp I \mid B\} \cup \\
\{\text{RES}_B \mid B \in \wp I \mid B\}
\]

both conditions are satisfied by \( \bigcap \mathcal{M} \).

We start with the simple condition \( M \cdot M \subseteq M \). We leave the (easy) proof that it holds for all the above constraints individually to the reader. For an arbitrary set of constraints \( \mathcal{M} \) we calculate

\[
S(\bigcap \mathcal{M} \cdot \bigcap \mathcal{M}) Q
\]

\[
\equiv \text{definition of } \cdot \\
\exists \langle K \mid S(\bigcap \mathcal{M}) K \land K(\bigcap \mathcal{M}) Q\rangle
\]

\[
\equiv \text{point-wise definition of } \bigcap \\
\exists \langle K \mid \forall \langle M \mid M \in \mathcal{M} \mid S(M) K \land K(M) Q\rangle \rangle
\]

\[
\Rightarrow \forall \langle M \mid M \in \mathcal{M} \mid \exists \langle K \mid S(M) K \land K(M) Q\rangle \rangle
\]

\[
\equiv \text{definition of } \cdot \\
\forall \langle M \mid M \in \mathcal{M} \mid S(M \cdot M) Q\rangle
\]

\[
\Rightarrow \forall \langle M \mid M \in \mathcal{M} \Rightarrow M \cdot M \subseteq M \rangle
\]

\[
\forall \langle M \mid M \in \mathcal{M} \mid S(M) Q\rangle
\]

\[
\equiv \text{point-wise definition of } \bigcap
\]

\[
S(\bigcap \mathcal{M}) Q
\]

The other condition on \( M \):

\[
S(M) Q \Rightarrow S(M) K \land K(M) Q
\]

\[
\equiv \{\bullet K = f_0 S \cdot \{\otimes\} \cup f_1 S \cdot \{\otimes\} \cup k\}
\]

\[
f_0 \in I \iff S \cdot \{\otimes\} \land f_1 \in I \iff S \cdot \{\otimes\} \land k \in I \iff (S \cdot \{2\})
\]

we also first check for each constraint individually:

\textit{UNI}: \( S \) has a unique input, so \( (K \cdot = S \cdot) K \) has a unique input.

\textit{FUN}: \( S \) is functional, so \( (f_0, f_1 \text{ and } k \text{ are functional}) K \) is functional.

\textit{NMP}: \( S \) is non-empty, so \( (K \cdot = S \cdot) K \) is non-empty.
DEQ: $S^\cdot = Q^\cdot$ and $K^\cdot = S^\cdot$, so $K^\cdot = Q^\cdot$.

TOT$_B$: $S$ is total on $B$, so $(K^\cdot = S^\cdot)$ $K$ is total on $B$.

RES$_B$: $S$ is restricted to $B$, so $(K^\cdot = S^\cdot)$ $K$ is restricted to $B$.

For an arbitrary set of constraints $\mathcal{M}$ we calculate:

$$S ((\bigcap M) K \land (\bigcap M) Q)$$
\[\equiv\]
{point-wise definition of $\bigcap$}
\[\forall \langle M \mid M \in \mathcal{M} \mid S(M) K \land (\bigcap M) Q \rangle \]
\[\equiv\]
\[\forall \langle M \mid M \in \mathcal{M} \mid S(M) \rangle \]
\[\equiv\]
{point-wise definition of $\bigcap$}
\[S ((\bigcap M) Q)\]

Putting everything together, we have proved that for each $\mathcal{M}$ with

$$\mathcal{M} \subseteq \{\text{UNI, FUN, NMP, DEQ}\} \cup \{\text{TOT}_B \mid B \in \wp I \mid B\} \cup \{\text{RES}_B \mid B \in \wp I \mid B\}$$

the following abidence theorem holds:

$$R_0^\cdot T_0^\cdot \otimes_M R_1^\cdot T_1$$
\[=\]
\[\{\cdot M = (\bigcap M)\}
$$R_0^\cdot \otimes_M R_1^\cdot \circ T_0^\cdot \otimes_M T_1$$

5.4.7 Identity relation

Similar to section 4.4.6, we do not have in general that $A_0^\cdot \otimes_M A_1^\cdot$ is a set for sets $A_0$ and $A_1$. We can prove however that $\otimes_M$ preserves $R_{no}$ for the $M$s we are considering and that $\otimes_M$ is diagonal preserving for arbitrary $M$.

The binary $M$-constrained cylindric product preserves $R_{no}$ if $M \in R_{no}$:

$$R_0, R_1 \in R_{no}$$
\[\Rightarrow\]
\[\{\cdot M \in R_{no}\}
\]
$$R_0^\cdot \otimes_M R_1^\cdot \in R_{no}$$

proof

108
\[(R_0 \otimes_M R_1)^c : \]
\[
\{ \text{definition of } \otimes_M \text{ and } \otimes \} \\
\{(R_0 \circ \{ \circ \}) \cap (R_1 \circ \{ \circ \}) \cap M \}^c \\
\subseteq \{(R \cap S) \subseteq R \cap S\} \\
(R_0 \circ \{ \circ \})^c \cap (R_1 \circ \{ \circ \})^c \cap M^c \\
\subseteq \{(\circ \{ i \}) \text{ preserves } R_{no}, \bullet R_0, R_1, M \in R_{no}\} \\
(R_0 \circ \{ \circ \}) \cap (R_1 \circ \{ \circ \}) \cap M \\
= \{ \text{definition of } \otimes \text{ and } \otimes_M \} \\
R_0 \otimes_M R_1^c
\]

We leave it to the reader to verify that all combinations of the constraints we consider, are an element of \( R_{no} \) (hint: prove it for each of the constraints individually and then for an arbitrary intersection of constraints that are an element of \( R_{no} \)).

For arbitrary \( M \) we have that \( \otimes_M \) is diagonal preserving:

\[
(R_0 \cap I \otimes_M R_1 \cap I) \cap I \\
= \{ \bullet R_0, R_1 \in R_{no} \} \\
R_0 \otimes_M R_1 \cap I
\]
proof

\[
(R_0 \cap I \otimes_M R_1 \cap I) \cap I \\
= \{ \text{definition of } \otimes_M \text{ and } \otimes \} \\
(R_0 \cap I \circ \{ \circ \}) \cap (R_1 \cap I \circ \{ \circ \}) \cap M \cap I \\
= \{ \text{idempotency, commutativity and associativity of } \cap \} \\
(R_0 \cap I \circ \{ \circ \}) \cap I \cap (R_1 \cap I \circ \{ \circ \}) \cap I \cap M \\
= \{(\circ \{ i \}) \text{ is diagonal preserving, } \bullet R_0, R_1 \in R_{no}\} \\
(R_0 \circ \{ \circ \}) \cap I \cap (R_1 \circ \{ \circ \}) \cap I \cap M \\
= \{ \text{idempotency, commutativity and associativity of } \cap \} \\
(R_0 \circ \{ \circ \}) \cap (R_1 \circ \{ \circ \}) \cap M \cap I \\
= \{ \text{definition of } \otimes \text{ and } \otimes_M \} \\
R_0 \otimes_M R_1 \cap I
\]

5.4.8 Converse

The converse distributes over the binary \( M \)-constrained cylindric product if \( M \) is a symmetric subrelation of \( FUN \cap DEQ \):

\[
R_0 \circ \otimes_M R_1 \sim \\
= \{ \bullet M \subseteq FUN \cap DEQ \land M \text{ is symmetric} \} \\
(R_0 \circ \otimes_M R_1) \sim
\]

109
proof

\[
R_0^\sim \otimes_M R_1^\sim
\]
\[
= \{\text{definition of } \otimes_M \text{ and } \otimes\}
\]
\[
R_0^\sim \cdot \sim \circ \{\circ \} \cap R_1^\sim \cdot \sim \circ \{\circ \} \cap M
\]
\[
= \{\cdot \circ \text{ commutes with } \sim, \bullet M \subseteq \text{FUN} \cap \text{DEQ}\}
\]
\[
(R_0^\sim \cdot \sim \circ \{\circ \}) ^\sim \cap (R_1^\sim \cdot \sim \circ \{\circ \}) ^\sim \cap M
\]
\[
= \{\bullet M \text{ is symmetric}\}
\]
\[
(R_0^\sim \cdot \sim \circ \{\circ \}) ^\sim \cap (R_1^\sim \cdot \sim \circ \{\circ \}) ^\sim \cap M^\sim
\]
\[
= \{\text{distribution of } \sim \text{ over } \cap\}
\]
\[
(R_0^\sim \cdot \sim \circ \{\circ \} \cap R_1^\sim \cdot \sim \circ \{\circ \} \cap \cap M^\sim
\]
\[
= \{\text{definition of } \otimes \text{ and } \otimes_M\}
\]
\[
(R_0 \otimes_M R_1)^\sim
\]

5.5 Arbitrary products

Next to the binary version of the \(M\)-constrained cylindric product, we also define an arbitrary version. The arbitrary \(M\)-constrained cylindric product \(\otimes_M W \in ((I \sim) \sim (I \sim)) \sim (I \sim) \sim (I \sim)\) is defined by

\[
\otimes_M W \ = \ \bigcap \{R \cdot \sim \circ \{i\} \mid R(W.i) \cap R.i \cap M\}
\]

The way we model tuples by relations gives us

\[
\otimes_M (R_0, R_1) = \otimes_M (R_0, R_1)
\]

A property of the arbitrary \(M\)-constrained cylindric product that is important in the context of object-oriented programming, is that it is anti-monotonic (‘becomes smaller when a relation is added’):

\[
W \subseteq V \Rightarrow \otimes_M W \supseteq \otimes_M V
\]

An arbitrary product \(\otimes \in ((I \sim) \sim (I \sim)) \sim (I \sim) \sim (I \sim)\) is information preserving if for each tag there exists a (total) function that from a package constructed with \(\otimes\), can fully retrieve the relation with that tag:

\[
\forall i \parallel \exists f \parallel f \in (I \sim) \sim (I \sim) \sim (I \sim) \sim (I \sim) \sim (I \sim) \sim (I \sim)
\]
\[
\forall (W.i) \parallel W \in (I \sim) \sim (I \sim) \sim (I \sim) \sim (I \sim) \sim (I \sim) \sim (I \sim) \sim (I \sim)
\]
\[
f(W.i) = W.i)
\]

We leave a structured investigation of the properties of the arbitrary \(M\)-constrained cylindric product for future research. The intricate cases will be the ones where \(W\) is infinite.
5.6 Conjoint sum

From our investigation of the constrained cylindric products, some new products with useful properties emerge. The **conjoint sum** \( \cap \) is such a product. It is defined by

\[
\cap = \odot \text{UNI} \cap \text{FUN} \cap \text{NMP} \cap \text{DEQ}
\]

The conjoint sum is similar to the cartesian sum, also called the **disjoint sum**. The main difference between the conjoint sum and the cartesian sum is that adding a relation to a conjoint sum results in a smaller relation whereas the arbitrary cartesian sum, defined as straightforward generalisation of the allegoric definition of the binary cartesian sum:

\[
+ f = \bigcup \{ f \circ i : i \in f \mid i \}
\]

constructs packages that become larger if a relation is added.

The following two theorems show the close relationship between the conjoint sum and the cartesian sum:

\[
\begin{align*}
R_0 + R_1 &= R_0 \cap R_1 \cap \text{RES}_1 \\
R_0 \cap R_1 &= R_0 + R_1 \cup +\langle \Pi \mid i \notin 2 \mid i \rangle
\end{align*}
\]

**proof**

The first theorem we already proved in section 5.2, except for the expansion of some definitions. For the second theorem we calculate

\[
\begin{align*}
R_0 \cap R_1 &= \{ \text{definition of } \cap \} \\
R_0 \odot R_1 \cap \text{UNI} \cap \text{FUN} \cap \text{NMP} \cap \text{DEQ} &= \{ \text{UNI} \cap \text{FUN} \cap \text{NMP} \cap \text{DEQ} = \text{TAG}_1 = \text{TAG}_1 \cup \text{TAG}_1 \cap \text{TAG}_1 \cap \text{TAG}_1, \text{left to the reader} \} \\
R_0 \odot R_1 \cap (\text{TAG}_1 \cup \text{TAG}_1) &= \{ \text{distribution of } \cap \text{ over } \cup \} \\
(R_0 \odot R_1 \cap \text{TAG}_1) \cup (R_0 \odot R_1 \cap \text{TAG}_1) &= \{ \text{TAG}_1 \subseteq R_0 \odot R_1, \text{left to the reader} \} \\
(R_0 \odot R_1 \cap \text{TAG}_1) \cup \text{TAG}_1 &= \{ \text{see section 5.2, } \text{TAG}_1 = +\langle \Pi \mid i \notin 2 \mid i \rangle, \text{left to the reader} \} \\
R_0 + R_1 &= R_0 + R_1 \cup +\langle \Pi \mid i \notin 1 \mid i \rangle
\end{align*}
\]

In chapter 7 we show how the conjoint sum can be used to model the ‘classes’ and ‘packages’ of object-oriented languages.

If we have a look at the properties of the constrained cylindric products that we introduced, the product \( \odot \text{FUN} \cap \text{DEQ} \) appears worthy of note as it has all the useful...
properties that we treated, with the least I/O constraints. From the theorems in 5.2 it follows that if we restrict its I/O to pairs, we obtain the cartesian product and if we restrict its I/O to values that are tagged with ⊙ or ⊙, we obtain the cartesian sum. The product ⊗_{\text{FUN} \cap \text{DEQ}} thus seems to ‘combine the best of both worlds’, although it does not preserve the identity (I ⊗_{\text{FUN} \cap \text{DEQ}} I does not equal I). In the next chapter we show a possible application of ⊗_{\text{FUN} \cap \text{DEQ}} in the construction of non-deterministic expressions.

5.7 Properties of the constrained cylindric packs

In section 5.3 we already proved some theorems about the $M$-constrained cylindric pack:

$$
R = R \otimes M \triangleright i \quad \Leftarrow \quad \text{TAG}(i) \subseteq M \\
\Pi = R \otimes M \triangleright j \quad \Leftarrow \quad \text{TAG}(j) \subseteq M \quad \text{and} \quad j \neq i \\
R \cap T \equiv R \otimes M \triangleright i \quad \Leftarrow \quad \text{TAG}(i) \subseteq M \\
R = T \equiv R \otimes M \triangleright i \quad \Leftarrow \quad \text{TAG}(i) \subseteq M
$$

Some other theorems are almost a trivial consequence of the theorems about the cylindric-type operator (section 4.4) and are left to the reader:

$$
R \cap T \otimes M i = R \otimes M i \cap T \otimes M i \quad \Leftarrow \quad M \subseteq \Pi \cdot (I \leftarrow I) \\
R \cup T \otimes M i = R \otimes M i \cup T \otimes M i \quad \Leftarrow \quad M \subseteq (I \leftarrow I) \cdot \Pi \\
R \in R_{no} \Rightarrow R \otimes M i \in R_{no} \quad \Leftarrow \quad M \in R_{no} \\
R \cap I \otimes M i \cap I = R \otimes M i \cap I \quad \Leftarrow \quad M \subseteq \text{FUN} \cap \text{DEQ} \quad \text{and} \quad M \text{ is symmetric}
$$

Distribution of $\otimes_M$ over composition is more tricky. The following theorem, the proof of which is left to the reader, enables us to reuse the corresponding theorem about the binary $M$-constrained cylindric product:

$$
R \otimes M \circ \equiv \{ \bullet M \subseteq \text{DEQ} \} \\
R \otimes M \Pi
$$

We use this theorem to prove that for $M$ with

$$
M \subseteq \{ \text{UNI, FUN, NMP, DEQ} \} \cup \\
\{ TOT_B \mid B \in \varphi \} \cup \\
\{ RES_B \mid B \in \varphi \}
$$

we have

$$
R \cdot T \otimes M \circ = R \otimes M \circ \cdot T \otimes M \circ \quad \Leftarrow \quad M = \bigcap M \quad \text{and} \quad M \subseteq \text{DEQ}
$$
proof

\[
\begin{align*}
R \cdot T \otimes_M \odot M &= \{ \text{above theorem, } \bullet M \subseteq DEQ \} \\
R \cdot T \otimes_M \Pi &= \{ \Pi = \Pi \Pi \} \\
R \cdot T \otimes_M \Pi \Pi &= \{ \text{abidence of } \otimes_M \text{ with } \cdot, \bullet M = \cap M \} \\
R \otimes_M \Pi \otimes_M \Pi &= \{ \text{above theorem, } \bullet M \subseteq DEQ \} \\
R \otimes_M \odot T \otimes_M \odot M &= \{ \text{above theorem, } \bullet M \subseteq DEQ \} \\
R \otimes_M \odot T \otimes_M \odot M &= \{ \text{above theorem, } \bullet M \subseteq DEQ \}
\end{align*}
\]

5.8 Loose product

If the cylindric product of two relations \( R \otimes R_1 \) connects an input with some output, it also connects this input with all subrelations of that output. Furthermore, for every input there exists a single largest output. We prove these facts by showing that there exists a construct \( R \otimes R_1 \) that has this largest output as output, rewriting the definition of \( R \otimes R_1 \) as follows:

\[
S( R \otimes R_1 ) Q \equiv \{ \text{definition of } \otimes \text{ and } \rightarrow \}
\]

\[
S \cdot \{ (0) \} \subseteq R_0 \cdot Q \land S \cdot \{ (1) \} \subseteq R_1 \cdot Q
\]

\[
\equiv \{ \text{point-free definition of } / \}
\]

\[
S \subseteq (R_0 \cdot Q)/(\{0\}) \land S \subseteq (R_1 \cdot Q)/(\{1\})
\]

\[
\equiv \{ \text{point-free definition of } \cap \}
\]

\[
S \subseteq (R_0 \cdot Q)/(\{0\}) \cap (R_1 \cdot Q)/(\{1\})
\]

\[
\equiv \{ \text{function comprehension} \}
\]

\[
S \subseteq (R_0 \cdot X)/(\{0\}) \cap (R_1 \cdot X)/(\{1\}) || X).Q
\]

\[
\equiv \{ \text{shunting} \}
\]

\[
S \subseteq (R_0 \cdot X)/(\{0\}) \cap (R_1 \cdot X)/(\{1\}) || X)
\]

So, defining the **binary loose product** \( R \otimes R_1 \in ((I \rightarrow I) \leftarrow (I \rightarrow I)) \leftarrow (I \rightarrow I) \times (I \rightarrow I) \) by

\[
( R \otimes R_1 ) Q = (R_0 \cdot Q)/(\{0\}) \cap (R_1 \cdot Q)/(\{1\})
\]

the cylindric product can be defined in terms of this binary loose product as follows:

\[
R \otimes R_1 = \subseteq \ast R \otimes R_1
\]

An important theorem about the binary loose product is that it abides with se-
sequential composition:

\[ R_0 \circ R_1 \ast T_0 \circ T_1 = R_0 \ast T_0 \circ R_1 \circ T_1 \]

**proof**

\[
(R_0 \circ R_1 \ast T_0 \circ T_1).Q
= (f \circ g).x = f(g.x)
\]

\[ (R_0 \circ R_1).((T_0 \circ T_1).Q) \]

\[ = \{\text{definition of } \circ\} \]

\[ (R_0 \circ R_1).((T_0 \circ Q)/\{\emptyset\} \cap (T_1 \circ Q)/\{\emptyset\}) \]

\[ = \{\text{definition of } \circ\} \]

\[ R_0^0((T_0 \circ Q)/\{\emptyset\} \cap (T_1 \circ Q)/\{\emptyset\}) / \{\emptyset\} \cap \]

\[ R_1^1((T_0 \circ Q)/\{\emptyset\} \cap (T_1 \circ Q)/\{\emptyset\}) / \{\emptyset\} \]

\[ = \{X/B = (X \cdot B)/B \text{ for sets } B, \text{ distribution of } \circ \text{ over } \cap \text{ for sets}\} \]

\[ R_0^0(((T_0 \circ Q)/\{\emptyset\}) \cap \{\emptyset\} \cap ((T_1 \circ Q)/\{\emptyset\}) \cap \{\emptyset\}) \cap \]

\[ R_1^1(((T_0 \circ Q)/\{\emptyset\}) \cap \{\emptyset\} \cap ((T_1 \circ Q)/\{\emptyset\}) \cap \{\emptyset\}) \cap \{\emptyset\} \]

\[ = \{\text{distribution of } \circ \text{ over } \cap \text{ for sets}\} \]

\[ R_0^0(T_0 \circ Q \cap \Pi \cdot \{\emptyset\}) / \{\emptyset\} \cap \]

\[ R_1^1(\Pi \ast T_1 \circ Q \cdot \{\emptyset\}) / \{\emptyset\} \]

\[ = \{\Pi \text{ is unit of } \cap\} \]

\[ R_0^0 T_0 \circ Q \cdot \emptyset / \{\emptyset\} \cap \]

\[ R_1^1 T_1 \circ Q \cdot \emptyset / \{\emptyset\} \]

\[ = \{X \cdot B = X / B \text{ for sets } B\} \]

\[ R_0^0 T_0 \circ Q / \{\emptyset\} \cap R_1^1 T_1 \circ Q / \{\emptyset\} \]

\[ = \{\text{definition of } \circ\} \]

\[ (R_0 \circ R_1 \ast T_0 \circ T_1).Q \]

This sequential-composition abstinence theorem for \( \circ \) can on the other hand also be used to prove it for \( \circ \) (hint: prove that \((R_0 \circ R_1)^{-} \subseteq \) equals \(R_0 \circ R_1\)).

We also introduce a variation of the loose product that restricts the inputs and outputs to elements of type \( 1 \rightarrow 2 \). The **binary restricted loose product** \( \ast \) \( \in (1 \rightarrow 2) \leftarrow (1 \rightarrow 2) \) \( \leftarrow (1 \rightarrow 1) \times (1 \rightarrow 1) \) is defined by

\[ (R_0 \circ R_1).Q = (R_0 \circ R_1).Q \ast 2 \]
or equivalently:

\[ (R_0 \circ R_1).Q = R_0^0 Q \cdot \{\emptyset\} \cup R_1^1 Q \cdot \{\emptyset\} \]

The following calculation proves the equivalence of both definitions:

114
(R₀⊙R₁).Q = 2

= \{\text{definition of } \circ\}

(\{(R₀ Q)/\{\circ\} \cap (R₁ Q)/\{\circ\}\} \circ 2)

= \{\text{distribution of } \circ \text{ over } \cap \text{ for sets and distribution of } \circ \text{ over } \cup\}

((R₀ Q_{\{\circ\}} \cup (\Pi_{\{\circ\}}) \cap (R₁ Q_{\{\circ\}} \cup (\Pi_{\{\circ\}}))) \circ 2)

= \{\text{for sets } B \text{ and } D \text{ we have } B \cup D = B \text{ if } B \subseteq D \text{ and } (\Pi \setminus B) \cup D = \Pi \setminus (D \setminus B)\}

(R₀ Q_{\{\circ\}} \cup (R₁ Q_{\{\circ\}} \cap (\Pi_{\{\circ\}}) \cap (R₁ Q_{\{\circ\}} \cup (\Pi_{\{\circ\}}))

= \{\text{distribution of } \circ \text{ over } \cap \text{ for sets, } \Pi \text{ is unit of } \cap\}

(R₀ Q_{\{\circ\}} \cup R₁ Q_{\{\circ\}} \cup (\Pi_{\{\circ\}})

To prove abidence of * with ⊙, we could use the first definition of *, together with the abidence theorem of ⊙. The second definition of * makes abidence of * with ⊙ however easy to prove directly:

\((R₀ \ast R₁ \ast T₀ \ast T₁).Q\)

= \{(f \ast g).x = f.(g.x) \text{ for total functions } f \text{ and } g\}

\((R₀ \ast R₁).((T₀ \ast T₁).Q)\)

= \{\text{second definition of } \ast\}

\((R₀ \ast R₁).((T₀ \ast T₁).Q_{\{\circ\}})\)

= \{\text{second definition of } \ast\}

\(R₀(T₀ \ast T₁ \ast Q_{\{\circ\}} \cup (T₀ \ast T₁ \ast Q_{\{\circ\}}) \cup R₁ \ast (T₀ \ast T₁ \ast Q_{\{\circ\}} \cup (T₀ \ast T₁ \ast Q_{\{\circ\}})) \circ 2)\)

= \{\text{distribution of } \circ \text{ over } \cup\}

\(R₀ \ast T₀ \ast Q_{\{\circ\}} \cup (R₀ \ast T₁ \ast Q_{\{\circ\}} \cup R₁ \ast (T₀ \ast T₁ \ast Q_{\{\circ\}})) \circ 2\)

= \{\text{for disjoint sets } B \text{ and } D, \emptyset \text{ is unit of } \cup\}

\(R₀ \ast T₀ \ast Q_{\{\circ\}} \cup R₁ \ast T₁ \ast Q_{\{\circ\}} \cup \circ\)

= \{\text{second definition of } \ast\}

\((R₀ \ast T₀ \ast R₁ \ast T₁).Q\)

In the next chapter we use the loose product in the construction of a general model for expressions that incorporates non-strict and even non-deterministic evaluation.
5.9 Conclusions

In this chapter we presented several constructs that enable us to combine relations into a package that satisfies several useful algebraic properties. One of the main reasons for this investigation was the fact that both anti-monotonicity and information preservingness are important properties for our purposes. We introduced several products that, different from the cartesian sum and product, do combine both properties, although we have to sacrifice the property that the product of two sets is again a set.
Chapter 6

Expressions

In this chapter we show how we model expressions in our formalism. The different products of the previous chapter enable us to model different kinds of evaluation like strict, non-strict and even non-deterministic evaluation. The chapter contains several theorems that show how concepts like typing and substitution fit into the model.

6.1 Definition

A language is usually divided into a syntax, describing the structure of the language and a semantics, describing the meaning of the language. The syntax is often described in some form of BNF-notation, for example

\[
\begin{align*}
\text{Exp} & : = \quad \text{var} \ Var \mid \text{con} \ Con \mid \text{un} \ Un \ Exp \mid \text{bin} \ Bin \ Exp \ Exp \\
\text{Var} & : = \quad a \mid \ldots \mid z \\
\text{Con} & : = \quad 0 \mid 1 \mid \ldots \\
\text{Un} & : = \quad - \\
\text{Bin} & : = \quad + \mid * 
\end{align*}
\]

The meaning of a language is described by a semantics function \( \ll \). The semantics function transforms a syntactic expression into what we call a semantic expression. A semantic expression is a function that maps a context to the value of the expression in that context. An often used context is a collection of variables, or more precise, a function that maps variable names to variable values. The straightforward definition of \( \ll \) for the above example is
\[
\begin{align*}
\langle \text{var } x \rangle.h &= h.x \\
\langle \text{con } n \rangle.h &= \langle n \rangle.h \\
\langle \text{un } e \rangle.h &= (\langle e_0 \rangle.h) \cdot (\langle e \rangle.h) \\
\langle \text{bin } o \ e_0 \ e_1 \rangle.h &= (\langle e_0 \rangle.h) \cdot (\langle e_1 \rangle.h) \\
\langle - \rangle.h &= - \\
\langle + \rangle.h &= + \\
\langle * \rangle.h &= \times \\
\langle 0 \rangle.h &= 0 \\
\langle 1 \rangle.h &= 1 \\
\ldots
\end{align*}
\]

This way of describing the semantics of a language is known as \textbf{denotational semantics}. In the above example, the mapping of \(-, +, *, 0, 1, \ldots\) to \(-, +, \times, 0, 1, \ldots\) appears rather 'clumsy'. This 'clumsiness' can be avoided by directly using semantic expressions. In that case an expression \textit{is} its semantics, being a function that maps contexts to values. If we allow every element as variable name and variable value and assume that each variable has a single value, then the type of contexts is \(I \leftarrow I\) and expressions are elements of type

\[
I \leftarrow (I \leftarrow I)
\]

A problem with this model for expressions is that every operator that we use to construct an expression, needs to be a total function (or a function that is total on the set of all pairs in case of binary operators). Most operators of everyday life are not total functions however. A possible solution could be to extend the collection of elements by an artificial element (often denoted by \(\perp\)) and use this as output for the inputs in which an operator was originally not single-valued. There is however a more natural approach possible that does not require the addition of such artificial elements. If we model expressions as functions that are not necessarily total on the type of contexts, i.e. as functions of type

\[
I \leftarrow (I \leftarrow I)
\]

then an expression can simply be empty in a certain context instead of having to output an artificial element. We even go a step further and model expressions as \textit{relations} with contexts as upper domain. This allows us to deal not only with \textbf{partiality}, but also with \textbf{non-determinism} in expressions. We also abstract from the specific \textbf{shape} of contexts, allowing arbitrary elements as contexts. Some common shapes are treated in chapter 10. For the theory of the upcoming chapters, choosing a particular shape for the context is irrelevant and would only distract from the core issues. The type of contexts is thus \(I\) and the type of expressions is

\[
I \leftarrow I
\]

In the next subsections we introduce several operators that enable us to construct expressions.
6.1.1 Context

The context expression $C \in I \mapsto I$ represents the value of the context:

$$y(C)z \equiv y = z$$

It is actually just another notation for the identity relation:

$$C = I$$

The value of the context expression $C$ in some context, is that context:

$$C.z = z$$

The reason why we use different symbols for the same thing is to somewhat re-obtain the separation of concerns that a semantics function provides: we separate the fact that we want to express the value of the context from the specific way this value is obtained. If we decide to model expressions differently, we do not need to change the “$I$” of every expression into the new thing that obtains the value, but only need to change the definition of $C$.

6.1.2 Constants

The constant lift $\bar{\cdot} \in (I \mapsto I) \mapsto I$ transforms an element into a constant expression:

$$y(\bar{a})z \equiv y = a$$

The constant lift is actually the constant function:

$$\bar{\cdot} = K$$

The value of constant expression $\bar{a}$ is $a$ in every context:

$$\bar{a}.z = a$$

6.1.3 Strict unary operators

The strict unary lift $\hat{\cdot} \in ((I \mapsto I) \mapsto (I \mapsto I)) \mapsto (I \mapsto I)$ transforms an arbitrary relation into a strict unary expression operator:

$$y(\hat{R}.E)z \equiv \exists(x \mid y(R)x \land x(E)z)$$

An equivalent definition in terms of the sequential composition is:
\[ R.E = R\cdot E \]

We use the convention that fix notation is preserved by \( \cdot \), so \( \hat{\cdot}E \) is equal to \( \hat{\cdot}E \).

For \( z \in \mathbb{R} \) with \( z \geq 0 \), we have for example

\[ (\cdot \sqrt{z})z = -\sqrt{z} \]

The reason for the adjective “strict” is that if expression \( E \) is empty in a context \( z \), then \( R.E \) is also empty in \( z \). The strict unary lift can for example not be used to construct an operator that checks whether or not an expression is ‘defined’. These kinds of operators are treated in section 6.1.5.

### 6.1.4 Loose binary operators

The loose binary lift \( \hat{\cdot} \in ((I\rightarrow I) \leftarrow (I\rightarrow I) \times (I\rightarrow I) \leftarrow (I\rightarrow (I\rightarrow \mathbb{2}))) \) transforms a relation with upper domain \( I\rightarrow \mathbb{2} \) into a loose binary expression operator:

\[
y(\hat{R}.(E_0,E_1))z \\
\equiv \\
\exists(T \parallel y(R)T) \land \forall(x \parallel x(T) \odot \equiv x(E_0)z) \land \forall(x \parallel x(T) \oplus \equiv x(E_1)z))
\]

The \( T \) of the existential quantification actually always exists and is uniquely determined by \( E_0, E_1 \) and \( z \). An equivalent definition is (see section 3.3.8 for the definitions of \( \blacklozenge \) and \( \blacklozenge \))

\[
y(\hat{R}.(E_0,E_1))z \\
\equiv \\
y(R) ((E_0\blacklozenge z) \blacklozenge \cup (E_1\blacklozenge z) \blacklozenge)
\]

As its name suggests, the loose binary lift is closely related to the loose product. The following equation shows how \( \hat{\cdot} \) can be defined in terms of this product:

\[ \hat{R}.(E_0,E_1) = R \odot E_0 \blacklozenge E_1 \blacklozenge \Delta \]

We leave the proof of equivalence of the above equations to the reader.

As an example of a loose binary operator, we model the conjunction of the four-valued logic E4 (see [42]). Its behaviour is described by the following table (\( \bot \) represents the case where an expression is empty in the context and \( \text{false} \circ \text{true} \) represents the case where an expression connects both booleans to the context):
A formal definition of \( \wedge_4 \in \mathcal{B} \mapsto (\mathcal{B} \mapsto \mathcal{T}) \) is given by the following equation:

\[
y(\wedge_4 \in \mathcal{B} \mapsto \mathcal{T}) =
\begin{align*}
&T \in \{\text{false}\} \iff \{\top\} \land \neg y \quad \lor \\
&T \in \{\text{false}\} \iff \{\bot\} \land \neg y \quad \lor \\
&T = ([\text{true}, \top], [\text{true}, \bot]) \land y \quad \lor \\
&T = ([\text{false}, \top], [\text{false}, \bot], [\text{true}, \bot]) \quad \lor \\
&T = ([\text{false}, \top], [\text{false}, \bot], [\text{false}, \bot], [\text{true}, \bot]) \quad \lor \\
&T = ([\text{false}, \bot], [\text{true}, \bot], [\text{true}, \bot], [\text{true}, \bot])
\end{align*}
\]

For the relation \( R \) defined by

\[ R = [(\text{true}, \text{true}), (\text{false}, \text{true})] \]

we have for example (\( \lnot \) also preserves fix notation):

\[
\begin{align*}
(\mathcal{R}.C \wedge_4 \text{false}) & \iff \text{false} = \{\text{false}\} \\
(\mathcal{R}.C \wedge_4 \text{false}) & \iff \text{true} = \{\text{false}\} \\
(\mathcal{R}.C \wedge_4 \text{true}) & \iff \text{false} = \emptyset \\
(\mathcal{R}.C \wedge_4 \text{true}) & \iff \text{true} = \{\text{false}, \text{true}\}
\end{align*}
\]

\subsection{Loose unary operators}

To be able to incorporate operators like the 'proper' \( \Delta_4 \) (this operator is not in any way related to our \( \Delta \)) and the 'defined' \( \tau_4 \) of the logic \( E4 \), we introduce the \textit{loose unary lift} \( \tilde{\in} \in ((I \mapsto I) \mapsto (I \mapsto I)) \mapsto (I \mapsto (I \mapsto \{\top\})) \) that transforms a relation with upper domain \( I \mapsto \{\top\} \) into a loose unary expression operator:

\[
y(\mathcal{R}.E) z \equiv \exists(T \mapsto y(\mathcal{R}) T) \land \forall(x \mapsto x(T) \equiv x(E) z)
\]

or equivalently:

\[
y(\mathcal{R}.E) z \equiv y(\mathcal{R})(E \mapsto z) \mapsto \top
\]

The operators \( \Delta_4 \) and \( \tau_4 \), whose behaviour is described by the following table:
can for example be defined by \( \Delta_4, \tau_4 \in B \leftrightarrow (B \twoheadrightarrow \{0\}) \) together with the following equations:

\[
\begin{align*}
\Delta_4 T & \equiv T \in I \leftrightarrow \{0\} \\
\tau_4 T & \equiv T \neq \emptyset
\end{align*}
\]

If we define the relation \( R \) again by

\[
R = [([true, true], (false, true)]
\]

we have (also also preserves fix notation)

\[
\begin{align*}
(\Delta_4 (R,C)) \cdot false & = false \\
(\Delta_4 (R,C)) \cdot true & = false \\
(\tau_4 (R,C)) \cdot false & = false \\
(\tau_4 (R,C)) \cdot true & = true
\end{align*}
\]

Loose unary operators are hardly ever used in this thesis, so we do not pay much attention to them anymore.

## 6.2 Non-strict evaluation

In 3.1.24 we described non-strict evaluation rather informally. Now that we have introduced expressions, we are able to formalise our notion of non-strict evaluation. Opposed to the negation (\( \neg \)) and the equivalence (\( \equiv \)) which are strict:

\[
\begin{align*}
\neg \in & { B \leftrightarrow B } \\
\equiv \in & { B \leftrightarrow B \times B }
\end{align*}
\]

the operators \( \land, \lor, \Rightarrow \) and \( \Leftarrow \) are non-strict, meaning that they are not only defined for pairs of booleans. The conjunction, disjunction, implication and consequence \( \land \lor \Rightarrow \Leftarrow \in B \leftrightarrow (I \twoheadrightarrow 2) \) satisfy the following equations:

\[
\begin{align*}
\gamma (\land) T & = \\
& (T \in \{false\} \leftrightarrow \{0\} \land \neg y) \lor \\
& (T \in \{false\} \leftrightarrow \{0\} \land \neg y) \lor \\
& (T = (true, true) \land y)
\end{align*}
\]
These equations are not definitions due to the fact that the involved formulas contain operators that they should define. They illustrate the main idea however.

If we restrict our attention to expressions that are functional and boolean-valued (expressions of type $\mathbb{B} \leftrightarrow \mathbb{I}$), the behaviour of our implication is in accordance with the following table:

<table>
<thead>
<tr>
<th>$\Rightarrow$</th>
<th>$\bot$</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>$\bot$</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>$\bot$</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

This is different from the implication $\Rightarrow_3$ of the logic E3 (see [42]) that is defined a little ‘looser’:

<table>
<thead>
<tr>
<th>$\Rightarrow_3$</th>
<th>$\bot$</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>$\bot$</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

So in case the expression at the left-hand side of $\Rightarrow_3$ (the antecedent) is ‘undefined’, the outcome is also always true. We think that in practice it is a good habit to make sure that antecedents (assumptions) are ‘defined’ in the contexts where their value is relevant. So for $y, z \in \mathbb{R}$ we do not write a formula like

$$x = (y/z = 1 \Rightarrow_3 y \neq z)$$

but instead write
\[ x = (z = 0 \lor (z \neq 0 \land (y/z = 1 \Rightarrow y \neq z))) \]

This example might convince the reader that ‘undefined’ antecedents are not that good an idea as people could inadvertently think that the first formula is equivalent to \( x = \text{false} \) whereas it is equivalent to \( x = (z = 0) \). Non-determinism in expressions we avoid because of our lack of experience with it. We do not know if it could be a serious source of errors. We however now show some facts that could be of interest to people that are more experienced in this subject.

If we write out the point-free definitions of \( \land, \neg, \cdot \) and \( \vdash \) for the expression

\[ C \in B \quad \vdash \neg C \]

we obtain

\[ \land \neg \land (\in \neg \Delta \star \neg I) \star \Delta \]

The use of \( \star \) enables non-strict evaluation. If we feed the above expression with context \texttt{piano}, it produces output \texttt{false}. However, if we replace the loose product by the cartesian product:

\[ \land \neg \in \Delta \times \neg I \star \Delta \]

then no output is connected with input \texttt{piano}.

We can thus select between different kinds of expression evaluation by plugging in another product. The product \( \otimes \text{FUN} \cap \text{RES} \), for example also enables non-strict evaluation. The expression

\[ \land \neg \in (I \otimes \text{FUN} \cap \text{RES}, K.B) \star \Delta \otimes \text{FUN} \cap \text{RES}, \neg I \star \Delta \]

also outputs \texttt{false} on input \texttt{piano}. The difference between \( \otimes \text{FUN} \cap \text{RES} \) and \( \star \) is that \( \otimes \text{FUN} \cap \text{RES} \), provides a default treatment of non-determinism whereas the use of \( \star \) requires this to be encoded in the operators (on the other hand providing more freedom in how to deal with non-determinism). In case we use \( \otimes \text{FUN} \cap \text{RES} \), then our non-strict conjunction:

\[
\begin{array}{c|ccc}
\land & \bot & false & true \\
\bot & false & \bot & false \\
false & false & false & false \\
true & false & true & \\
\end{array}
\]

would behave according to the following table in case non-determinism is involved:
The \textit{false} at the crosspoints of \texttt{⊥} and \textit{false}\texttt{□true} makes it different from \texttt{∧} of the logic E4 that has \texttt{⊥} at these crosspoints. Intuitively, the above table for \texttt{∧} does not seem that strange however. If one of the components is \textit{false}, the outcome is always \textit{false}. So, if one of the components is either \textit{false} or \textit{true} and the other one is ‘undefined’, it does not seem that strange to say that the outcome \textit{false} is ‘angelically’ chosen.

We get the same behaviour if we replace \texttt{∆} by the \textbf{cylindric-doubling function} \( \texttt{⊆} \triangleq (I \rightarrow (1)) \rightarrow (I) \), defined by

\[
T(\texttt{⊆})x =
\]

\[
T = (x, x) \lor T = [(x, \ominus)] \lor T = [(x, 0)] \lor T = \emptyset
\]

or equivalently:

\[
\texttt{⊆} = \texttt{⊆} \cdot \texttt{Δ}
\]

After this replacement, replacing \( \otimes_{\text{FUN∩RES}} \), by the product \( \otimes_{\text{FUN∩DEQ}} \) (see the end of section 5.6) still does not change the behaviour:

\[
E_0 \otimes_{\text{FUN∩DEQ}} E_1 \cdot \texttt{⊆} = E_0 \otimes_{\text{FUN∩DEQ}} E_1 \cdot \texttt{Δ}
\]

We leave it to the reader to verify this fact. Our knowledge about non-determinism in expressions is too limited to judge if this default treatment of non-determinism has all the nice properties it should have. We leave this for future research. For now, we simply stick to our strict unary and loose binary expression operators and avoid the use of non-determinism in expressions. Luckily not every theorem we construct is useless in case we would decide to use \( \otimes_{\text{FUN∩DEQ}} \) and \( \texttt{⊆} \) instead of * and \( \texttt{Δ} \). Using the first two is the same as using the last two and replacing every binary operator \( R \in I \rightarrow (I \rightarrow 2) \) by \( R \cdot (I \leftarrow I) \cdot \subseteq \):

\[
R \cdot E_0 \otimes_{\text{FUN∩DEQ}} E_1 \cdot \texttt{⊆} = R \cdot (I \leftarrow I) \cdot \subseteq \cdot E_0 \ast E_1 \cdot \texttt{Δ}
\]

So, if we proved something about \( \hat{R} \) for all \( R \), we also proved it in case we would use \( \otimes_{\text{FUN∩DEQ}} \) and \( \texttt{⊆} \) instead of * and \( \texttt{Δ} \).
6.3 Single-valuedness and typing

As stated before, we mainly use expressions in contexts where they are single-valued. The following single-valuedness type rules, whose proofs we leave to the reader, enable us to prove that an expression is single-valued and outputs elements of a specific type for certain contexts. For sets \(A, B(i)\) and \(C\) we have:

\[
\begin{align*}
C &\in C \leftarrow C \\
\bar{a} &\in A \leftarrow C \leftarrow a \in A \\
\bar{R},E &\in A \leftarrow C \leftarrow R \in A \leftarrow B \land E \in B \leftarrow C \\
\bar{R}(E_0, E_1) &\in A \leftarrow C \leftarrow R \in A \leftarrow B_0 \times B_1 \land E_0 \in B_0 \leftarrow C \land E_1 \in B_1 \leftarrow C
\end{align*}
\]

As an example we prove:

\[
C \bar{\cdot} \sqrt{C + 1} {\in} B \leftarrow N
\]

proof

\[
\begin{align*}
C \bar{\cdot} \sqrt{C + 1} &\in B \leftarrow N \\
\leftarrow &\{\text{single-valuedness type rule for binary operators, } \bar{\cdot} {\in} B \leftarrow \{\times\}\} \\
C \in R \leftarrow N \land \sqrt{C + 1} &\in R \leftarrow N \\
\leftarrow &\{\text{single-valuedness type rule for context, } R \leftarrow R \subseteq R \leftarrow N \land \text{single-valuedness type rule for unary operators, } \sqrt{\in} R \leftarrow N\} \\
C + 1 &\in N \leftarrow N \\
\leftarrow &\{\text{single-valuedness type rule for binary operators, } {\bar{\cdot} +} {\in} N \leftarrow N \times N\} \\
C &\in N \leftarrow N \land 1 &\in N \leftarrow N \\
\equiv &\{\text{single-valuedness type rule for context, } \text{single-valuedness type rule for constants, } 1 \in N\}
\end{align*}
\]

true

In case we use non-strict operators to construct an expression, it is possible that an expression is still single-valued in certain contexts if one of the expressions it is constructed from, is not. For these cases we can use the following rules:

\[
\begin{align*}
E_0 \bar{\land} E_1 &\in B \leftarrow C \leftarrow E_0 \in \{\text{false}\} \leftarrow C \\
E_0 \bar{\lor} E_1 &\in B \leftarrow C \leftarrow E_1 \in \{\text{false}\} \leftarrow C \\
E_0 \bar{\Rightarrow} E_1 &\in B \leftarrow C \leftarrow E_0 \in \{\text{true}\} \leftarrow C \\
E_0 \bar{\Leftarrow} E_1 &\in B \leftarrow C \leftarrow E_1 \in \{\text{true}\} \leftarrow C \\
E_0 \bar{\hat{\cdot}} E_1 &\in B \leftarrow C \leftarrow E_0 \in \{\text{false}\} \leftarrow C \\
E_0 \bar{\hat{\cdot}} E_1 &\in B \leftarrow C \leftarrow E_1 \in \{\text{false}\} \leftarrow C \\
E_0 \bar{\hat{\cdot}} E_1 &\in B \leftarrow C \leftarrow E_0 \in \{\text{true}\} \leftarrow C \\
E_0 \bar{\hat{\cdot}} E_1 &\in B \leftarrow C \leftarrow E_1 \in \{\text{true}\} \leftarrow C
\end{align*}
\]

We usually also need some more advanced rules for these cases, like for example the domain-split type rule:
\[ E \in A \iff C_0 \cup C_1 \equiv E \in A \iff C_0 \land E \in A \iff C_1 \]

and the disjoint-type rule:

\[ E_0 \notin E_1 \in \{\text{false}\} \iff C \iff E_0 \in A \iff C \land E_1 \in \wp B \iff C \land A \cap B = \emptyset \]

The following example illustrates the use of these rules:

\[ C \notin B \land C \in B \iff I \]

**proof**

\[ C \notin B \land C \in B \iff I \]

\[ \equiv \{ \text{domain-split type rule, } I = B \cup I \land B \} \]

\[ C \notin B \land C \in B \iff I \land B \]

\[ \equiv \{ \text{the truth of the first conjunct is left to the reader} \} \]

\[ C \notin B \land C \in B \iff I \land B \]

\[ \equiv \{ \text{type rule for non-strict conjunction} \} \]

\[ C \notin B \in \{ \text{false} \} \iff I \land B \]

\[ \equiv \{ \text{disjoint-type rule, } B \cap I \land B = \emptyset \} \]

\[ C \in I \land B \iff I \land B \land B \in \wp B \iff I \land B \]

\[ \equiv \{ \text{type rule for context and type rule for constants, } B \in \wp B \} \]

\[ \text{true} \]

### 6.4 Theorem lifting

Suppose that an expression \( E \) is constructed with \( C, \overline{\cdot}, \hat{\cdot}, \) and \( \overline{\cdot} \). In case \( E \) and all its syntactic subexpressions (all expressions that are used for the construction of \( E \)) are single-valued in a context \( z \), the result of applying \( E \) to \( z \) is a formula that is equal to \( E \) where, at the syntactic level, all accents (\( \overline{\cdot}, \hat{\cdot}, \) and \( \overline{\cdot} \)) are removed and each \( C \) is replaced by \( z \):

\[
\begin{align*}
C .z &= z \\
\overline{a} .z &= a \\
(R.E) .z &= R.(E.z) & \iff E \in I \iff \{z\} \land R \in I \iff \{E.z\} \\
(R.(E_0,E_1)).z &= R.(E_0.z,E_1.z) & \iff E_0,E_1 \in I \iff \{z\} \land R \in I \iff \{(E_0.z,E_1.z)\}
\end{align*}
\]

Because we proved for example that

\[ C \in \sqrt{C+1} \]
and all its syntactic subexpressions are single-valued in contexts that are natural numbers, we may apply it to any context \( z \in \mathbb{N} \), with result

\[
z < \sqrt{z+1}
\]

This enables us to straightforwardly transform theorems from our mathematical language into theorems about expressions. Because in our mathematical language we have the rule that for \( z \in \mathbb{N} \)

\[
z < \sqrt{z+1} = z \times z \leq z
\]

we also know that

\[
(C \leq \sqrt{C+1}) \mathbb{N} = (C \times C \leq C) \mathbb{N}
\]

We call this theorem lifting. The "\( \mathbb{N} \)" makes explicit that the equality holds under the assumption that the context is of type \( \mathbb{N} \).

### 6.5 Assignment and substitution

The usual definition of the assignment \( a := E \) is that it inputs a context \( h \) that consists of a collection of variables and outputs a context \( g \) where the value of the variable with name \( a \) has acquired the value that expression \( E \) has in context \( h \). All other variables are left unchanged. In case of shapeless contexts, we could say that we only have one (nameless) variable: the context itself. The expression of an assignment is the assignment then. The assignment that transforms the value of the context into the value of the context plus 1, is for example the expression

\[
C + 1
\]

Suppose now that we have an expression \( E \) that is constructed with \( \odot, \odot, \odot \) and \( \odot \). A consequence of the following rules is that for an assignment \( S \in I \leftarrow I \), in contexts that make \( S \) single-valued, \( E \cdot S \) is equal to \( E \) where, at the syntactic level, every \( C \) is replaced (substituted) by \( S \):

\[
\begin{align*}
C \cdot S &= S \\
\tilde{a} \cdot S &= \tilde{a} \cdot S \\
\bar{R}.E \cdot S &= \bar{R}.E \cdot S \\
\bar{R}.\langle E_0, E_1 \rangle \cdot S &= \bar{R}.\langle E_0 \cdot S, E_1 \cdot S \rangle \cdot S \quad \text{\( S \) is a function}
\end{align*}
\]

We leave it to the reader to prove the first three rules. The fourth rule can be proved as follows:
\( \tilde{R}.(E_0, E_1) * S \)
\[ = \quad \{ \text{definition of } \tilde{\ } \}
\]
\[ R * E_0 * E_1 * \Delta * S \]
\[ = \quad \{ \bullet S \text{ is a function, see below} \}
\]
\[ R * E_0 * E_1 * S * S * \Delta * S \]
\[ = \quad \{ \text{abundance of } * \text{ with } \tilde{\ } \}
\]
\[ R * (E_0 * S * E_1 * S) * \Delta * S \]
\[ = \quad \{ \text{definition of } \tilde{\ } \}
\]
\[ \tilde{R}.(E_0 * S, E_1 * S) * S \]

The second step is justified by the following calculation:

\[ T (S * S * \Delta * S) * S \]
\[ \equiv \quad \{ \text{shunting, } S \text{ is a set} \}
\]
\[ T (S * S * \Delta) * S \]
\[ \equiv \quad \{ \text{shunting (twice), } \Delta \text{ and } S * S \text{ are total functions} \}
\]
\[ T = (S * S)(\Delta) * S \]
\[ \equiv \quad \{ \text{definition of } * \} \]
\[ T = (S)(\Delta)(S) * z \quad \land \quad z \in S \]
\[ = \quad \{ \text{left to the reader, } \bullet S \text{ is a function} \}
\]
\[ T = [(S)(z), (S)(z)] * z \quad \land \quad z \in S \]
\[ = \quad \{ \text{tuple enumeration} \}
\]
\[ T = (S)z, (S)z \quad \land \quad z \in S \]
\[ = \quad \{ \text{definition of } \Delta \}
\]
\[ T = \Delta (S)z \quad \land \quad z \in S \]
\[ \equiv \quad \{ \text{shunting, } \Delta \text{ is a total function} \}
\]
\[ T (\Delta) S z \quad \land \quad z \in S \]
\[ \equiv \quad \{ \text{shunting, } \bullet S \text{ is a function} \}
\]
\[ T (\Delta) S z \quad \land \quad z \in S \]

As an example, we show how a relation \( f \in I \leftrightarrow B \) distributes over the expression

\[ \tilde{\mathcal{C}} \tilde{\mathcal{O}} + 2 \]

Notice that \( f * B \) is a function with domain \( B \).
\[(\vdash C \oplus 2) \circ f \circ B\]
\[= \quad \{\text{substitution rule for binary operators}\}\]
\[(\vdash (C \circ f \circ B) \oplus 2 \cdot f \cdot B) \circ B\]
\[= \quad \{\text{substitution rule for unary operators and constants}\}\]
\[(\vdash (C \circ f \circ B) \oplus 2 \cdot B) \circ B\]
\[= \quad \{\text{substitution rule for context}\}\]
\[(\vdash (f \circ B) \oplus 2 \cdot B) \circ B\]
\[= \quad \{\text{substitution rule for unary operators the other way around}\}\]
\[(\vdash f \circ B \oplus 2 \cdot B) \circ B\]
\[= \quad \{\text{substitution rule for binary operators the other way around, a set is a function and is equal to its own domain}\}\]
\[(\vdash f \oplus 2) \circ B\]

Notice how the ‘reversed’ substitution rules enable us to get rid of the \(B\)-type restrictions in the expression. From now on, we do not write out all these substitution steps anymore but only write
\[(\vdash C \oplus 2) \circ f \circ B\]
\[= \quad \{\text{substitution, } f \in I \leftarrow B\}\]
\[(\vdash f \oplus 2) \circ B\]

### 6.6 Collection-comprehension operator

Collections are often described by predicates. The **collection-comprehension operator** \([], [\cdot]\) \(\in P I \leftarrow\{1\leftarrow\{1\}\}\) transforms an expression into a collection. The collection \([E]\) contains element \(x\), exactly when expression \(E\) is single-valued in \(x\), with output \(true\):

\[x \in [E] \equiv E \in \{true\} \leftarrow \{x\}\]

So, in case \(E \in B \leftarrow \{x\}\), we have

\[x \in [E] \equiv E.x\]

An immediate result is that for a collection \(C\)

\([C \in C]\) \(= C\)

The definition of the collection-comprehension operator is such that the conjunction of expressions corresponds to intersection of the corresponding collections, the disjunction corresponds to the union, the constant-\(false\) expression corresponds to the empty collection and the constant-\(true\) expression corresponds to the full collection:
We leave the rather straightforward proofs to the reader.

Because \[\llbracket \text{false} \rrbracket = \emptyset\] sort of identifies ‘undefinedness’ with false, \(\llbracket \neg E \rrbracket \neq \llbracket E \rrbracket\) does not hold in general (take for example \(E = \emptyset\)). However, in case \(E\) is a total predicate, we have a similar rule for \(\neg\):

\[\llbracket \neg E \rrbracket = \llbracket \text{false} \rrbracket \iff E \in \mathcal{B} \leftarrow I\]

A similar thing holds for \(\Rightarrow\), \(\Leftarrow\) and \(\equiv\):

\[\llbracket E_0 \Rightarrow E_1 \rrbracket = \llbracket E_0 \rrbracket \supset \llbracket E_1 \rrbracket \iff E_0 \in \mathcal{B} \Leftarrow I\]
\[\llbracket E_0 \Leftarrow E_1 \rrbracket = \llbracket E_0 \llbracket \supset \llbracket E_1 \rrbracket \iff E_1 \in \mathcal{B} \Leftarrow I\]
\[\llbracket E_0 \equiv E_1 \rrbracket = \llbracket E_0 \rrbracket \llbracket E_1 \rrbracket \iff E_0, E_1 \in \mathcal{B} \Leftarrow I\]

The proofs are again left to the reader.

For \(\Rightarrow\) and \(\Leftarrow\), the logic E3 eliminates the need for these ‘definedness’ restrictions. However, as we already mentioned, we prefer to only use antecedents that, in the contexts where their value is relevant, are single-valued (and boolean-valued of course).

We sometimes want to prove an inclusion or equality between collections that are described by expressions. The following rules show how the proof can be kept at the level of expressions (although our rather incomplete set of rules for expressions usually still forces us to use theorem lifting):

\[\llbracket E_0 \Rightarrow E_1 \rrbracket = [\text{true}] \iff \llbracket E_0 \rrbracket \supset \llbracket E_1 \rrbracket \iff E_0 \in \mathcal{B} \Leftarrow I\]
\[\llbracket E_0 \Leftarrow E_1 \rrbracket = [\text{true}] \iff \llbracket E_0 \rrbracket \supset \llbracket E_1 \rrbracket \iff E_1 \in \mathcal{B} \Leftarrow I\]
\[\llbracket E_0 \equiv E_1 \rrbracket = [\text{true}] \iff \llbracket E_0 \rrbracket \equiv \llbracket E_1 \rrbracket \iff E_0, E_1 \in \mathcal{B} \Leftarrow I\]

We leave the proofs again to the reader (hint: use everywhere, defined in section 3.3.11).

Although we formulated all these theorems for total predicates, they also hold for not necessarily total predicates if proper domain restrictions are added. We refrain from spelling out the formulas as it probably does not provide more insight.

### 6.7 Set-comprehension operator

A set is also often described by means of a predicate. The **set-comprehension operator** \(\{ \} \in \varnothing \Leftarrow (I \mapsto I)\) transforms an expression into a set. The set \(\llbracket E \rrbracket\)
owns exactly every element for which expression $E$ is single-valued with output true:

$$x \in \{E\} \equiv E \in \{\text{true}\} \leftrightarrow \{x\}$$

So, in case $E \in \mathcal{B} \leftrightarrow \{x\}$, we have

$$x \in \{E\} \equiv E.x$$

An immediate result is that for a set $A$

$$\{C \in A\} = A$$

The fact that

$$x \in \{E\} \equiv x \in \{E\}$$

enables us to transform theorems about $\llbracket \cdot \rrbracket$ into theorems about $\lbrack \cdot \rbrack$:

$$\begin{align*}
\llbracket E_0 \land E_1 \rrbracket &= \llbracket E_0 \rrbracket \cap \llbracket E_1 \rrbracket \\
\llbracket E_0 \lor E_1 \rrbracket &= \llbracket E_0 \rrbracket \cup \llbracket E_1 \rrbracket \\
\llbracket E_0 \Rightarrow E_1 \rrbracket &= (\llbracket E_0 \rrbracket \Rightarrow \llbracket E_1 \rrbracket) \cap I \iff E_0 \in \mathcal{B} \leftarrow I \\
\llbracket E_0 \Leftarrow E_1 \rrbracket &= (\llbracket E_0 \rrbracket \Leftarrow \llbracket E_1 \rrbracket) \cap I \iff E_1 \in \mathcal{B} \leftarrow I \\
\llbracket \text{false} \rrbracket &= \emptyset \\
\llbracket \text{true} \rrbracket &= I \\
\llbracket \neg E \rrbracket &= (\neg \llbracket E \rrbracket) \cap I \iff E \in \mathcal{B} \leftarrow I \\
\llbracket E_0 \Rightarrow E_1 \rrbracket = \{\text{true}\} &= \llbracket E_0 \rrbracket \subseteq \llbracket E_1 \rrbracket \iff E_0 \in \mathcal{B} \Leftarrow I \\
\llbracket E_0 \Leftarrow E_1 \rrbracket = \{\text{true}\} &= \llbracket E_0 \rrbracket \supseteq \llbracket E_1 \rrbracket \iff E_1 \in \mathcal{B} \Leftarrow I \\
\llbracket E_0 \equiv E_1 \rrbracket = \{\text{true}\} &= \llbracket E_0 \rrbracket = \llbracket E_1 \rrbracket \iff E_0, E_1 \in \mathcal{B} \Leftarrow I
\end{align*}$$

### 6.8 Connectional expressions

In this section we treat the special case of expressions with connections as contexts.

#### 6.8.1 Definition

Where expressions have arbitrary elements as input, **connectional expressions** have connections $(y, z)$ as input. The type of connectional expressions is thus $\mathcal{I} \leftarrow (\mathcal{I} \times \mathcal{I})$. In the setting of connectional expressions, we call $\llbracket \cdot \rrbracket$ the relation-comprehension operator.
6.8.2 Constructing connectional expressions

The output conditioner and input conditioner \( \ast \in (I \leftarrow I) \) can be used to transform an expression into a connectional expression that talks about the output of the context or the input of the context respectively:

\[
\begin{align*}
\hat{w}(\hat{E})(y,z) & \equiv w(E) y \\
\check{w}(\check{E})(y,z) & \equiv w(E) z
\end{align*}
\]

In particular:

\[
\begin{align*}
\hat{C}.(y,z) & = y \\
\check{C}.(y,z) & = z
\end{align*}
\]

We leave it to the reader to verify the following distribution theorems:

\[
\begin{align*}
\hat{C}.(\bar{a}) & = \bar{a} \\
\check{C}.(\dot{R}.E) & = \dot{R}.E \\
\hat{C}.(\dot{R}.\langle E_0, E_1 \rangle) & = \dot{R}.\langle \hat{E}_0, \hat{E}_1 \rangle \\
\check{C}.(\dot{R}.\langle E_0, E_1 \rangle) & = \dot{R}.\langle \check{E}_0, \check{E}_1 \rangle
\end{align*}
\]

If an expression \( E \) is constructed with \( \ast, \bar{\ast}, \dot{\ast} \), then applying \( \hat{\ast} \) or \( \check{\ast} \) to it, is thus the same as syntactically replacing every \( \ast \) by \( \hat{\ast} \). The same holds for \( \check{\ast} \).

6.8.3 Assignment and substitution

In case the assignment is the connectional product of two relations \( S_0 \) and \( S_1 \) (see section 3.7), we have the following rules for \( \hat{C} \) and \( \check{C} \):

\[
\begin{align*}
\hat{C} \circ S_0 \ast S_1 & = S_0 \ast \hat{C} \circ (I \ast S_1) \\
\check{C} \circ S_0 \ast S_1 & = S_1 \ast \check{C} \circ (S_0 \ast I)
\end{align*}
\]

The proofs of these theorems are left to the reader.

If \( S_0 \) is constructed with the operators \( \ast, \bar{\ast} \) and \( \dot{\ast} \) then the substitution rules for expressions (section 6.5) allow us to syntactically transform \( S_0 \ast \hat{C} \) into \( S_0 \) where each \( \ast \) is replaced by \( \hat{\ast} \) (notice that \( \hat{C} \) is a total function). The same holds for \( S_1 \ast \check{C} \).

A theorem that is closely related to the shunting theorems and that we also refer to as shunting, is that for a connectional expression \( E \) and relations \( S_0 \) and \( S_1 \):

\[
S_0 \ast \lfloor E \rfloor \ast S_1 = \lfloor E \ast S_0 \ast S_1 \rfloor
\]

The proof is left to the reader.
The proof of the following equation shows the above theorems in action.

\[ R \circ (C \downarrow 2) \circ [\hat{C} \preceq \hat{C}] \circ C \downarrow 1 \circ R = R \circ [\hat{C} \downarrow 2 \preceq \hat{C} \downarrow 1] \circ R \]

**proof**

\[
R \circ (C \downarrow 2) \circ [\hat{C} \preceq \hat{C}] \circ C \downarrow 1 \circ R = R \circ [\hat{C} \downarrow 2 \preceq \hat{C} \downarrow 1] \circ R
\]

= (shunting)

\[
\hat{C} \preceq \hat{C} \circ (C \downarrow 2 \circ C \downarrow 1) \circ R
\]

= (shunting)

\[
[\hat{C} \preceq \hat{C} \circ (C \downarrow 2 \circ C \downarrow 1) \circ R \circ R]
\]

= {substitution rule for binary operators, (C \downarrow 2 \circ C \downarrow 1) \in I \leftrightarrow R \circ R}

\[
[\hat{C} \circ (C \downarrow 2 \circ C \downarrow 1) \preceq \hat{C} \circ (C \downarrow 2 \circ C \downarrow 1) \circ R \circ R]
\]

= {substitution rule for \(\circ\) and \(\cdot\), the trick of the example in section 6.5 can again be used to get the \(\downarrow\)s in and out}

\[
[(C \downarrow 2 \circ \hat{C}) \preceq (C \downarrow 1 \circ \hat{C}) \circ R \circ R]
\]

= {substitution, \(\hat{C}\) and \(\hat{C}\) are total functions}

\[
[(C \downarrow 2 \circ \hat{C}) \preceq (C \downarrow 1 \circ \hat{C}) \circ R \circ R]
\]

= (shunting)

\[
\hat{C} \downarrow 2 \preceq \hat{C} \downarrow 1 \circ R
\]

From now on this is shortened to

\[
R \circ (C \downarrow 2) \circ [\hat{C} \preceq \hat{C}] \circ C \downarrow 1 \circ R = R \circ [\hat{C} \downarrow 2 \preceq \hat{C} \downarrow 1] \circ R
\]

\[
\{[E] \circ \Pi = [\hat{E}]
\]

\[\Pi \cdot [E] = [\hat{E}]\]

Together with the fact that

\[I = [\hat{C} \equiv \hat{C}]
\]

and \([E_0] \wedge [E_1] = [E_0] \cap [E_1]\) (see section 6.6), it is also easy to prove

\[\{[E] = [\hat{C} \equiv \hat{C} \wedge \hat{E}]
\]

\[\{[E] = [\hat{C} \equiv \hat{C} \wedge \hat{E}]
\]

134
The connectional expression on the right-hand side of $\bar{\lambda}$ may also contain a mix of “$C$”s and “$\bar{C}$”s.

To write an expression in terms of the relation-comprehension operator, we can use the theorem that for a set $B$

$$E \cdot B = [[C \equiv \bar{E}] \cdot B \iff E \in I \leftrightarrow B]$$

### 6.9 Notational conventions

For readability, we omit from now on the accents $\bar{\cdot}$, $\cdot$ and from formulas written within $[[\cdot]]$ or $\{\cdot\}$, assuming that the reader is able to put these accents at the right places. We furthermore prefer the notation $[[C = \bar{E}]]$ to $E$, although we have to be careful that $E$ is properly typed. So, the last equation of section 6.8.3 is from now on written as

$$R \cdot [[C = \hat{C} + 2] \cdot [[C \leq \hat{C}] \cdot [[C = \hat{C} + 1] \cdot R] = \{\text{substitution, } C+2, C+1 \in I \leftrightarrow R\} \cdot R \cdot [[\hat{C} + 2 \leq \hat{C} + 1]] \cdot R$$

### 6.10 Conclusions

The purpose of this chapter was to introduce a model for expressions that incorporates non-strictness in an elegant way. The idea of ‘semantic expressions’ is not new, although we do not know to whom this idea should be attributed. Our treatment of non-strictness appears to be new however, being the result of our detailed analysis of product-like constructs.

This chapter contains several theorems that show how fundamental concepts like typing and substitution fit into the presented model for expressions. Although most theorems are formulated in terms of a single variable (the context), the theory about connectional expressions illustrates how to treat contexts that consist of multiple variables, like the type of contexts that we introduce in chapter 10.
Chapter 7

Specifications

In this chapter we show how several concepts that are often found in practical specification languages can be formalised if we use relations as model for specifications.

7.1 Refinement

Refinement plays an essential role in the development of software. If we choose a certain refinement relation for our development process, we fix what we consider to be ‘acceptable specialisations’ (refinements) of some specification. In the context of a refinement where \( S' \) refines \( S \), we call \( S' \) the fine specification and \( S \) the coarse specification.

If we talk about refinement, we mean refinement with refinement relation \( \subseteq \), also called partial refinement. A specification \( S' \) is a thus refinement of a specification \( S \) if \( S' \) is a subrelation of \( S \):

\[
S' \subseteq S
\]

Refinements of a relation are created by reducing the collection of outputs that are possible on an input, even if the collection becomes empty. The specification that tells that the inputs and outputs are integer numbers and that the output is at most the input:

\[
\mathbb{Z} \cdot [\hat{\mathbb{C}} \leq \hat{\mathbb{C}}] \cdot \mathbb{Z}
\]

is for example refined by the specification that tells that the input is an integer number and that the output is one less than the input:

\[
[\hat{\mathbb{C}} = \hat{\mathbb{C}} - 1] \cdot \mathbb{Z}
\]
It is however also refined by the specification that tells that the output is integer and that the input is the square of the output:

\[ \mathbb{Z} \models \hat{c}^2 = \hat{c} \]

The original specification connects every integer input to some output, but this refinement does not. It is for example empty in 2. This emptiness issue is addressed in section 7.8.

### 7.2 Post and pre

A specification is often described by means of a \textbf{post} that tells what the specification guarantees and a \textbf{pre} that tells under which circumstances the specification guarantees this. In our relational model, a post is a relation \( R \), describing which outputs the specification connects with which inputs, and a pre is a set \( P \), describing the inputs for which the specification guarantees something. The specification \( S \) that is described by them, is defined by

\[
S = R/P
\]

or equivalently:

\[
S = R \in \Pi \cdot P
\]

or point wise:

\[ y(S) \equiv y(R) \iff z \in P \]

An example is the specification \( \text{sqrt} \) that outputs the square root of its input if this input is a real number that is at least zero:

\[
\begin{align*}
\text{sqrt} &= \text{sqrt}_{\text{post}} / \text{sqrt}_{\text{pre}} \\
\text{sqrt}_{\text{post}} &= [\hat{c} = \sqrt{\hat{c}}] \\
\text{sqrt}_{\text{pre}} &= [\hat{c} \in \mathbb{R} \land \hat{c} \geq 0]
\end{align*}
\]

Inputs that do not adhere to the pre, in the example the inputs that are less than zero or not even a real number, are connected to every output. This means that a refinement is completely free in its behaviour for such inputs. Specifically, a specification that is described by a post \( R \) and a pre \( P \) is refined by a specification that is described by a post \( R' \) and a pre \( P' \) if \( P \) refines \( P' \) and \( R' \) refines \( R \) for inputs from \( P \):

\[
\begin{align*}
R'/P' \subseteq R/P \\
\iff \{ \text{\( P \) and \( P' \) are sets} \} \\
R' \subseteq P' \land R' \cdot P \subseteq R
\end{align*}
\]
A more general case is proved in section 7.3.5.

7.3 Partial correctness

By generalising pres to be relations instead of sets, it is possible to construct a proof system for refinement that is very similar to Hoare logic for partial correctness [24].

The partial-correctness operator \( \texttt{−−}) \) \( \in \mathcal{B} \) \( \text{←} (I \text{→} I) \times (I \text{→} I) \times (I \text{→} I) \) is defined by any of the following three equations:

\[
\begin{align*}
R \text{−−}) S(P) & \equiv S \subseteq R/P \\
R \text{−−}) S(P) & \equiv S \cdot P \subseteq R \\
R \text{−−}) S(P) & \equiv P \subseteq S \setminus R
\end{align*}
\]

If \( R \text{−−}) S(P) \) holds, we call specification \( S \) partially correct with respect to post \( R \) and pre \( P \). The pre \( P \) is ‘what has already been established’, the post \( R \) is ‘what has to be established’ and the specification \( S \) is ‘how it is established’. Notice that we place the post on the left and the pre on the right. An example is

\[
\mathcal{Z} \cdot [\hat{C} > \hat{C}] \cdot \mathcal{Z} \cdot [\hat{C} = \hat{C} + 1] \cdot \mathcal{Z} \cdot [\hat{C} \geq \hat{C}] \cdot \mathcal{Z}
\]

telling that if it has already been established that the output is an integer number that is at least as large as the integer input, then adding 1 transforms this output into an integer number that is larger than the input.

In the following subsections we introduce several partial-correctness rules that show how \( R \text{−−}) S(P) \) can be proved in certain cases. An example shows these rules in action. Except for the rule for \( / \), no rule strengthens the original proof obligation. This gives some sense of completeness, a property that is often strived for in proof systems. The rules whose proof is omitted, are trivial results of the three definitions of \( \text{−−}) \).

7.3.1 Conjunctivity of post

The partial-correctness proof of a specification with respect to a post that is the intersection of two relations, can be split up into two partial-correctness proofs:

\[
R_0 \cap R_1 \text{−−}) S(P)
\]

\[
\equiv
\]

\[
R_0 \text{−−}) S(P) \land R_1 \text{−−}) S(P)
\]
7.3.2 Disjunctivity of pre

The partial-correctness proof of a specification with respect to a pre that is the union of two relations, can also be split up into two partial-correctness proofs:

\[
R \to S(P_0 \cup P_1) \\
\equiv \\
R \to S(P_0 \land R) \land S(P_1)
\]

7.3.3 Binary union

Partial correctness of the union of two specifications can be split up into partial correctness of the individual specifications:

\[
R \to S_0 \cup S_1(P) \\
\equiv \\
R \to S_0(P \land R) \lor S_1(P)
\]

7.3.4 Sequential composition

For the sequential composition we have three partial-correctness rules. The first one says that we have to invent a relation \( X \) that acts as post for the ‘firstly executed’ specification \( S_1 \) and as pre for the ‘secondly executed’ specification \( S_0 \):

\[
R \to S_0 \circ S_1(P) \\
\equiv \\
\exists \langle X \mid R \to S_0(X \land X) \circ S_1(P) \rangle
\]

proof

\[
R \to S_0 \circ S_1(P) \\
\equiv \{ \text{definition of } \circ \} \\
S_0 \circ S_1(P) \subseteq R \\
\equiv \{ \Rightarrow: \text{take } X = S_1 \circ P, \Rightarrow: \text{monotonicity of } \circ \text{ and transitivity of } \subseteq \} \\
\exists \langle X \parallel R \to S_0(X) \subseteq X \land S_1 \circ P \subseteq X \rangle \\
\equiv \{ \text{definition of } \parallel \} \\
\exists \langle X \parallel R \to S_0(X \land X) \circ S_1(P) \rangle
\]

The second rule tells how the pre should be transformed in order to eliminate the ‘firstly executed’ specification \( S_1 \):
\[ R)S_0 \leq S_1(P) \]
\[ \equiv \]
\[ R)S_0(S_1 \leq P) \]

The third rule tells how the post should be transformed in order to eliminate the 'secondly executed' specification \( S_0 \):
\[ R)S_0 \leq S_1(P) \]
\[ \equiv \]
\[ S_0 \setminus R)S_1(P) \]

### 7.3.5 Over

Partial correctness of a specification that is described by means of a post/pre combination, can be proved with the following rule:
\[ R)R'/P'(P \]
\[ \equiv \]
\[ \{ \bullet P' \text{ is a set and } \bullet P' \subseteq P' \} \]
\[ R)R'(P \]

**proof**

\[ R)R'/P'(P \]
\[ \equiv \]
\[ \{ \text{definition of } \bigcup \} \]
\[ (R'/P').P \subseteq R \]
\[ \equiv \]
\[ \{ R'/P' = R' \in \Pi \text{ if } \bullet P' \text{ is a set} \} \]
\[ (R' \in \Pi \cdot P').P \subseteq R \]
\[ \equiv \]
\[ \{ X \equiv Y = X \cup -Y \} \]
\[ (R' \cup -((\Pi \cdot P')).P \subseteq R \]
\[ \equiv \]
\[ \{ \text{distribution of } \cdot \text{ over } \cup \} \]
\[ R' \cdot P \cup -((\Pi \cdot P')).P \subseteq R \]
\[ \equiv \]
\[ \{ -(\Pi \cdot P').P = \emptyset \text{ if } \bullet P' \subseteq P' \} \]
\[ R' \cdot P \cup \emptyset \subseteq R \]
\[ \equiv \]
\[ \{ \emptyset \text{ is unit of } \cup \} \]
\[ R' \cdot P \subseteq R \]
\[ \equiv \]
\[ \{ \text{definition of } \bigcup \} \]
\[ R)R'(P \]

Informally, this rule can be read as: “The pre of the call should imply the pre of the called specification and the post of the called specification should imply the post of the call under the assumption of the pre of the call.”.
7.3.6 Identity relation

The identity relation is correct with respect to post $R$ and pre $P$ if $R$ is refined by

$P$:

\[
R \sqsubseteq P
\]

7.3.7 Iteration

Partial correctness of the iteration of a specification can be proved by inventing a proper loop invariant $X$:

\[
R \sqsubseteq S^*(P)
\]

\[
\equiv \exists \langle X \parallel R \supseteq X \land X \rangle S(X \land X \supseteq P)
\]

proof

We split the proof into $(\Rightarrow)$ and $(\Leftarrow)$:

$(\Rightarrow)$:

\[
R \sqsubseteq S^*(P)
\]

\[
\equiv \quad \{ \text{definition of } \sqsubseteq \}
\]

\[
S^* \subseteq R
\]

\[
\Rightarrow \quad \{ \text{take } X = S^* \land P \text{ and use the facts } S \cdot S^* \subseteq S^* \text{ and } I \subseteq S^* \}
\]

\[
\exists \langle X \parallel X \subseteq R \land S \cdot X \subseteq X \land P \subseteq X \rangle
\]

\[
\equiv \quad \{ \text{definition of } \sqsubseteq \}
\]

\[
\exists \langle X \parallel X \subseteq R \land X \rangle S(X \land P \subseteq X)
\]

$(\Leftarrow)$:
7.3.8 If-then-else

The if-then-else \( \text{if}_\text{then}_\text{else} \in (I \Rightarrow I) \leftarrow \emptyset I \times (I \Rightarrow I) \times (I \Rightarrow I) \) is defined by

\[
\text{if } B \text{ then } S_0 \text{ else } S_1 = S_0 \cup B \cup S_1 \setminus (I \setminus B)
\]

or equivalently:

\[
\text{if } B \text{ then } S_0 \text{ else } S_1 = S_0 \cup B \cap S_1 \setminus (I \setminus B)
\]

The equivalence of both definitions is a result of the fact that the guards \( B \) and \( I \setminus B \) are disjoint \( (B \cap I \setminus B = \emptyset) \) and set-conjoint \( (B \cup I \setminus B = I) \).

A partial-correctness rule for if \( B \) then \( S_0 \) else \( S_1 \) can be derived from the rules of the constituent components:

\[
R)\text{if } B \text{ then } S_0 \text{ else } S_1(P) \\
\equiv \{ \text{definition of } \text{if}_\text{then}_\text{else} \} \\
R)S_0 \cup B \cup S_1 \setminus (I \setminus B)(P) \\
\equiv \{ \text{partial-correctness rule for } \cup \} \\
R)S_0 \cup B(P \land R)S_1 \setminus (I \setminus B)(P) \\
\equiv \{ \text{second partial-correctness rule for } \cup \} \\
R)S_0(B \setminus P \land R)S_1((I \setminus B) \setminus P)
\]

7.3.9 While

Another common construct in programming languages is the while. The program
while \( [\mathbb{C} \neq 0 \land \mathbb{C} \neq 1] \) do \( \hat{\mathbb{C}} = \hat{\mathbb{C}} - 2 \)

outputs for example 0 if the input is a non-negative even integer and 1 if the input is a non-negative odd integer. If \( x \) is a negative integer, the program enters an infinite loop.

How should we define the ‘while’ in our relational model? The following equation shows the general intuition behind its meaning:

\[
\text{while } B \text{ do } S = \text{if } B \text{ then } (\text{while } B \text{ do } S) \circ S \text{ else } I
\]

So, while \( B \) do \( S \) is a fixed point of the function

\[
\langle \text{if } B \text{ then } X \circ S \text{ else } I \mid | \mid X \rangle
\]

Which fixed point should we now take? In case we want to be able to describe infinite behaviour with the ‘while’, the least fixed point does not seem a proper candidate because it identifies while \( I \) do \( S \) with \( \emptyset \) for each \( S \):

\[
\mu(\text{if } I \text{ then } X \circ S \text{ else } I \mid | \mid X) =
\mu(X \circ S \cup I \setminus (I \setminus I) \mid | \mid X) =
\{ I \text{ is unit of } \circ, I \setminus I = \emptyset, \emptyset \text{ is unit of } \cup \}
\mu(X \circ S \mid | \mid X) =
\{ \emptyset \text{ is the smallest } X \text{ such that } X \circ S \subseteq X \}
\emptyset
\]

If all infinite loops are identified with \( \emptyset \), then a generalisation of relations that captures information about events that occur while a program executes, would not be able to capture events that occur in an infinite loop.

Maybe the greatest fixed point or even another one? We refrain from making a choice by using the ‘while’ only in case we know for sure that the function \( \langle \text{if } B \text{ then } X \circ S \text{ else } I \mid | \mid X \rangle \) has exactly one fixed point. In [17] it is shown that this is equivalent to the relation \( S \circ B \) being well-founded. A compact definition of well-foundedness of a relation \( R \) is \( \nu(X \circ R \mid | \mid X) = \emptyset \) (or using section notation: \( \nu(\circ R) = \emptyset \)).

Using the theorems of [17], the least fixed point of \( \langle \text{if } B \text{ then } X \circ S \text{ else } I \mid | \mid X \rangle \) can be shown to be equal to \( I \setminus B \circ (S \circ B)^* \). In other words, we have
\[
\text{while } B \text{ do } S
\]
\[
= \{ \bullet S \cdot B \text{ is well-founded} \}
\]
\[
I \setminus B \cdot (S \cdot B)^* \]

This rule is however a little too weak for our purposes. If we take a look at the example that we presented at the beginning of this section:

\[
\text{while } \{ C \mid C \neq 0 \land C \neq 1 \} \text{ do } [\text{\[C'} = \text{\[C'} - 2]\]
\]

we want to be able to use the fact that this program does not enter an infinite loop if its input is a non-negative integer. We therefore generalise the above theorem, adding an input restriction:

\[
(\text{while } B \text{ do } S) \cdot A
\]
\[
= \{ \bullet S \cdot B \cdot A \text{ is well-founded and } \bullet S \cdot B \in A \rightarrow \leftarrow A \}
\]
\[
I \setminus B \cdot (S \cdot B)^* \cdot A
\]

So if we restrict the inputs to elements of some set \( A \), we only need to prove that \( S \cdot B \cdot A \) is well-founded and that \( S \cdot B \) is such that we stay within \( A \). To prove this theorem, we prove the slightly more general (a monotonic total function \( f \) has a single fixed point exactly when its least fixed point equals its greatest fixed point)

\[
\nu(T \cup X \cdot R \mid| X) \cdot A
\]
\[
= \{ \bullet R \cdot A \text{ is well-founded and } \bullet R \in A \rightarrow \leftarrow A \}
\]
\[
\mu(T \cup X \cdot R \mid| X) \cdot A
\]

\textbf{proof}

The proof is a slight variation on a proof in [17]. The addition of the input constraint required some small modifications.

\[
\nu(T \cup X \cdot R \mid| X) \cdot A \equiv \mu(T \cup X \cdot R \mid| X) \cdot A
\]
\[
\equiv \{ \nu f \supseteq \mu f \text{ for every } f \in \text{PL} \leftarrow \text{PL, monotonicity of } \cdot \}
\]
\[
\nu(T \cup X \cdot R \mid| X) \cdot A \subseteq \mu(T \cup X \cdot R \mid| X) \cdot A
\]
\[
\equiv \{ U \cdot A \subseteq V \cdot A \equiv U \cdot A \subseteq V \}
\]
\[
\nu(T \cup X \cdot R \mid| X) \cdot A \subseteq \mu(T \cup X \cdot R \mid| X)
\]
\[
\equiv \{ \text{definition of } \mu \}
\]
\[
\nu(T \cup X \cdot R \mid| X) \cdot A \subseteq \bigcap \{ Y \mid T \cup Y \cdot R \subseteq Y \mid Y \}
\]
\[
\equiv \{ \text{point-free definition of } \bigcap \}
\]
\[
\forall(Y \mid T \cup Y \cdot R \subseteq Y \mid Y) \nu(T \cup X \cdot R \mid| X) \cdot A \subseteq Y
\]
\[
\equiv \{ \text{see below} \}
\]
\[
\nu(X \cdot R \cdot A \mid| X) = \emptyset \land R \in A \rightarrow \leftarrow A
\]

The last step in this proof is justified by the following calculation:
\[ \nu(T \cup X \cdot R \parallel X) \cdot A \subseteq Y \]
\[ \equiv \quad \{ U \cdot A \subseteq V \equiv U \subseteq V \cup \Pi \cdot (I \cdot \neg A) \} \]
\[ \nu(T \cup X \cdot R \parallel X) \subseteq Y \cup \Pi \cdot (I \cdot \neg A) \]
\[ \equiv \quad \{ \nu(X \cdot R \cdot A \parallel X) = \emptyset, \emptyset \text{ is unit of } \cup \} \]
\[ \nu(T \cup X \cdot R \parallel X) \subseteq Y \cup \Pi \cdot (I \cdot \neg A) \cup \nu(X \cdot R \cdot A \parallel X) \]
\[ \equiv \quad \{ \text{fusion with the following instantiations:} \}
\[ G = (Y \cup \Pi \cdot (I \cdot \neg A) \cup X \parallel X), \text{ distributes over } \cap \]
\[ H = (X \cdot R \cdot A \parallel X) \]
\[ K = (T \cup X \cdot R \parallel X) \}
\[ \forall(X \parallel T \cup (Y \cup \Pi \cdot (I \cdot \neg A) \cup X) \cdot R \subseteq Y \cup \Pi \cdot (I \cdot \neg A) \cup X \cdot R \cdot A) \]
\[ \equiv \quad \{ \text{distribution of } \cdot \text{ over } \cup \} \]
\[ \forall(X \parallel T \cup Y \cdot R \cup \Pi \cdot (I \cdot \neg A) \cdot R \cup X \cdot R \subseteq Y \cup \Pi \cdot (I \cdot \neg A) \cup X \cdot R \cdot A) \]
\[ \equiv \quad \{ U \subseteq V \cup \Pi \cdot (I \cdot \neg A) \equiv U \cdot A \subseteq V \} \]
\[ \forall(X \parallel (T \cup Y \cdot R \cup \Pi \cdot (I \cdot \neg A) \cdot R \cup X \cdot R) \cdot A \subseteq Y \cup X \cdot R \cdot A) \]
\[ \equiv \quad \{ \text{distribution of } \cdot \text{ over } \cup \} \]
\[ \forall(X \parallel T \cdot A \cup Y \cdot R \cdot A \cup \Pi \cdot (I \cdot \neg A) \cdot R \cdot A \subseteq Y \cup X \cdot R \cdot A) \]
\[ \equiv \quad \{ (\equiv): \text{ take } X = \emptyset, (\Rightarrow): \text{ monotonicity of } \cup \} \]
\[ T \cdot A \cup Y \cdot R \cdot A \cup \Pi \cdot (I \cdot \neg A) \cdot R \cdot A \subseteq Y \]
\[ \equiv \quad \{ \bullet R \in A \rightarrow \emptyset A, \text{ then } (I \cdot \neg A) \cdot R \cdot A = \emptyset \} \]
\[ T \cdot A \cup Y \cdot R \cdot A \cup \Pi \cdot \emptyset \subseteq Y \]
\[ \equiv \quad \{ \emptyset \text{ is zero of } \cdot, \emptyset \text{ is unit of } \cup \} \]
\[ T \cdot A \cup Y \cdot R \cdot A \subseteq Y \]
\[ \equiv \quad \{ \text{distribution of } \cdot \text{ over } \cup \} \]
\[ (T \cup Y \cdot R) \cdot A \subseteq Y \]
\[ \equiv \quad \{ U \cdot A \subseteq U, \text{ transitivity of } \subseteq \} \]
\[ T \cup Y \cdot R \subseteq Y \]

If we use the theorem about the ‘while’ with the input restriction and write out the partial-correctness rules of the constituent components, we arrive at the ‘well-known’ rule for the ‘while’:

\[ R \text{ while } B \text{ do } S(P) \]
\[ \equiv \quad \{ \bullet S ; B \cdot (P) \text{ is well-founded and } \bullet S ; B \in P, \quad \rightarrow \quad P \} \]
\[ R \cdot (I \cdot B) \cdot (S \cdot B)^{\ast}(P) \]
\[ \equiv \quad \{ \text{third partial-correctness rule for } \cdot \} \]
\[ ((I \cdot B) \cdot R)(S \cdot B)^{\ast}(P) \]
\[ \equiv \quad \{ \text{partial-correctness rule for } ^{\ast} \} \]
\[ \exists(X \parallel (I \cdot B) \cdot R \supseteq X \land X) S \cdot B(X \wedge X \supseteq P) \]
\[ \equiv \quad \{ \text{second partial-correctness rule for } \cdot \} \]
\[ \exists(X \parallel (I \cdot B) \cdot R \supseteq X \land X) S(B \cdot X \land X \supseteq P) \]
\[ \equiv \quad \{ \text{definition of } \cdot \} \]
\[ \exists(X \parallel R \supseteq (I \cdot B) \cdot X \land X) S(B \cdot X \land X \supseteq P) \]
7.3.10 Assignment

In case a specification is functional and total on the range of the pre, we can use the following rule that we suggestively call the partial-correctness rule for the assignment:

\[ (R) S(P) \equiv \{ \bullet S \in I \leftarrow P \} \]

\[ S^c \circ R \supseteq P \]

\[ S \circ P \subseteq R \]

\[ P \subseteq S^c \circ R \]

\[ \{ \text{definition of } \cdots \} \]

\[ \{ \text{shunting, } \bullet S \in I \leftarrow P \} \]

\[ P \subseteq S^c \circ R \]

7.3.11 Example

In this section we present an example to demonstrate the partial-correctness rules. The specification \( \text{more} \) outputs a real number smaller than 1 and larger than the absolute value of the input if its input is a real number between \(-1 \) and \( 1 \), unequal to 0:

\[ \text{more} = \text{more}_{\text{post}} / \text{more}_{\text{pre}} \]

\[ \text{more}_{\text{post}} = [ \dot{\textit{C}} \in \mathbb{R} \land |\dot{\textit{C}}| < \dot{\textit{C}} \land \dot{\textit{C}} < 1 ] \]

\[ \text{more}_{\text{pre}} = [ \dot{\textit{C}} \in \mathbb{R} \land -1 < \dot{\textit{C}} \land \dot{\textit{C}} \neq 0 \land \dot{\textit{C}} < 1 ] \]

We now prove that \( \text{more} \) is refined by \( \text{more}' \), defined by

\[ \text{more}' = \text{sqrt} \circ \text{abs} \]

\[ \text{sqrt} = \text{sqrt}_{\text{post}} / \text{sqrt}_{\text{pre}} \]

\[ \text{sqrt}_{\text{post}} = [ \dot{\textit{C}} = \sqrt{\textit{C}} ] \]

\[ \text{sqrt}_{\text{pre}} = [ \dot{\textit{C}} \in \mathbb{R} \land \dot{\textit{C}} \geq 0 ] \]

\[ \text{abs} = 1 \cdot [ \dot{\textit{C}} \geq 0 ] \cup [ \dot{\textit{C}} = -\dot{\textit{C}} ] \cdot [ \dot{\textit{C}} \leq 0 ] \]

We thus have to prove \( \text{more}' \subseteq \text{more}_{\text{post}} / \text{more}_{\text{pre}} \), or writing it with the partial-correctness operator:

\[ \text{more}_{\text{post}} \text{more}'( \text{more}_{\text{pre}} ) \]

Using the first partial-correctness rule for the composition, we have to invent a
relation $more_{int}$ such that

\[
more_{post} \overline{\text{sqt}}(more_{int})
\]

\[
more_{int} \overline{\text{abs}}(more_{pre})
\]

We start with $more_{post} \overline{\text{sqt}}(more_{int})$. Writing out the definition of $\text{sqt}$, the partial-correctness rule for the ‘over’ leaves us with

\[
more_{int} \subseteq \text{sqt}_{pre} \land more_{post} \text{sqt}_{post}(more_{int})
\]

which can be rewritten into the following point-wise form:

\[
\forall (x_0, x_1, x_2 \mid x_1 (more_{int}) x_2 \\
\quad \text{\mid } x_1 \in \text{sqt}_{pre} \land (x_0 (\text{sqt}_{post}) x_1 \Rightarrow x_0 (more_{post}) x_2))
\]

The following calculation shows a possible choice for $more_{int}$:

\[
x_1 \in \text{sqt}_{pre} \land (x_0 (\text{sqt}_{post}) x_1 \Rightarrow x_0 (\text{more}_{post}) x_2)
\]

\[
\equiv \{\text{definition of \text{sqt}_{pre}}\}
\]

\[
x_1 \in [\{C \in \mathbb{R} \land C \geq 0\} \land (x_0 (\text{sqt}_{post}) x_1 \Rightarrow x_0 (\text{more}_{post}) x_2)]
\]

\[
\equiv \{\text{sections 6.7 and 6.4}\}
\]

\[
x_1 \in \mathbb{R} \land x_1 \geq 0 \land (x_0 (\text{sqt}_{post}) x_1 \Rightarrow x_0 (\text{more}_{post}) x_2)
\]

\[
\equiv \{\text{definition of \text{sqt}_{post}}\}
\]

\[
x_1 \in \mathbb{R} \land x_1 \geq 0 \land (x_0 (\text{sqrt} C) x_1 \Rightarrow x_0 (\text{more}_{post}) x_2)
\]

\[
\equiv \{\text{sections 6.6, 6.4 and 6.8.2}\}
\]

\[
x_1 \in \mathbb{R} \land x_1 \geq 0 \land (x_0 = \sqrt{x_1} \Rightarrow x_0 (\text{more}_{post}) x_2)
\]

\[
\equiv \{\text{definition of \text{more}_{post}}\}
\]

\[
x_1 \in \mathbb{R} \land x_1 \geq 0 \land (x_0 = \sqrt{x_1} \Rightarrow x_0 (\{\overset{\ast}{C} \in \mathbb{R} \land |\overset{\ast}{C}| < \overset{\ast}{C} \land \overset{\ast}{C} < 1\}))
\]

\[
\equiv \{\text{sections 6.6, 6.4 and 6.8.2} \ast x_2 \in \mathbb{R}\}
\]

\[
x_1 \in \mathbb{R} \land x_1 \geq 0 \land (x_0 = \sqrt{x_1} \Rightarrow (x_0 \in \mathbb{R} \land |x_2| < x_0 \land x_0 < 1))
\]

\[
\equiv \{(x_1 \in \mathbb{R} \land x_1 \geq 0 \land x_0 = \sqrt{x_1}) \Rightarrow x_0 \in \mathbb{R}\}
\]

\[
x_1 \in \mathbb{R} \land x_1 \geq 0 \land (x_0 = \sqrt{x_1} \Rightarrow (|x_2| < x_0 \land x_0 < 1))
\]

\[
\equiv \{\text{Leibniz}\}
\]

\[
x_1 \in \mathbb{R} \land x_1 \geq 0 \land (x_0 = \sqrt{x_1} \Rightarrow (|x_2| < x_0 \land \sqrt{x_1} < 1))
\]

\[
\equiv \{\ast x_1 = |x_2| \text{ together with previous } \ast, \text{ then } x_1 \in \mathbb{R} \land x_1 \geq 0\}
\]

\[
x_0 = \sqrt{x_1} \Rightarrow (x_1 < \sqrt{x_1} \land \sqrt{x_1} < 1)
\]

\[
\equiv \{\ast x_1 \neq 0 \land x_1 < 1 \text{ together with previous } \ast\}'s\}
\]

true
Notice that if we want to do this calculation at the level of expressions rather than at the (not completely formalised) level of formulas in the mathematical language that we use, we need expressions that can talk about three points instead of only two ($\hat{C}$ and $\check{C}$). To be able to do complicated calculations like the one above, the expression formalism should also be developed much more than it is now. We leave that for future research.

Putting it all together, we found the following restrictions on $x_1$ and $x_2$:

$$x_2 \in \mathbb{R} \land x_1 = |x_2| \land x_1 \neq 0 \land x_1 < 1$$

which means the following choice for $more_{int}$:

$$more_{int} = [\hat{C} \in \mathbb{R} \land \hat{C} = |\hat{C}| \land \hat{C} \neq 0 \land \hat{C} < 1]$$

We now arrive at the other part of the composition: $more_{int} \text{abs}(more_{pre})$. Writing out the definition of $\text{abs}$:

$$more_{int} \text{I}[C \geq 0] \cup [\hat{C} = -\hat{C}] \cdot [C \leq 0] \{more_{pre}\}$$

the partial-correctness rule for the union leaves us with the following two proof obligations:

$$more_{int} \text{I}[C \geq 0] \{more_{pre}\}$$
$$more_{int} [\hat{C} = -\hat{C}] \cdot [C \leq 0] \{more_{pre}\}$$

Writing out the definition of $more_{pre}$:

$$more_{int} \text{I}[C \geq 0] \{more_{pre}\}$$
$$more_{int} [\hat{C} = -\hat{C}] \cdot [C \leq 0] \{more_{pre}\}$$

we apply the second partial-correctness rule for the composition and use the fact that $\{E_0 \cap E_1\}$, $\{E_0\}$, and $\{E_1\}$ are equal:

$$more_{int} \text{I}[C \geq 0] \{\{C \in \mathbb{R} \land -1 < C \land C \neq 0 \land C < 1\}\}$$
$$more_{int} [\hat{C} = -\hat{C}] \cdot [C \leq 0] \{\{C \in \mathbb{R} \land -1 < C \land C \neq 0 \land C < 1\}\}$$

We start with the first of these two proof obligations. The partial-correctness rule for $I$ leaves us with:

$$more_{int} \supseteq \{C \geq 0 \land C \in \mathbb{R} \land -1 < C \land C \neq 0 \land C < 1\}$$

This can be proved as follows:
\[ \text{more}_{\text{int}} = \{\text{definition of more}_{\text{int}}\} \]
\[ [\hat{C} \in \mathbb{R} \land \hat{C} = |\hat{C}| \land \hat{C} \neq 0 \land \hat{C} < 1] \]

\[ \geq \{\text{section 6.6 and theorem lifting (section 6.4)}\} \]
\[ [\hat{C} \geq 0 \land \hat{C} \in \mathbb{R} \land -1 < \hat{C} \land \hat{C} < 0 \land \hat{C} < 1] \]

For the second one,
\[ \text{more}_{\text{int}} [\hat{C} = -\hat{C}] [\{\mathbb{C} \leq 0 \land \mathbb{C} \in \mathbb{R} \land -1 < \hat{C} \land \hat{C} \neq 0 \land \hat{C} < 1\} \]

we use the partial-correctness rule for the assignment (notice that \(-C \in I \leftrightarrow \mathbb{R}\)) and calculate:
\[ [\hat{C} = -\hat{C}] [\{\mathbb{C} \leq 0 \land \mathbb{C} \in \mathbb{R} \land -1 < \hat{C} \land \hat{C} \neq 0 \land \hat{C} < 1\} \]
\[ = [\mathbb{C} \geq 0 \land \mathbb{C} \in \mathbb{R} \land -1 < \hat{C} \land \hat{C} < 0 \land \hat{C} < 1] \]

\[ \geq \{\text{section 6.8.4}\} \]
\[ \{\mathbb{C} \geq 0 \land \mathbb{C} \in \mathbb{R} \land -1 < \hat{C} \land \hat{C} \neq 0 \land \hat{C} < 1\} \]

7.4 Type, declarative and operational

It is often convenient to split up a post into a post-type, a post-condition and an action clause, and a pre into a pre-type and a pre-condition. Pre-types and post-types are sets that are thought of to be ‘simple’ like \( \mathbb{Z} \) or \( \mathbb{R} \). A pre-condition is a set that is described by means of some kind of unary predicate-logic formula and a post-condition is a relation that is described by means of some kind of binary predicate-logic formula. The action clause deserves some more explanation. In most theories about sequential programming, the pre-condition and post-condition specify the ‘desired behaviour’ of a program in a declarative way. Then, separate
from the pre-condition and post-condition, there is code that implements the behaviour that is specified by them in an operational way. The operational code is however often more readable than the declarative specification. This does not really stimulate the use of pre-conditions and post-conditions. The fact that declarative specifications are often much harder to read than the corresponding operational code is not that strange because many specifications of everyday life have an operational nature. A pragmatic specification language, like ISpec, therefore combines both declarative and operational descriptions, resulting in compact and understandable specifications. Hence the action clause.

An example of a specification mod2 for the ‘modulo 2’ of a ‘two’s complement byte’ is

\[
\text{mod2} = \text{post} / \text{pre}
\]

\[
\text{post} = \text{postType} \cap \text{postCondition} \cap \text{actionClause}
\]

\[
\text{pre} = \text{preType} \cap \text{preCondition}
\]

\[
\text{postType} = \mathbb{Z}
\]

\[
\text{preType} = \mathbb{Z}
\]

\[
\text{postCondition} = \llbracket 0 \leq \hat{C} \wedge \hat{C} < 2 \rrbracket
\]

\[
\text{preCondition} = \llbracket -128 \leq C \wedge C < 128 \rrbracket
\]

\[
\text{actionClause} = (\llbracket \hat{C} = \hat{C} - 2 \rrbracket \cup \llbracket \hat{C} = \hat{C} + 2 \rrbracket)^* \]

The pre-type and pre-condition together say that the input should be a two’s complement byte. If this is the case, then the post-type and post-condition together tell us that the output is either 0 or 1 and the action clause tells us that 2 is added/subtracted several times to/from the input. This specification thus clearly describes the modulo 2 of a two’s complement byte.

We now prove that \( \text{mod2}' \), defined by

\[
\text{mod2}' = \text{while } \llbracket C < 0 \vee 2 \leq C \rrbracket \text{ do}
\]

\[
\text{if } \llbracket C \geq 0 \rrbracket
\]

\[
\text{then } \llbracket \hat{C} = \hat{C} - 2 \rrbracket
\]

\[
\text{else } \llbracket \hat{C} = \hat{C} + 2 \rrbracket
\]

is a refinement of \( \text{mod2} \).

An advantage of separating a post into type, declarative and operational aspects, is that partial-correctness proofs can also be separated into these aspects. This is a trivial consequence of the conjunctivity of the post (see section 7.3.1):
\[ \text{mod}' \text{mod}' \{ \text{post} \} \equiv \{ \text{definition of post} \} \]
\[ (\text{postType} \land \text{pre} \land \text{actionClause} ) \text{mod}' \{ \text{pre} \} \]

Before we continue, we first introduce some shorthands:

\[ \text{mod}' = \text{while WG do IF} \]
\[ IF = \text{if IG then } S2 \text{ else } A2 \]
\[ WG = \{ \mathcal{C} < 0 \lor 2 \leq \mathcal{C} \} \]
\[ IG = \{ \mathcal{C} \geq 0 \} \]
\[ S2 = \{ \mathcal{C} = \mathcal{C} - 2 \} \]
\[ A2 = \{ \mathcal{C} = \mathcal{C} + 2 \} \]

We now show that we are allowed to use the least–fixed-point interpretation of the ‘while’. Section 7.3.9 tells us that this follows from the fact that \( IF \circ WG \circ Z \) is well-founded and \( IF \circ WG \in Z \circ Z \). We have not introduced practical theory for proving well-foundedness, but assume that the reader believes us when we say that \( IF \circ WG \circ Z \) is well-founded. The fact that \( IF \circ WG \in Z \circ Z \), can be proved by a simple type proof (the cylindric-type rules can be found in section 4.1):

\[ IF \circ WG \in Z \circ Z \]
\[ \iff \{ \text{cylindric-type rule for } \circ \} \]
\[ IF \in Z \circ Z \land WG \in Z \circ Z \]
\[ \iff \{ \text{definition of } IF, WG \in Z \circ Z \text{ follows from the cylindric-type rules for I and } \subseteq \} \]
\[ \text{if IG then } S2 \text{ else } A2 \in Z \circ Z \]
\[ \iff \{ \text{definition of if then else } \} \]
\[ S2 \cdot IG \cup A2 \cdot (I \backslash IG) \in Z \circ Z \]
\[ \iff \{ \text{cylindric-type rule for } \cup \} \]
\[ S2 \cdot IG \in Z \circ Z \land A2 \cdot (I \backslash IG) \in Z \circ Z \]
\[ \iff \{ \text{cylindric-type rule for } \cup \} \]
\[ S2 \in Z \circ Z \land IG \in Z \circ Z \land A2 \in Z \circ Z \land I \backslash IG \in Z \circ Z \]
\[ \iff \{ IG, I \backslash IG \in Z \circ Z \text{ follows from the cylindric-type rules for I and } \subseteq \} \]
\[ S2 \in Z \circ Z \land A2 \in Z \circ Z \]
\[ \iff \{ A \leftarrow B \subseteq A \circ Z \} \]
\[ S2 \in Z \leftrightarrow Z \land A2 \in Z \leftrightarrow Z \]
\[ \iff \{ \text{chapter 6} \}
\[ \text{true} \]

We thus know that we can use the least–fixed-point interpretation of the ‘while’ if
the inputs are of type \( \mathbb{Z} \). We now prove

\[
postType \Pi \mod' (preType \cap preCondition)
\]

We can simply stay on the type level and do not need the pre-condition. Using the fact that

\[
A \Pi S(B) \equiv S \in A \rightarrow B
\]

we again have a simple type proof:

\[
\begin{align*}
\mod' & \in \mathbb{Z} \rightarrow \mathbb{Z} \\
& \equiv \{ \text{definition of } \mod' \}\ 
\text{while } WG \text{ do } IF & \in \mathbb{Z} \rightarrow \mathbb{Z} \\
& \equiv \{ \text{least-fixed-point interpretation of the 'while', see above text} \} \\
I \setminus WG \circ (IF \circ WG)^* & \in \mathbb{Z} \rightarrow \mathbb{Z} \\
& \leftarrow \{ \text{cylindric-type rule for } \circ, I \text{ and } \subseteq \} \\
(IF \circ WG)^* & \in \mathbb{Z} \rightarrow \mathbb{Z} \\
& \leftarrow \{ \text{cylindric-type rule for } ^* \} \\
IF \circ WG & \in \mathbb{Z} \rightarrow \mathbb{Z} \\
& \equiv \{ \text{see above proof} \} \\
& \text{true}
\end{align*}
\]

Now for the post-condition. Again we only need the fact that the input is of type \( \mathbb{Z} \):
\[\text{postCondition} \mod 2' (\mathbb{Z})\]
\[\equiv\{\text{definition of } \mod 2'\}\]
\[\text{postCondition} \text{while } WG \text{ do } IF (\mathbb{Z})\]
\[\equiv\{\text{least fixed point interpretation of the 'while'}\}\]
\[\text{postCondition} I \setminus WG \circ (IF \ast WG) \ast (\mathbb{Z})\]
\[\equiv\{\text{definition of } \setminus \}\]
\[I \setminus WG \circ (IF \ast WG) \ast \mathbb{Z} \subseteq \text{postCondition}\]
\[\equiv\{IF \ast WG \ast \mathbb{Z} \in \text{postCondition}, \text{see previous proof}\}\]
\[\text{postCondition} \leftarrow \{R \ast S \subseteq T \leftarrow R \ast \Pi \subseteq T\}\]
\[I \setminus WG \circ \mathbb{Z} \ast \Pi \subseteq \text{postCondition}\]
\[\equiv\{\text{definition of } WG \text{ and postCondition}\}\]
\[I \setminus \{[\mathbb{C} | \mathbb{C} < 0 \lor 2 \leq \mathbb{C}] \ast \mathbb{Z} \ast \Pi \subseteq [0 \leq \hat{\mathbb{C}} \land \hat{\mathbb{C}} < 2]\}\]
\[\equiv\{\text{similar to previous calculation}\}\]
\[IF \ast \subseteq \text{actionClause}\]
\[\equiv\{\text{definition of } IF\}\]
\[\text{if } IG \text{ then } S2 \text{ else } A2 \ast \subseteq \text{actionClause}\]
\[\equiv\{\text{definition of } \text{if then else}\}\]
\[S2 \ast IG \cup A2 \ast (I \setminus IG) \ast \subseteq \text{actionClause}\]
\[\equiv\{IG \text{ and } I \setminus IG \text{ are sets}\}\]
\[S2 \ast A2 \ast \subseteq \text{actionClause}\]
\[\equiv\{\text{definition of } S2, A2 \text{ and actionClause}\}\]
\[([\hat{\mathbb{C}} = \hat{\mathbb{C}} - 2] \cup [\hat{\mathbb{C}} = \hat{\mathbb{C}} + 2]) \ast \subseteq ([\hat{\mathbb{C}} = \hat{\mathbb{C}} - 2] \cup [\hat{\mathbb{C}} = \hat{\mathbb{C}} + 2])^*\]
\[\equiv\{\text{reflexivity of } \subseteq\}\]
\[true\]

Notice that the only thing we used from the 'while' is its guard.

The last thing we need to prove is the action clause:

\[\text{actionClause} \mod 2' (\text{preType})\]
\[\equiv\{\text{similar to previous calculation}\}\]
\[I \setminus WG \circ (IF \ast WG) \ast \text{preType} \subseteq \text{actionClause}\]
\[\equiv\{I \setminus WG, WG \text{ and preType are sets}\}\]
\[IF \ast \subseteq \text{actionClause}\]
\[\equiv\{\text{definition of } IF\}\]
\[\text{if } IG \text{ then } S2 \text{ else } A2 \ast \subseteq \text{actionClause}\]
\[\equiv\{\text{definition of } \text{if then else}\}\]
\[S2 \ast IG \cup A2 \ast (I \setminus IG) \ast \subseteq \text{actionClause}\]
\[\equiv\{IG \text{ and } I \setminus IG \text{ are sets}\}\]
\[S2 \ast A2 \ast \subseteq \text{actionClause}\]
\[\equiv\{\text{definition of } S2, A2 \text{ and actionClause}\}\]
\[([\hat{\mathbb{C}} = \hat{\mathbb{C}} - 2] \cup [\hat{\mathbb{C}} = \hat{\mathbb{C}} + 2]) \ast \subseteq ([\hat{\mathbb{C}} = \hat{\mathbb{C}} - 2] \cup [\hat{\mathbb{C}} = \hat{\mathbb{C}} + 2])^*\]
\[\equiv\{\text{reflexivity of } \subseteq\}\]
\[true\]
Notice that we could do this proof completely at the operational level. This clean separation of type, declarative and operational reasoning is not always possible, but should in our opinion be strived for as much as possible in order to keep refinement comprehensible.

7.5 Structural hierarchy

An important concept in specification/programming languages is structural hierarchy in specifications/programs. In traditional sequential programming, a program consists of a collection of procedures. The structural hierarchy in classical object-oriented programming has one layer extra. There a program consists of classes which again consist of methods. Moreover, most modern object-oriented languages allow to put classes into packages and even allow to put packages into packages, enabling the construction of arbitrary structural hierarchies. Some object-oriented languages allow to put classes into classes. The class inside another class is then called an inner class of that class.

The pack operators that we introduced in chapter 5 enable the hierarchical structuring of specifications. We use the pack operator that corresponds to the conjoint sum because that one has the properties that we find desirable. This includes the fact that it resembles the disjoint sum, making it ‘more familiar’ than other candidates. We therefore define the pack $\triangleright$ by

\[ \triangleright = \emptyset \cup FUN \cup NMPP \cup DEQ \]

We illustrate the use of $\triangleright$ by means of a simple specification $MATH$ that contains a class with name $\text{Crease}$ and body $\text{Crease}1$ with a method named $\text{inc}$ whose body $\text{inc1}$ increases its input by 1 and a method named $\text{dec}$ whose body $\text{dec1}$ decreases its input by 1. Specification $MATH$ also contains a class $\text{Three}$ with a method $\text{direct}$ that outputs 3 on any input, calculated by subtracting 1 from 4, using method $\text{dec}$ of class $\text{Crease}$. Class $\text{Three}$ also has a method $\text{indirect}$ that outputs 3 on any input, but calculates this by subtracting 1 from the output of procedure $\text{four}$, using method $\text{dec}$ of class $\text{Crease}$. The procedure $\text{four}$ again calculates its output by adding 1 to the output of method $\text{direct}$ of class $\text{Three}$, using method $\text{inc}$ of class $\text{Crease}$:

\[
\begin{align*}
MATH & = \text{Crease}1 \triangleright \text{Crease} \cap \text{Three} \triangleright \text{Three} \cap \text{four} \triangleright \text{four} \\
\text{Crease1} & = \text{inc1} \triangleright \text{inc} \cap \text{dec1} \triangleright \text{dec} \\
\text{Three} & = \text{direct3} \triangleright \text{direct} \cap \text{indirect3} \triangleright \text{indirect} \\
\text{inc1} & = \left[ \hat{\mathcal{C}} = \hat{\mathcal{C}} + 1 \right] \\
\text{dec1} & = \left[ \hat{\mathcal{C}} = \hat{\mathcal{C}} - 1 \right] \\
\text{direct3} & = \left( MATH \triangleright \text{Crease} \right) \triangleright \text{dec} \cap \left[ \hat{\mathcal{C}} = 4 \right] \\
\text{indirect3} & = \left( MATH \triangleright \text{Crease} \right) \triangleright \text{dec} \cap MATH \triangleright \text{four} \\
\text{four} & = \left( MATH \triangleright \text{Crease} \right) \triangleright \text{inc} \cap \left( MATH \triangleright \text{Three} \right) \triangleright \text{direct}
\end{align*}
\]

The following equation illustrates the behaviour of $MATH$:
\[ ((MATH \triangleright Three) \triangleright indirect).z = 3 \]

**proof**

\[ ((MATH \triangleright Three) \triangleright indirect).z \]
\[ = \quad \{ MATH \triangleright Three = Three, see below \} \]
\[ (Three \triangleright indirect).z \]
\[ = \quad \{ Three \triangleright indirect = indirect3, similar \} \]
\[ indirect3.z \]
\[ = \quad \{ definition of indirect3 \}\]
\[ ((MATH \triangleright Crease) \triangleright dec \ast MATH \triangleright four).z \]
\[ = \quad \{ left to the reader \}\]
\[ (dec1 \ast inc1 \ast dec1 \ast [\overline{C} = 4]).z \]
\[ = \quad \{ left to the reader \}\]
\[ 3 \]

\[ MATH \triangleright Three \]
\[ = \quad \{ definition of MATH \}\]
\[ (Crease1 \triangleright Crease \odot Three < Three \odot four < four) \triangleright Three \]
\[ = \quad \{ distribution of (\triangleright i) over \odot \}\]
\[ (Crease1 \triangleright Crease \triangleright Three) \odot \]
\[ (Three < Three \triangleright Three) \odot \]
\[ (four < four \triangleright Three) \]
\[ = \quad \{ cancellation \}\]
\[ \Pi \odot Three \odot \Pi \]
\[ = \quad \{ \Pi is unit of \odot \}\]
\[ Three \]

In the following example we show a way to prove refinement betwe en two structural-hierarchical specifications. The specification *Crease*, consisting of two procedures *inc* and *dec* where *inc* outputs an integer that is larger than its integer input and *dec* outputs an integer that is smaller than its integer input:

\[
\begin{align*}
\text{Crease} &= \quad inc \ast inc \odot dec \ast dec \\
inc &= \quad \text{Z} \cdot \overline{C} > \overline{C} \cdot \text{Z} \\
\text{dec} &= \quad \text{Z} \cdot \overline{C} < \overline{C} \cdot \text{Z}
\end{align*}
\]

is refined by the specification *Crease'*, that consists of three procedures *inc*, *dec* and *chg* where *inc* adds 1 to its integer input, *dec* subtracts 1 from its integer input and *chg* outputs an integer that is different from its integer input:
\[ \text{Crease} = \text{inc}' \triangleright \text{dec} \cap \text{chg}' \triangleright \text{chg} \]
\[ \text{inc}' = [\hat{C} = \hat{C}+1] \cdot \mathbb{Z} \]
\[ \text{dec}' = [\hat{C} = \hat{C}-1] \cdot \mathbb{Z} \]
\[ \text{chg}' = \mathbb{Z} \cdot [\hat{C} \neq \hat{C}] \cdot \mathbb{Z} \]

**proof**

\[ \text{Crease}' \subseteq \text{Crease} \]
\[ \equiv \{ \text{definition of Crease}' \text{ and Crease} \} \]
\[ \text{inc}' \triangleright \text{inc} \cap \text{dec}' \triangleright \text{dec} \triangleright \text{chg}' \triangleright \text{chg} \subseteq \text{inc} \cap \text{dec} \triangleright \text{dec} \]
\[ \Leftarrow \{ \text{R} \cap \text{S} \subseteq \text{S} \} \]
\[ \text{inc}' \triangleright \text{inc} \cap \text{dec}' \triangleright \text{dec} \subseteq \text{inc} \cap \text{dec} \]
\[ \Leftarrow \{ \text{monotonicity of } \cap \} \]
\[ \text{inc}' \triangleright \text{inc} \wedge \text{dec}' \triangleright \text{dec} \subseteq \text{dec} \]
\[ \equiv \{ \text{see below} \} \]
\[ \text{true} \]

\[ \text{inc}' \subseteq \text{inc} \]
\[ \equiv \{ \text{definition of inc}' \text{ and inc} \} \]
\[ [\hat{C} = \hat{C}+1] \cdot \mathbb{Z} \subseteq \mathbb{Z} \cdot [\hat{C} > \hat{C}] \cdot \mathbb{Z} \]
\[ \equiv \{ \text{x}_1 \in \mathbb{Z} \wedge x_0 = x_1+1 \Rightarrow x_0, x_1 \in \mathbb{Z} \wedge x_0 > x_1 \} \]
\[ \text{true} \]

Similar for \( \text{dec}' \subseteq \text{dec} \).

This example shows the anti-monotonicity of the conjoint sum in action. Because, in contrast to the disjoint sum, adding something to a conjoint sum is done by means of intersection instead of union, we can simply use inclusion for refinement and do not need to complicate things with functions that project away new procedures.

In the second and third step of the proof it seems as if we are strengthening our proof obligation. The following calculation however shows that this is not the case:
\[
\begin{align*}
\text{inc'} & \triangleleft \text{inc} \cap \text{dec'} \triangleleft \text{dec} \cap \text{chg'} \triangleleft \text{chg} \\
\Rightarrow \quad & \{\text{monotonicity of } (\sqsubseteq_i)\} \\
(\text{inc'} \triangleleft \text{inc} \cap \text{dec'} \triangleleft \text{dec} \cap \text{chg'}) & \subseteq (\text{inc'} \triangleleft \text{inc} \cap \text{dec'} \triangleleft \text{dec} \cap \text{chg'}) \land \\
(\text{inc'} \triangleleft \text{inc} \cap \text{dec'} \triangleleft \text{dec} \cap \text{chg'}) & \subseteq (\text{inc'} \triangleleft \text{inc} \cap \text{dec'} \triangleleft \text{dec}) \\
\equiv \quad & \{\text{distribution of } (\sqsubseteq_i) \text{ over } \cap \text{ and cancellation}\} \\
\text{inc'} \land \text{II} \land \text{II} & \subseteq \text{inc} \land \text{II} \land \text{II} \land \text{dec} \land \text{dec'} \subseteq \text{II} \land \text{dec} \\
\equiv \quad & \{\text{II is unit of } \cap\} \\
\text{inc'} & \subseteq \text{inc} \land \text{dec'} \subseteq \text{dec}
\end{align*}
\]

### 7.6 Components

With the increasing complexity of tasks that have to be performed by software and the increasing development effort that has to be spent in order to build such software, the wish came that software could be constructed from components that can be developed independently from each other. As a result, a new paradigm emerged, called component-oriented programming.

A prime ingredient of component-oriented programming is component composition. Suppose that several teams are refining distinct parts of a specification. Each team is responsible for a correct refinement of its part, without knowing how the other teams refine their parts. The component composition of all these refinements should now result in a specification that refines the original specification.

The model for specifications of this chapter (relations) provides a simple definition of components and their composition. Before giving the formal definitions, we first illustrate the main idea by means of an example.

We want two separate teams to refine the specification \(\text{Crease}\):

\[
\begin{align*}
\text{Crease} &= \text{inc'} \triangleleft \text{inc} \cap \text{dec'} \triangleleft \text{dec} \\
\text{inc} &= \mathbb{Z} \cdot [\dddot{C} > \dddot{C}] \cdot \mathbb{Z} \\
\text{dec} &= \mathbb{Z} \cdot [\dddot{C} < \dddot{C}] \cdot \mathbb{Z}
\end{align*}
\]

Team INC develops the following specification that refines procedure \(\text{inc}\):

\[
\begin{align*}
\text{Increase} &= \text{inc'} \triangleleft \text{inc} \cap \text{dec'} \triangleleft \text{dec} \\
\text{inc'} &= [\dddot{C} = \dddot{C}+1] \cdot \mathbb{Z}
\end{align*}
\]

Team DEC develops the following solution:

\[
\begin{align*}
\text{Decrease} &= \text{inc'} \triangleleft \text{inc} \cap \text{dec'} \triangleleft \text{dec} \\
\text{dec'} &= [\dddot{C} = \dddot{C}-1] \cdot \mathbb{Z}
\end{align*}
\]

The intersection of \(\text{Increase}\) and \(\text{Decrease}\), \(\text{Crease'}\), now consists of a procedure \(\text{inc}\)
with body \(\text{inc}'\) and a procedure \(\text{dec}\) with body \(\text{dec}'\). This matches the common intuition about what component composition means:

\[
\text{Crease}' = \text{inc}' \triangleleft \text{inc} \cap \text{dec}' \triangleleft \text{dec}
\]

**proof**

\[
\text{Crease}'
\]

\[
= \text{(definition of Crease')}
\]

\[
\text{Increase} \cap \text{Decrease}
\]

\[
= \text{(definition of Increase and Decrease)}
\]

\[
\text{inc}' \triangleleft \text{inc} \cap \text{dec} \triangleleft \text{dec} \cap \text{inc}' \triangleleft \text{dec}
\]

\[
= \text{(associativity and commutativity of \(\cap\))}
\]

\[
\text{inc}' \triangleleft \text{inc} \cap \text{inc} \triangleleft \text{dec} \triangleleft \text{dec} \cap \text{dec}' \triangleleft \text{dec}
\]

\[
= \text{(distribution of (\(\triangleleft\)) over \(\cap\))}
\]

\[
(\text{inc}' \cap \text{inc}) \triangleleft \text{inc} \cap (\text{dec} \cap \text{dec}') \triangleleft \text{dec}
\]

\[
= \text{(inc' \(\subseteq\) inc and dec' \(\subseteq\) dec)}
\]

\[
\text{inc}' \triangleleft \text{inc} \cap \text{dec}' \triangleleft \text{dec}
\]

Furthermore, \(\text{Crease}'\) clearly is a refinement of \(\text{Crease}\).

Separation of a specification into distinct parts is done by means of its input values. In the above example, two distinct parts of the specification are described by the two disjoint sets of input values \(\text{INC}\) and \(\text{DEC}\), defined by

\[
\text{INC} = I \rightarrow \{\text{inc}\}
\]

\[
\text{DEC} = I \rightarrow \{\text{dec}\}
\]

The following equation shows the partitioning of the specification into distinct parts:

\[
\text{Crease}' = \text{Increase} \cdot \text{INC} \cup \text{Decrease} \cdot \text{DEC} \cup \text{Crease} \cdot (I \setminus \text{INC} \cup \text{DEC})
\]

We now formalise the ideas in this example. We call a specification \(S'\) a \(Z\)-override of a specification \(S\) if \(S'\) is equal to \(S\) for inputs outside the set \(Z\). We define the \(\text{is-Z-override-of}_Z S \in \mathcal{B} \leftarrow (I \rightarrow I) \times (I \rightarrow I)\) by

\[
S' \circ_Z S \equiv S'(I \setminus Z) = S \cdot (I \setminus Z)
\]

If for disjoint sets \(Z_0\) and \(Z_1\), \(S_0\) is a \(Z_0\)-override of \(S\) and \(S_1\) is a \(Z_1\)-override of \(S\), then the component composition of \(S_0\) and \(S_1\) \((S_0 \cap S_1)\) is the specification \(S_0\) for inputs from \(Z_0\), the specification \(S_1\) for inputs from \(Z_1\) and the specification \(S\) for all other inputs:
\[ S_0 \cap S_1 = S_0 \cdot Z_0 \cup S_1 \cdot Z_1 \cup S^+(I \setminus Z_0 \cup Z_1) \]
\[ \iff \{ Z_0 \text{ and } Z_1 \text{ are disjoint sets} \} \]
\[ S_0 \circ Z_0 S \land S_1 \circ Z_1 S \]

A trivial consequence is that \( S_0 \cap S_1 \) is a \( Z_0 \cup Z_1 \)-override of \( S \) (disjointness of \( Z_0 \) and \( Z_1 \) is not needed):
\[ S_0 \cap S_1 \subseteq Z_0 \cup Z_1 S \]
\[ \iff S_0 \circ Z_0 S \land S_1 \circ Z_1 S \]

In the above example we have

\[ \text{Increase} \subseteq \text{INC} \quad \text{Crease} \]
\[ \text{Decrease} \subseteq \text{DEC} \quad \text{Crease} \]

and therefore

\[ \text{Increase} \cap \text{Decrease} \subseteq \text{INC} \cup \text{DEC} \quad \text{Crease} \]

We call a specification \( S' \) a \emph{Z-refinement} of a specification \( S \) if \( S' \) is a refinement and a \( Z \)-override of \( S \). The \emph{is-Z-refinement-of} \( \subseteq_{Z} \in \mathcal{B} \iff (I \leftarrow I) \times (I \leftarrow I) \) is defined by
\[ S' \subseteq_{Z} S \iff S' \subseteq S \land S' \circ Z S \]

For sets \( Z_0 \) and \( Z_1 \), the component composition of \( S_0 \) and \( S_1 \) is a \( Z_0 \cup Z_1 \)-refinement of \( S \) if \( S_0 \) is a \( Z_0 \)-refinement of \( S \) and \( S_1 \) is a \( Z_1 \)-refinement of \( S \):
\[ S_0 \cap S_1 \subseteq_{Z_0 \cup Z_1} S \]
\[ \iff S_0 \subseteq_{Z_0} S \land S_1 \subseteq_{Z_1} S \]

So, because the following refinements hold for our example:

\[ \text{Increase} \subseteq_{\text{INC}} \text{Crease} \]
\[ \text{Decrease} \subseteq_{\text{DEC}} \text{Crease} \]

we have

\[ \text{Increase} \cap \text{Decrease} \subseteq_{\text{INC} \cup \text{DEC}} \text{Crease} \]

A non-empty set of specifications \( S \) is called a \emph{decomposition} of a specification \( S \) if for each pair of different specifications \( S_0, S_1 \) from \( S \), there exist disjoint sets \( Z_0, Z_1 \) such that \( S_0 \) is a \( Z_0 \)-refinement of \( S \) and \( S_1 \) is a \( Z_1 \)-refinement of \( S \). Formally, for a non-empty set of specifications \( S \) we define:
\( S \) is a decomposition of \( S \)

\[
\forall \langle S_0, S_1 \rangle \mid S_0, S_1 \in S \land S_0 \neq S_1 \land \exists \langle Z_0, Z_1 \rangle \mid Z_0 \text{ and } Z_1 \text{ are disjoint sets} \land S_0 \subseteq Z_0 \land S_1 \subseteq Z_1 \rangle
\]

For any decomposition \( S \) of a specification \( S \), the component composition of all specifications in \( S \) is a refinement of \( S \). Formally, for a non-empty set of specifications \( S \):

\[
\bigcap S \subseteq S \iff S \text{ is a decomposition of } S
\]

This is a trivial result of the fact that \( S \) is non-empty and that all specifications in \( S \) are refinements of \( S \). Actually, we only need that at least one specification in \( S \) is a refinement of \( S \). However, when we require total refinement, the fact that \( S \) is a decomposition of \( S \) becomes more relevant (see section 7.8).

This all looks rather trivial and indeed it is. We neglected however an important aspect of components. Often there exist dependencies between components. In the model of components and component composition of this chapter, dependencies between components are impossible, although specification \( MATH \) of section 7.5 seems to suggest otherwise. Decomposition of \( MATH \) into separate components requires binding of variable \( MATH \) to be explicitly modeled however. This is the subject of the next chapter. Dependencies are then possible and we have to face the problems that they cause.

### 7.7 Aspects and invariants

A relatively new paradigm that is expected to have a big impact on the development of large software systems is aspect-oriented programming. Aspect-oriented programming languages enable one to isolate a common property of several procedures into an aspect. This prevents the property from being duplicated and scattered over the entire system.

#### 7.7.1 Aspect operator

In order to enable isolation of common properties of procedures, we define the aspect operator \( \ll \in ((I = I) \times (I = I)) \subseteq (I = I) \times (\phi I) \) by

\[
R \ll B = R \circ B \cap UNI \cap FUN \cap NMP \cap DEQ
\]

The aspect operator is thus the cylindric-type operator that is constrained by the conjoint-sum constraint. It is actually a straightforward generalisation of the pack and a specialisation of the arbitrary conjoint sum. The following alternative definitions of the aspect operator show the close relationship between these operators
(the "∩ UNI∩FUN∩NMP∩DEQ" is necessary for the case in which \( B \) is empty):

\[
R \prec B = \bigcap \{(R \cdot i) \mid i \in B \mid i\} \cap UNI\cap FUN\cap NMP\cap DEQ
\]

\[
R \prec B = \cap[(R, i) \mid i \in B \mid i]
\]

If we take \( B = \{i\} \) in the first of these two definitions, we see in what way the aspect operator is a generalisation of the pack:

\[
R \prec \{i\} = R \prec i
\]

The theorems about the pack can be generalised to the following theorems about the aspect operator:

- \( R = R \prec B \triangleright i \iff i \in B \)
- \( \Pi = R \prec B \triangleright j \iff j \notin B \)
- \( R \subseteq T \iff R \prec B \subseteq T \prec B \iff B \) is non-empty
- \( R \neq T \iff R \prec B = T \prec B \iff B \) is non-empty
- \( R \cap T \prec B = R \prec B \cap T \prec B \)
- \( R \cup T \prec B = R \prec B \cup T \prec B \iff B \) has a unique element
- \( R \prec T \prec B = R \prec B \circ T \prec B \)
- \( R \prec T \prec B = R \prec B \prec T \prec B \)

The first two theorems are straightforward results of the pack’s cancellation theorems and the fact that \((\triangleright i)\) distributes over the arbitrary intersection. The inclusion and equality theorems can be proved in the same way as the corresponding theorems about the binary constrained cylindric products. The intersection, union and converse theorems are straightforward results of the corresponding theorems about the cylindric-type operator. To prove the composition theorem, the fact can be used that \( S (UNI\cap FUN\cap NMP\cap DEQ) Q \) implies that \( S \) and \( Q \) are of the form \([(y, i)]\) and \([(z, i)]\) respectively.

The aspect operator enables us to add a property to relations that are packed with a certain tag. As an example we take specification \( Math \), defined by

\[
Math = \text{inc} \cdot \text{dec} \cap \text{chg} \cdot \text{skp} \cap \text{inc} \cdot \text{dec} \cap \text{chg} \cdot \text{skp}
\]

\[
\text{inc} = Z \cdot [C > C] \cdot Z
\]

\[
\text{dec} = Z \cdot [C < C] \cdot Z
\]

\[
\text{chg} = Z \cdot [C \neq C] \cdot Z
\]

\[
\text{skp} = Z \cdot [C = C] \cdot Z
\]

There is one property that all four procedures share and that is that the inputs and outputs are integer numbers. The procedures \( \text{inc} \), \( \text{dec} \) and \( \text{chg} \) also share another property: their output is unequal to their input. We now show an equivalent definition of \( Math \) where we factored out these two properties into separate aspects:
The following example shows how the aspect operator can be used to factor out
common behaviour of methods of different classes:

$$MATH = \text{int} \prec \{\text{inc}, \text{dec}, \text{chg}, \text{skp}\}$$
$$\cap \ \text{neq} \prec \{\text{inc}, \text{dec}, \text{chg}\}$$
$$\cap \ \text{inc}'\prec \text{inc} \cap \text{dec}'\prec \text{dec} \cap \text{chg}'\prec \text{chg} \cap \text{skp}'\prec \text{skp}$$

$$\text{int} = Z \cdot \Pi \cdot Z$$
$$\text{neq} = [\mathbb{C} \neq \mathbb{C}]$$

$$\text{inc}' = [\mathbb{C} \geq \mathbb{C}]$$
$$\text{dec}' = [\mathbb{C} \leq \mathbb{C}]$$
$$\text{chg}' = [\text{true}]$$
$$\text{skp}' = [\mathbb{C} = \mathbb{C}]$$

Because most programming languages do not have an intersection, aspects are usu-
ally added by means of sequential composition. An aspect $T$ is then usually im-
plemented by two pieces of code $T_0$ and $T_1$ that are placed respectively before and
after the original code. According to our definition of an aspect, these two pieces
should satisfy

$$T_0 \cdot S \cdot T_1 \subseteq S \cap T$$

for all method bodies $S$ to which the aspect is added. The aspect $\text{int}$ from our
example could for example be implemented by the pieces $Z$ and $Z$.

Using sequential composition to add aspects also enables one to alter the behaviour
of procedures in a non-refining manner. We leave the answer to the question whether
or not this is desirable to researchers from the field of aspect-oriented programming.

### 7.7.2 Invariants

In object-oriented specifications a class usually has an invariant attached to it. The
meaning of this is that all methods of the class adhere to the invariant. Invariants
are usually separated into two kinds: state invariants and history invariants. A state invariant is a property (modeled by a set) that the context (state) satisfies before and after each method call. A history invariant is a relation that tells what
kind of state change is guaranteed by each method. An invariant $I$ is thus defined
by a state invariant $SI$ and a history invariant $HI$ as follows:
\[ I = SI \cdot HI \cdot SI \]

We seem to deviate here from the traditional view where ‘a state invariant only needs to hold afterwards if it held initially’. We think however that this traditional view is based on a misunderstanding between what a specification defines and how one proves that an implementation satisfies this. Section 7.7.4 should clarify this.

Invariants are actually no different from aspects. Enforcing an invariant on all methods of a class can be done by using the universal set \( I \) as right argument of <:

\[
\begin{align*}
\text{Crease} &= \text{inv} \cup \text{inc} \cup \text{dec} \cup \text{chg} \\
\text{inv} &= \text{stateInv} \cup \text{historyInv} \cup \text{stateInv} \\
\text{stateInv} &= \{ C \in \mathbb{Z} \} \\
\text{historyInv} &= \{ [C \neq C] \} \\
\text{inc} &= \{ [C \geq C] \} \\
\text{dec} &= \{ [C \leq C] \} \\
\text{chg} &= \{ \text{true} \}
\end{align*}
\]

7.7.3 Refinement

Apart from the readability of specifications, factoring out aspects/invariants also has advantages for refinement. It is rather easy to refine the specification \( \text{Crease} \) of section 7.7.2 to the specification \( \text{Crease1} \) where the inputs and outputs are two’s complement bytes and where the increases and decreases are not by an arbitrary integer number, but always by 1:

\[
\begin{align*}
\text{Crease1} &= \text{inv1} \cup \text{inc} \cup \text{dec} \cup \text{chg} \\
\text{inv1} &= \text{stateInv1} \cup \text{historyInv1} \cup \text{stateInv1} \\
\text{stateInv1} &= \{ C \in \mathbb{Z} \land -128 \leq C \land C < 128 \} \\
\text{historyInv1} &= \{ [\bar{C} - \bar{C}] = 1 \}
\end{align*}
\]

We only need to copy (inherited) the procedure bodies from \( \text{Crease} \) and strengthen the invariant.

7.7.4 Proving invariants

Programming languages are usually not able to ‘magically’ ensure that procedures adhere to an invariant, although some invariants can be handled with aspect-oriented programming techniques. In general however, we have to prove that each procedure adheres to all invariants.

There is a simple ‘trick’ to ensure that a certain procedure cannot violate any invariant and that is to simply not allow that procedure to be called. This ‘trick’ is part of almost every programming language. Procedures that are allowed to be called, we refer to as supported procedures. In object-oriented languages for
example, a class is always associated with a collection of methods that are supported by the class. This collection is usually known at compile time but sometimes only at runtime. Calling an unsupported method results in an error (either at compile time or at runtime).

For procedures that are supported, it is usually undesirable to have to check the state invariant at the beginning of each procedure. It can be costly to have to check it and if it does not hold, the only option we have is to abort in one way or another. There is a way to avoid this. We then only need to prove that the state invariant holds after the body of the procedure has been executed, if it held when the procedure was called. The trick is that we provide the user of our procedures with an initialiser (comparable to the notion of constructor or factory in object-oriented languages) that establishes the state invariant somehow. If the specification that uses our procedures, from now on referred to as the main, first calls the initialiser and then only changes the state by calling supported procedures, we know that the state invariant ‘holds at every point in the main’. This ‘trick’ is called the datatype-induction principle. A similar ‘trick’, without the initialiser, is often used to ensure that a procedure maintains the state invariant. Let it only change the state by calling procedures of which it is known that they maintain the state invariant.

An example of a specification that uses the datatype-induction principle to ‘refine’ specification Crease1 of section 7.7.3 is

\[
\begin{align*}
\text{Crease1}' &= \text{inc}' \circ \text{inc} \cap \text{dec}' \circ \text{dec} \cap \text{chg}' \circ \text{chg} \cap \text{init}' \circ \text{init} \\
\text{inc}' &= \{\mathbb{C} = \mathbb{C} + 1\} \cdot \{\mathbb{C} \neq 127\} \\
\text{dec}' &= \{\mathbb{C} = \mathbb{C} - 1\} \cdot \{\mathbb{C} \neq -128\} \\
\text{chg}' &= \text{inc}' \\
\text{init}' &= \{\mathbb{C} = 0\}
\end{align*}
\]

Although Crease1’ is not a refinement of Crease1, it becomes one if we restrict to the supported procedures and assume that the state invariant holds for their inputs:

\[
\text{Crease1}' \circ (\text{stateInv1} \rightarrow \{\text{inc}, \text{dec}, \text{chg}\}) \subseteq \text{Crease1}
\]

Notice the guards at the beginning (right) of inc1’ and dec1’ which are necessary to avoid violation of the state invariant.

An example of a main that uses Crease1’ is (remember to read right-left and bottom-up)

\[
\text{if } \{\mathbb{C} > 0\} \\
\text{then Crease1}' \circ \text{dec} = \text{Crease1}' \circ \text{dec} \\
\text{else Crease1}' \circ \text{inc} = \text{Crease1}' \circ \text{inc} \circ \text{Crease1}' \circ \text{chg} \\
\circ \text{Crease1}' \circ \text{init}
\]

165
Because Crease1>'init establishes the state invariant, because all supported procedures maintain it and because of certain algebraic properties of the operators that are used to construct the main, we know that the state invariant ‘holds at every point in the main program’, meaning that the above formula is equal to

\[
\text{stateInv1}
\]

\[\text{if } \llbracket C > 0 \rrbracket \]

\[\text{then stateInv1} \circ \text{Crease1>'dec} \circ \text{stateInv1} \circ \text{Crease1>'dec} \circ \text{stateInv1} \]

\[\text{else stateInv1} \circ \text{Crease1>'inc} \circ \text{stateInv1} \circ \text{Crease1>'inc} \circ \text{stateInv1} \]

\[\text{stateInv1} \]

\[\text{Crease1>'chg} \]

\[\text{stateInv1} \]

\[\text{Crease1>'init} \]

This distribution of the state invariant over the main is a trivial consequence of the following properties:

\[
S_0 \cup S_1 \circ A = A \circ (A \circ S_0 \cup A \circ S_1 \circ A) \circ A \iff S_0, S_1 \in A \circ A
\]

\[
S_0 \circ S_1 \circ A = A \circ S_0 \circ A \circ S_1 \circ A \iff S_0, S_1 \in A \circ A
\]

\[
S^* \circ A = A \circ (A \circ S^* \circ A) \circ A \iff S \in A \circ A
\]

\[
B \circ A = A \circ B \circ A = A \circ B
\]

In object-oriented languages, an additional trick is used to make it easier to maintain state invariants. The state is then partitioned into the same classes as the procedures and procedures are only allowed to directly change the state partition of their class. The state partition of another class may only be changed by calling (directly or indirectly) a procedure of that class. This helps in keeping track of the state partitions that can be changed by a call to a certain procedure and consequently helps to see which invariants could be affected by a certain call.

### 7.7.5 Type, declarative and operational invariants

Just like a pre can be split up into a pre-type and a pre-condition, a state invariant can be split up into a **state-invariant type** and a **state-invariant condition**. A history invariant is usually split up into a **post invariant**, describing the state change in a declarative manner and an **action invariant**, describing the state change in an operational manner.

### 7.8 Totality

In many cases, (partial) refinement is not considered strong enough to model the concept of ‘acceptable specialisations’. The empty specification \(\emptyset\) is for example a (partial) refinement of every specification and cannot really be considered an ‘acceptable specialisation’ of every specification. This is where **totality** requirements come in.
7.8.1 Total refinement

If someone constructs a refinement $S'$ of a specification $S$, it is in many cases desirable that $S'$ is not refining something by means of emptiness if specification $S$ did not prescribe emptiness. In other words, the domain of $S'$ should be at least as large as the domain of $S$:

$$S' \supseteq S.$$

We say that $S'$ domain-includes $S$.

The combination of refinement and domain-inclusion is called total refinement. The **is-total-refinement-of** $\in A \rightleftharpoons (I \rightarrow I) \times (I \rightarrow I)$ is defined by

$$S' \subseteq S \equiv S' \subseteq S \land S' \supseteq S,$$

or equivalently ($\supseteq$ is monotonic):

$$S' \subseteq S \equiv S' \subseteq S \land S' = S.$$

A total refinement of the specification

$$\mathbb{Z} \mathbf{[\hat{C} \leq \hat{C}]} \mathbb{Z}$$

is for example

$$\mathbf{[\hat{C} = \hat{C} \cdot 1]} \mathbb{Z}$$

The specification

$$\mathbb{Z} \mathbf{[\hat{C}^2 = \hat{C}]}$$

is however not a total refinement because it is for example empty in $2$ whereas $\mathbb{Z} \mathbf{[\hat{C} \leq \hat{C}]} \mathbb{Z}$ is not empty in $2$.

7.8.2 Proving totality

The type operator $\rightarrow$ can be used to prove totality. The meaning of $S \in A \rightarrow B$ is that $S$ connects each input that is an element of $B$ with at least one output that is an element of $A$. Some **totality rules** that we can use, are that for sets $A^{(0)}$, $B^{(0)}$, $C$ and $D$:
\[ S \in A \rightarrow B' \iff S \in A' \rightarrow B \land A' \subseteq A \land B' \subseteq B \]
\[ S' \in A \rightarrow B \iff S \in A \rightarrow B \land S' \supseteq S \]
\[ S_0 \cup S_1 \in A \rightarrow B \cup D \iff S_0 \in A \rightarrow B \land S_1 \in A \rightarrow D \]
\[ S_0 \cap S_1 \in A \rightarrow B \iff S_0 \in A \rightarrow C \land S_1 \in C \rightarrow B \]
\[ B \in B \rightarrow B \]

We have for example:

\[
\begin{align*}
\lceil \hat{C} = -\sqrt{C} \rceil \cdot \{C \leq 0\} & \cup \{C = \sqrt{C} \} \cdot \{C \geq 0\} \in I \rightarrow \mathbb{R}^+ \\
\rightharpoonup \{ \text{totality rule for } \cup, \mathbb{R} = \mathbb{R}^- \cup \mathbb{R}^+ \} \\
\{C \leq 0\} &= \{ x \mid x \in \mathbb{R} \land x \leq 0 \mid x \} \\
\mathbb{R}^- &= \{ x \mid x \in \mathbb{R} \land x \geq 0 \mid x \} \\
\lceil \hat{C} = -\sqrt{C} \rceil \cdot \{C \leq 0\} & \in I \rightarrow \mathbb{R}^- \land \\
\lceil \hat{C} = \sqrt{C} \rceil \cdot \{C \geq 0\} & \in I \rightarrow \mathbb{R}^+ \land \\
\{ \text{totality rule for } \cdot \} \\
\{ \hat{C} = -\sqrt{C} \} & \in I \rightarrow \mathbb{R}^- \land \\
\{ \hat{C} = \sqrt{C} \} & \in I \rightarrow \mathbb{R}^+ \land \\
\{ \text{totality rule for } \leq, \mathbb{R}^- \subseteq \{ C \leq 0 \}, \mathbb{R}^+ \subseteq \{ C \geq 0 \} \} \\
\lceil \hat{C} = -\sqrt{C} \rceil & \in I \rightarrow \mathbb{R}^- \land \\
\mathbb{R}^- & \in I \rightarrow \mathbb{R}^- \land \\
\lceil \hat{C} = \sqrt{C} \rceil & \in I \rightarrow \mathbb{R}^+ \land \\
\mathbb{R}^+ & \in I \rightarrow \mathbb{R}^+ \\
\{ \text{totality rule for } \leq, \mathbb{R} \subseteq \{ C \leq 0 \}, \mathbb{R} \subseteq \{ C \geq 0 \} \} \\
\{ \text{chapter 6} \} \\
\end{align*}
\]

\[ \text{true} \]

### 7.8.3 Total correctness

The notion of partial correctness that was presented in section 7.3 can be extended to a notion of **total correctness** by adding the requirement that the range of the pre is a subset of the domain of the body. We define the **total-correctness operator** \( \_ \_ \in \mathbb{B} \) \( \_ \_ \in (I \rightarrow I) \times (I \rightarrow I) \times (I \rightarrow I) \) by

\[ R \cup S \mid P \equiv R \cup S \cup P \land P \subseteq S \]

If \( R \cup S \mid P \) holds, we call specification \( S \) **totally correct** with respect to post \( R \) and pre \( P \). The following **total-correctness rules** hold:
We leave the proofs to the reader. Notice that where the partial-correctness rules for
the union and sequential composition are equivalences, the total-correctness rules
are only implications. Furthermore, we only give one total-correctness rule for the
sequential composition whereas for partial correctness, we had three rules. The
total-correctness rule for the if-then-else, whose proof we also leave to the reader,
provides some compensation:

\[ R \{ \text{if } B \text{ then } S_0 \text{ else } S_1 \} | P = R | S_0 | B \cdot P \wedge R | S_1 | (I \setminus B) \cdot P \]

The iteration does not seem an appropriate construct in the context of total-
correctness. For this case, the ‘while’ provides compensation:

\[ R \{ \text{while } B \text{ do } S \} | P = \{ \bullet S \cdot B \cdot (P) \text{ is well-founded and } \bullet S \cdot B \in P; \rightarrow P \} \]
\[ \exists (X \mid R \geq (I \setminus B) \cdot X \wedge X \mid S \cdot B \cdot X \wedge X \geq P) \]

The remainder of this subsection is dedicated to proving this theorem.

\[ R \{ \text{while } B \text{ do } S \} | P = \{ \text{definition of } \underline{\underline{\_}} \} \]
\[ R \{ \text{while } B \text{ do } S \} | P = \{ A \subseteq T^\ast \equiv A = (T \cdot A) \text{ for all sets } A \text{ and relations } T \} \]
\[ R \{ \text{while } B \text{ do } S \} | P = \{ \text{while } B \text{ do } S \} \cdot P \]
\[ \{ \bullet S \cdot B \cdot (P) \text{ is well-founded and } \bullet S \cdot B \in P; \rightarrow P \} \text{ (see section 7.3.9)} \]
\[ R \{ \text{while } B \text{ do } S \} | P = \{ \text{definition of } \underline{\underline{\_}} \} \]
\[ \exists (X \mid R \geq (I \setminus B) \cdot X \wedge X \mid S \cdot B \cdot X \wedge X \geq P) \]

We split the penultimate step of this proof into (⇒) and (⇐), starting with (⇒):

\[ R \subseteq \Rightarrow P \subseteq P' \subseteq I \]

\[ \exists (X \mid R \geq (I \setminus B) \cdot X \wedge X \mid S \cdot B \cdot X \wedge X \geq P) \]
The above lemma can now be proved as follows:

\[ R \setminus B \circ (S \cdot B)^* \subseteq P \]  
\[ \equiv \{ \text{definition of } \setminus \} \]  
\[ \forall A \subseteq T \equiv A = (T \cdot A) \text{ for all sets } A \text{ and relations } T \]  
\[ P \subseteq (1 \setminus B \circ (S \cdot B)^*) \]  
\[ \equiv \{ \text{definition of } \setminus \} \]  
\[ \exists (X \parallel (1 \setminus B) \cdot X \subseteq R \land S \cdot B \cdot X \subseteq X \land (B \cdot X) \subseteq S \land P \subseteq X) \]  
\[ \equiv \{ \text{definition of } \setminus \} \]  
\[ \exists (X \parallel R \supseteq (1 \setminus B) \cdot X \land X) \subseteq (B \cdot X) \subseteq S \land X \supseteq P) \]  

For the \( X \) that we took in the middle step, we focus on the proof of \( (B \cdot X) \subseteq S \). For the other properties that this \( X \) should satisfy, we refer to section 7.3.7.

\[ (B \circ (S \cdot B)^* \circ P) \subseteq S \]  
\[ \equiv \{(R \cdot T) = (R \cdot T) \text{ for all relations } R \text{ and } T \} \]  
\[ (B \circ (S \cdot B)^* \circ P) \subseteq S \]  
\[ \equiv \{ \text{for all } S: B \in P, \text{ then } (S \cdot B)^* \circ P = P \circ (S \cdot B)^* \circ P \} \]  
\[ (B \circ P \circ (S \cdot B)^* \circ P) \subseteq S \]  
\[ \equiv \{ R \subseteq T \text{ for all relations } R \} \]  
\[ (B \circ P \circ \Pi) \subseteq S \]  
\[ \equiv \{(A \cdot \Pi) = A \text{ for all sets } A \} \]  
\[ B \circ P \subseteq S \]  
\[ \equiv \{(1 \setminus B \circ (S \cdot B)^* \circ B) \subseteq S \text{ (left to the reader)}\} \]  
\[ B \circ P \subseteq (1 \setminus B \circ (S \cdot B)^* \circ B) \]  
\[ \equiv \{ \text{easy to prove using the fact that } A \subseteq T \equiv A = (T \cdot A) \text{ for all sets } A \text{ and relations } T \} \]  
\[ B \circ P \subseteq (1 \setminus B \circ (S \cdot B)^*) \]  
\[ \equiv \{ B \text{ is a set} \} \]  
\[ P \subseteq (1 \setminus B \circ (S \cdot B)^*) \]  

This completes the proof of \( \Rightarrow \). We now proceed with the \( \Leftarrow \). Partial correctness was already treated in section 7.3.9. The interesting part is the one that deals with totality. The key lemma that we use, is that if a relation is well-founded then there is a finite path from every element in its domain to an element outside its domain. This lemma can be formalised as follows:

\[ R \text{ is well-founded} \]  
\[ \Rightarrow \]  
\[ ((1 \setminus R) \circ R^*) = 1 \]  

To prove this lemma, we use a theorem from [17] that states that well-foundedness of a relation \( R \) is equivalent to

\[ \forall (S, T) : T = S \cup T \cdot R \equiv T = S \circ R^* \]  

The above lemma can now be proved as follows:
We now investigate components in the context of total refinement. We call a specification $S'$ a total Z-refinement of a specification $S$ if it is a total refinement and a Z-override of $S$. The is-total-Z-refinement-of $\exists_{Z \subseteq Z} \in B \iff (I \mapsto I) \times (I \mapsto I)$ is defined by

$$S' \subseteq Z \quad S' \subseteq S \land S' \circ Z \quad S$$

If $S_0$ is a total $Z_0$-refinement of $S$ and $S_1$ is a total $Z_1$-refinement of $S$ for disjoint sets $Z_0$ and $Z_1$, then the component composition of $S_0$ and $S_1$ is a total $Z_0 \cup Z_1$-refinement of $S$: 

171
A non-empty set of specifications $S$ is called a **total decomposition** of a specification $S$ if for each pair of different specifications $S_0$, $S_1$ from $S$ there exist disjoint sets $Z_0$, $Z_1$ such that $S_0$ is a total $Z_0$-refinement of $S$ and $S_1$ is a total $Z_1$-refinement of $S$. Formally, for a non-empty set of specifications $S$, we define:

$$S \text{ is a total decomposition of } S \iff \forall \langle S_0, S_1 | S_0, S_1 \in S \land S_0 \neq S_1 | \exists \langle Z_0, Z_1 | Z_0 \text{ and } Z_1 \text{ are disjoint sets} \mid S_0 \subseteq Z_0 \land S_1 \subseteq Z_1 \rangle \rangle$$

For any finite total decomposition $S$ of a specification $S$, the component composition of all specifications in $S$ is a total refinement of $S$. Formally, for a non-empty finite set of specifications $S$:

$$\bigcap S \subseteq S \iff S \text{ is a total decomposition of } S$$

For the case that $S$ consists of two specifications, this is a trivial consequence of the previous theorem. For other $S$, the two-element case can be used for the induction step in an inductive proof.

Things become more complicated in the next chapter where dependencies between components are possible.

### 7.8.5 Invariants

Looking at invariants, a problem arises in the context of total refinement. In general, we are not allowed to strengthen the state invariant anymore. The specification $\text{Crease}$ whose state invariant enforces the state to be an integer number:

$$\text{Crease} = \text{inv} \cap \text{inc} \cap \text{dec} \cap \text{chg}$$

$$\text{inv} = \text{stateInv} \cap \text{stateInv}$$

$$\text{stateInv} = \{ C \in \mathbb{Z} \}$$

$$\text{inc} = \{ C > C \}$$

$$\text{dec} = \{ C < C \}$$

$$\text{chg} = \{ C \neq C \}$$

does not have as total refinement the specification $\text{Crease}'$ whose state invariant enforces the state to also be even:
\[ Crease' = inv' < I \cap inc \cap dec \cap chg \]
\[ inv' = stateInv' \Pi stateInv' \]
\[ stateInv' = \{ C \in \mathbb{Z} \land \frac{C}{2} \in \mathbb{Z} \} \]

The datatype-induction principle again helps out. If we use this principle, we only need to prove total refinement for those inputs that adhere to the state invariant of the fine specification:

\[ Crease' \subseteq Crease \circ (stateInv' \rightarrow I) \]

This clearly holds because \( inc, dec \) and \( chg \) are total on \( stateInv' \).

The specification that restricts the state to two’s complement bytes:

\[ Crease'' = inv'' < I \cap inc \cap dec \cap chg \]
\[ inv'' = stateInv'' \Pi stateInv'' \]
\[ stateInv'' = \{ C \in \mathbb{Z} \land -128 \leq C \land C < 128 \} \]

is not a total refinement, even if we use the datatype-induction principle:

\[ Crease'' \nsubseteq Crease \circ (stateInv'' \rightarrow I) \]

Things go wrong for inputs \([127, inc] \) and \([-128, dec] \).

7.9 Conclusions

In this chapter we showed how several specification concepts can be modeled within the calculus that is used in this thesis. An important ingredient is the fact that we see pre-conditions, post-conditions, operational descriptions and invariants all as defining properties rather than one being a derived property of another. We have illustrated the gain in expressive power this can provide, especially in terms of simpler specifications and proofs.
Chapter 8

Statements

Recursion is an essential aspect of every realistic programming language. In this chapter we show how recursion is formalised by means of fixed-point theory.

8.1 Recursion

In most programming languages, procedures are able to call each other. What this essentially means, is that the program that contains the procedures is able to use its own behaviour. This can be modeled by defining programs as monotonic functions that have program behaviours as input and output. Taking a fixed point of a program now means that its own behaviour is used to determine its behaviour. This is called recursion.

In general we model program behaviours by collections and programs by monotonic total functions whose inputs and outputs have the same external type. The type of behaviours is thus $\mathcal{P} I$ and the type of programs is $\mathcal{P} I \leftarrow_{\nu} \mathcal{P} I$ for $I$ equal to $I$.

A program can have several fixed points. Similar to the ‘while’ (see section 7.3.9), we could now focus on programs with a single fixed point, but rather avoid this topic as it requires much extra theory. We simply always take the least fixed point, that we call the canonical behaviour of a program.

It is useful to have something that is more general than programs to allow for a rich collection of programming constructs. For this purpose, we introduce the notion of statements which are actually just total functions. A program is thus a monotonic statement whose inputs and outputs have the same external type.
8.2 Statement operators

In this section we introduce some useful statement operators.

8.2.1 Inclusion

The statement-inclusion \( \subseteq \) is defined by

\[
\forall b, s_0.s.b \subseteq s_1.b
\]

8.2.2 Binary intersection and union

The binary statement-intersection and binary statement-union \( \cap, \cup \) are defined by

\[
(s_0 \cap s_1).b = s_0.b \cap s_1.b
\]

\[
(s_0 \cup s_1).b = s_0.b \cup s_1.b
\]

8.2.3 Empty and universal statement

The empty statement and universal statement \( \varnothing, \Pi \) are defined by

\[
\varnothing.b = \varnothing
\]

\[
\Pi.b = \Pi
\]

8.2.4 Sequential composition

The sequential statement-composition \( \circ \) is defined by

\[
(s_0 \circ s_1).b = s_0.b \circ s_1.b
\]

8.2.5 Identity statement

The identity statement \( I \) is defined by

\[
I.b = I
\]
8.2.6 Pick and pack

The statement-pick \(^{\hat{\lozenge}}\) ∈ ((I → I) ← I) ← ((I → I) ← (I → I)) ← I) × I and the statement-pack \(^{\hat{\bowtie}}\) ∈ (((I → I) ← (I → I)) ← I) ← ((I → I) ← I) × I are defined by

\[(s \hat{\lozenge})_b = s.b \triangleright i\]
\[(s \hat{\bowtie})_b = s.b \triangleleft i\]

8.2.7 Call

The call call ∈ I ← I is defined by

\[\text{call}.b = \text{b}\]

The call enables recursion. In combination with the statement-pick, it enables us to call a procedure with a specific name. This is illustrated by an upcoming example.

8.2.8 Constrain

The constrain const ∈ (I ← I) ← I is defined by

\[(\text{const}.c)_b = c\]

The constrain enables us to describe behaviour by means of some fixed element (a relation for example).

8.2.9 Preservation of monotonicity

The operators that we introduced in section 8.2.2 till 8.2.8 are monotonicity preserving: they either construct a monotonic statement or transform monotonic statements into a monotonic statement. If we only use these operators to construct a statement, we know for sure that it is monotonic.

8.2.10 Other operators

We could of course define many more statement operators like \(^{\hat{\ast}}\), \(^{\hat{\cdot}}\), \(^{\hat{*}}\) and \(^{\hat{\circ}}\). Some of these would not be monotonicity preserving, but that is not a major problem because using them in the right way still enables us to construct monotonic statements. We simply left them out because we do not need them for the upcoming theory, except for a few that are introduced in chapter 11.
8.2.11 Correspondence with expressions

Statements are actually just expressions (see chapter 6) that are total and functional. The input of a statement is also called its context and determines the meaning of a call. In particular we have:

\[
\begin{align*}
\cap & = \hat{\cap} \\
\cup & = \hat{\cup} \\
\mathcal{O} & = \mathcal{O} \\
\mathcal{I} & = \mathcal{I} \\
\hat{\vdash} & = \hat{\vdash} \\
\vdash & = \vdash \\
(\hat{\triangleright}) & = (\triangleright) \\
(\hat{\triangleleft}) & = (\triangleleft) \\
\text{call} & = \mathcal{C} \\
\text{const} & = \mathcal{C}
\end{align*}
\]

8.2.12 Theorem lifting

Similar to theorems about expressions, many of the theorems that hold for \(\subseteq, \cap, \cup, \mathcal{O}, \mathcal{I}, \vdash, \triangleright, \triangleleft\) can be directly transformed into theorems about \(\hat{\subseteq}, \hat{\cap}, \hat{\cup}, \hat{\mathcal{O}}, \hat{\mathcal{I}}, \hat{\vdash}, \hat{\triangleright}, \hat{\triangleleft}\). In the context of statements we also call this theorem lifting.

Distribution of \((\hat{\triangleright})\) over \(\cap\) can for example be proved as follows:

\[
\begin{align*}
(s_0 \cap s_1) & \hat{\vdash} . b \\
= & \{\text{definition of } \hat{\vdash}\} \\
(s_0 \cap s_1) . b \triangleright i \\
= & \{\text{definition of } \cap\} \\
(s_0 . b \cap s_1 . b) & \triangleright i \\
= & \{\text{distribution of } (\triangleright) \text{ over } \cap\} \\
(s_0 . b \triangleright i) & \cap (s_1 . b \triangleright i) \\
= & \{\text{definition of } \hat{\vdash}\} \\
(s_0 \hat{\vdash} i) & \cap (s_1 \hat{\vdash} i) . b \\
= & \{\text{definition of } \cap\} \\
(s_0 \hat{\vdash} i \cap s_1 \hat{\vdash} i) . b
\end{align*}
\]

It is possible to treat constants, unary operators and binary operators in a uniform way, using functions with a 0-tuple as input instead of constants and functions with a 1-tuple as input instead of unary operators. Another way to treat them in a uniform way is to use constant-valued functions as constants and the objects of a product category (see [11]) as pairs. This would enable the definition of a generic lift operator \(\hat{\vdash}\) and a generic formulation of the above theorem. Such generic theory is outside the scope of this thesis though.
Algebraic frameworks like allegory theory [11] provide many free theorems by proving some simple properties of \( \hat{\subseteq} \), \( \hat{\cap} \), \( \hat{\cup} \), \( \hat{\circ} \), \( \hat{\setminus} \), \( \hat{\Pi} \), and \( \hat{I} \). An algebraic framework that includes \( \hat{\triangleright} \), \( \hat{\triangleleft} \), const, and call is left for future research. We stick to ‘proofs on demand’: if we need a certain theorem, we include an ‘ad-hoc’ proof.

8.2.13 Example

We now show how we can model the recursion that occurs in the example of section 7.5. The program \( MATH \) is defined by

\[
\begin{align*}
MATH & = \text{Crease} \hat{\triangleright} \text{Crease} \hat{\cap} \text{Three} \hat{\triangleright} \text{Three} \hat{\cap} \text{four} \hat{\triangleright} \text{four} \\
\text{Crease1} & = \text{inc1} \hat{\triangleright} \text{inc} \hat{\cap} \text{dec1} \hat{\triangleright} \text{dec} \\
\text{Three} & = \text{direct3} \hat{\triangleright} \text{direct} \hat{\cap} \text{indirect3} \hat{\triangleright} \text{indirect} \\
\text{inc1} & = \text{const}.[\hat{C} = \hat{C}+1] \\
\text{dec1} & = \text{const}.[\hat{C} = \hat{C}-1] \\
\text{direct3} & = (\text{call} \hat{\triangleleft} \text{Crease} \hat{\triangleright} \text{dec} \hat{\circ} \text{const}.[\hat{C} = 4]) \\
\text{indirect3} & = (\text{call} \hat{\triangleleft} \text{Crease} \hat{\triangleright} \text{dec} \hat{\circ} \text{call} \hat{\triangleleft} \text{four}) \\
\text{four} & = (\text{call} \hat{\triangleleft} \text{Crease} \hat{\triangleright} \text{inc} \hat{\circ} (\text{call} \hat{\triangleleft} \text{Three} \hat{\triangleright} \text{direct}))
\end{align*}
\]

The following equations demonstrate its canonical behaviour:

\[
\begin{align*}
(\mu MATH \hat{\triangleright} \text{Crease}) \hat{\triangleright} \text{inc} & = [\hat{C} = \hat{C}+1] \\
(\mu MATH \hat{\triangleright} \text{Crease}) \hat{\triangleright} \text{dec} & = [\hat{C} = \hat{C}-1] \\
(\mu MATH \hat{\triangleright} \text{Three}) \hat{\triangleright} \text{direct} & = [\hat{C} = 3] \\
(\mu MATH \hat{\triangleright} \text{Three}) \hat{\triangleright} \text{indirect} & = [\hat{C} = 3] \\
\mu MATH \hat{\triangleright} \text{four} & = [\hat{C} = 4]
\end{align*}
\]

In the following section we present a theorem that helps to prove these kinds of equations.

8.3 Call expansion

A well-known rule in programming is that “a call to a procedure may be replaced by the procedure’s body”. We introduce a similar rule that we name call expansion. For a function \( F \) that maps statements to statements (\( F \in (PI \leftrightarrow I) \leftrightarrow (PI \leftrightarrow I) \)), that is statement-monotonic:

\[
s' \hat{\subseteq} s \Rightarrow F.s' \hat{\subseteq} F.s
\]

monotonicity preserving:

\[
s \text{ is monotonic} \Rightarrow F.s \text{ is monotonic}
\]

and context preserving:

\[
\text{179}
\]
\[
\forall (s, s', b \mid s, s' \in I \land s.b = s'.b \mid (\mathcal{F}.s).b = (\mathcal{F}.s').b)
\]

the following equation holds:

\[
\mu(\mathcal{F}.\text{call}) = \mu(\mathcal{F}.(\mathcal{F}.\text{call}))
\]

proof

We define the following shorthands:

\[
\begin{align*}
s &= \mathcal{F}.\text{call} \\
 s' &= \mathcal{F}.(\mathcal{F}.\text{call}) \quad (= \mathcal{F}.s) \\
 s'' &= \mathcal{F}.(\mathcal{F}.\text{call} \cap \text{call}) \quad (= \mathcal{F}.(s\cap\text{call}))
\end{align*}
\]

and prove \( \mu s'' \subseteq \mu s' \subseteq \mu s \subseteq \mu s'' \):

\[
\begin{align*}
\mu s'' \subseteq \mu s' & \quad \Leftarrow \quad \{ \text{monotonicity of } \mu \} \\
 s'' \subseteq s' & \quad \Leftarrow \quad \{ \text{definition of } s'' \text{ and } s' \} \\
 \mathcal{F}.(s\cap\text{call}) \subseteq \mathcal{F}.s & \quad \Leftarrow \quad \{ \text{• statement-monotonicity of } \mathcal{F} \} \\
 s\cap\text{call} \subseteq s & \quad \Leftarrow \quad \{ s_0\cap s_1 \subseteq s_0 \} \\
\text{true} & \quad \Leftarrow \quad \{ \text{• } \mathcal{F} \text{ is context preserving} \}
\end{align*}
\]

\[
\mu s' \subseteq \mu s & \quad \Leftarrow \quad \{ \text{induction} \} \\
 s'.\mu s = \mu s & \quad \Leftarrow \quad \{ \text{computation} \} \\
 s'.\mu s = s.\mu s & \quad \Leftarrow \quad \{ \text{definition of } s' \text{ and } s \} \\
 (\mathcal{F}.s).\mu s = (\mathcal{F}.\text{call}).\mu s & \quad \Leftarrow \quad \{ \text{• } \mathcal{F} \text{ is context preserving} \} \\
 s.\mu s = \text{call}.\mu s & \quad \Leftarrow \quad \{ \text{definition of call} \} \\
 s.\mu s = \mu s & \quad \Leftarrow \quad \{ \text{computation} \} \\
\text{true} & \quad \Leftarrow \quad \{ \text{• } \mathcal{F} \text{ is context preserving} \} 
\]
\[ \mu s \subseteq \mu s'^{\prime} \]
\[ \Leftarrow \{ \text{induction} \} \]
\[ s.\mu s'' = \mu s'' \]
\[ \equiv \{ \text{computation} \} \]
\[ s.\mu s'' = s''.\mu s'' \]
\[ \equiv \{ \text{definition of } s \text{ and } s'' \} \]
\[ (\mathcal{F}.\text{call}).\mu s'' = (\mathcal{F}.(s\hat{\cap}\text{call})).\mu s'' \]
\[ \Leftarrow \{ \bullet \mathcal{F} \text{ is context preserving} \} \]
\[ \text{call.}\mu s'' = (s\hat{\cap}\text{call}).\mu s'' \]
\[ \equiv \{ \text{definition of } \hat{\cap} \} \]
\[ \text{call.}\mu s'' = s.\mu s'' \cap \text{call.}\mu s'' \]
\[ \equiv \{ \text{definition of call} \} \]
\[ \mu s'' = s.\mu s'' \cap \mu s'' \]
\[ \equiv \{ \text{computation} \} \]
\[ \mu s'' = s.\mu s'' \cap s''.\mu s'' \]
\[ \equiv \{ \text{definition of } \cap \} \]
\[ \mu s'' = (s\hat{\cap}s'').\mu s'' \]
\[ \equiv \{ \bullet \text{statement-monotonicity of } \mathcal{F}, \text{ we then have } s'' \subseteq s, \text{ or equivalently, } s\hat{\cap}s'' = s'' \} \]
\[ \mu s'' = s''.\mu s'' \]
\[ \equiv \{ \text{computation} \} \]
\[ true \]

Although it appears that we did not use the monotonicity preservingness of \( \mathcal{F} \), we used it implicitly by assuming several fixed points to exist.

All \( \mathcal{F} \)s that can be finitely constructed with the statement operators that we introduced in section 8.2, are statement-monotonic, monotonicity preserving and context preserving. This can be proved by induction on the structure of such \( \mathcal{F} \)s.

Call expansion seems closely related to the square rule of fixed-point calculus (see for example [4]). The square rule states that for \( f \in \Pi \leftarrow_{m} \Pi \) (see section 3.3.7 for the definition of \( n \))
\[ \mu f = \mu(f^2) \]

We were unable to exploit this rule in the proof of the call-expansion theorem.

As an example of call expansion in action, we prove that program \( p_0 \), defined by
\[
\begin{align*}
p_0 &= m_0 \hat{\cap} m_0 \hat{\cap} m_0 \\
m_0 &= \text{const.} [\hat{C} = 4] \\
n_0 &= \text{const.} [\hat{C} \geq \hat{C}] \Diamond \text{call} \hat{\cap} m
\end{align*}
\]
has the same canonical behaviour as program $p_1$, defined by

$$p_1 = m_0 \triangleleft m \cap n_1 \triangleleft n$$

$$n_1 = \text{const.}[\hat{C} \geq 4]$$

**proof**

$$\mu p_0$$

= \{ definition of $p_0$ \}

$$\mu(m_0 \triangleleft m \cap n_0 \triangleleft n)$$

= \{ definition of $n_0$ \}

$$\mu(m_0 \triangleleft m \cap (\text{const.}[\hat{C} \geq \hat{C}] : \text{call} \triangleleft m) \triangleleft n)$$

= \{ call expansion with $F = (m_0 \triangleleft m \cap (\text{const.}[\hat{C} \geq \hat{C}] : X \triangleleft m) \parallel X)$, $F.\text{call} = p_0$ \}

$$\mu(m_0 \triangleleft m \cap (\text{const.}[\hat{C} \geq \hat{C}] : p_0 \triangleleft m \triangleleft n) \triangleleft n)$$

= \{ see below \}

$$\mu(m_0 \triangleleft m \cap n_1 \triangleleft n)$$

= \{ definition of $p_1$ \}

$$\mu p_1$$

$$\text{const.}[\hat{C} \geq \hat{C}] : p_0 \triangleleft m$$

= \{ cancellation \}

$$\text{const.}[\hat{C} \geq \hat{C}] : m_0$$

= \{ definition of $m_0$ \}

$$\text{const.}[\hat{C} \geq \hat{C}] : \text{const.}[\hat{C} = 4]$$

= \{ left to the reader \}

$$\text{const.}[\hat{C} \geq 4]$$

= \{ definition of $n_1$ \}

$$n_1$$

### 8.4 Refinement

In this section we discuss refinement between statements.

#### 8.4.1 Statement-refinement

An obvious candidate for refinement between statements is statement-inclusion. We call a statement $s'$ a **statement-refinement** of a statement $s$ if

$$s' \subseteq s$$
Statement-refinement is a (PI ⊲ I)–partial-order.

8.4.2 Behavioural refinement

More fine-grained than statement-refinement, is refinement between behaviours of programs. We call a program \( s' \) a behavioural refinement of a program \( s \) if the canonical behaviour of \( s' \) refines the canonical behaviour of \( s \). The is-behavioural-refinement-of \( \hat{\subseteq} \in \mathfrak{B} \) is defined by

\[
s' \hat{\subseteq} s \equiv \mu s' \subseteq \mu s
\]

The relation \( \hat{\subseteq} \) is a (PI ⊲ I)-preorder. In contrast to \( \hat{\subseteq} \), it is not anti-symmetric.

The fact that the least–fixed-point operator is monotonic, implies that behavioural refinement is weaker than statement-refinement. Formally, for all programs \( s' \) and \( s \) we have

\[
s' \hat{\subseteq} s \iff s' \subseteq s
\]

8.4.3 Simple refinement rules

Some simple refinement rules that enable us to prove refinement for some simple cases are:

\[
s_0 \cap s_1' \hat{\subseteq} s_0 \cap s_1 \iff s_0' \hat{\subseteq} s_0 \land s_1' \hat{\subseteq} s_1
\]

\[
s_0' \cup s_1' \hat{\subseteq} s_0 \cup s_1 \iff s_0' \hat{\subseteq} s_0 \land s_1' \hat{\subseteq} s_1
\]

\[
\emptyset \hat{\subseteq} \emptyset \equiv \text{true}
\]

\[
\hat{\Pi} \hat{\subseteq} \hat{\Pi} \equiv \text{true}
\]

\[
s_0' \hat{\cup} s_1' \hat{\subseteq} s_0 \hat{\cup} s_1 \iff s_0' \hat{\subseteq} s_0 \land s_1' \hat{\subseteq} s_1
\]

\[
\hat{\Pi} \hat{\cup} \emptyset \hat{\subseteq} \hat{\Pi} \equiv \text{true}
\]

\[
s_0' \hat{\cdot} s_1' \hat{\subseteq} s_0 \hat{\cdot} s_1 \iff s_0' \hat{\subseteq} s_0 \land s_1' \hat{\subseteq} s_1
\]

\[
call \hat{\subseteq} \text{call} \equiv \text{true}
\]

\[
\text{const}.C' \hat{\subseteq} \text{const}.C' \equiv C' \subseteq C
\]

8.4.4 Example

Although the simple refinement rules are rather weak, we can handle already some interesting examples if we combine them with call expansion and some other basic algebraic rules. We show how we can prove that program \( p \), defined by

183
\[
p = m \supseteq m \cap n \supseteq n
\]
\[
m = \text{const.}[\hat{C} \geq 3]
\]
\[
n = \text{const.}[\hat{C} \geq \hat{C}] \cup \text{call} \cdot m
\]
is behaviourally refined by program \(p'\), defined by
\[
p' = m' \supseteq m \cap n' \supseteq n \cap o' \supseteq o
\]
\[
m' = \text{const.}[\hat{C} = 4]
\]
\[
n' = \text{const.}[\hat{C} = 5]
\]
\[
o' = \text{const.}[\hat{C} \geq \hat{C}] \cup \text{call} \cdot n
\]

The first step that we take, is to use the simple refinement rules to prove that \(p\) is behaviourally refined by \(p_0\), defined by
\[
p_0 = m' \supseteq m \cap n' \supseteq n
\]
The proof is as follows:

\[
p_0 \subseteq p
\]
\[
\Leftarrow \quad \{\text{behavioural refinement is weaker than statement-refinement}\}
\]
\[
p_0 \subseteq p
\]
\[
\Leftarrow \quad \{\text{definition of } p_0 \text{ and } p\}
\]
\[
m' \supseteq m \cap n' \supseteq m \cap n
\]
\[
\Leftarrow \quad \{\text{simple refinement rule for } \cap\}
\]
\[
m' \supseteq m \cap n \supseteq n
\]
\[
\Leftarrow \quad \{(PI \leftarrow Q)-\text{reflexivity of } \subseteq\}
\]
\[
m' \supseteq m \supseteq m
\]
\[
\Leftarrow \quad \{\text{definition of } m' \text{ and } m\}
\]
\[
\text{const.}[\hat{C} = 4] \subseteq \text{const.}[\hat{C} \geq 3]
\]
\[
\Leftarrow \quad \{\text{simple refinement rule for const}\}
\]
\[
[\hat{C} = 4] \subseteq [\hat{C} \geq 3]
\]
\[
\Leftarrow \quad \{4 \geq 3\}
\]
\[
\text{true}
\]

In section 8.3 we already showed that the canonical behaviour of \(p_0\) is equal to the canonical behaviour of \(p_1\) defined by
\[
p_1 = m' \supseteq m \cap n_1 \supseteq n
\]
\[
n_1 = \text{const.}[\hat{C} \geq 4]
\]
The last step is to again use the simple refinement rules to show that $p'$ is a behavioural refinement of $p_1$:

$$p' \subseteq p_1$$

\[ \text{\{similar to the previous proof\}} \]

$$m' \circ m \cap n' \circ n \subseteq m' \circ m \cap n_1 \circ n$$

\[ \text{\{similar to the previous proof\}} \]

$$m' \circ m \cap n' \circ n \subseteq m' \circ m \cap n_1 \circ n$$

\[ \text{true} \]

The fact that $\subseteq$ is a (PI $\in$ n PI)-preorder now gives us that $p' \subseteq p$.

### 8.4.5 A ‘strange’ refinement

Choosing relations as model for behaviour has a ‘strange’ consequence. If we again take program $p$, defined by

$$p = m \circ m \cap n \circ n$$

$$m = \text{const.}[\mathbb{C} \geq 3]$$

$$n = \text{const.}[\mathbb{C} \geq 3]$$

its appears to be also behaviourally refined by $p''$, defined by

$$p'' = m'' \circ m \cap n'' \circ n$$

$$m'' = \text{const.}[\mathbb{C} = 4]$$

$$n'' = \text{const.}[\mathbb{C} = 3]$$

We leave it to the reader to check this fact (hint: start with call expansion). Intuitively this refinement looks a bit strange because we ‘naturally’ interpret statement $n$ as: “The output is at least as large as the output of $m$.”, where it should be interpreted as: “The output is at least as large as some output of the given specification of $m$.”. These kinds of ‘unnatural’ refinements are due to the fact that relations abstract from information that we ‘naturally’ see as part of the semantics, like the call to $m$. The semantics of a call in this chapter is referred to as an invisible call. In the next chapter we introduce a richer semantics where calls can be made visible. We can then write down something that really means: “The output is at least as large as the output of $m$.”.
8.5 Components

In this section we show how components and their composition can be modeled within the framework of statements. We take behavioural refinement as the preferred kind of refinement and investigate different ways to compose components.

8.5.1 Example

We want two separate teams to construct programs whose component composition behaviourally refines program $q$, defined by

$$
q = c \triangleleft c \cap d \triangleleft d
$$

$$
c = \text{const.}[\hat{C} \geq 3]
$$

$$
d = \text{const.}[\hat{C} \geq 4]
$$

Team C develops the following program that is clearly a behavioural refinement of $q$:

$$
qc = c' \triangleleft c \cap d' \triangleleft d
$$

$$
c' = \text{const.}[\hat{C} = 3]
$$

Team D decides to reuse whatever team C develops and simply add 1:

$$
qd = c \triangleleft c \cap d' \triangleleft d
$$

$$
d' = \text{const.}[\hat{C} = \hat{C} + 1] \cdot \text{call}\hat{c}\cdot c
$$

We now investigate three different kinds of component composition. One important requirement that we have, is that the component composition of two behavioural refinements of a program is a behavioural refinement of that program.

8.5.2 Statement-intersection

Had we taken statement-refinement as the preferred kind of refinement, statement-intersection would be the natural choice for component composition. In the context of behavioural refinement however, statement-intersection seems too ‘coarse’. For the statement-intersection of $qc$ and $qd$ we have for example

$$
qc \cap qd = c' \triangleleft c \cap d' \triangleleft d \cap d \triangleleft d
$$

The “$\cap d' \triangleleft d$” can however not be eliminated. The ‘knowledge’ that the call to $c$ in $d'$ results in a value that is at least 3, is not ‘transferred’ if we use $\cap$ as composition operator. This means that the information that $d$ outputs a value that is at least 4, as specified by $d$, cannot be identified as redundant.

Although statement-refinement does not seem the perfect candidate, it does comply
with the requirement that the component composition of two behavioural refinements of a program is a behavioural refinement of that program:

\[ s_0 \subseteq s \land s_1 \subseteq s \Rightarrow s_0 \hat{\cap} s_1 \subseteq s \]

### 8.5.3 Behavioural intersection

A composition operator that does incorporate the ‘knowledge transfer’ that the statement-intersection operator \( \hat{\cap} \) lacks, is the **binary behavioural intersection** \( \hat{\cap} \in (PI \leftarrow PI) \leftarrow (PI \leftarrow m PI) \times (PI \leftarrow m PI) \), defined by

\[
(s_0 \hat{\cap} s_1).b = \mu s_0 \cap \mu s_1
\]

The following theorem holds:

\[ s \hat{\subseteq} s_0 \hat{\cap} s_1 \equiv s \hat{\subseteq} s_0 \land s \hat{\subseteq} s_1 \]

This theorem is not a definition due to the fact that \( \hat{\subseteq} \) is not a partial order, but only a preorder.

The behavioural intersection satisfies our main requirement:

\[ s_0 \hat{\subseteq} s \land s_1 \hat{\subseteq} s \Rightarrow s_0 \hat{\cap} s_1 \hat{\subseteq} s \]

Although the behavioural intersection seems the right candidate in the context of behavioural refinement, it does not represent the kind of component composition that is considered ‘common practice’. For the behavioural intersection of the \( qc \) and \( qd \) of section 8.5.1, we have for example

\[
\mu(qc \hat{\cap} qd) = [[C = 3] \subseteq c \cap [\hat{C} \geq 4] \subseteq d
\]

This shows that components interact at the level of specifications rather than at the level of implementations if \( \hat{\cap} \) is used as component-composition operator. In the following subsection we introduce a component-composition operator where interaction takes place at the level of implementations.

### 8.5.4 Statement-component composition

The kind of component composition that is considered ‘common practice’, is the one where components interact at the level of implementations. In that case the behaviour of the component composition of \( qc \) and \( qd \) of section 8.5.1 is equal to

\[
[[\hat{C} = 3] \subseteq c \cap [\hat{C} = 4] \subseteq d
\]

187
This is equal to the behaviour of \(q'\) defined by
\[
q' = c' \circ c \cap d' \circ d
\]

We call this form of component composition **statement-component composition**.

The program \(q'\) is a behavioural refinement of \(q\). In general however, it is not a trivial fact that the statement-component composition of two behavioural refinements of a program is a behavioural refinement of that program. This is shown in section 8.5.7.

We now show how the notion of statement-component composition can be formalised. For disjoint sets \(Z_0\) and \(Z_1\), the \((Z_0, Z_1)\)-component composition \(Z_0 \circ Z_1 \in (1 \iff 1) \iff (1 \iff 1) \times (1 \iff 1)\) is defined by
\[
S_0 \circ Z_1 S_1 = \{ \circ Z_0 \text{ and } Z_1 \text{ are disjoint sets} \}
\]
and the statement-\((Z_0, Z_1)\)-component composition \(Z_0 \circ Z_1 \in ((1 \iff 1) \iff (1 \iff 1)) \iff (1 \iff 1) \times (1 \iff 1)\) is defined by
\[
(s_0 \circ Z_1 s_1).b = \{ \circ Z_0 \text{ and } Z_1 \text{ are disjoint sets} \}
\]
\[
s_0.b \circ Z_1 s_1.b
\]
or equivalently:
\[
S_0 \circ Z_1 S_1 = \{ \circ Z_0 \text{ and } Z_1 \text{ are disjoint sets} \}
\]
\[
(s_0 \circ (\hat{\Pi} \circ \text{ const.} Z_1)) \cap (s_1 \circ (\hat{\Pi} \circ \text{ const.} Z_0))
\]

For the example of section 8.5.1 we have
\[
qc \circ c \circ d = c' \circ c \cap d' \circ d
\]
where \(C = 1 \iff \{c\}\) and \(D = 1 \iff \{d\}\).

### 8.5.5 Override

We call a statement \(s'\) a **statement-\(Z\)-override** of a statement \(s\) if for all behaviours \(b\), \(s'.b\) is equal to \(s.b\) for inputs outside the set \(Z\). We define the is-statement-\(Z\)-override-of \(\hat{\circ} Z \in \mathbb{B} \iff ((1 \iff 1) \iff 1) \times ((1 \iff 1) \iff 1)\) by
or equivalently:

\[ s' \circ_Z s \equiv s' \circ \text{const.}(I \setminus Z) = s \circ \text{const.}(I \setminus Z) \]

If for disjoint sets \(Z_0\) and \(Z_1\), a statement \(s_0\) is a statement-\(Z_0\)-override of some statement \(s\) and a statement \(s_1\) is a statement-\(Z_1\)-override of that same \(s\), then the statement-(\(Z_0, Z_1\))-component composition of \(s_0\) and \(s_1\) equals \(s_0\) for inputs from \(Z_0\), \(s_1\) for inputs from \(Z_1\) and \(s\) for all other inputs:

\[
{s_0}_{Z_0} \circ_{Z_1} {s_1} = (s_0 \circ \text{const.}(Z_0)) \cup (s_1 \circ \text{const.}(Z_1)) \cup (s \circ \text{const.}(I \setminus Z_0 \cup Z_1))
\]

\[
\equiv \{ \bullet Z_0 \text{ and } Z_1 \text{ are disjoint sets} \}
\]

\[
s_0 \circ_{Z_0} s \land s_1 \circ_{Z_1} s
\]

A trivial consequence is that it is a statement-\(Z_0 \cup Z_1\)-override of \(s\):

\[
(s_0 \circ_{Z_0} \cup \circ_{Z_1} s_1) \equiv (s_0 \circ \text{const.}(Z_0)) \cup (s_1 \circ \text{const.}(Z_1)) \cup (s \circ \text{const.}(I \setminus Z_0 \cup Z_1))
\]

\[
\equiv \{ \bullet Z_0 \text{ and } Z_1 \text{ are disjoint sets} \}
\]

\[
s_0 \circ_{Z_0} s \land s_1 \circ_{Z_1} s
\]

For the example we have

\[
qc \circ_{C} q \land qd \circ_{D} q
\]

and thus

\[
(qc \circ_{C} \cup \circ_{D} qd) \circ_{C \cup D} q
\]

The component-composition operators \(\circ\) and \(\hat{\circ}\) treat components in a symmetric way. Two operators that are closely related to the component-composition operators, but that are more suitable for the asymmetric treatment that is to follow, are the override operators.

For a set \(Z\), the \(Z\)-override operator \(\_\circ_{Z} \_\) \((I \longleftarrow 1) \longleftarrow (I \longleftarrow 1) \times (I \longleftarrow I)\) is defined by

\[
S \circ_Z S' = S \times (I \longleftarrow Z) \cup S' \times Z
\]

and the statement-\(Z\)-override operator \(\_\hat{\circ}_{Z} \_\) \(((I \longleftarrow 1) \longleftarrow 1) \longleftarrow ((I \longleftarrow 1) \longleftarrow 1) \times ((I \longleftarrow I) \longleftarrow 1)\) is defined by

\[
(s \hat{\circ}_Z s').b = s.b \circ_Z s'.b
\]

or equivalently:

\[
189
\]
These two operators satisfy several algebraic properties like idempotency and associativity (for a fixed $Z$). We address these properties on a need-to-know basis. The following theorem shows the close relationship between $\hat{\circ}$ and $\hat{\circ}^\dagger$:

$$s_0 \hat{\circ} Z_0 \hat{\circ} Z_1 s_1 = \{\bullet Z_0 \text{ and } Z_1 \text{ are disjoint sets, } s_0 \hat{\circ} Z_0 s \land s_1 \hat{\circ} Z_1 s \text{ for some } s\}$$

8.5.6 Statement-refinement

We call a statement $s'$ a statement-$Z$-refinement of a statement $s$ if $s'$ is a statement-refinement and a statement-$Z$-override of $s$. We define the is-statement-$Z$-refinement-of $\hat{\subseteq}^Z \in \mathbb{B} \leftarrow ((I \leftarrow I) \leftarrow (I \leftarrow I)) \times ((I \leftarrow I) \leftarrow (I \leftarrow I))$ by

$$s' \hat{\subseteq}^Z s \equiv \forall \langle b_1 \rangle \ s', b \subseteq Z \ s, b$$

or equivalently:

$$s' \hat{\subseteq}^Z s \equiv s' \hat{\subseteq} s \land s' \hat{\circ}^Z s$$

We leave it to the reader to verify that for disjoint sets $Z_0$ and $Z_1$, the statement-$(Z_0, Z_1)$-component composition of statements $s_0$ and $s_1$ is a statement-$Z_0 \cup Z_1$-refinement of a statement $s$ if $s_0$ is a statement-$Z_0$-refinement of $s$ and $s_1$ is a statement-$Z_1$-refinement of $s$:

$$\langle s_0 \hat{\circ} Z_0 Z_1 s_1 \rangle \hat{\subseteq} Z_0 \cup Z_1 s \equiv \{\bullet Z_0 \text{ and } Z_1 \text{ are disjoint sets}\}$$

8.5.7 Behavioural refinement

We call a program $s'$ a behavioural $Z$-refinement of a program $s$ if $s'$ is a behavioural refinement and a statement-$Z$-override of $s$. The is-behavioural-$Z$-refinement-of $\hat{\subseteq}^Z \in \mathbb{B} \leftarrow ((I \leftarrow I) \leftarrow (I \leftarrow I)) \times ((I \leftarrow I) \leftarrow (I \leftarrow I))$ is defined by

$$s' \hat{\subseteq}^Z s \equiv s' \hat{\subseteq} s \land s' \hat{\circ}^Z s$$

We now try to discover under which circumstances, for disjoint sets $Z_0$ and $Z_1$, the statement-$(Z_0, Z_1)$-component composition of programs $s_0$ and $s_1$ is a behavioural $Z_0 \cup Z_1$-refinement of a program $s$ if $s_0$ is a behavioural $Z_0$-refinement of $s$ and $s_1$
is a behavioural $Z_1$-refinement of $s$:

$$(s_0 Z_0 \cap Z_1 s_1) \subseteq Z_{0 \cup \cap} Z_1 s$$

$\Leftarrow \quad \{ \cdot Z_0 \text{ and } Z_1 \text{ are disjoint sets and } \ldots \}$

$s_0 \subseteq Z_0 s \land s_1 \subseteq Z_1 s$

For the example of section 8.5.1 we have

$$qc \subseteq C q \land qd \subseteq D q$$

and also

$$(qc C \cap D) \subseteq C \cup D q$$

The following example shows however that the “???” is not equivalent to true. Suppose program $r$ is defined by

$$r = c \triangleleft c \cap d \triangleleft d \cap e \triangleleft e$$

$c = \text{const.}[C \geq 3]$  
$d = \text{const.}[C \geq 4]$  
$e = \text{call}\triangleleft e$

The sets of input values $CD = I \rightarrow \{c, d\}$ and $E = I \rightarrow \{e\}$ describe two disjoint parts of this program. A behavioural $CD$-refinement of $r$ is the program $rcd$, defined by

$$rcd = c_1 \triangleleft c \cap d_1 \triangleleft d \cap e \triangleleft e$$

$c_1 = \text{const.}[C = 4]$  
$d_1 = \text{call}\triangleleft e$

and a behavioural $E$-refinement of $r$ is the program $re$, defined by

$$re = c \triangleleft c \cap d \triangleleft d \cap e_1 \triangleleft e$$

$e_1 = \text{const.}[C = 3]$  

The statement-$(CD, E)$-component composition of $rcd$ and $re$ is equal to $r_1$, defined by

$$r_1 = c_1 \triangleleft c \cap d_1 \triangleleft d \cap e_1 \triangleleft e$$

which is not a behavioural $CD \cup E$-refinement of $r$. The use of call expansion gives us that $r_1$’s canonical behaviour is equal to the canonical behaviour of $r_1’$ defined by
\[ r_1' = c_1 \triangleleft c \cap d_1' \triangleleft d \cap c_1 \triangleleft e \]
\[ d_1' = \text{const.}[\hat{C} = 3] \]

This is obviously not a behavioural refinement of \( r \).

The root cause of the problem is the mutual dependency between the two components that are composed. In general, such a mutual dependency does not allow us to behaviourally refine components in isolation if \( \hat{\circ} \) is used as composition operator. It is however allowed to behaviourally refine one component if we statement-refine the other:

\[ (s_0 \hat{\circ} Z_1 s_1) \subseteq Z_0 \cup Z_1 s \]
\[ \Leftarrow \quad \bullet Z_0 \text{ and } Z_1 \text{ are disjoint sets} \]
\[ s_0 \subseteq Z_0 s \land s_1 \subseteq Z_1 s \]

**proof**

The override part of the proof can be found in section 8.5.5. For the behavioural-refinement part, assuming \( Z_0 \) and \( Z_1 \) to be disjoint sets, \( s_0 \) to be a statement-\( Z_0 \)-override of \( s \) and \( s_1 \) to be a statement-\( Z_1 \)-override of \( s \), we calculate:

\[ (s_0 \hat{\circ} Z_1 s_1) \subseteq s \]
\[ \equiv \quad \{ \text{relationship between } \hat{\circ} \text{ and } \hat{\circ} \text{ (see section 8.5.5)} \} \]
\[ (s_0 \hat{\circ} Z_1 s_1) \subseteq s \]
\[ \Leftarrow \quad \{ \text{transitivity of } \hat{\circ} \circ \bullet \} s_0 \subseteq s \]
\[ (s_0 \hat{\circ} Z_1 s_1) \subseteq s_0 \]
\[ \equiv \quad \{ \text{because } Z_0 \text{ and } Z_1 \text{ are disjoint sets and } s_0 \hat{\circ} Z_0 s, \text{ we have } s_0 = s_0 \hat{\circ} Z_0 s \} \]
\[ (s_0 \hat{\circ} Z_1 s_1) \subseteq (s_0 \hat{\circ} Z_1 s) \]
\[ \equiv \quad \{ \text{definition of } \hat{\circ} \circ \} \mu(s_0 \hat{\circ} Z_1 s_1) \subseteq \mu(s_0 \hat{\circ} Z_1 s) \]
\[ \Leftarrow \quad \{ \text{monotonicity of } \mu \} s_0 \hat{\circ} Z_1 s_1 \subseteq s_0 \hat{\circ} Z_1 s \]
\[ \Leftarrow \quad \{ \text{statement-monotonicity of } (x \hat{\circ} Z_1) \} s_1 \subseteq s \]

The last three steps are actually the core of the proof. The other steps reformulate the problem in terms of the override operator. We repeat this core as a theorem:

\[ (s_0 \hat{\circ} Z_1 s_1) \subseteq (s_0 \hat{\circ} Z_1 s) \]
\[ \Leftarrow \quad s_1 \subseteq s \]
In the following subsection we show a restriction that allows us to behaviourally refine both components.

8.5.8 Independence

We call a set $Z$ an independent part of a statement $s$ if the behaviour of $s$ for inputs from $Z$ is not influenced by the way its context behaves on inputs outside $Z$:

$$\forall \langle b, b' \mid b \cdot Z = b' \cdot Z \mid s \cdot b = s \cdot b' \cdot Z \rangle$$

If we take the example of the previous section:

$$r = c \circ c \cap d \circ d \cap e \circ e$$
$$c = \text{const.} \lceil C \geq 3 \rceil$$
$$d = \text{const.} \lceil C \geq 4 \rceil$$
$$e = \text{call} \circ m$$

and consider in this case all combinations of the parts $C = I \rightarrow \{c\}$, $D = I \rightarrow \{d\}$ and $E = I \rightarrow \{e\}$, then $C$, $D$, $C \cup D$, $C \cup E$ and $C \cup D \cup E$ are independent parts of $r$ and $E$ and $D \cup E$ are not. If we only use the statement operators of section 8.2 to construct statements then, informally stated, independence of a part can be ensured by having no calls from that part to another part.

The main theorem that we pursue, is that for disjoint sets $Z_0$ and $Z_1$, the statement-(2)-component composition of programs $s_0$ and $s_1$ is a behavioural $Z_0 \cup Z_1$-refinement of a program $s$ if $s_0$ is a behavioural $Z_0$-refinement of $s$, $s_1$ is a behavioural $Z_1$-refinement of $s$ and $Z_1$ is an independent part of $s$ and $s_1$:

$$(s_0 \widehat{\circ} Z_0 \circ Z_1 \circ Z_1) \subseteq Z_0 \cup Z_1 \circ s$$

$\Leftarrow$ $\{Z_0$ and $Z_1$ are disjoint sets and $Z_1$ is an independent part of $s$ and $s_1\}$

$s_0 \subseteq Z_0 \circ s \land s_1 \subseteq Z_1 \circ s$

Similar to the theorem of the previous subsection, this theorem is a trivial consequence of the following theorem that is a reformulation of the theorem in terms of the statement-override operator:

$$(s_0 \circ Z_1 \circ s_1) \subseteq (s_0 \circ Z_1 \circ s)$$

$\Leftarrow$ $\{Z_1$ is an independent part of $s$ and $s_1\}$

$s_1 \subseteq s$

To prove this theorem, we prove some lemmas that enable us to replace an independent part of a program by the ‘const’ of its canonical behaviour. We can then use that statement-refining this ‘const’ is the same as behaviourally refining the independent part.
The first lemma says that if \( Z \) is an independent part of a program \( s \), then the canonical behaviour of \( s \) restricted to inputs from \( Z \), is equal to the canonical behaviour of \( s \circ \text{const.}Z \):

\[
\mu s \circ Z = \mu (s \circ \text{const.}Z) \quad \iff \quad Z \text{ is an independent part of } s
\]

**proof**

\[
\begin{align*}
\mu s \circ Z &= \mu (s \circ \text{const.}Z) \\
&\iff (\text{fusion with } F = (Z), H = s \text{ and } K = s \circ \text{const.}Z) \\
&\quad \forall \langle b \mid s.b \circ Z = (s \circ \text{const.}Z) \circ (b \circ Z) \rangle \\
&\quad \quad (\text{definition of } \circ \text{ and const}) \\
&\quad \forall \langle b \mid s.b \circ Z = s.(b \circ Z) \circ Z \rangle \\
&\quad \quad (\bullet \text{ } Z \text{ is an independent part of } s) \\
&\quad \forall \langle b \mid b \circ Z = b \circ Z \circ Z \rangle \\
&\quad \quad (Z \text{ is a set, so } Z \circ Z = Z) \\
&\quad \text{true}
\end{align*}
\]

The second lemma states that if \( Z \) is an independent part of a program \( s \), then the canonical behaviour of \( s \) is equal to the canonical behaviour of the result of statement-\( Z \)-overriding \( s \) with the \text{const} of the canonical behaviour of \( s \):

\[
\mu s = \mu (s \circ Z \text{ const.}\mu s) \quad \iff \quad Z \text{ is an independent part of } s
\]

**proof**

We split the proof into \( \supseteq \) and \( \subseteq \). For the \( \supseteq \) we use induction, showing that \( \mu s \) is a fixed point of \( (s \circ Z \text{ const.}\mu s) \):

\[
\begin{align*}
(s \circ Z \text{ const.}\mu s).\mu s &= (s \circ Z \text{ const.}\mu s).\mu s \\
&= (s, \mu s) \circ Z (\text{const.}\mu s).\mu s \\
&= (\text{computation, definition of const}) \\
&\quad \mu s \circ Z \mu s \\
&= (\text{idempotency of } \circ_Z) \\
&\quad \mu s
\end{align*}
\]

For the \( \subseteq \) we calculate

194
\[ \mu s \subseteq \mu (s \diamond Z \text{ const.} \mu s) \]
\[ \Leftarrow \{ \mu g \subseteq \mu h \text{ if each fixed point of } h \text{ is a fixed point of } g \} \]
\[ \forall \langle b \mid (s \diamond Z \text{ const.} \mu s), b = b \mid s.b = b \rangle \]
\[ \Leftarrow \{ \text{definition of } \diamond \} \]
\[ \forall \langle b \mid s.b \diamond Z (\text{const.} \mu s), b = b \mid s.b = b \rangle \]
\[ \Leftarrow \{ \text{definition of const} \} \]
\[ \forall \langle b \mid s.b \diamond Z \mu s = b \mid s.b = b \rangle \]
\[ \Leftarrow \{ \text{Leibniz} \} \]
\[ \forall \langle b \mid s.b \diamond Z \mu s = b \mid s.b = s.b \diamond Z \mu s \rangle \]
\[ \Leftarrow \{ S = S \diamond Z \Rightarrow S = S \diamond Z \} \]
\[ \forall \langle b \mid s.b \diamond Z \mu s = b \mid s.b \diamond Z = \mu s \diamond Z \rangle \]
\[ \Leftarrow \{ \text{Z is an independent part of } s, \text{ then } s.b \diamond Z = s.\mu s \diamond Z \text{ follows from } b.Z = \mu s \diamond Z \text{ which again follows from } b = s.b \diamond Z \mu s \} \]
\[ \forall \langle b \mid s.b \diamond Z \mu s = b \mid s.\mu s \diamond Z = \mu s \diamond Z \rangle \]
\[ \Leftarrow \{ \text{computation} \} \]
\[ \text{true} \]

Using these two lemmas, we prove a third one that we call the isolation lemma:

\[ \mu (s \diamond Z s') = \mu (s \diamond Z \text{ const.} \mu s') \Leftarrow Z \text{ is an independent part of } s' \]

**proof**

We assume that \( Z \) is an independent part of \( s' \). A trivial consequence is that it then also is an independent part of \( s \diamond Z s' \). We now calculate:
\[
\mu(s \hat{\otimes}_Z s')
= \{\text{second lemma, } Z \text{ is an independent part of } s \hat{\otimes}_Z s'\}
\]
\[
\mu(s \hat{\otimes}_Z s' \hat{\otimes}_Z \text{const.}\mu(s \hat{\otimes}_Z s'))
= \{x_0 \hat{\otimes}_Z x_1 \hat{\otimes}_Z x_2 = x_0 \hat{\otimes}_Z x_2\}
\]
\[
\mu(s \hat{\otimes}_Z \text{const.}\mu(s \hat{\otimes}_Z s'))
= \{x \hat{\otimes}_Z x' = x \hat{\otimes}_Z (x' \hat{\otimes} \text{const.}Z)\}
\]
\[
\mu(s \hat{\otimes}_Z (\text{const.}\mu(s \hat{\otimes}_Z s') \hat{\otimes} \text{const.}Z))
= \{\text{const.}S_0 \hat{\otimes} \text{const.}S_1 = \text{const.}(S_0 \cap S_1)\}
\]
\[
\mu(s \hat{\otimes}_Z \text{const.}(\mu(s \hat{\otimes}_Z s') \hat{\otimes} Z))
= \{\text{first lemma, } Z \text{ is an independent part of } s \hat{\otimes}_Z s'\}
\]
\[
\mu(s \hat{\otimes}_Z \text{const.}(\mu((s \hat{\otimes}_Z s') \hat{\otimes} \text{const.}Z)))
= \{(x \hat{\otimes}_Z x') \hat{\otimes} \text{const.}Z = x' \hat{\otimes} \text{const.}Z\}
\]
\[
\mu(s \hat{\otimes}_Z \text{const.}(\mu(s' \hat{\otimes} \text{const.}Z)))
= \{\text{first lemma, } Z \text{ is an independent part of } s'\}
\]
\[
\mu(s \hat{\otimes}_Z \text{const.}(\mu s' \hat{\otimes} Z))
= \{\text{const.}(S_0 \cap S_1) = \text{const.}(S_0 \hat{\otimes} \text{const.}S_1)\}
\]
\[
\mu(s \hat{\otimes}_Z (\text{const.}\mu s' \hat{\otimes} \text{const.}Z))
= \{x \hat{\otimes}_Z (x' \hat{\otimes} \text{const.}Z) = x \hat{\otimes}_Z x'\}
\]
\[
\mu(s \hat{\otimes}_Z \text{const.}\mu s')
\]

With this isolation lemma, we are now able to prove the theorem that we were after:

\[
(s_0 \hat{\otimes}_Z s_1) \hat{\subseteq} (s_0 \hat{\otimes}_Z s)
\]
\[
\Rightarrow \{\bullet Z_1 \text{ is an independent part of } s_1 \text{ and } s\}
\]
\[
s_1 \hat{\subseteq} s
\]

**proof**

\[
(s_0 \hat{\otimes}_Z s_1) \hat{\subseteq} (s_0 \hat{\otimes}_Z s)
\]
\[
\Rightarrow \{\text{definition of } \hat{\subseteq}\}
\]
\[
\mu(s_0 \hat{\otimes}_Z s_1) \subseteq \mu(s_0 \hat{\otimes}_Z s)
\]
\[
\Rightarrow \{\text{isolation lemma, } \bullet Z_1 \text{ is an independent part of } s_1 \text{ and } s\}
\]
\[
\mu(s_0 \hat{\otimes}_Z \text{const.}\mu s_1) \subseteq \mu(s_0 \hat{\otimes}_Z \text{const.}\mu s)
\]
\[
\Rightarrow \{\text{section 8.5.7}\}
\]
\[
\text{const.}\mu s_1 \hat{\subseteq} \text{const.}\mu s
\]
\[
\Rightarrow \{\text{simple refinement rule for const}\}
\]
\[
\mu s_1 \subseteq \mu s
\]
\[
\Rightarrow \{\text{definition of } \hat{\subseteq}\}
\]
\[
s_1 \hat{\subseteq} s
\]
As already mentioned, the following theorem is a trivial consequence:

\[
(s_0, Z_0 \hat{\circ} Z_1, s_1) \subseteq_{Z_0 \cup Z_1, s} s
\]

\[\iff\]

\{• Z_0 and Z_1 are disjoint sets and • Z_1 is an independent part of s and s_1\}

\[s_0 \subseteq_{Z_0, s} s \land s_1 \subseteq_{Z_1, s} s\]

If we have a look at the example of section 8.5.1 and the parts \(M = I \mapsto \{m\}\) and \(N = I \mapsto \{n\}\), then we see that \(M\) is an independent part of \(p\) and \(pm\). This means that the fact that \(pm\) is a behavioural \(M\)-refinement of \(p\) and that \(pn\) is a behavioural \(N\)-refinement of \(p\), gives us that the statement-(\(M, N\))-component composition of \(pm\) and \(pn\) is a behavioural \(M\cup N\)-refinement of \(p\).

If we compose two components with \(\hat{\circ}\), it is not necessary that one of them is an independent part to be able to apply the main theorem of this section. As an example, we again take program \(MATH\), defined by

\[
MATH = Crease1 \hat{\circ} Crease \cap Three \hat{\circ} Three \cap four \hat{\circ} four
\]

\[Crease1 = inc1 \hat{\circ} \text{inc} \cap dec1 \hat{\circ} \text{dec}\]

\[Three = direct3 \hat{\circ} \text{direct} \cap indirect3 \hat{\circ} \text{indirect}\]

\[inc1 = \text{const.}[\hat{C} = \hat{C} + 1]\]

\[dec1 = \text{const.}[\hat{C} = \hat{C} - 1]\]

\[direct3 = (\text{call}\hat{Crease})\hat{\circ} \text{dec} \hat{\circ} \text{const.}[\hat{C} = 4]\]

\[indirect3 = (\text{call}\hat{Crease})\hat{\circ} \text{dec} \hat{\circ} \text{call}\hat{four}\]

\[four = (\text{call}\hat{Crease})\hat{\circ} \text{inc} \hat{\circ} (\text{call}\hat{Three})\hat{\circ} \text{direct}\]

Now, consider the three parts

\[Z_1 = I \mapsto \{\text{Crease}\}\]

\[Z_3 = I \mapsto \{\text{Three}\}\]

\[Z_4 = I \mapsto \{\text{four}\}\]

Part \(Z_1\) clearly is an independent part of \(MATH\). However, the parts \(Z_3\) and \(Z_4\) depend on each other, so we cannot apply the main theorem to them. The crux is to consider the parts

\[Z_{30} = (I \mapsto \{\text{direct}\}) \mapsto \{\text{Three}\}\]

\[Z_{31} = (I \mapsto \{\text{indirect}\}) \mapsto \{\text{Three}\}\]

\[Z_4 = I \mapsto \{\text{four}\}\]

We see that \(Z_1 \cup Z_{30}\) and \(Z_1 \cup Z_{30} \cup Z_4\) are independent parts of \(MATH\). Behaviourally refining \(Z_3\) as a whole is not allowed, but behaviourally refining the individual parts \(Z_1, Z_{30}, Z_{31}\) and \(Z_4\) is allowed, as long as \(Z_1, Z_1 \cup Z_{30}\) and \(Z_1 \cup Z_{30} \cup Z_4\) remain independent parts:
This recipe can be used to handle many practical situations. Suppose that we have a program $s$ and a set of disjoint parts $Z \in \wp(\wp I)$. A $Z$-preorder $D \in Z \leftarrow I$ is called a dependency relation of $(s, Z)$ if for each part $Z \in Z$, the union of all parts that $D$ connects with input $Z$, is an independent part of $s$:

$$\forall Z \mid Z \in Z \mid \bigcup(D \uparrow Z)$$

The dependency relation that we used for our example looks as follows (we omitted the arrows from a part to itself):

An independent part in this graph is a collection of nodes without outgoing arrows.

If $Z$ is finite and the dependency relation is a $Z$-partial-order (meaning that its graph contains no cycles), then all parts in $Z$ may be behaviourally refined in isolation. Using the main theorem of this section, this can be proved by induction in the way suggested by the above calculation.

The main message is that in the context of statement-component composition (the way components are composed in practice) mutual dependencies should be avoided as much as possible. A simple solution could be to treat mutually dependent parts...
as one part and behaviourally refine this part as a whole. We may still refine parts of this whole in isolation by choosing one part of this whole that we really want to behaviourally refine, and statement-refine the other parts of this whole (using the theorem of section 8.5.7). Although these solutions might be sufficiently powerful for many practical situations, they also might not be for some cases. The main problem seems to be the fact that a call to a procedure may be replaced by the specification of that procedure. This means that the information that it is a call to that procedure is lost. We already mentioned in section 8.4.5 that in the next chapter, we introduce a model where calls can be made visible. The visibility of calls between mutually dependent parts might bring us closer to behavioural refinement of mutually dependent programs in isolation. We leave that for future research however.

8.6 Totality

In this section we investigate the addition of totality requirements on refinement between statements.

8.6.1 Total statement-refinement

A statement \( s' \) is a total statement-refinement of a statement \( s \) if for each context \( b \), \( s'.b \) is a total refinement of \( s.b \). Formally, the is-total-statement-refinement-of \( \triangleleft \subseteq \) is defined by

\[
\forall (b \mid s'.b \subseteq s.b)
\]

8.6.2 Total behavioural refinement

A program \( s' \) is a total behavioural refinement of a program \( s \) if the canonical behaviour of \( s' \) is a total refinement of that of \( s \). The is-total-behavioural-refinement-of \( \triangleleft \subseteq \) is defined by

\[
\forall (b \mid \mu s'.b \subseteq \mu s)
\]

8.6.3 Example

In section 8.5.7 we saw that the addition of recursion already introduces some problems in the field of components. These problems were caused by mutual dependencies. Switching from partial refinement to total refinement introduces some new challenges that are unrelated to mutual dependencies.
We again take the $q$, $qc$ and $qd$ of section 8.5.1:

\[
q = c \triangleleft c \land d \triangleleft \bar{d}
\]
\[
c = \text{const.}[\hat{C} \geq 3]
\]
\[
d' = \text{const.}[\hat{C} \geq 4]
\]
\[
qc = c' \triangleleft c \land d' \triangleleft \bar{d}
\]
\[
c' = \text{const.}[\hat{C} = 3]
\]
\[
qd = c \triangleleft c \land d \triangleleft \bar{d}
\]
\[
d' = \text{const.}[\hat{C} = \hat{C}+1] \triangleright \text{call} \triangleright \bar{c}
\]

The statement-$(C, D)$-component composition of $qc$ and $qd$, where $C = I \mapsto \{c\}$ and $D = I \mapsto \{d\}$:

\[
qc \hat{\odot}_D qd = c' \triangleleft c \land d' \triangleleft \bar{d}
\]

‘clearly’ is a total behavioural refinement of $q$. One might be inclined to think that, similar to section 7.8.4, this is a trivial consequence of the fact that $qc$ and $qd$ are total behavioural refinements of $q$. However, another total behavioural refinement of $q$ is $qd_2$, defined by

\[
qd_2 = c \triangleleft c \land d_2 \triangleleft \bar{d}
\]
\[
d_2 = \text{const.}[\hat{C} \geq 4] \triangleright \text{call} \triangleright \bar{c}
\]

Its statement-$(C, D)$-component composition with $qc$:

\[
qc \hat{\odot}_D qd_2 = c' \triangleleft c \land d_2 \triangleleft \bar{d}
\]

is not a total behavioural refinement of $q$. The problem is that after statement-component composition, the effect of a call to $c$ changes in a way ‘the designers of $qd_2$ did not foresee’. The call expansion rule shows the exact problem. It tells us that the canonical behaviour of $qd_2$ is equal to the canonical behaviour of $qd_2'$, defined by

\[
qd_2' = c \triangleleft c \land d_2' \triangleleft \bar{d}
\]
\[
d_2' = \text{const.}[\hat{C} \geq 4] \triangleright \text{const.}[\hat{C} \geq 3]
\]

or simplifying the definition of $d_2'$:

\[
qd_2' = c \triangleleft c \land d_2' \triangleleft \bar{d}
\]
\[
d_2' = \text{const.}[\hat{C} \geq 4]
\]

In other words, $\mu qd_2$ equals $\mu q$. The canonical behaviour of $(qc \hat{\odot}_N qd_2)$ is however equal to the canonical behaviour of $qcd_2$, defined by
\[ \text{qcd}_2 = c' \circ \text{c} \cap d_2'' \circ \text{d} \]
\[ d_2'' = \text{const.}[C \geq 4] \circ \text{const.}[\text{c'} = 3] \]

or simplifying the definition of \( d_2'' \):
\[ \text{qcd}_2 = c' \circ \text{c} \cap d_2'' \circ \text{d} \]
\[ d_2'' = \emptyset \]

which clearly is not a total behavioural refinement of \( q \).

### 8.6.4 Conservative users

In this section we investigate the totality issue of the previous subsection. We do not consider mutual dependencies, assuming that we have a team A writing a statement \( s \) that makes use of some total refinement \( b' \) that a team B is developing of some specification \( b \). The issue is that team A does not exactly know what team B's \( b' \) looks like. Team A only has an abstract description \( b \) and knows that the concrete \( b' \) is a total refinement of that description. Of course team A wants its statement \( s \) to 'work' for each possible total refinement that team B can come up with. With 'work' we mean that for every total refinement \( b' \) of \( b \), \( s.b' \) should be a total refinement of \( s.b \):

\[ \forall \langle b' | b' \subseteq b | s.b' \subseteq s.b \rangle \]

If this property holds, we call \( s \) a **conservative user** of \( b \).

If we would only demand partial refinement, we would obtain the property

\[ \forall \langle b' | b' \subseteq b | s.b' \subseteq s.b \rangle \]

which we get for free if \( s \) is monotonic. From section 8.2.9 we know that monotonicity of \( s \) is not hard to achieve. However, monotonicity alone is not enough in the context of total refinement. In the following calculation we isolate the extra property that we need besides monotonicity:

\[ \forall \langle b' | b' \subseteq b | s.b' \subseteq s.b \rangle \]

\[ \equiv \{ \text{definition of } \subseteq \} \]
\[ \forall \langle b' | b' \subseteq b | s.b' \subseteq s.b \wedge (s.b') \supseteq (s.b) \rangle \]

\[ \equiv \forall \langle b' | b' \subseteq b | s.b' \subseteq s.b \rangle \wedge \forall \langle b' | b' \subseteq b | (s.b') \supseteq (s.b) \rangle \]

\[ \equiv \{ \bullet \text{monotonicity of } s \} \]
\[ \forall \langle b' | b' \subseteq b | (s.b') \supseteq (s.b) \rangle \]

This extra property is not so trivial to accomplish. Take for example the specification \( b_9 \) that outputs an arbitrary integer number that is less than 9:

201
Statement \( s_6 \), defined by

\[
s_6 = \text{const.} [ \hat{c} \in \mathbb{Z} \land \hat{c} \leq 6 \land \hat{c} \geq \hat{c} ] \odot \text{call}
\]

is not a conservative user of \( b_9 \), although \( s_6.b_9 \) is total. For the total refinement \( b_8 \) of \( b_9 \), defined by

\[
b_8 = [ \hat{c} = 8 ]
\]

\( s_6.b_8 \) is equal to \( \emptyset \), which clearly is not total.

Another way to show that \( s_6 \) is not a conservative user of \( b_9 \), is to derive a \( b' \) that can act as counter example. Assuming that \( b' \sqsubseteq b_9 \), we calculate:

\[
s_6.b' \sqsubseteq s_6.b_9
\]

\[
\equiv \quad \text{monotonicity of } s_6
\]

\[
(s_6.b') \supseteq (s_6.b_9)
\]

\[
\equiv \quad \{(s_6.b_9) = \emptyset\}
\]

\[
(s_6.b') \supseteq \emptyset
\]

\[
\equiv \quad \forall \langle z \mid \exists \langle y \mid y (s_6.b') z \rangle\rangle
\]

\[
\equiv \quad \text{definition of } s_6
\]

\[
\forall \langle z \mid \exists \langle y, x \mid y (\hat{c} \in \mathbb{Z} \land \hat{c} \leq 6 \land \hat{c} \geq \hat{c} \odot b') z \rangle\rangle
\]

\[
\equiv \quad \text{definition of } \odot
\]

\[
\forall \langle z \mid \exists \langle y, x \mid y (\hat{c} \in \mathbb{Z} \land \hat{c} \leq 6 \land \hat{c} \geq \hat{c} \odot x \odot (b') z) \rangle\rangle
\]

\[
\equiv \quad \{ (=) : \ x \leq y \land y \leq 6 \Rightarrow x \leq 6, (=): \text{take } y = x\}
\]

\[
\forall \langle z \mid \exists \langle x \mid x \in \mathbb{Z} \land x \leq 6 \land x \odot (b') z \rangle\rangle
\]

So, by assuming \( s_6.b' \) to be a total refinement of \( s_6.b_9 \), we derived that each total refinement \( b' \) of \( b_9 \) should satisfy

\[
\forall \langle z \mid \exists \langle x \mid x \in \mathbb{Z} \land x \leq 6 \land x \odot (b') z \rangle\rangle
\]

This property clearly does not hold for every total refinement of \( b_9 \) (\( b_8 \) provides again a counter example).

In general, for a specification \( b \) and a statement \( s \), we have that \( s \) is not a conservative user of \( b \) exactly when there exists a total predicate \( p \) that holds for each specification \( b' \) for which \( s.b' \) is a total refinement of \( s.b \), but that does not hold for each total refinement of \( b \):
\[\neg \forall \langle b' \mid b' \sqsubseteq b \mid s.b' \sqsubseteq s.b \rangle \equiv \exists \langle p \mid p \in B \nleftrightarrow (I \Rightarrow I) \mid \forall \langle b' \mid s.b' \sqsubseteq s.b \mid p.b' \rangle \land \neg \forall \langle b' \mid b' \sqsubseteq b \mid p.b' \rangle \rangle\]

Taking the negation of both sides of the equivalence, we see that this is equivalent to the fact that \( s \) is a conservative user exactly when each total predicate \( p \) that holds for each specification \( b' \) for which \( s.b' \) is a total refinement of \( s.b \), also holds for each total refinement of \( b \):

\[\forall \langle b' \mid b' \sqsubseteq b \mid s.b' \sqsubseteq s.b \rangle \equiv \forall \langle p \mid p \in B \nleftrightarrow (I \Rightarrow I) \mid \forall \langle b' \mid s.b' \sqsubseteq s.b \mid p.b' \rangle \Rightarrow \forall \langle b' \mid b' \sqsubseteq b \mid p.b' \rangle \rangle\]

The \((\Rightarrow)\) is trivial. For \((\Leftarrow)\), take the total predicate \( p \) defined by

\[p.b' \equiv s.b' \sqsubseteq s.b\]

A third way to look at conservativity, is that the set of all total refinements of \( b \) for which \( s \) ‘works’, should be equal to the set of all total refinements of \( b \). In other words:

\[\forall \langle b' \mid b' \sqsubseteq b \mid s.b' \sqsubseteq s.b \rangle \equiv \{b' \mid b' \sqsubseteq b \land s.b' \sqsubseteq s.b \mid b' \} = \{b' \mid b' \sqsubseteq b \mid b' \}\]

This is a trivial result of the fact that \((x \Rightarrow y) \equiv (x \land y \equiv x)\)

We thus found three equivalent formulations of conservativity:

\[\forall \langle b' \mid b' \sqsubseteq b \mid s.b' \sqsubseteq s.b \rangle \equiv \forall \langle p \mid p \in B \nleftrightarrow (I \Rightarrow I) \mid \forall \langle b' \mid s.b' \sqsubseteq s.b \mid p.b' \rangle \Rightarrow \forall \langle b' \mid b' \sqsubseteq b \mid p.b' \rangle \rangle\]

\[\equiv \{b' \mid b' \sqsubseteq b \land s.b' \sqsubseteq s.b \mid b' \} = \{b' \mid b' \sqsubseteq b \mid b' \}\]

Notice that the equivalence of these three notions is actually independent of the refinement relation that we choose (\(\sqsubseteq\) in the above case).

The reason why we focus on these three formulations in particular, is that these are all fundamental notions in ISpec. To clarify this somewhat, we introduce a few ad-hoc notions. We read \((s, b)\) as “specification \( b \) extended with \( s \)”, calling \( b \) the “base” and \( s \) the “extension”. Taking the identity function \( I \) for \( s \) means “no extension”. We define \((s, b) \models p\) (\( p \) is “valid” for \((s, b)\)) by

\[\neg \forall \langle b' \mid b' \sqsubseteq b \mid s.b' \sqsubseteq s.b \rangle \equiv \exists \langle p \mid p \in B \nleftrightarrow (I \Rightarrow I) \mid \forall \langle b' \mid s.b' \sqsubseteq s.b \mid p.b' \rangle \land \neg \forall \langle b' \mid b' \sqsubseteq b \mid p.b' \rangle \rangle\]
\[(s, b) \models p \equiv \forall (b' | s, b' \sqsubseteq s, b | p, b')\]

and \(B(s, b)\) (the “behaviour of the base” of \((s, b)\)) by

\[B(s, b) = \{ b' | b' \sqsubseteq b \land s, b' \sqsubseteq s, b | b' \}\]

We can now reformulate the above three equivalent formulations of conservativity as

\[\forall (b' | b' \sqsubseteq b | s, b' \sqsubseteq s, b)\]

\[\equiv \forall (p | p \in B \iff (I_\bowtie (I) | (s, b) \models p) \Rightarrow (I, b) \models p)\]

\[\equiv B(s, b) = B(I, b)\]

In ISpec, the first formulation would be referred to as “\(b\) is substitutable in \(s\)”, the second one would be referred to as “\(s\) is a conservative extension of \(b\)” and the third one would be referred to as “\(s\) does not influence the behaviour of \(b\)”.

We thus showed that these three notions are equivalent in the framework that is used here.

Now that we know that conservativity can be disproved by finding an appropriate counter-example predicate, the question naturally rises how to prove conservativity.

We again take our specification \(b_9\), defined by

\[b_9 = [\hat{\mathcal{C}} \in \mathbb{Z} \land \hat{\mathcal{C}} \leq 9]\]

The statement \(s_{12}\), defined by

\[s_{12} = \text{const.}[\hat{\mathcal{C}} \in \mathbb{Z} \land \hat{\mathcal{C}} \leq 12 \land \hat{\mathcal{C}} \geq \hat{\mathcal{C}}] : \text{call}\]

is a conservative user of \(b_9\). To prove conservativity, we usually need to combine totality properties with partial-refinement properties. This is where the type operator \(\rightarrow\) comes in. The meaning of \(S \in A \rightarrow B\) is that \(S\) connects each input that is an element of \(B\) to at least one element of \(A\) and only to elements of \(A\). Some conservativity rules that we can use, are that for sets \(A^{(s)}, B^{(s)}, C\) and \(G\):

\[
\begin{align*}
S & \in A \rightarrow B' \iff S \in A' \rightarrow B \land A' \subseteq A \land B' \subseteq B \\
S' & \in A \rightarrow B \iff S \in A \rightarrow B \land S' \subseteq S \\
\text{if } G \text{ then } S_0 & \in A \rightarrow B \iff S_0 \in A \rightarrow B \land S_0 \subseteq G \land S_1 \in A \rightarrow B \land S_1 \subseteq G \\
S_0, S_1 & \in A \rightarrow B \iff S_0 \in A \rightarrow C \land S_1 \in C \rightarrow B \\
B & \in B' \rightarrow B' \iff B' \subseteq B
\end{align*}
\]

Comparing these conservativity rules to the totality rules:
\[
S \in A \implies B' \iff S \in A' \implies B \land A' \subseteq A \land B' \subseteq B
\]
\[
S' \in A \implies B \iff S \in A \implies B \land S' \supseteq S
\]
\[
S_0 \cup S_1 \in A \implies B \cup D \iff S_0 \in A \implies B \land S_1 \in A \implies D
\]
\[
S_0 \cup S_1 \in A \implies B \iff S_0 \in A \implies C \land S_1 \in C \implies B
\]
\[
B \in B \implies B
\]

and the cylindric-type rules:
\[
S \in A \implies B' \iff S \in A' \implies B \land A' \subseteq A \land B' \subseteq B
\]
\[
S' \in A \implies B \iff S \in A \implies B \land S' \supseteq S
\]
\[
S_0 \cup S_1 \in A \implies B \iff S_0 \in A \implies B \land S_1 \in A \implies B
\]
\[
S_0 \cup S_1 \in A \implies B \iff S_0 \in A \implies C \land S_1 \in C \implies B
\]
\[
S_0 \cup S_1 \in A \implies B \iff S_0 \in A \implies C \land S_1 \in C \implies B
\]
\[
S \in A \implies B' \iff S \in A' \implies B \land A' \subseteq A \land B' \subseteq B
\]
\[
S' \in A \implies B \iff S \in A \implies B \land S' \supseteq S
\]
\[
S_0 \cup S_1 \in A \implies B \iff S_0 \in A \implies B \land S_1 \in A \implies B
\]
\[
S_0 \cup S_1 \in A \implies B \iff S_0 \in A \implies C \land S_1 \in C \implies B
\]
\[
S \in A \implies B' \iff S \in A' \implies B \land A' \subseteq A \land B' \subseteq B
\]
\[
S' \in A \implies B \iff S \in A \implies B \land S' \supseteq S
\]
\[
S_0 \cup S_1 \in A \implies B \iff S_0 \in A \implies B \land S_1 \in A \implies B
\]
\[
S_0 \cup S_1 \in A \implies B \iff S_0 \in A \implies C \land S_1 \in C \implies B
\]
\[
S \in A \implies B' \iff S \in A' \implies B \land A' \subseteq A \land B' \subseteq B
\]
\[
S' \in A \implies B \iff S \in A \implies B \land S' \supseteq S
\]
\[
S_0 \cup S_1 \in A \implies B \iff S_0 \in A \implies B \land S_1 \in A \implies B
\]
\[
S_0 \cup S_1 \in A \implies B \iff S_0 \in A \implies C \land S_1 \in C \implies B
\]

we see that at the places where the totality rules and cylindr ic-type rules differ, the conservativity rules show a way to integrate them: the inclusion rules have been integrated into a rule about total refinement, the union rules have been integrated into an if-then-else rule and the guard rules have been integrated into a generalised rule, where the generalisation compensates for the fact that the inclusion rules have been replaced by the total-refinement rule.

The following calculation shows how the conservativity rules can be used to prove that \( s_{12} \) is a conservative user of \( b_9 \):

\[
s_{12}.b' \in I \implies I
\]
\[
\equiv\{\text{definition of } s_{12}\}
\]
\[
(const.\llbracket\hat{C} \in \mathbb{Z} \land \hat{C} \leq 12 \land \hat{C} \geq \hat{C} \rrbracket \triangleright \text{call}).b' \in I \implies I
\]
\[
\equiv\{\text{definition of } \triangleright, \text{const and call}\}
\]
\[
\llbracket\hat{C} \in \mathbb{Z} \land \hat{C} \leq 12 \land \hat{C} \geq \hat{C} \rrbracket.b' \in I \implies I
\]
\[
\equiv\{\text{conservativity rule for } \implies\}
\]
\[
\llbracket\hat{C} \in \mathbb{Z} \land \hat{C} \leq 12 \land \hat{C} \geq \hat{C} \rrbracket.\llbracket\hat{C} \in \mathbb{Z} \land \hat{C} \leq 12 \rrbracket \land b' \in I \implies I
\]
\[
\equiv\{\text{first conjunct is left to the reader, conservativity rule for total refinement, } b' \subseteq b_9\}
\]
\[
\llbracket\hat{C} \in \mathbb{Z} \land \hat{C} \leq 12 \rrbracket \land b' \in I \implies I
\]
\[
\equiv\{\text{left to the reader}\}
\]
\[
true
\]

The conservativity rules show the fundamental issues but do not make up a powerful proof system. For that the total-correctness rules that were presented in section 7.8.3 seem the obvious candidate, but we have to leave this for future research.

How does all this relate to components and their composition? Suppose that we have a program with an independent part. Then the above conservativity story can be applied where the canonical behaviour of the independent part is the used specification and the rest of the program is the user. To be precise:
\[(s_0 \hat{\circ} Z_1, s_1) \sqsubseteq Z_0 \cup Z_1, s\]

\[\iff \begin{cases} 
\bullet Z_0 \text{ and } Z_1 \text{ are disjoint sets and } \\
\bullet Z_1 \text{ is an independent part of } s_1 \text{ and } s
\end{cases}
\]

\[\mu(s_0 \hat{\circ} Z_1, \text{const} \cdot b) \parallel b \text{ is a conservative user of } \mu s\]

\[s_0 \sqsubseteq Z_0, s \land s_1 \sqsubseteq Z_1, s\]

Similar to the sections 8.5.7 and 8.5.8, the core step of the proof is the following theorem:

\[(s_0 \hat{\circ} Z_1, s_1) \sqsubseteq (s_0 \hat{\circ} Z_1, s)\]

\[\iff \begin{cases} 
\bullet Z_1 \text{ is an independent part of } s_1 \text{ and } s
\end{cases}
\]

\[\mu(s_0 \hat{\circ} Z_1, \text{const} \cdot b) \parallel b \text{ is a conservative user of } \mu s\]

\[s_1 \sqsubseteq s\]

**proof**

\[(s_0 \hat{\circ} Z_1, s_1) \sqsubseteq (s_0 \hat{\circ} Z_1, s)\]

\[\equiv \{ \text{definition of } \sqsubseteq \} \]

\[\mu(s_0 \hat{\circ} Z_1, s_1) \sqsubseteq \mu(s_0 \hat{\circ} Z_1, s)\]

\[\equiv \{ \text{isolation lemma, } \bullet Z_1 \text{ is an independent part of } s_1 \text{ and } s \} \]

\[\mu(s_0 \hat{\circ} Z_1, \text{const} \cdot s_1) \sqsubseteq \mu(s_0 \hat{\circ} Z_1, \text{const} \cdot s)\]

\[\iff \{ \bullet (\mu(s_0 \hat{\circ} Z_1, \text{const} \cdot b) \parallel b \text{ is a conservative user of } \mu s \} \]

\[\mu s_1 \sqsubseteq \mu s\]

\[\equiv \{ \text{definition of } \sqsubseteq \} \]

\[s_1 \sqsubseteq s\]

The above theorems do not give a detailed path on how to prove total behavioural refinement in a modular fashion within our framework. That is left for future research, as is the investigation whether or not the theory of the next chapter could provide a more powerful theory that allows mutually dependent parts to be refined in isolation.

### 8.7 Conclusions

In this chapter we showed how recursion is modeled. We paid special attention to partial and total behavioural refinement and investigated component composition in these settings. We pointed out that the use of relations as model for behaviour has as a consequence that communication between components is invisible. This seems to be the main cause for the problem that we encountered with statement-component composition in section 8.5.7. To illustrate the problem in a more practical setting, we now reformulate it in terms of the observer pattern of section 2.2.4.

Suppose that the observer pattern is behaviourally refined by two teams in isolation where one team refines *Subject* and the other team refines *Observer*. Further-
more, the fact that \texttt{fillWith} really calls \texttt{update}, instead of only realising the given specification, cannot be expressed. A behavioural refinement of \texttt{Subject} would then be that \texttt{fillWith} updates the \texttt{Observer}'s \texttt{copy} by a different means than calling \texttt{update}. Suppose that in the behavioural refinement of \texttt{Observer}, method \texttt{insertInto} sets some boolean \texttt{b} to \texttt{true} before the call to \texttt{fillWith} and \texttt{update} sets \texttt{b} again to \texttt{false}. After the call to \texttt{fillWith}, \texttt{insertInto} now changes the value of \texttt{copy} into 481 if \texttt{b} is \texttt{true}. Both teams have constructed correct behavioural refinements of the observer pattern. However, in the statement-component composition of these refinements, the observer pattern’s invariant is violated in general. The reason is that \texttt{b} is not set to \texttt{false} when \texttt{insertInto} calls \texttt{fillWith}, because \texttt{fillWith} does not call \texttt{update}.

One could simply deal with the above problem by requiring that if one combines the implementations of several specifications by statement-component composition (called \texttt{implementation inheritance}), one has to prove that the combination is a behavioural refinement of (implements) all these specifications. This means that for each implementation we reuse, we have to construct a new correctness proof. The theory of section 8.5.8 shows that if we prove that there exist no mutual dependencies, these correctness proofs are not necessary. In other words, the theory enables modular reuse of implementations. In the next chapter we introduce a model where communication between components can be made visible. Whether or not visible communication can provide the fundament for a more powerful theory that even works when there exist mutual dependencies, is left for future research.
Chapter 9

Protocols

In the previous chapter we observed that it is not possible to specify ‘visible calls’ if relations are used as model for behaviour. In this chapter we present a generalisation of relations, called connection protocols, that can capture the extra information that is needed for this. Where a relation is a collection of connections, a connection protocol is a collection of so-called connection traces.

9.1 Definitions

In this section we introduce some external types that form the basis of connection protocols.

9.1.1 Sequences

We assume that there exists an external type of (finite) sequences. We do not define this external type formally, but limit ourselves to saying that we use the same notation for sequences as for tuples, except that we decorate them with a \( V \). A sequence of length \( n \) (\( n \in \mathbb{N} \)) that consists of the respective elements \( x_0, \ldots, x_{n-1} \) is denoted by

\[
\langle x_0, \ldots, x_{n-1} \rangle_V
\]

For a set \( A \), the set of sequences with elements of type \( A \) is denoted by \( VA \):

\[
\langle x_0, \ldots, x_{n-1} \rangle_V \in VA \equiv x_0, \ldots, x_{n-1} \in A
\]

The formula \( v \in V \) means that \( v \) is a sequence and thus that \( v = \langle x_0, \ldots, x_{n-1} \rangle_V \) for some \( x_0, \ldots, x_{n-1} \) for some \( n \in \mathbb{N} \). The formula itself is equivalent to \( \text{true} \).
The sequence \(\langle \rangle\) is called the **empty sequence**.

The **concatenation** \(\oplus\in VI\leftarrow VI \times VI\) pastes two sequences together:

\[
\langle y_0, \ldots, y_{n-1}\rangle_{\oplus} + \langle z_0, \ldots, z_{n-1}\rangle_{\oplus} = \langle y_0, \ldots, y_{n-1}, z_0, \ldots, z_{n-1}\rangle_{\oplus}
\]

The **number-of** \(\#\in (\mathbb{N} \leftarrow VI)\leftarrow I\) counts the number of occurrences of a certain element in a sequence:

\[
\begin{align*}
\#_{a} \langle \rangle_{\oplus} &= 0 \\
\#_{a} \langle x \rangle_{\oplus} &= 0 & \iff a \neq x \\
\#_{a} \langle x \rangle_{\oplus} &= 1 & \iff a = x \\
\#_{a} (v_0 \oplus v_1) &= \#_{a} v_0 + \#_{a} v_1
\end{align*}
\]

The **occurs-in** \(\in \in \leftarrow I \times VI\) checks whether or not a certain element occurs in a sequence:

\[
\begin{align*}
a \in \langle \rangle_{\oplus} &\equiv \text{false} \\
a \in \langle x \rangle_{\oplus} &\equiv a = x \\
a \in v_0 \oplus v_1 &\equiv a \in v_0 \lor a \in v_1
\end{align*}
\]

The **all-contained-by** \(<\equiv \leftarrow VI \times PI\) checks if all elements in a sequence are contained by a given collection:

\[
\forall (x \in v \mid x \in C)
\]

9.1.2 Traces

We also assume that there exists an external type of (finite) **traces**. A trace is denoted by \((x, v)_\tau\) where the **root element** \(x\) is an arbitrary element and the **visual** \(v\) is a sequence of traces.

A node trace of a trace is either the trace itself or a node trace of a trace in the visual:

\[
t \text{ is a node trace of } (x, v)_\tau \\
\equiv t = (x, v)_\tau \lor \exists (t' \mid \exists t' \text{ in } v \mid \exists t' \text{ is a node trace of } t')
\]

A node element of a trace is the root element of some node trace of the trace:

\[
x \text{ is a node element of } t \\
\equiv \exists (v \parallel (x, v)_\tau \text{ is a node trace of } t)
\]

For a set \(A\), the set of all traces with node elements of type \(A\), is denoted by \(TA\):
\[ t \in TA \equiv \forall (x \mid x \text{ is a node element of } t \mid x \in A) \]

An example of a trace of type \( \mathbb{T} \mathbb{N} \), consisting of 7 nodes, is

\[ (0, (1, (2, (\emptyset)_v)_r)_r, (2, (0, (\emptyset)_v)_r, (3, (\emptyset)_v)_r, (3, (\emptyset)_v)_r)_r \]

The two occurrences of \((3, (\emptyset)_v)_r\) are different nodes with the same node trace.

The following representation of the above trace is called a tree picture:

```
  0
 /\  \
(1, 2, 3)
  \  \
   2 0 3
```

### 9.1.3 Connection traces

Traces with connections as node elements are called \textbf{connection traces}. Connection traces thus have type \( \mathbb{T}(I \star I) \). Instead of \((y, z)_v\), we usually write \((y : v : z)_r\). We call \(y\) the \textbf{output} and \(z\) the \textbf{input} of this connection trace.

Two examples of connection traces are

\[ (0 : ((1 : (\emptyset)_v : 1)_r, \emptyset : (2)_r : (\emptyset)_v : 1)_r) \]

and

\[ (0 : ((1 : (1 : (\emptyset)_v : 0)_r : (\emptyset)_v : 2)_r : (\emptyset)_v : 1)_r) \]

In terms of tree pictures:

```
(0, 1)
  / \  \
(1 , 1) (0, 2)
```

for the first connection trace and

```
(0, 1)
  |  \
(1 , 2)
  |  \
(1, 0)
```

211
for the second one.

Another way to visualise connection traces is by means of a **connection-trace picture**. For the first connection trace, this picture is

![Connection Trace Picture 1]

and for the second connection trace it is

![Connection Trace Picture 2]

In these pictures a connection trace is divided into an **observation sequence**, \( \langle 0, 1, 1, 0, 2, 1 \rangle \), for both traces, and a **connection pattern**, \( \langle 0, 0, \text{ I }, \text{ I }, 0, \text{ I } \rangle \) and \( \langle 0, 0, \text{ O }, \text{ I }, \text{ I }, \text{ I } \rangle \) respectively. A connection pattern is not an arbitrary sequence of 0s and 1s. It is non-empty with an equal number of 0s and 1s and for each 0, the number of 1s on its right minus the number of 0s on its right is at least 1. Formally, a sequence \( l \in V \{0, 1\} \) is a connection pattern if

\[
\#_I \cdot l = \#_0 \cdot l \geq 1
\]

and

\[
\forall \langle l_0, l_1 \rangle \ | \ l = l_0 + (0) \#_I + l_1 \ | \ #_1 \cdot l_1 - #_0 \cdot l_1 \geq 1
\]

An equivalent definition is

\[
l = \langle 0, \text{ I } \rangle \\
l = (0) \#_I + l' + (\text{ I }) \#_I \quad \text{ for some connection pattern } l' \quad \forall \\
l = l_0 + l_1 \quad \text{ for connection patterns } l_0 \text{ and } l_1
\]

Each connection trace corresponds to the combination of an observation sequence and a connection pattern of equal length. If we walk from right to left through the connection-trace picture and connect each 0 we encounter with the nearest 1 on its right that is not already connected to some 0, then each connection corresponds to a node of the connection trace. For the first example connection trace:
9.1.4 Protocols

A protocol is a collection of traces. The set of all protocols is thus $P(T(I))$.

9.1.5 Connection protocols

A connection protocol is a protocol where the node elements of the traces are connections. The set of all connection protocols is thus $P(T(I*1))$.

The set of all connection protocols whose traces have a root element (a connection) whose output is of type $A$ and input is of type $B$, is denoted by $A \searrow B$:

$$P \in A \searrow B$$

$$\equiv \forall (y,v,z \mid (y:v;z) \in P \mid y \in A \land z \in B)$$

A connection protocol can be visualised by drawing all its connection traces into one connection-protocol picture. For the connection protocol

$$[(0:(1:)_1)_1, (0:(0;2))_3)_1, (1:(0;2)_1)_1, (0:(0;0))_1]$$

this picture looks for example like
9.2 Protocol operators

In this section we define some operators that, next to the collection operators, can be used to manipulate protocols.

9.2.1 Sequential composition

The sequential protocol composition \( (I \rightharpoonup I) \leftarrow (I \rightharpoonup I) \times (I \rightharpoonup I) \) is defined by

\[
(y:v:z) \in P_0 \circ p P_1 \\
= \exists \langle v_0, x, v_1 | v = v_0 \oplus v_1 | (y:v_0:x) \in P_0 \land (x:v_1:z) \in P_1 \rangle
\]

Connection-protocol pictures provide a good means to understand how this operator works.

9.2.2 Identity protocol

The identity protocol \( I_p \in (I \rightharpoonup I) \) is defined by

\[
(y:v:z) \in I_p \equiv y = z \land v = \langle \rangle
\]

9.2.3 Domain and range

The protocol-domain and protocol-range \( \preceq_p, \succeq_p \in (I \rightharpoonup I) \leftrightarrow (I \rightharpoonup I) \) are defined by

\[
(y:v:z) \in P \preceq_p \\
= y = z \land v = \langle \rangle \land \exists (y', v' \parallel (y':v':z) \in P)
\]

\[
(y:v:z) \in P \succeq_p \\
= y = z \land v = \langle \rangle \land \exists (v', z' \parallel (y:v':z') \in P)
\]
9.2.4 Pick and pack

The protocol-pick \( \mathcal{P}_p \in (I \rightarrow I) \leftarrow ((I \rightarrow I) \rightarrow (I \rightarrow I)) \times I \) and the protocol-pack \( \mathcal{Q}_p \in ((I \rightarrow I) \rightarrow (I \rightarrow I)) \leftarrow (I \rightarrow I) \times I \) are defined by

\[
(y:x:z)_\tau \in \mathcal{P}_p \equiv \langle (y, i) : v : (z, i) \rangle_\tau \in P
\]

\[
(Y:x:Z)_\tau \in \mathcal{Q}_p \equiv \exists \langle y, z, j \mid Y = \langle (y, j) \rangle \land Z = \langle (z, j) \rangle \mid j = i \Rightarrow (y:x:z) \in P \rangle
\]

The correspondence with the pick and pack on relations should be clear from the following definitions for those:

\[
y (R \triangleright i) z \equiv \langle (y, i) \rangle (R) \langle (z, i) \rangle
\]

\[
Y (R \triangleleft i) Z
\]

\[
\equiv \exists \langle y, z, j \mid Y = \langle (y, j) \rangle \land Z = \langle (z, j) \rangle \mid j = i \Rightarrow y (R) z \rangle
\]

We leave it to the reader to verify that this definition of \( \triangleleft \) is equivalent to the original one.

9.2.5 Universal dark protocol

The universal dark protocol \( \text{dark} \in \mathcal{P}(TI) \) is defined by

\[
(x, v)_\tau \in \text{dark} \equiv v = \langle \rangle
\]

A subcollection of \( \text{dark} \) is called a dark protocol. A dark protocol is a protocol that ‘performs no visible actions’.

9.2.6 Show

The show \( \text{show} \in \mathcal{P}(TI) \leftarrow \mathcal{P}(TI) \) is defined by

\[
(x, v)_\tau \in \text{show}.P \equiv \exists \langle v' \mid v = \langle (x, v') \rangle_\tau \rangle \mid (x, v')_\tau \in P \rangle
\]

The show transforms a protocol \( P \) into one that contains the traces \( (x, \langle (x, v')_\tau \rangle_\tau) \) for all \( (x, v')_\tau \) that \( P \) contains. It makes the root element of each trace in \( P \) visible.

215
9.2.7 Spec

The spec $\text{spec} \in \mathcal{P}(\mathcal{T}I) \leftrightarrow \mathcal{PI}$ is defined by

$$(x, v)_\tau \in \text{spec}.C \equiv x \in C$$

The spec enables the specification of possible root elements by means of a collection.

9.2.8 Example

In this section we present a small example to show the above operators in action.

We define the connection protocols $M$ and $N$ by

$$M = \text{dark} \cap \text{spec.}[\mathcal{C} \geq 3]$$
$$N = \text{dark} \cap \text{spec.}[\mathcal{C} \geq \mathcal{C}]$$

The following equivalences show the meaning of some connection protocols. We leave it to the reader to verify these equivalences:

$$(y:v;z)_\tau \in M \equiv v = \langle \rangle \land y \geq 3$$
$$(y:v;z)_\tau \in N \equiv v = \langle \rangle \land y \geq z$$
$$(y:v;z)_\tau \in (N \circ_p M) \equiv \exists (x \mid v = \langle (x;v;z)_\tau \rangle \land y \geq x \land x \geq 3)$$

The last equivalence demonstrates how show can be used to make the intermediate $x$ between $M$ and $N$ visible in the semantics.

9.3 Visual independence

A protocol $P$ is called visual independent if

$$\forall (x, v, v') \parallel (x, v)_\tau \in P \equiv (x, v')_\tau \in P$$

or equivalently:

$$\forall (x, v, v') \mid (x, v)_\tau \in P \mid (x, v')_\tau \in P$$

In other words, $(x, v)_\tau \in P$ holds for every $v$ if (and only if) it holds for some $v$.

Visual-independent protocols model collections as protocols in a similar way as conditions model collections as relations (see section 3.4.4).

For every collection $C$, $\text{spec}.C$ is visual independent. As a matter of fact, a protocol $P$ is visual independent exactly when there exists a collection $C$ such that $P$ equals $\text{spec}.C$: 

216
$P$ is visual independent

≡

$\exists (C \parallel P = \text{spec}.C)$

If $P \subseteq R$ for a visual-independent protocol $R$, then we can ‘extract this visual-independent protocol from a show.$P$’. The visual-independent extraction theorem formalises what we mean by this:

\[
\text{show}.P \subseteq R \\
\equiv \{
\bullet R \text{ is visual independent}
\}
\]

\[
P \subseteq R
\]

\textbf{proof}

\[
\begin{align*}
\text{show}.P \subseteq R & \\
\equiv & \{\text{definition of } \subseteq\} \\
\forall (x, v \parallel (x, v)_{\tau} \in \text{show}.P \mid (x, v)_{\tau} \in R) & \\
\equiv & \{\text{definition of show}\} \\
\forall (x, v \parallel \exists v' \parallel v = ((x, v')_{\tau} \parallel (x, v')_{\tau} \in P \mid (x, v)_{\tau} \in R) & \\
\equiv & \{\text{predicate calculus}\} \\
\forall (x, v, v' \parallel v = ((x, v')_{\tau} \land (x, v')_{\tau} \in P \mid (x, v)_{\tau} \in R) & \\
\equiv & \{1\text{-point rule}\} \\
\forall (x, v' \parallel (x, v')_{\tau} \in P \mid (x, (x, v')_{\tau})_{\tau} \in R) & \\
\equiv & \{\bullet R \text{ is visual independent}\} \\
\forall (x, v' \parallel (x, v')_{\tau} \in P \mid (x, v')_{\tau} \in R) & \\
\equiv & \{\text{definition of } \subseteq\} \\
P \subseteq R
\end{align*}
\]

Visual-independent extraction enables extraction of relational behaviour from a protocol. If we again take the $M$ and $N$ of section 9.2.8:

\[
\begin{align*}
M & = \text{dark} \cap \text{spec}.[\hat{C} \geq 3] \\
N & = \text{dark} \cap \text{spec}.[\hat{C} \geq \hat{C}]
\end{align*}
\]

then we can use visual-independent extraction to prove that the output of the connection protocol $(N \circ_p \text{show}.M)$ is at least 3, or in the theorem’s words, to extract the visual-independent connection protocol $\text{spec}.[\hat{C} \geq 3]$:

\[
N \circ_p \text{show}.M \subseteq \text{spec}.[\hat{C} \geq 3]
\]

\textbf{proof}
\[ N \cdot_p \text{show}.M \subseteq \text{spec.}[\hat{C} \geq 3] \]
\[ \equiv \{ \text{show}.M \subseteq \text{spec.}[\hat{C} \geq 3], \text{see below} \} \]
\[ N \cdot_p \text{spec.}[\hat{C} \geq 3] \subseteq \text{spec.}[\hat{C} \geq 3] \]
\[ \equiv \{ \text{left to the reader} \} \]
\[ \text{true} \]

\text{show}.M \subseteq \text{spec.}[\hat{C} \geq 3]
\[ \equiv \{ \text{visual-independent extraction} \} \]
\[ M \subseteq \text{spec.}[\hat{C} \geq 3] \]
\[ \equiv \{ \text{definition of } M \} \]
\[ \text{dark} \cap \text{spec.}[\hat{C} \geq 3] \subseteq \text{spec.}[\hat{C} \geq 3] \]
\[ \equiv \{ P_0 \cap P_1 \subseteq P_1 \} \]
\[ \text{true} \]

### 9.4 Refinement

For (partial) refinement between protocols, we use inclusion (\(\subseteq\)). If we again take the \(M\) and \(N\) of section 9.2.8:

\[ M = \text{dark} \cap \text{spec.}[\hat{C} \geq 3] \]
\[ N = \text{dark} \cap \text{spec.}[\hat{C} \geq \hat{C}] \]

then the following refinement holds:

\[ M \cdot M \subseteq N \cdot M \]

This means that first computing a number \(x\) greater than 3 and then a number that is larger than \(x\), is refined by first computing a number greater than 3 and then an arbitrary number greater than 3. This is a trivial consequence of the fact that the intermediate \(x\) is not visible.

If we put the \(M\) into a \text{show}, such a refinement is not possible anymore because the intermediate \(x\) is now visible:

\[ M \cdot \text{show}.M \not\subseteq N \cdot \text{show}.M \]

The left protocol contains the trace \((4:((3;\langle \cdot,0\rangle_{\tau},0)_{\cdot},0)_{\cdot})_{\cdot})_{\cdot}\), whereas the right protocol does not.

For simple cases the following simple protocol-refinement rules can be used to prove refinement between protocols:
We leave it to the reader to verify these rules. Notice that they cannot deal with the above example.

We leave the study of total refinement in the context of protocols for future research. We give however a definition that we think is appropriate. The total protocol refinement \( P' \sqsubseteq_p P \) is defined by

\[ P' \sqsubseteq_p P \equiv P' \subseteq P \land P'_p \supseteq P_p \]

9.5 Protocol statements

Because protocols are a special kind of collections, the theory of chapter 8 that is about collections in general, also holds for protocols. In this particular setting, we talk about protocol programs and protocol statements.

Next to the operators of chapter 8 that can be used for collections in general (\( \sqsubseteq, \sqcap, \sqcup, \sqcap, \sqsupset, \sqsubseteq, \text{call and const} \)), we introduce some operators specifically for protocol statements.

9.5.1 Sequential composition

The sequential protocol-statement composition \( \hat{\circ}_{p} \) is defined by

\[ (s_0 \circ_p s_1).b = s_0.b \circ_p s_1.b \]

9.5.2 Identity protocol statement

The identity protocol statement \( \hat{I}_p \) is defined by
\( \tilde{I}_p.b = I_p \)

### 9.5.3 Pick and pack

The protocol-statement–pick \( \tilde{s}_p \in (\langle I \rangle \leftarrow I) \leftarrow \langle (((I) \leftarrow I) \leftarrow I) \leftarrow I \rangle \times I \) and protocol-statement–pack \( \tilde{s}_p \in ((\langle I \rangle \leftarrow I) \leftarrow I) \leftarrow \langle (((I) \leftarrow I) \leftarrow I) \leftarrow I \rangle \times I \) are defined by

\[
(s \tilde{v}_p).b = s.b \triangleright_p i \\
(s \tilde{v}_p).b = s.b \triangleleft_p i
\]

### 9.5.4 Universal statement-dark statement

The universal statement-dark statement \( \tilde{d} \in \tilde{P}(TI) \leftarrow I \) is defined by

\[
\tilde{d}.b = \text{dark}
\]

If \( s \triangleleft \tilde{d} \) holds, we call \( s \) statement-dark.

### 9.5.5 Show

The statement-show \( \tilde{sh} \in \tilde{P}(TI) \leftarrow I \) is defined by

\[
(\tilde{sh}.s).b = \text{show} (s.b)
\]

### 9.5.6 Spec

The statement-spec \( \tilde{sp} \in \tilde{P}(TI) \leftarrow PI \) is defined by

\[
(\tilde{sp}.C).b = \text{spec} .C
\]

### 9.6 Visual-independent statements

A protocol statement \( s \) is called statement–visual-independent if \( s.b \) is visual independent for all \( b \).

The following statement–visual-independent—extraction theorem is a trivial consequence of the visual-independent extraction theorem of section 9.3:
9.7 Visibility of calls

In this section we investigate how call information can be made visible in a protocol program. In the first subsection we show a call that is completely invisible in the semantics. The second subsection shows a call that is itself not visible, but whose input/output is visible. The third subsection shows a call that appears to be completely visible, but that could be ‘fake’.

9.7.1 Invisible calls

The protocol program $p_0$, defined by

$$p_0 = m \triangleright p_m \cap n_0 \triangleright p_n$$

$$m = \text{dark} \cap \text{spec.}[C \geq 3]$$

$$n_0 = (\text{dark} \cap \text{spec.}[C \geq C]) \triangleright p \text{call} \triangleright p_m$$

has the same canonical behaviour as $p_0'$, defined by

$$p_0' = m \triangleright p_m \cap n_0' \triangleright p_n$$

$$n_0' = \text{dark} \cap \text{spec.}[C \geq 3]$$

In other words, because the call to $m$ in procedure $n$ is not made visible, the body of $n$ of the canonical behaviour only says that the output is at least 3 and that the visual is empty:

$$\langle y : v : z \rangle_r \in \mu p_0 \triangleright p \text{ n}$$

$$\equiv v = \langle \rangle_v \land y (\geq) 3$$

9.7.2 Visible input/output

The protocol program $p_1$, defined by

$$p_1 = m \triangleright p_m \cap n_1 \triangleright p_n$$

$$m = \text{dark} \cap \text{spec.}[C \geq 3]$$

$$n_1 = (\text{dark} \cap \text{spec.}[C \geq C]) \triangleright p \text{ show.} \triangleright p_m$$

has the same canonical behaviour as $p_1'$, defined by
\[ p_1' = m_{\hat{\phi}_m} \cap n_1'_{\hat{\phi}_n} \]
\[ n_1' = (\text{dark} \cap \text{spéc.} [C \geq C]) \cap \hat{\text{sh}} \text{ów.m} \]

The fact that we can replace the “\(\text{shów.(call}\hat{\phi}_m)\)” by “\(\text{shów.m}\)” means that the information that it is a call to \(m\) is invisible. However, the input/output of \(m\) is visible. The body of \(n\) of the canonical behaviour says that the visual is \([[x;\triangledown z]_\nu,\nu\rangle\]
where \(z\) is the input of \(n\) and \(x\) is some element that is at least 3. The output is at least \(x\):

\[ (y;v;z)_\tau \in \mu p_1 \triangleright_p n \]
\[ \equiv \exists x \mid v = ((x;\triangledown z)_\nu,\nu \rangle y \geq x \land x \geq 3) \]

### 9.7.3 ‘Fake’ visible calls

If we use \("(\text{shów.call})\hat{\phi}_m\)" instead of \("\text{shów.(call}\hat{\phi}_m)\)":

\[ p_2 = m_{\hat{\phi}_m} \cap n_2_{\hat{\phi}_n} \]
\[ m = \text{dark} \cap \text{spéc.}[C \geq 3] \]
\[ n_2 = (\text{dark} \cap \text{spéc.}[C \geq C]) \cap \hat{\text{sh}} \text{ów.(call)}\hat{\phi}_m \]

then the \(\text{shów}\) makes it impossible for \((\hat{\phi}_m)\) to completely cancel the \((\hat{\phi}_m)\). However, it can still be shown that \(p_2\) has the same canonical behaviour as \(p_2'\), defined by

\[ p_2' = m_{\hat{\phi}_m} \cap n'_2_{\hat{\phi}_n} \]
\[ m = \text{dark} \cap \text{spéc.}[C \geq 3] \]
\[ n_2' = (\text{dark} \cap \text{spéc.}[C \geq C]) \cap \hat{\text{sh}} \text{ów.(call)}\hat{\phi}_m \]

The body of \(n\) of the canonical behaviour expresses that the visual is equal to \((([[x;m];\triangledown[z;m]]_\nu,\nu\rangle y \geq x \land x \geq 3)\)
where \(z\) is the input of \(n\) and \(x\) is some element that is at least 3. The output is at least \(x\):

\[ (y;v;z)_\tau \in \mu p_2 \triangleright_p n \]
\[ \equiv \exists x \mid v = (([x;m];\triangledown[z;m])_\nu,\nu \rangle y \geq x \land x \geq 3) \]

Although \(m\) now appears in the visual, we see that the \(n\) of \(p_2'\) contains no \(\text{call}\). This means that we still have not achieved to formalise what we mean by: “The output is at least as large as the output of \(m\)”.

In the next section we address this issue.
9.8 Genuine statements

Putting a call into a show still does not make the call itself really visible. Intuitively speaking, this is a result of the fact that the notions ‘protocol’ and ‘statement’ are ‘orthogonal’ in protocol statements. We want the two notions to ‘interact’ appropriately, by which we mean that ‘the only thing that is shown are real calls’. In this section we formalise this, enabling us to write a procedure whose body really means that its output is at least as large as the output of some other procedure.

9.8.1 Introspectivity

A protocol $P$ is **introspective** if for every trace $t$ it contains, it also contains all traces in the visual of $t$:

$$\forall \langle x, v \mid (x, v), t \epsilon P \mid v \epsilon P \rangle$$

For a set $A$, the set $PT, A$ of introspective protocols where node elements have type $A$, is defined by

$$P \in PT, A \equiv P \in P(T, A) \land P \text{ is introspective}$$

We leave it to the reader to verify that the operators $\cap$, $\cup$, $\ominus$, $\Pi$, $I_p$ and dark are **introspectivity preserving**:

$$P_0 \cap P_1 \in PT, I \Leftarrow P_0, P_1 \in PT, I$$
$$P_0 \cup P_1 \in PT, I \Leftarrow P_0, P_1 \in PT, I$$
$$\ominus \in PT, I$$
$$\Pi \in PT, I$$
$$I_p \in PT, I$$
$$\text{dark} \in PT, I$$

The operators $\triangleleft_p$, $(\triangleright_p)$, $(\triangleleft_\phi)$, show and spec.C are not introspectivity preserving. We leave it to the reader to find counter examples.

The definition of introspectivity is recursive in the sense that if $P$ is introspective, then for each trace $t_0$ that $P$ contains, $P$ also contains each trace $t_1$ that occurs in the visual of $t_0$ and thus, because $P$ is introspective, $P$ also contains each trace $t_2$ that occurs in the visual of $t_1$, etcetera. Because traces are finite, this implies that introspectivity is equivalent to the property that a protocol contains each of its node traces:

$$\forall \langle x, v \mid (x, v), t \epsilon P \mid v \epsilon P \rangle$$

$$\equiv$$

$$\forall \langle t_0 \mid t_0 \epsilon P \mid \forall \langle t \mid t \text{ is a node trace of } t_0 \mid t \epsilon P \rangle)$$

The $(\Leftarrow)$ is trivial. The $(\Rightarrow)$ can be proved by induction on the depth of $t_0$ where
the depth of a trace is 0 when its visual is the empty sequence and otherwise it is 1 plus the maximum of the depths of the traces in its visual.

In case we would allow traces to be infinite, the above equivalence would not hold in general, but would represent some kind of ‘inductiveness’ property of traces.

9.8.2 Genuineness

A protocol statement $s$ is genuine if for each protocol $b$, the visual of each trace in $s.b$ consists only of traces contained by $b$:

$$\forall (b, x, v | (x, v) \in s.b | v < b)$$

For sets $A$ and $B$ that are equal to $I$, the set of genuine statements $P(TA) \leftarrow^g PB$ is defined by

$$s \in P(TA) \leftarrow^g PB \equiv \{ \bullet A = I \land B = I \}$$

$s \in P(TA) \leftarrow PB \land s$ is genuine

Informally stated, a protocol statement is genuine if ‘the only things that are shown are real calls’. This is formalised by the fact that for each context, the traces that are ‘shown’ by the protocol statement, must be traces from the context.

The operators $\cap$, $\cup$, $\emptyset$, $\hat{I}_p$, $(\hat{\bowtie}_p i)$ and $\text{darker}$ are genuineness preserving:

$$s_0 \cap s_1 \in P(TI) \leftarrow^g PI \Leftarrow s_0, s_1 \in P(TI) \leftarrow^g PI$$

$$s_0 \cup s_1 \in P(TI) \leftarrow^g PI \Leftarrow s_0, s_1 \in P(TI) \leftarrow^g PI$$

$$\emptyset \in P(TI) \leftarrow^g PI$$

$$s_0 \bowtie_p s_1 \in P(TI) \leftarrow^g PI \Leftarrow s_0, s_1 \in P(TI) \leftarrow^g PI$$

$$\hat{I}_p \in P(TI) \leftarrow^g PI$$

$$s \hat{\bowtie}_p i \in P(TI) \leftarrow^g PI \Leftarrow s \in P(TI) \leftarrow^g PI$$

$$\text{darker} \in P(TI) \leftarrow^g PI$$

We leave it to the reader to verify these facts.

The operator $\text{show}$ is not genuineness preserving. If it were, this would contradict the fact that ‘the only things that may be shown are real calls’. Also the operator $\text{call}$ is not genuineness preserving. This fact is discussed in section 9.9.4. The combination of these two operators, $\text{show}.\text{call}$, is genuineness preserving however:

$$\text{show}.\text{call} \in P(TI) \leftarrow^g PI$$

The operators $\hat{I}_p$, $(\hat{\bowtie}_p i)$ and $\text{spéc}. C$ are also not genuineness preserving. For these operators, we introduce special versions that do preserve genuineness.
The universal genuine statement $\Pi_g \in P(TI) \leftrightarrow PI$ is defined by

$$(x,v)_T \in \Pi_g.b \equiv v < b$$

The genuine-statement–pack $\hat{\bowtie}_g \in ((I \mapsto I) \mapsto (I \mapsto I)) \leftrightarrow PI \leftrightarrow ((I \mapsto I) \mapsto I \times I)$ is defined by

$$(Y:v,Z)_T \in (s \bowtie_g)i.b$$

$$\equiv \exists \langle y,z,j \mid Y = [(y,j)] \land Z = [(z,j)] \mid j = i \Rightarrow (y:v:z)_T \in s.b \land j \neq i \Rightarrow v < b \rangle$$

The genuine-statement–spec $\sp\bowtie \in (P(TI) \leftrightarrow PI) \leftrightarrow PI$ is defined by

$$(x,v)_T \in (s \sp\bowtie C).b \equiv x \in C \land v < b$$

These last two operators can be equivalently defined in terms of their non-genuine counterparts as follows:

$$s \bowtie_gi = \Pi_g \cap s \bowtie_gi$$

$$s \sp\bowtie C = \Pi_g \cap s \sp\bowtie C$$

The proofs are again left to the reader. Notice that the $s$ in the first equation ranges over genuine statements.

The universal genuine statement is the largest genuine statement and all smaller statements are also genuine:

$$s \in P(TI) \xleftarrow{\cdot g} PI \equiv s \subseteq \Pi_g$$

A direct consequence is that the operators $\Pi_g$, $(\bowtie_gi)$ and $\sp\bowtie C$ are genuineness preserving:

$$\Pi_g \in P(TI) \xleftarrow{\cdot g} PI$$

$$s \bowtie_gi \in P(TI) \xleftarrow{\cdot g} PI \equiv s \in P(TI) \xleftarrow{\cdot g} PI$$

$$\sp\bowtie C \in P(TI) \xleftarrow{\cdot g} PI$$

The operator $\sp\bowtie$ ‘magically’ ensures that a constructed statement is genuine. Ordinary programming languages do not have such a ‘magical’ operator and such a statement must then be implemented by means of a call or some statement-dark statement.

225
9.8.3 Continuity

Monotonicity is not always a strong enough requirement for statements to have all properties that we desire. Strengthening monotonicity to continuity often helps. The operators \( \cap, \cup, \emptyset, I, I_p, (\hat{\delta}_p), (\hat{\delta}_d), \text{d\text{\textasciicircum}rk}, \text{sh\text{\textasciicircum}ow}, \text{sp\text{\textasciicircum}c.C}, \text{call}, \text{const.C}, \Pi_g, (\hat{\delta}_d) \) and \( \text{sp\text{\textasciicircum}c.C} \) are all continuity preserving:

\[
\begin{align*}
\hat{s}_0 \cap s_1 & \in \Pi \iff s_0, s_1 \in \Pi \iff s_0 \cap s_1 \subseteq s_0, s_1 \subseteq \Pi \\
\hat{s}_0 \cup s_1 & \in \Pi \iff s_0, s_1 \in \Pi \iff s_0 \cup s_1 \subseteq s_0, s_1 \subseteq \Pi \\
\emptyset & \in \Pi \iff s \in \Pi \iff \emptyset \subseteq s \subseteq \Pi \\
\Pi & \in \Pi \iff s \in \Pi \iff \Pi \subseteq s \subseteq \Pi \\
\hat{s}_{\hat{\delta}_p} & \in \Pi \iff s \in \Pi \iff \hat{s}_{\hat{\delta}_p} \subseteq s \subseteq \Pi \\
\hat{s}_{\hat{\delta}_d} & \in \Pi \iff s \in \Pi \iff \hat{s}_{\hat{\delta}_d} \subseteq s \subseteq \Pi \\
\text{show.s} & \in \Pi \iff s \in \Pi \iff \text{show.s} \subseteq s \subseteq \Pi \\
\text{sp\text{\textasciicircum}c.C} & \in \Pi \iff s \in \Pi \iff \text{sp\text{\textasciicircum}c.C} \subseteq s \subseteq \Pi \\
\text{call} & \in \Pi \iff s \in \Pi \iff \text{call} \subseteq s \subseteq \Pi \\
\text{const.C} & \in \Pi \iff s \in \Pi \iff \text{const.C} \subseteq s \subseteq \Pi \\
 \Pi_g & \in \Pi \iff s \in \Pi \iff \Pi_g \subseteq s \subseteq \Pi \\
\hat{s}_{\hat{\delta}_d} & \in \Pi \iff s \in \Pi \iff \hat{s}_{\hat{\delta}_d} \subseteq s \subseteq \Pi \\
\text{sp\text{\textasciicircum}c.C} & \in \Pi \iff s \in \Pi \iff \text{sp\text{\textasciicircum}c.C} \subseteq s \subseteq \Pi \\
\end{align*}
\]

proof

We first prove it for \( \text{const.C} \), for an arbitrary collection \( C \):

\[
\text{const.C} \text{ is continuous} \\
= \quad \{\text{definition of continuity}\} \\
\forall \{W : W \text{ is a chain} \} : W \neq \emptyset \quad \{\text{(const.C)} \cup W = \bigcup \{(\text{const.C}).X \mid X \in W \mid X\}\} \\
= \quad \{\text{definition of const}\} \\
\forall \{W : W \text{ is a chain} \} : W \neq \emptyset \quad \{\text{C} \mid X \in W \mid X\} = \bigcup \{C \mid X \in W \mid X\} \\
= \quad \{W \neq \emptyset \Rightarrow \bigcup \{C \mid X \in W \mid X\} = C\}
\]

true

We now also have tackled the cases \( \emptyset, \Pi, I_p, \text{dark} \) and \( \text{sp\text{\textasciicircum}c.C} \) because

\[
\begin{align*}
\emptyset & = \text{const.}\emptyset \\
\Pi & = \text{const.}\Pi \\
I_p & = \text{const.}I_p \\
\text{dark} & = \text{const.}\text{dark} \\
\text{sp\text{\textasciicircum}c.C} & = \text{const.}(\text{spec.C}) \\
\end{align*}
\]
For $\bigcap$, we assume that $W$ is a non-empty chain and calculate:

$$x \in (s_0 \bigcap s_1). (\bigcup W)$$

$$= \text{definition of } \bigcap$$

$$x \in s_0. (\bigcup W) \cap s_1. (\bigcup W)$$

$$= \text{definition of } \cap$$

$$x \in s_0. (\bigcup W) \land x \in s_1. (\bigcup W)$$

$$= \{s_0, s_1 \text{ are continuous}\}$$

$$x \in \bigcup\{s_0. b_0 : b_0 \in W \mid b_0\} \land x \in \bigcup\{s_1. b_1 : b_1 \in W \mid b_1\}$$

$$= \text{definition of } \bigcup$$

$$\exists (b_0 : b_0 \in W \mid x \in s_0. b_0) \land \exists (b_1 : b_1 \in W \mid x \in s_1. b_1)$$

$$= \{\text{($\Leftarrow$): take } b_0 = b_1 = b\}$$

$$= \{\text{($\Rightarrow$): take } b = b_0 \cup b_1, \text{ this is an element of } W \text{ because } W \text{ is a chain,}\}$$

$$\exists (b : b \in W \mid x \in s_0. b \land x \in s_1. b)$$

$$= \text{definition of } \cap$$

$$\exists (b : b \in W \mid x \in (s_0 \cap s_1). b)$$

$$= \text{definition of } \bigcup$$

$$x \in \bigcup\{(s_0 \cap s_1). b : b \in W \mid b\}$$

$$= \text{definition of } \cap$$

$$x \in \bigcup\{(s_0 \cap s_1). b : b \in W \mid b\}$$

The proofs for the other non-$g$ protocol-statement operators are left to the reader (remark: the proof for $\langle \triangleleft \rangle$ needs the non-emptiness of $W$ as much as the proof for $\text{const.C}$). For $\Pi_g$ we prove for a non-empty chain $W$:

$$(x, v)_t \in \Pi_g.(\bigcup W)$$

$$= \text{definition of } \Pi_g$$

$$v \ll \bigcup W$$

$$= \text{definition of } \ll$$

$$\forall (t : t \in v \mid t \in \bigcup W)$$

$$= \text{point-wise definition of } \bigcup$$

$$\forall (t : t \in v \mid \exists (P : P \in W \mid t \in P))$$

$$= \{\text{($\Leftarrow$): trivial, ($\Rightarrow$): see below}\}$$

$$\exists (P : P \in W \mid \forall (t : t \in v \mid t \in P))$$

$$= \text{definition of } \ll$$

$$\exists (P : P \in W \mid v \ll P)$$

$$= \text{definition of } \Pi_g$$

$$\exists (P : P \in W \mid (x, v)_t \in \Pi_g. P)$$

$$= \text{point-wise definition of } \Pi_g$$

$$(x, v)_t \in \bigcup \Pi_g. P \mid P \in W \mid P)$$

227
Now for the \( \Rightarrow \) of the fourth step:

\[
\forall t \mid t \in v \mid \exists (P \mid P \in W \mid t \in P)
\]

\[
\equiv
\exists (f \mid f \in I \iff \{ t \mid t \in v \mid t \} \mid \forall (t \mid t \in v \mid f.t \in W \land t \in f.t))
\]

\[
\Rightarrow
\{ \text{Take } P = \bigcup \{ f.t \mid t \in v \mid t \}. \text{ The set } \{ f.t \mid t \in v \mid t \} \text{ is finite because sequences are finite by definition. We thus have } P \in W \text{ because the union of the elements of a finite subset of a chain is an element of that chain.} \}
\]

\[
\exists (P \mid P \in W \mid \forall (t \mid t \in v \mid t \in P))
\]

The continuity preservingness of \( \hat{g} \) and \( \text{sp}_{\hat{g}} C \) follows directly from their definition in terms of their non-genuine counterparts.

The canonical behaviour of a continuous genuine statement is introspective:

\[
\mu s \text{ is introspective } \iff s \text{ is a continuous genuine statement}
\]

**proof**

Assume that \( s \) is a continuous genuine statement.

\[
\mu s \text{ is introspective}
\]

\[
\equiv
\{ \text{definition of introspectivity} \}
\]

\[
\forall (x, v \mid (x, v), \in \mu s \mid v \in \mu s)
\]

\[
\equiv
\{ \text{definition of } < \}
\]

\[
\forall (x, v \mid (x, v) \in \mu s \mid \forall (t \mid t \in v \mid t \in \mu s))
\]

\[
\equiv
\{ s \text{ is continuous, so } \mu s = \bigcup \{ s^m.\emptyset \mid n \in \mathbb{N} \mid n \} \text{ (see section 3.6)} \}
\]

\[
\forall (x, v \mid (x, v) \in \mu s \mid \forall (t \mid t \in v \mid t \in \mu s))
\]

\[
\equiv
\{ \text{definition of } \bigcup \}
\]

\[
\forall (x, v \mid \exists (n \mid n \in \mathbb{N} \mid (x, v) \in s^n.\emptyset) \mid \forall (t \mid t \in v \mid \exists (m \mid m \in \mathbb{N} \mid t \in s^m.\emptyset))
\]

\[
\equiv
\{ \text{predicate calculus} \}
\]

\[
\forall (n \mid n \in \mathbb{N} \mid \forall (x, v \mid (x, v) \in s^n.\emptyset) \mid \forall (t \mid t \in v \mid \exists (m \mid m \in \mathbb{N} \mid t \in s^m.\emptyset))
\]

\[
\equiv
\{ \text{take } m = n \}
\]

\[
\forall (n \mid n \in \mathbb{N} \mid \forall (x, v \mid (x, v) \in s^n.\emptyset) \mid \forall (t \mid t \in v \mid t \in s^n.\emptyset))
\]

\[
\equiv
\{ \text{definition of } < \}
\]

\[
\forall (n \mid n \in \mathbb{N} \mid \forall (x, v \mid (x, v) \in s^n.\emptyset) \mid v \in s^n.\emptyset)
\]

\[
\equiv
\{ \text{definition of introspectivity} \}
\]

\[
\forall (n \mid n \in \mathbb{N} \mid s^n.\emptyset \text{ is introspective})
\]

We prove this last formula by mathematical induction. The basis \( n = 0 \) holds because \( \emptyset \) is introspective. For the step we calculate:

228
\[ s^{n+1}, \emptyset \]
\[ = \{ R^{n+1} = R^n \cdot R \} \]
\[ s^n, (s, \emptyset) \]
\[ = \{ \emptyset \text{ is unit of } \cup \} \]
\[ s^n, (\emptyset \cup s, \emptyset) \]
\[ = \{ \text{left to the reader (hint: prove by induction that } s^n \text{ is continuous)} \} \]
\[ s^n, \emptyset \cup s^n, (s, \emptyset) \]
\[ = \{ R^n \cdot R = R \cdot R^n \} \]
\[ s^n, \emptyset \cup s, (s^n, \emptyset) \]

The induction hypothesis gives us that \( s^n, \emptyset \) is introspective. We now complete the proof by showing that for each introspective protocol \( b \), \( b \cup s \cdot b \) is also introspective:

\[ \forall \langle b, x, v \mid (x, v) \in s \cdot b \mid v < b \rangle \]
\[ = \{ s \text{ is genuine} \} \]

true

9.9 Refinement of protocol statements

In this section we discuss refinement of protocol statements.

9.9.1 Simple refinement rules

Some simple protocol-statement–refinement rules are:
The proofs of these rules are straightforward.

9.9.2 A ‘strange’ refinement revisited

In section 8.4.5 we observed that the use of relations as model for behaviour has as a result that program \( p \), defined by

\[
p = m \triangleright m \cap n \triangleright n
\]

is behaviourally refined by \( p'' \), defined by

\[
p'' = m'' \triangleright m'' \cap n'' \triangleright n''
\]

We now show that if we restrict to continuous genuine programs and make the call to \( m \) visible, such a ‘strange’ refinement is not possible anymore. Suppose program \( p_3 \) is defined by

\[
p_3 = m_3 \triangleright m_3 \cap n_3 \triangleright n_3
\]

and \( p_4 \), defined by

\[
p_4 = m_4 \triangleright m_4 \cap n_4 \triangleright n_4
\]
for some statement \( n_4 \), is a continuous genuine program that behaviourally refines \( p_3 \). We now prove that the output of procedure \( n \) of \( p_4 \) is at least 4:

\[
(p_4) \triangleright p_n \subseteq \text{spec}.[\bar{C} \geq 4]
\]

**proof**

We first investigate what can be derived from the fact that \( p_4 \) is a behavioural refinement of \( p_3 \):

\[
\begin{align*}
& (y:v:z) \vDash (p_4) \triangleright p_n \\
& \Rightarrow \quad \{ p_4 \not\subset p_3 \}
\end{align*}
\]

\[
\begin{align*}
& (y:v:z) \vDash (p_4) \triangleright p_n \\
& \equiv \quad \{ \text{left to the reader (see also section 9.7.3)} \}
\end{align*}
\]

\[
\begin{align*}
& \forall v \leq \mu p_3 \land \exists x \mid v = \langle (((x,m):(\emptyset):(z,m))_\lor) \lor \mid y \geq x \land x \geq 3 \rangle \\
& \Rightarrow \quad \{ \text{we remove the things that we do not need} \}
\end{align*}
\]

\[
\begin{align*}
& \exists x \mid v = \langle (((x,m):(\emptyset):(z,m))_\lor) \lor \mid y \geq x \rangle
\end{align*}
\]

We now calculate:

\[
\begin{align*}
& (y:v:z) \vDash (p_4) \triangleright p_n \\
& \equiv \quad \{ \text{definition of } \triangleright \}
\end{align*}
\]

\[
\begin{align*}
& (((y,n):v:((z,n)))_\lor) \vDash \mu p_4 \\
& \Rightarrow \quad \{ \text{the property we just derived, gives us the shape of } v \}
\end{align*}
\]

\[
\begin{align*}
& \exists x \parallel y \geq x \land (((y,n):v:((z,n)))_\lor) \vDash \mu p_4 \\
& \Rightarrow \quad \{ \mu p_4 \text{ is introspective because } p_4 \text{ is continuous and genuine (section 9.8.3)} \}
\end{align*}
\]

\[
\begin{align*}
& \exists x \parallel y \geq x \land (((y,n):v:((z,n)))_\lor) \vDash \mu p_4 \\
& \equiv \quad \{ \text{left to the reader} \}
\end{align*}
\]

\[
\begin{align*}
& \exists x \parallel y \geq x \land x = 4 \\
& \equiv \quad \{ 1\text{-point rule} \}
\end{align*}
\]

\[
\begin{align*}
& y \geq 4
\end{align*}
\]

### 9.9.3 Components

In section 8.5.3 we saw that behavioural intersection of program \( q_c \), defined by

\[
q_c = c' \cap d \cap \text{spec}.[C = 3]
\]

\[
c' = \text{const}.[C = 3]
\]

\[
d = \text{const}.[C \geq 4]
\]

with program \( q_d \), defined by
\[
qd = c \triangleleft \hat{d} \triangleleft \hat{d}
\]
\[
c = \text{const.}[\hat{C} \geq 3]
\]
\[
d' = \text{const.}[\hat{C} = \hat{C} + 1] \cdot \text{call} \cdot \text{c}
\]

has as behaviour:
\[
[\hat{C} = 3] \times c \cap [\hat{C} \geq 4] \times d
\]
rather than what we would ‘expect’:
\[
[\hat{C} = 3] \times c \cap [\hat{C} = 4] \times d
\]

The reason is that “\text{call} \cdot \text{c}” does not mean that a call to procedure \text{c} is performed, but that the given specification of \text{c} is somehow realised. In other words, the call to \text{c} is invisible. We now show that if we make the call to \text{c} visible, we get the behaviour we ‘expect’.

The program \(qc_3\) is defined by
\[
qc_3 = c_3' \triangleleft \hat{d}_y \triangleleft \hat{d} \triangleleft \hat{d}
\]
\[
c_3' = \text{dark} \cap \text{specific} \cdot [\hat{C} = 3]
\]
\[
d_3 = \text{specific} \cdot [\hat{C} \geq 4]
\]

and the program \(qd_3\) is defined by
\[
qd_3 = c_3 \triangleleft \hat{d}_y c \triangleleft \hat{d}_y \triangleleft \hat{d}
\]
\[
c_3 = \text{dark} \cap \text{specific} \cdot [\hat{C} \geq 3]
\]
\[
d_3' = (\text{dark} \cap \text{specific} \cdot [\hat{C} = \hat{C} + 1]) \cdot \text{sh} \cdot \text{call} \cdot \text{c}
\]

We now prove for the behaviour of the behavioural intersection of \(qc_3\) and \(qd_3\) that the output of procedure \text{d} is equal to 4:
\[
\mu(qc_3 \cap qd_3) \triangleright d \subseteq \text{specific} \cdot [\hat{C} = 4]
\]

**proof**
\[(y:v) \in \mu(qc_3 \cap qd_3) \triangleright_p d\]
\[\equiv \quad \{\text{definition of } \triangleright_p\}\]
\[
\begin{align*}
\{(y,d) : v \in [(z,d)]\}_r &\in \mu(qc_3 \cap qd_3) \\
\equiv &\quad \{y_0 \cap y_1 = \text{const.} (\mu s_0 \cap \mu s_1)\}
\end{align*}
\]
\[
\begin{align*}
\{(y,d) : v \in [(z,d)]\}_r &\in \mu(\text{const.} (\mu qc_3 \cap \mu qd_3)) \\
\equiv &\quad \{\mu(\text{const.} c) = c\}
\end{align*}
\]
\[
\begin{align*}
\{(y,d) : v \in [(z,d)]\}_r &\in \mu qc_3 \cap \mu qd_3 \\
\equiv &\quad \{\text{definition of } \cap\}\]
\end{align*}
\]
\[
\begin{align*}
\{(y,d) : v \in [(z,d)]\}_r &\in \mu qc_3 \equiv x = 3 \\
\end{align*}
\]

The last step shows how the visual enables us to use the fact that procedure \(c\) outputs 3. This shows how our model incorporates an implicit form of dynamic binding (see also section 2.5.7).

### 9.9.4 Invisible calls

In section 9.8.2 we remarked that \texttt{call} is not genuineness preserving. A way to make \texttt{call} genuineness preserving, is to change the definition of genuineness by adding \(b \in PT_1\) to the domain of the quantification, changing it into:

\[
\forall \langle b, x, v \mid b \in PT_1 \land (x,v)_r \in s.b \mid v < b\rangle
\]

To guarantee that some basic algebraic properties of the formalism are preserved, the definition of \(\hat{\Pi}_g\) then needs to be changed into

\[
(x,v)_r \in \hat{\Pi}_g.b \equiv (b \in PT_1 \Rightarrow v < b)
\]

This new \(\hat{\Pi}_g\) is not continuous however.

Invisible calls thus seem not to fit well into our framework. There is another solution to incorporate invisible calls however which involves the introduction of an operator that explicitly hides visible information. The \texttt{hide} \texttt{hide} \(\in P(TI) \leftrightarrow P(TI)\) is defined by

\[
233
\]
\((x, v), \in \text{hide}.P \equiv v = \langle \rangle \land \exists (v', \eta) (x, v'), \in P\)

and the statement-hide \text{hide} \in (P(TI) \iff I) \iff (P(TI) \iff I) by

\((\text{hide}.s).b = \text{hide}.(s.b)\)

The statement-hide creates a genuine statement from any statement:

\[
\text{hide}.s \in P(TI) \iff PI \iff s \in P(TI) \iff PI
\]

This fact holds because \((x, v), \in (\text{hide}.s).b\) implies that the visual \(v\) is empty. The proof of statement-hide being continuity preserving:

\[
\text{hide}.s \in PI \iff PI \iff s \in PI \iff PI
\]

is also straightforward.

A result of all this is that invisible calls can be done with a \text{hide}.call. Notice though that the visible calls of a procedure that is called with a \text{hide}.call, become invisible within that call. We leave further investigation of invisible calls for future research.

### 9.10 Conclusions

In this chapter we showed how visible calls can be modeled. Visible calls enable one to specify that a certain procedure should be called. This gain in expressivity is essential for some design patterns. Suppose in the plug-pattern example of section 2.2, the call of \text{insertInto} to \text{fillWith} would be invisible. If someone would have implemented \text{insertInto} according to the plug-pattern specification and we implement \text{fillWith} according to the observer-pattern specification, there is no way of knowing for sure that calling \text{insertInto} of an observer, results in calling our \text{fillWith}. Making the call visible does provide this assurance.
Chapter 10

Variables

In this chapter we show how we model variables. Special attention is paid to the particular kind of variable-concepts of object-oriented specification languages like ISpec.

10.1 Maps

We assume the existence of an external type of maps. Maps are isomorphic to total functions. The set of all maps is denoted by $I \leftarrow I$. The selection $f \in I \leftarrow (I \leftarrow I) \times I$ is the equivalent of the application. If $f.z = y$ holds, we say that $f$ maps $z$ to $y$. Equality between maps $g$ and $h$ is defined by

$$g = h \equiv \forall (x \parallel g.x = h.x)$$

The operator $\llbracket R \llbracket P \leq (I \leq I) \leq (I \rightarrow I) \times (I \rightarrow I)$ is defined by

$$g \llbracket R \llbracket P \leq \forall (y, z \parallel g.y \cup P \llbracket h.z \leq y \llbracket (P) \llbracket z)$$

For sets $A$ and $B$, $A \leftarrow B$ is not a set in general. For reasons mentioned in section 4.2.7, a set-forced version $\leftarrow \in ((I \leftarrow I) \rightarrow (I \leftarrow I)) \rightarrow (I \rightarrow I) \times (I \rightarrow I)$ is introduced:

$$R \leftarrow P \equiv (R \leftarrow P) \cap I$$

The set $A \leftarrow B$ consists of all maps that map elements of type $B$ to elements of type $A$.

The reason why we introduce maps and not simply use total functions, is that the fact $I \leftarrow I = I$ often allows for cleaner formulas.
10.2 Variables

We model variables by a map from variable names to variable values. The fact that we use maps, means that each variable name is mapped to exactly one value. If we want to model existence (creation and deletion) of variables, we can model the non-existence of a variable by a special null value or switch to (partial) functions, similar to what we did in chapter 6. We leave the incorporation of variable existence in our framework for future research however.

10.2.1 Variable expressions

The selection enables the retrieval of the value of a variable with a specific name. The expression that represents the value of the variable (with name) \( a \) is \((C \, \triangleleft \, a)\). In this particular case where a context is a map from variable names to variable values, we normally use the term “state” instead of “context”. In case of connectional expressions (see section 6.8), we talk about “new state” instead of “output of the context” and about “old state” instead of “input of the context”. The connectional expression that represents the new value of variable \( a \) (the value of variable \( a \) in the new state) is \((\hat{C} \, \triangleleft \, \hat{a})\) and the connectional expression that represents the old value of variable \( a \) (the value of variable \( a \) in the old state) is \((\hat{C} \, \triangleleft \, \hat{a})\).

The relation that tells that variable \( a \) is increased at least by the old value of variable \( b \), can for example be written as

\[
[\hat{C}.a \geq \hat{C}.a + \hat{C}.b]
\]

10.2.2 Modify

The relation \([\hat{C}.a \geq \hat{C}.a + \hat{C}.b]\) tells something about the new value of variable \( a \), but nothing about the new values of other variables. This means that these variables may change arbitrarily. For expressing the fact that all variables outside a certain set remain equal, we define the modify modify \( modify \in ((I \prec\prec I)\prec\prec(I \prec\prec I)) \prec\prec \wp I \) by:

\[
g (modify.A) h \equiv \forall(x \mid x \notin A \mid g.x = h.x)
\]

The relation

\[
[\hat{C}.a \geq \hat{C}.a + \hat{C}.b] \cap modify.\{a\}
\]

expresses that variable \( a \) is increased at least by the value of variable \( b \) and only variable \( a \) possibly changes.

Notice that if the value of a variable does not change (like the value of \( b \) in the above example), we talk about “the” value of that variable instead of “the old” or “the new” value. If it does not matter whether or not we use a ` or a ´ in a relation
comprehension, we prefer ` . This corresponds to the fact that in ISpec, “x” is used for “the new” or “the” value of variable x and “x” for “the old” value of variable x.

The proofs of the following theorems are left to the reader:

\[
\text{modify}. A_0 \cap \text{modify}. A_1 = \text{modify}. (A_0 \cap A_1)
\]
\[
\text{modify}. A_0 \circ \text{modify}. A_1 = \text{modify}. (A_0 \cup A_1)
\]

10.2.3 Declared variables

By a “modify.A” it is often not meant that each variable whose name is not an element of A, remains equal. Usually there exists some set of variable names D that represents the declared variables: the variables that the specification ‘talks about’. A “modify.A” is then expected to only talk about these declared variables. Variables that are not declared, can be added in a refinement and may change arbitrarily. In other words, a “modify.A” then actually is a modify.D.A, where for a set (of declared-variable names) D, modifyD ∈ ((I≺− − −I)≺− − −℘I) is defined by

\[
\text{modify}_D.A = \text{modify}.(I \setminus D \cup A)
\]

The relation

\[
\left[ \hat{\mathcal{C}}.a \geq \hat{\mathcal{C}}.a + \hat{\mathcal{C}}.b \right] \cap \text{modify}_D.\{a\}
\]

where D = \{a, b, c\}, is for example refined by

\[
\left[ \hat{\mathcal{C}}.a > \hat{\mathcal{C}}.a + \hat{\mathcal{C}}.b \wedge \hat{\mathcal{C}}.d = \hat{\mathcal{C}}.a \right] \cap \text{modify}_{D'}.\{a, d\}
\]

where D' = \{a, b, c, d, e\}.

10.2.4 Pointers

A pointer is a variable whose value is the name of a variable. The relation

\[
\left[ \hat{\mathcal{C}}.(\hat{\mathcal{C}}.p) \geq \hat{\mathcal{C}}.(\hat{\mathcal{C}}.p) + \hat{\mathcal{C}}.(\hat{\mathcal{C}}.q) \right]
\]

expresses for example that the variable whose name is the old value of variable p (that pointer p points to in the old state), is increased at least by the old value of the variable that q points to in the old state.

Like before, we often want all other variables to remain equal. For this purpose we introduce a generalisation of modify. To indicate the possibly modified variables, we use total functions that are evaluated in the old state. The pointer-modify
\( \text{pmodify} \in ((I \leadsto I) \leadsto (I \leadsto I)) \leftarrow \varnothing (I \leadsto (I \leadsto I)) \) is defined by

\[
g (\text{pmodify}.K) h \equiv g (\text{modify}.\{ k.h \mid k \in K \mid k \}) h
\]

Instead of defining the \( \text{pmodify} \) in terms of the \( \text{modify} \), the \( \text{modify} \) could also be defined in terms of the \( \text{pmodify} \):

\[
\text{modify}.A = \text{pmodify}.\{ \bar{a} \mid a \in A \mid a \}
\]

The relation

\[
[\mathcal{C}, (\mathcal{C} \triangleright p)] \geq \mathcal{C}, (\mathcal{C} \triangleright q) \cap \text{pmodify}.\{ C \triangleright p \}
\]

expresses for states where \( p \) and \( q \) point to variables (whose value is) of type \( \mathbb{R} \), that the variable \( a \) that \( p \) points to, is increased at least by the old value of the variable that \( q \) points to and \( a \) is the only possibly modified variable. Notice that if \( p \) and \( q \) point to the same variable, also the value of the variable that \( q \) points to possibly changes.

In case we ‘execute’ the relation

\[
[\mathcal{C}, (\mathcal{C} \triangleright p)] = \mathcal{C}, (\mathcal{C} \triangleright q) \cap \text{pmodify}.\{ C \triangleright p \}
\]

in a state where variable \( p \) points to itself then, in the new state, \( p \) points to the variable that was pointed at by the variable that \( q \) was pointing to in the old state. In case \( p \) would point to \( q \) in the old state, then \( q \) would point to that variable in the new state. In our interpretation of the relation of the previous example, the assumption that \( p \) and \( q \) point to variables of type \( \mathbb{R} \) and the (not explicitly stated) fact that \( \mathbb{R} \) and \( \mathbb{I} \) are disjoint, ensure that \( p \) cannot possibly point to \( p \) or \( q \).

The fact that pointers give rise to such intricate behaviour is reflected by the fact that \( \text{pmodify}.K_0 \cdot \text{pmodify}.K_1 \) is not equal to \( \text{pmodify}.(K_0 \cup K_1) \) in general. We have for example

\[
\text{pmodify}.\{ C \triangleright q \} \cdot \text{pmodify}.\{ C \triangleright p \} \neq \text{pmodify}.\{ (C \triangleright q), (C \triangleright p) \}
\]

If we ‘execute’ the right formula, only the variables where \( p \) and \( q \) point to in the old state, can change. If we ‘execute’ the left formula however, then every variable can change if \( p \) points to \( q \) in the old state.

The above inequality can be turned into an equality if we restrict to old states where \( p \) does not point to \( q \):

\[
g (\text{pmodify}.\{ (C \triangleright q), (C \triangleright p) \}) h \\
\equiv \{ h.p \neq q \} \\
g (\text{pmodify}.\{ C \triangleright q \}, (C \triangleright p)) h
\]

238
This theorem is a trivial consequence of the following theorem, whose proof we leave
to the reader:

\[
g (\text{pmodify}.\{k\} \circ \text{pmodify}.\{l\}) h \\
\equiv \{\forall f : f \, (\text{pmodify}.\{l\}) \, h \mid k, f = k, h\} \\
g (\text{pmodify}.\{k, l\}) h
\]

The condition in this theorem expresses that the value of \( k \) is not affected by the
modification of a variable whose name is the value of \( l \) in the old state.

We also define a pointer-modify that incorporates declared variables. For a set (of
declared-variable names) \( D \), \( \text{pmodify}_D \in ((I \leftarrow I) \rightsquigarrow (I \leftarrow I)) \leftarrow \varnothing(I \leftarrow (I \leftarrow I)) \)
is defined by

\[
\text{pmodify}_D.K = \text{pmodify}.(\{\bar{x} \mid x \notin D \cup x\} \cup K)
\]

In other words, all variables whose name is an element of \( D \) and not the value of
any \( k \in K \) in the old state, remain equal.

## 10.2.5 Assignment

The common way to change the values of variables, is by means of assignments. The
assignment \( g (R=:k) h \) is defined by

\[
g (R=:k) h = g.(k.h) (R) h \land g (\text{pmodify}.\{k\}) h
\]

The relation \( R=:k \) expresses that the variable whose name is the value of \( k \) in the
old state, is (non-deterministically) assigned a value of relation \( R \) in the old state.
All other variables remain equal.

We use the symbol \( ":=\" \) instead of \( ":=\" \) as a reminder of the fact that the new state
is placed on the left and the old state on the right.

The assignment

\[
(\tilde{C} \vdash \bar{a}) \tilde{\oplus} (\tilde{C} \vdash \bar{b}) =: \bar{a}
\]

expresses for example that variable \( a \) is increased by the value of variable \( b \) and \( a \)
is the only possibly modified variable.

An example of a non-deterministic assignment is

\[
[\hat{C} \geq \hat{C}, a + \tilde{C}, b] =: \bar{a}
\]

expressing that variable \( a \) is \textit{at least} increased by the value of variable \( b \) and \( a \) is
the only possibly modified variable.

An example of a *pointer-assignment* is

$$(C ⊢ (C ⊢ p)) \vdash (C ⊢ (C ⊢ q)) =: (C ⊢ p)$$

which, in case $p$ and $q$ point to variables of type $R$, expresses that the variable $a$ that $p$ points to, is increased by the old value of the variable that $q$ points to and $a$ is the only possibly modified variable.

For specifications we advise the combination of a relation comprehension and a *modify* (or $p$*modify* if necessary) to express changes to variables. This is in line with section 7.4 where we show the advantages of splitting a specification into an operational and a declarative part. The operational part (the *modify*) tells *which* variables possibly change and the declarative part (the relation comprehension) tells *how* they change.

### 10.3 Objects

In this section we discuss the particular kind of states that are found in object-oriented programming.

#### 10.3.1 Global states

In object-oriented programming, a program has access to a *global state* that is partitioned into *fields* where a field is identified by an *object name* and an *attribute name*. Formally, a global state is a map from *field names* to *field values* where the field name with object name $o$ and attribute name $a$ is denoted by $(o, a)_F$. We talk about “*attribute* $a$ of *object* $o$” to indicate the field with name $(o, a)_F$.

We assume the existence of an external type of field names. The set of all $(o, a)_F$ with $o \in O$ and $a \in A$ is denoted by $F(O, A)$:

$$(o, a)_F \in F(O, A) \equiv o \in O \land a \in A$$

We denote the set of all object names by $O$ and the set of all attribute names by $A$. We define both equal to $I$ to allow for cleaner formulas:

- $O = I$
- $A = I$

The set $G$ of all global states is now defined by

$$G = I \leftarrow F(O, A)$$

The formula
represents the value of attribute \(a\) of object \(o\) (field \((o, a)\)) in global state \(g\).

### 10.3.2 Field selection

The field selection \(\_ \rightarrow \_ \in (I \leftarrow G) \leftrightarrow (O \leftarrow G) \times A\) is defined by

\[
(e \rightarrow a).g = g.(e.g.a)
\]

The expression

\(o \rightarrow a\)

represents the value of attribute \(a\) of object \(o\). Assuming that attribute \(a\) of object \(o\) is pointing to some object (has an object name as value), the expression

\((o \rightarrow a) \rightarrow b\)

represents the value of attribute \(b\) of that object.

It is uncommon in object-oriented languages to indicate an object directly by its name (\(o\) in the example). This issue is dealt with in section 10.4 where we introduce the notion of a ‘this’-object name.

For the evaluation of the new and old value of a field, we introduce the functions \(\_ \rightarrow \_ \_, \_ \rightarrow \_\_ \) in \((I \leftarrow G \times G) \leftrightarrow (O \leftarrow G \times G) \times A\), defined by

\[
\begin{align*}
(e' \rightarrow a).(g, h) &= (K.(e.(g, h)) \rightarrow a).g \\
(e'' \rightarrow a).(g, h) &= (K.(e.(g, h)) \rightarrow a).h
\end{align*}
\]

Notice the ‘trick’ of using a constant-valued function \(K.(e.(g, h))\) as a single-state expression that actually depends on two states. This enables us to define \(\rightarrow\) and \(\rightarrow\) in terms of \(\rightarrow\). We use this ‘trick’ at several places.

The relation

\[
\left[ o' \rightarrow a \geq o' \rightarrow a + o' \rightarrow b \right]
\]

expresses that attribute \(a\) of object \(o\) is increased at least by the old value of attribute \(b\) of object \(o\).
10.3.3 Field-modify

The field-modify \( f\text{modify} \in (G \leftarrow G) \leftarrow \wp((\bigodot \leftarrow G) \times \mathbb{A}) \) can be used to express modification of fields:

\[
g (f\text{modify}.X) h \equiv g (\text{modify.} \{(k,h,a), (k,a) \in X | k, a\}) h
\]

The relation

\[
\lceil \delta \ast a \geq \delta \ast a + \delta \ast b \rceil \cap f\text{modify}.\{(\delta, a)\}
\]

expresses for example that attribute \( a \) of object \( \delta \) is increased at least by the value of attribute \( b \) of object \( \delta \) and attribute \( a \) of object \( \delta \) is the only possibly modified field.

Similar to what we observed for \( p\text{modify} \), \( f\text{modify}.X_0 \ast f\text{modify}.X_1 \) is not equal to \( f\text{modify}.(X_0 \cup X_1) \) in general. We do have

\[
g (f\text{modify}.\{(k,a)\} \ast f\text{modify}.\{(l,b)\}) h \equiv \\
\{ \bullet \forall f | f (f\text{modify}.\{(l,b)\}) h | k.f = k.h \}
\]

\[
g (f\text{modify}.\{(k,a), (l,b)\}) h
\]

The fact that attribute names in the field-selection notation are constants, makes it possible to syntactically (by which we mean “regardless of the states \( g \) and \( h \)”) ensure that a modification cannot possibly affect the value of a certain expression. The following theorem holds for example:

\[
f\text{modify}.\{(\langle \delta \ast a \rangle, b)\} \ast f\text{modify}.\{(k,c)\} = \\
\{ \bullet a \neq c \}
\]

\[
f\text{modify}.\{(\langle \delta \ast a \rangle, b), (k,c)\}
\]

The fact that \( a \) and \( c \) are different attribute names, means that modifying a field with attribute name \( c \) cannot possibly affect the value of expression \( (\delta \ast a) \).

Field modification can also be combined with the notion of declared variables. However, instead of field names we use attribute names to indicate the declared variables and we talk about “declared attributes”. For a set \( D \) of (declared) attribute names, \( f\text{modify}_D \in (G \leftarrow G) \leftarrow \wp((\bigodot \leftarrow G) \times \mathbb{A}) \) is defined by

\[
f\text{modify}_D.X
\]

\[
= f\text{modify}.\{(\langle \delta, a \rangle | a \notin D | \delta, a \} \cup X\}
\]

The relation

\[
\lceil \delta \ast a \geq \delta \ast a + \delta \ast b \rceil \cap f\text{modify}_{\{a,b,c\}}.\{(\delta, a)\}
\]

242
expresses for example that attribute \( a \) of object \( o \) is increased at least by the value of attribute \( b \) of object \( o \). From the attributes \( a, b \) and \( c \) of any object, attribute \( a \) of object \( o \) is the only possibly modified field. For every object, all attributes other than \( a, b \) or \( c \) may change arbitrarily.

## 10.4 Carriers

Fields are not the only variables that occur in object-oriented programming. We also encounter things like parameters, a result and a special variable that is usually called “\textit{this}” or “\textit{self}”.

In this section we introduce the notion of a \textbf{carrier}. A carrier is a variable that represents some piece of information that a specification can talk about, like for example the global state or the parameters that are passed to a method. Carriers are described by a \textbf{carrier record} which is a map from \textbf{carrier names} to \textbf{carrier values}. The set of all carrier records is denoted by \( C \). In the following subsections we introduce some common carriers.

### 10.4.1 Global state

The carrier named \texttt{globalState} holds a global state:

\[
C \subseteq G \leftarrow \{\text{globalState}\}
\]

If we talk about the global state of a carrier record, we mean the value of this particular carrier. The function \texttt{globalState} \( \in G \leftarrow C \) returns the global state of a carrier record:

\[
\text{globalState.c} = c,\text{globalState}
\]

### 10.4.2 This

In object-oriented languages, the keyword “\textit{this}” (or sometimes “\textit{self}”) is used to identify the ‘currently active object’. The carrier that holds the ‘\textit{this}’-object name, is named \texttt{this}:

\[
C \subseteq O \leftarrow \{\text{this}\}
\]

The function \texttt{this} \( \in O \leftarrow C \) returns the ‘\textit{this}’-object name of a carrier record:

\[
\text{this.c} = c,\text{this}
\]
10.4.3 Field selection

In order to express field selection in case states are carrier records, we introduce the carrierised field selection $\_ \rightarrow_c \_ \in (I \leftarrow C) \leftarrow (O \leftarrow C \times \mathcal{A})$, defined by

$$(e \rightarrow_c a).c = (K.(e.c) \rightarrow_a).(\text{globalState}.c)$$

An expression that represents the value of attribute $b$ of the object that attribute $a$ of the ‘this’ object is pointing to, is

$$(\text{this} \rightarrow_c a) \rightarrow_c b$$

For the evaluation of the new and old value of an attribute, we introduce the functions $\_ \rightarrow_c \_ \rightarrow_c \_ \in (I \leftarrow C) \leftarrow (O \leftarrow C \times \mathcal{A})$, defined by

$$(e \rightarrow_c a).(c_0, c_1) = (K.(e.(c_0, c_1)) \rightarrow_a).(\text{globalState}.c_0, \text{globalState}.c_1)$$
$$(e \rightarrow_c a).(c_0, c_1) = (K.(e.(c_0, c_1)) \rightarrow_a).(\text{globalState}.c_0, \text{globalState}.c_1)$$

The relation

$$[\text{this} \rightarrow_c a \geq \text{this} \rightarrow_c a + \text{this} \rightarrow_c b] \cap \text{modify}.\{\text{globalState}\}$$

expresses for example that attribute $a$ of the ‘this’ object is increased at least by the old value of attribute $b$ of the ‘this’ object. Furthermore, the only carrier that is possibly modified is $\text{globalState}$.

10.4.4 Field-modify

Also the field-modify has to be adapted in case states are carrier records. The carrierised field-modify $\text{fmodify}_c \in (C \rightarrow C) \leftarrow \varphi((O \leftarrow C) \times \mathcal{A})$ is defined by

$$c_0 \text{ fmodify}_c X \equiv \text{globalState}.c_0 \text{ fmodify}_c \{\langle K.(k.c_1), a \rangle \mid \langle k, a \rangle \in X \mid k, a \} \text{ globalState}.c_1$$

The only fields of the global state that are possibly modified, are the ones for which $\langle k, a \rangle \in X$, where $a$ is the attribute name of the field and the value of $k$ in the old state is the object name of the field.

The relation

$$[\text{this} \rightarrow_c a \geq \text{this} \rightarrow_c a + \text{this} \rightarrow_c b] \cap \text{fmodify}_c.\{(\text{this}, a)\} \cap \text{modify}.\{\text{globalState}\}$$

expresses for example that attribute $a$ of the ‘this’ object is increased at least by the value of attribute $b$ of the ‘this’ object and only attribute $a$ of the ‘this’ object
is possibly modified.

The function $f_{\text{modify},C,D} \in (C \leftarrow C) \leftarrow \wp((\emptyset \leftarrow C) \times A)$, where $D$ is a set of attribute names, combines $f_{\text{modify},C}$ with the notion of declared attributes:

$$f_{\text{modify},C,D}.X = f_{\text{modify},C}((\{\langle \bar{o}, a \rangle \mid a \notin D \mid a, a \} \cup X))$$

10.4.5 Parameters

Parameters are held by the carrier named $\text{parameters}$. This carrier holds a $\text{parameter record}$ which is a map from parameter names to parameter values. The set of all parameter names is denoted by $P$ and the set of all parameter records is $I \leftarrow P$:

$$C \subseteq (I \leftarrow P) \leftarrow \{\text{parameters}\}$$

The function $\text{parameters} \in (I \leftarrow P) \leftarrow C$ returns the parameter record of a carrier record:

$$\text{parameters}.c = c.\text{parameters}$$

The function $p_{\text{par}} \in (I \leftarrow C) \leftarrow P$ returns the value of a certain parameter of a carrier record:

$$p_{\text{par}} = \text{parameters} \cdot \bar{p}$$

10.4.6 Result

The carrier for the result is named $\text{result}$:

$$C \subseteq I \leftarrow \{\text{result}\}$$

The function $\text{result} \in I \leftarrow C$ returns the result of a carrier record:

$$\text{result}.c = c.\text{result}$$

10.4.7 Example

The relation

$$[[\text{result} \geq (\text{this} \cdot \rightarrow_c a) + b_{\text{par}}] \cap \text{modify} \cdot \{\text{result}\}]$$

245
expresses that the new value of the result is at least the value of attribute \( a \) of the ‘this’ object plus the value of parameter \( b \). The only possibly modified carrier is result.

## 10.5 Conclusions

In this chapter we showed how we model several variable-concepts that play a role in object-oriented specification languages like ISpec. We did not pay too much attention to the construction of proof rules, except for a few that illustrate some common ‘pointer issues’.
Chapter 11

Suites

In this chapter we show how the thus far developed theory corresponds to ISpec by defining a formal language for a subset of ISpec.

11.1 Some more operators

We first introduce a few more operators for the semantic models that we introduced in chapters 8 and 9.

11.1.1 Over and under

The statement-over and statement-under \( \hat{\text{over}} \) and \( \hat{\text{under}} \) are defined by

\[
\begin{align*}
(s/s_1).b &= s.b / s_1.b \\
(s_0\setminus s).b &= s_0.b \setminus s.b
\end{align*}
\]

The protocol-over and protocol-under \( \hat{\text{over}} \) and \( \hat{\text{under}} \) are defined by

\[
\begin{align*}
(y;v_0;x)_\tau &\in P/\rho P_1 \\
\equiv & \forall \langle y, v_1, z \mid v = v_0\oplus v_1 \mid (y;v;z)_\tau \in P \leftarrow (x;v_1;z)_\tau \in P_1 \rangle \\
&(x;v_1;z)_\tau \in P_0\setminus\rho P \\
\equiv & \forall \langle y, v_0, v \mid v = v_0\oplus v_1 \mid (y;v_0;x)_\tau \in P_0 \Rightarrow (y;v;z)_\tau \in P \rangle
\end{align*}
\]
The protocol-statement–over and protocol-statement–under \( \tilde{p}_r, \tilde{p}_s \in ((I \rightsquigarrow I) \leftarrow I \leftarrow ((I \rightsquigarrow I) \leftrightarrow I) \times ((I \rightsquigarrow I) \leftarrow I) \) are defined by

\[
(s/p)_{\tilde{r}} \cdot b = s \cdot b / p \cdot s, b \\
(s_0 \cdot p)_{\tilde{s}} \cdot b = s_0 \cdot s, b
\]

The genuine-statement–over and genuine-statement–under \( \tilde{g}_r, \tilde{g}_s \in ((I \rightsquigarrow I) \leftarrow PI \leftarrow ((I \rightsquigarrow I) \leftarrow PI) \times ((I \rightsquigarrow I) \leftarrow PI) \) are defined by

\[
(y:v_0:x) \cdot (s/p)_{\tilde{r}} = (s/p)_{\tilde{s}} \cdot b \\
(y:v_0:x) \cdot (s_0 \cdot p)_{\tilde{s}} = (s_0 \cdot s) \cdot b
\]

The following point-free definitions are equivalent to the above point-wise ones:

\[
\begin{align*}
& s_0 \preceq s_1 \equiv s_0 \cdot s_1 \preceq s \\
& s_1 \preceq s_0 \equiv s_0 \cdot s_1 \preceq s \\
& P_0 \preceq P_1 \equiv P_0 / p \cdot P_1 \preceq P \\
& P_1 \preceq P_0 \cdot P \equiv P_0 / p \cdot P_1 \preceq P \\
& s_0 \preceq s_0 \cdot p \cdot s_1 \equiv s_0 \cdot s_1 \preceq s \\
& s_1 \preceq s_0 \cdot p \cdot s \equiv s_0 \cdot s_1 \preceq s \\
& s_0 \preceq s_0 \cdot p \cdot s_1 \equiv s_0 \cdot p \cdot s_1 \preceq s \\
& s_1 \preceq s_0 \cdot p \cdot s \equiv s_0 \cdot p \cdot s_1 \preceq s
\end{align*}
\]

where the point-free definitions of \( \tilde{g}_r, \tilde{g}_s \) require the additional assumption that \( \tilde{p}_r, \tilde{p}_s \in ((I \rightsquigarrow I) \leftarrow PI) \leftarrow ((I \rightsquigarrow I) \leftarrow PI) \times ((I \rightsquigarrow I) \leftarrow PI) \). So the \( s, s_0 \) and \( s_1 \) in those definitions range over \((I \rightsquigarrow I) \leftarrow PI\).

To prove that these point-free definitions are equivalent with their point-wise counterparts, the only thing that needs to be shown, is that the point-free equations are a consequence of the point-wise definitions. We leave these proofs to the reader. The other direction then automatically holds because the point-free equations are Galois connections, although we should then use a definition of Galois connection in the context of an arbitrary poset \( (A, \preceq) \) instead of only the particular poset \( (PI, \subseteq) \) that we used in section 3.5. Because adjoints in a Galois connection are unique with
respect to their behaviour within the poset, the point-wise defined operators must be those defined by the point-free equations.

The operators \( \hat{\wedge}, \hat{\lor}, \hat{\land}, \hat{\lor}_p, \hat{\wedge}_g \) and \( \hat{\lor}_q \) are not continuity preserving in general. The following theorems show when they do preserve continuity:

\[
\begin{align*}
\hat{s}^1 & \in \Pi \iff s \in \Pi \iff s_1 \subseteq \text{const.I} \\
\hat{s}_0 & \in \Pi \iff s \in \Pi \iff s_0 \subseteq \text{const.I} \\
\hat{s}_p & \in \Pi \iff s \in \Pi \iff s_1 \subseteq \text{dark \, \spc.I} \\
\hat{s}_0 & \in \Pi \iff s \in \Pi \iff s_0 \subseteq \text{dark \, \spc.I} \\
\hat{s}_g & \in \Pi \iff s \in \Pi \iff s_1 \subseteq \text{dark \, \spc.I} \\
\hat{s}_0 & \in \Pi \iff s \in \Pi \iff s_0 \subseteq \text{dark \, \spc.I}
\end{align*}
\]

These theorems are a trivial consequence of the following theorems, whose proofs we leave to the reader (\( B \) is a set):

\[
\begin{align*}
\hat{s}^1(\text{const.B}) & = s^1(\text{const.B}) \cup \hat{\Pi}^1(\text{const.}(I \backslash B)) \\
(\text{const.B})\hat{s} & = (\text{const.B})\hat{s} \cup (\text{const.}(I \backslash B))\hat{\Pi} \\
\hat{s}_p(\text{dark \, \spc.B}) & = s_p^1(\text{dark \, \spc.B}) \cup \hat{\Pi}_p(\text{dark \, \spc.}(I \backslash B)) \\
(\text{dark \, \spc.B})\hat{s}_p & = (\text{dark \, \spc.B})\hat{s}_p \cup (\text{dark \, \spc.}(I \backslash B))\hat{\Pi} \\
\hat{s}_g & = (\hat{s}_p s_1) \backslash \hat{\Pi}_g \\
\hat{s}_0 & = (\hat{s}_0 s_0) \backslash \hat{\Pi}_g
\end{align*}
\]

In the language that we introduce in this chapter, the operator \( \hat{\lor}_g \) is used to combine pres and posts (see section 11.2.4).

### 11.1.2 Arbitrary intersection

The arbitrary statement-intersection \( \bigcap \in (\Pi \leftarrow I) \iff \varphi(\Pi \leftarrow I) \) is defined by

\[
(\bigcap w).b = \bigcap \{s, b \mid s \in w \} \backslash s
\]

The arbitrary genuine-statement–intersection \( \bigcap_{\rho} \in (\Pi \leftarrow I) \iff \varphi(\Pi \leftarrow_g I) \) is defined by

\[
\begin{align*}
\bigcap_{\rho} & = \hat{\Pi}_g \\
\bigcap_{\rho} w & = \bigcap w \iff w \neq \emptyset
\end{align*}
\]

We leave it to the reader to verify that \( \bigcap w \) is continuous for a finite set \( w \) of con-
timuous statements. The fact that $\bigcap_w \cap g$ is continuous for a finite set $w$ of continuous
genuine statements, is a trivial consequence of the followi
g theorem, whose proof
is also left to the reader:
\[
\bigcap_w \cap g = \bigcap \cap g
\]

11.1.3 Aspect operator

The statement–aspect-operator $\hat{\ell} < | \in ((I \rightleftharpoons I) \rightleftharpoons \rightleftharpoons I) \leftarrow (I \rightleftharpoons I) \rightleftharpoons (I \rightleftharpoons I) \leftarrow (I \rightleftharpoons I) \times (\wp I)$ is defined by
\[
(s \hat{\ell} < | B).b = s.b < | B
\]
The protocol–aspect-operator $\hat{\ell} p < | \in ((I \rightleftharpoons I) \rightleftharpoons \rightleftharpoons I) \leftarrow (I \rightleftharpoons I) \times (\wp I)$ is defined by
\[
(Y:x:Z) \in P \hat{\ell} p B
\]
\[
= \exists (y, j) \in Y = [(y, j)] \land Z = [(z, j)] \land j \in B \Rightarrow (y:x:z) \in P
\]
The protocol-statement–aspect-operator $\hat{\ell} p \hat{\ell} \in ((I \rightleftharpoons I) \rightleftharpoons \rightleftharpoons I) \leftarrow (I \rightleftharpoons I) \times (\wp I)$ is defined by
\[
(s \hat{\ell} < | p B).b = s.b < | p B
\]
and the genuine-statement–aspect-operator $\hat{\ell} g < | \in ((I \rightleftharpoons I) \rightleftharpoons \rightleftharpoons I) \leftarrow (I \rightleftharpoons I) \times (\wp I)$ is defined by
\[
(Y:x:Z) \in (s \hat{\ell} < | g B).b
\]
\[
= \exists (y, j) \in Y = [(y, j)] \land Z = [(z, j)] \land j \in B \Rightarrow (y:x:z) \in s.b
\]
\[
\land j \notin B \Rightarrow v < b
\]
or in terms of $\hat{\ell} p$:
\[
s \hat{\ell} g B = s \hat{\ell} p B \cap \hat{\Pi} g
\]

The operators $(\hat{\ell} B)$, $(\hat{\ell} p B)$ and $(\hat{\ell} g B)$ are continuity preserving:
\[
s \hat{\ell} B \in \Pi \leftarrow \Pi \leftarrow \Pi
\]
\[
s \hat{\ell} p B \in \Pi \leftarrow \Pi \leftarrow \Pi
\]
\[
s \hat{\ell} g B \in \Pi \leftarrow \Pi \leftarrow \Pi
\]

We leave it to the reader to verify these facts (notice that it is not possible to
simply use the definition of the aspect operator in terms of the pack, because the
arbitrary intersection of an infinite set of continuous statements is not necessarily continuous).

11.2 Formal ISpec language

We now introduced enough machinery for the definition of our formal ISpec language. We do not define an explicit syntax together with a denotational semantics function, but simply introduce a number of (semantic) operators. The operators that are introduced in this section are called ISpec operators. Although in principle every mathematical operator may be used to construct input for an ISpec operator (as long as the types match), the following picture shows the intended information flow:

The operators interface and roleInvariant are for example intended to be used for the construction of input for operator role. Section 11.3 contains an example that shows the intended use of the operators.

Comparing the above structure to the syntax that was presented in chapter 2, several differences can be observed. These differences are a result of the fact that the above structure follows the intended semantic structure of ISpec whereas the syntax that was presented in chapter 2 follows traditional syntactic structures.

The top-most operator, suite, constructs a suite statement. We do not call this a “suite”, because we prefer to use the term “suite” for a broader concept a part of which is a suite statement. Other parts could be a name for the suite and a
graphical icon representing the suite. In informal explanations, we often talk in terms of these broader concepts to keep text readable.

A suite is a hierarchically structured collection of procedures. The hierarchy is constituted by role names, interface names and method names, represented by the sets $\mathbb{R}$, $\mathbb{I}$ and $\mathbb{M}$ respectively. Although in principle we leave these sets unspecified, we often use the set $\mathbb{I}$ of identifiers for them. The procedures that a suite consists of, operate on states that are carrier records (see section 10.4). In this chapter we fix the type $\mathbb{C}$ of carrier records by the equation

$$\mathbb{C} = \mathbb{G} \sqcap \mathbb{O} \sqcap \mathbb{I} \sqcap \mathbb{M}$$

The set of the four carrier names that are used for our formal ISpec language, is denoted by $IC$ (ISpec carriers):

$$IC = \{\text{globalState}, \text{this}, \text{parameters}, \text{result}\}$$

This set of declared-carrier names is used at places where we need to modify carriers.

Because procedures should be able to call other procedures, a suite statement is defined as a statement, with suite behaviours as input and output. Suite behaviours are connection protocols of type $\mathbb{B}$, defined by

$$\mathbb{B} = ((\mathbb{C} \rightarrow \mathbb{M}) \rightarrow \mathbb{I}) \rightarrow \mathbb{R}) \rightarrow ((\mathbb{C} \rightarrow \mathbb{M}) \rightarrow \mathbb{I}) \rightarrow \mathbb{R}$$

This type reflects the fact that a product-like construct is used to constitute the hierarchy of procedures (see sections 11.2.1, 11.2.2 and 11.2.3).

The suite statements that we are actually interested in, are the ones that are genuine and continuous. We made genuineness part of the types of the relevant operators. This means that we, the designers of the operators, ensure that if an operator is applied to an element of its domain, the result is a genuine statement. For these particular operators this is obvious from the fact that they are defined in terms of genuineness-preserving operators. We do not mention this fact explicitly anymore. We could have done the same thing for continuity, but then we needed to introduce again more notation. We refrained from doing so, also because it was not necessary for obtaining elegant formulas as opposed to the genuineness case.

We now define the output types of the main ISpec operators:
Suite = (((((C) → M) → (I)) → (R)) → (suites)) ∈ y B
Role = (((C) → M) → (I)) → y B
Interface = ((C) → (M)) → (I) → y B
Method = (C) → C → y B
Post = (C) → C → y B
ActionClause = (C) → C → y B
PostCondition = (C) → C → y B
ResultType = (C) → C → y B
Pre = (C) → C → y B
PreCondition = (C) → C → y B
ParametersType = (C) → C → y B
RoleInvariant = (((C) → M) → (I)) → (C) → (M) → y B
ActionInvariant = (C) → C → y B
PostInvariant = (C) → C → y B
StateInvariant = (C) → C → y B
AttributeType = (C) → C → y B

The following table shows how the elements of the above types are called:

<table>
<thead>
<tr>
<th>Type</th>
<th>Statement Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suite</td>
<td>suite statement</td>
</tr>
<tr>
<td>Role</td>
<td>role statement</td>
</tr>
<tr>
<td>Interface</td>
<td>interface statement</td>
</tr>
<tr>
<td>Method</td>
<td>method statement</td>
</tr>
<tr>
<td>Post</td>
<td>post statement</td>
</tr>
<tr>
<td>ActionClause</td>
<td>action-clause statement</td>
</tr>
<tr>
<td>PostCondition</td>
<td>post-condition statement</td>
</tr>
<tr>
<td>ResultType</td>
<td>result-type statement</td>
</tr>
<tr>
<td>Pre</td>
<td>pre statement</td>
</tr>
<tr>
<td>PreCondition</td>
<td>pre-condition statement</td>
</tr>
<tr>
<td>ParametersType</td>
<td>parameters-type statement</td>
</tr>
<tr>
<td>RoleInvariant</td>
<td>role-invariant statement</td>
</tr>
<tr>
<td>ActionInvariant</td>
<td>action-invariant statement</td>
</tr>
<tr>
<td>PostInvariant</td>
<td>post-invariant statement</td>
</tr>
<tr>
<td>StateInvariant</td>
<td>state-invariant statement</td>
</tr>
<tr>
<td>AttributeType</td>
<td>attribute-type statement</td>
</tr>
</tbody>
</table>

In the following subsections we formally define the ISpec operators.

### 11.2.1 Suites

The ISpec operator suite ∈ Suite ← φ(Role×R), defined by
suite.roles
= \bigcap_{\text{role } \triangleright g \text{ r} | \langle \text{role}, \text{r} \rangle \in \text{roles} | \text{role}, \text{r}}
constructs a suite statement from a set of \langle \text{role statement}, \text{role name} \rangle-pairs, using a conjoint-sum kind of construct.

11.2.2 Roles
The ISpec operator \text{role } \in \text{Role} \leftarrow \varphi(\text{Interface} \times \text{I} \times \text{RoleInvariant}), defined by

\text{role.}\langle \text{interfaces}, \text{roleInv} \rangle
= \bigcap_{\text{interface } \triangleright g \text{ i} | \langle \text{interface}, \text{i} \rangle \in \text{interfaces} | \text{interface}, \text{i}} \cap \text{roleInv}
constructs a role statement from a set of \langle \text{interface statement}, \text{interface name} \rangle-pairs. A role-invariant statement \text{roleInv} imposes additional constraints.

11.2.3 Interfaces
The ISpec operator \text{interface } \in \text{Interface} \leftarrow \varphi(\text{Method} \times \text{M}), defined by

\text{interface.}\langle \text{methods} \rangle
= \bigcap_{\text{method } \triangleright g \text{ m} | \langle \text{method}, \text{m} \rangle \in \text{methods} | \text{method}, \text{m}}
constructs an interface statement from a set of \langle \text{method statement}, \text{method name} \rangle-pairs.

11.2.4 Methods
The ISpec operator \text{method } \in \text{Method} \leftarrow \varphi(\text{Post} \times \text{Pre}), defined by

\text{method.}\langle \text{post}, \text{pre} \rangle
= post \triangleright g \text{ pre}
constructs a method statement from a post statement and a pre statement. The operators that we introduce to construct pre statements are such that each constructed pre statement \text{pre} satisfies \text{pre} \subseteq \text{dArk} \cap \text{spec.I}. This allows us to interpret \text{post } \triangleright g \text{ pre} as “\text{post} holds if \text{pre} holds” and ensures preservation of continuity as
shown in section 11.1.1.

### 11.2.5 Posts

The ISpec operator \( \text{post} \in \text{Post} \leftarrow \text{ActionClause} \times \text{PostCondition} \times \text{ResultType} \), defined by

\[
\text{post}(\text{actionClause}, \text{postCondition}, \text{resultType}) = \text{actionClause} \cap \text{postCondition} \cap \text{resultType}
\]

constructs a **post statement** from an action-clause statement, a post-condition statement and a result-type statement.

### 11.2.6 Action clauses

The ISpec operator \( \text{actionClause} \in \text{ActionClause} \leftarrow (\mathcal{C} \rightarrow \mathcal{C} \leftarrow \mathcal{B}) \), defined by

\[
\text{actionClause} = \text{actionClause}.s
\]

constructs an **action-clause statement** from a statement \( s \) of the proper type. Some operators for the construction of these \( s \)s are introduced in sections 11.2.18, 11.2.19 and 11.2.20.

### 11.2.7 Post-conditions

The ISpec operator \( \text{postCondition} \in \text{PostCondition} \leftarrow (\mathcal{C} \rightarrow \mathcal{C}) \), defined by

\[
\text{postCondition}.S = \text{spec}_p.S
\]

constructs a **post-condition statement** from a binary relation \( S \), ensuring \( S \) to hold between the new and old state.

### 11.2.8 Result types

The ISpec operator \( \text{resultType} \in \text{ResultType} \leftarrow \mathcal{I} \), defined by
resultType.\( V \) \\
= \[
\text{sp\( \varepsilon \_g_r\)\( \varepsilon \_r\) result} \in V
\]
constructs a result-type statement from a set \( V \), ensuring that the new value of carrier \( \text{result} \) is an element of \( V \).

11.2.9 Pres

The ISpec operator \( \text{pre} \in \text{Pre} \leftarrow \text{PreCondition} \times \text{ParametersType} \), defined by

\[
\text{pre}.\langle \text{preCondition}, \text{parametersType} \rangle \\
= \text{preCondition} \cap \text{parametersType}
\]
combines a pre-condition statement and a parameters-type statement into a pre statement.

11.2.10 Pre-conditions

The ISpec operator \( \text{preCondition} \in \text{PreCondition} \leftarrow \varphi C \), defined by

\[
\text{preCondition}.C \\
= \text{dark} \cap \text{sp\( \varepsilon \_g_r\)C}
\]
constructs a pre-condition statement from a set of states \( C \).

11.2.11 Parameters types

The ISpec operator \( \text{parametersType} \in \text{ParametersType} \leftarrow \varphi (\varphi I \times \mathbb{P}) \), defined by

\[
\text{parametersType}.W \\
= \text{dark} \cap \text{sp\( \varepsilon \_g_r\)}(f.W)
\]
where \( f \in \varphi C \leftarrow \varphi (\varphi I \times \mathbb{P}) \) is defined by

\[
c \in f.W \\
\equiv \forall \{V, p \mid \{V, p\} \in W \mid (p_{par}).c \in V\}
\]
constructs a parameters-type statement from a set of \((\text{parameter type}, \text{parameter name})\)-pairs \(W\).

### 11.2.12 Role invariants

The ISpec operator \(\text{roleInvariant} \in \text{RoleInvariant} \leftarrow (\varphi(\text{ActionInvariant} \times \varnothing) \times \varphi(\text{PostInvariant} \times \varnothing) \times \varphi(\text{StateInvariant} \times \varnothing) \times \varphi(\text{AttributeType} \times \varnothing))\), defined by

\[
\text{roleInvariant}.(\text{actionInvs}, \text{postInvs}, \text{stateInvs}, \text{attributeTypes}) = \\
\bigcap_b \{ (\text{ai} < g M) \prec g I \setminus \text{is} | (\text{ai}, \text{is}) \in \text{actionInvs} \big| \text{ai}, \text{is} \} \cap \\
\bigcap_b \{ (\text{pi} < g M) \prec g I \setminus \text{is} | (\text{pi}, \text{is}) \in \text{postInvs} \big| \text{pi}, \text{is} \} \cap \\
\bigcap_b \{ (\text{si} < g M) \prec g I \setminus \text{is} | (\text{si}, \text{is}) \in \text{stateInvs} \big| \text{si}, \text{is} \} \cap \\
\bigcap_b \{ (\text{at} < g M) \prec g I \setminus \text{is} | (\text{at}, \text{is}) \in \text{attributeTypes} \big| \text{at}, \text{is} \}
\]

combines four kinds of invariant statements into one role-invariant statement. The four kinds of invariants that a role-invariant is constructed from, are action invariants, post invariants, state invariants and attribute types. Their construction is treated in sections 11.2.13, 11.2.14, 11.2.15 and 11.2.16.

The set of interface names attached to an invariant, denotes excluded interfaces. (The methods of) these interfaces do not necessarily adhere to that particular invariant. The crux of the observer pattern is the re-establishment of the state invariants that postulate that the copy of an observer is equal to the orig of its subject. The interfaces that are used for re-establishing a state invariant, often have to be excluded from the state invariant itself (see sections 2.4.2 and 11.4).

### 11.2.13 Action invariants

The ISpec operator \(\text{actionInvariant} \in \text{ActionInvariant} \leftarrow ((C \rightarrow C) \leftarrow g B)\), defined by

\[
\text{actionInvariant}.s = \\
s
\]

constructs an action-invariant statement from a statement of the proper type. An action invariant is like an action clause, except for the fact that it is intended to be distributed over multiple methods.

### 11.2.14 Post invariants

The ISpec operator \(\text{postInvariant} \in \text{PostInvariant} \leftarrow (C \rightarrow C)\), defined by
postInvariant.\( S \)
\[
= \text{spec}_p.\( S \)
\]
constructs a \textbf{post-invariant statement} from a binary relation between states. Post invariants are a less expressive kind of action invariants (similar to post-conditions versus action clauses), describing only a relation between the new and old state.

### 11.2.15 State invariants

The ISpec operator \texttt{stateInvariant} \( \in \text{StateInvariant} \), defined by

\[
\text{stateInvariant}.\( C \)
= \text{spec}_p.(C \cdot \Pi \cdot C)
\]

constructs a \textbf{state-invariant statement} from a set \( C \) of states. It can be used to ensure that the old and new state are an element of \( C \). See section 7.7.2 for details about the specific way we model state invariants.

### 11.2.16 Attribute types

The ISpec operator \texttt{attributeType} \( \in \text{AttributeType} \), defined by

\[
\text{attributeType}.\langle V, a \rangle
= \text{stateInvariant}.\{ \text{this} \rightarrow_c a \in V \}
\]

constructs an \textbf{attribute-type statement} from an \( \langle \text{attribute type, attribute name} \rangle \)-pair. An attribute type is a less expressive kind of state invariant that only expresses that a certain attribute of the ‘this’ object has some specific type.

### 11.2.17 Primitive types

The ISpec operators \texttt{void, bool, int, object} \( \in \wp I \), defined by

\[
\begin{align*}
\text{void} & = I \\
\text{bool} & = B \\
\text{int} & = \mathbb{Z} \\
\text{object} & = \mathbb{O}
\end{align*}
\]

represent some \textbf{primitive types} intended to be used as parameter types, result
types and attribute types. Notice that we use one type (object) for objects instead of a type for each role name. This is explained in section 11.2.20.

11.2.18 Sequential composition

The ISpec-sequential-composition is defined by

\[ s_0; s_1 = s_1 \circ p s_0 \]

We reversed the order of statements because most people prefer reading from left to right.

11.2.19 Modify

In ISpec, a ‘modify’ indicates which attributes of which objects possibly change. Apart from this, in ISpec a ‘modify’ also means that no calls are performed on methods of the suite’s declared interfaces, also not indirectly via a method of an undeclared interface. We call a modify of this kind a dark modify. The declared interfaces of a suite are the interfaces that the suite ‘talks about’. Opposed to methods of declared interfaces, methods of undeclared interfaces can be called during a dark modify.

The fact that no (direct or indirect) calls may be performed on declared interfaces, described by a set of interface names \( I \), can be expressed by means of the operator \( \text{dark}_I \), defined by

\[ \text{dark}_I \in (\mathcal{C} \leftarrow \mathcal{I} \mathcal{M}) \leftarrow \mathcal{I}(\mathcal{R}) \leftarrow \mathcal{B} \]

Notice that we use sets of interface names to describe declared interfaces and not sets of (role name, interface name)-pairs. This simplification is justified by the fact that, as mentioned in section 2.4.1, we assume all interfaces in a suite to have different names.

The statement-version of \( \text{dark}_I \) is \( \text{dark}^s_I \in (\mathcal{C} \leftarrow \mathcal{I} \mathcal{M}) \leftarrow \mathcal{B} \), defined by

\[ \text{dark}^s_I \in (\mathcal{C} \leftarrow \mathcal{I} \mathcal{M}) \leftarrow \mathcal{B} \]

The genuine version \( \text{dark}^g_I \in (\mathcal{C} \leftarrow \mathcal{I} \mathcal{M}) \leftarrow \mathcal{B} \) is defined by
\[
dark_{gI} = \dark_I \cap \bar{I}_g
\]
The dark modify \(\text{dmodify}_{A,I} \in ((C \cdots C) \leftarrow g \ B) \leftarrow \varphi((\Omega \leftarrow C \times A))\) is now defined by
\[
\text{dmodify}_{A,I}.X = \text{smody}_{A,X} \cap \dark_{gI}
\]
where the spec-modify \(\text{smody}_{A} \in ((C \cdots C) \leftarrow g \ B) \leftarrow \varphi((\Omega \leftarrow C \times A))\) is defined by
\[
\text{smody}_{A}.X = \text{spec}_{p}(\text{modify}_{IA,X} \cap \text{modify}_{IC} \cdot \{\text{globalState}, \text{result}\})
\]
The statement \(\text{dmodify}_{\{a,b,c\},\{IA,IB\},\{\langle \text{this}, a \rangle\}}\) for example means that no calls are performed on the interfaces \(IA\) and \(IB\), that the carriers \text{parameters} and this remain equal and that from the attributes \(a, b, c\) of any object, only attribute \(a\) of the 'this' object possibly changes. The reason for allowing carrier \text{result} to arbitrarily change, is to avoid undesirable result dependencies.

### 11.2.20 Call

The most complicated ISpec operator is the one for method calls. The operator \((- > \text{call} \cdot i \cdot m) \in ((C \cdots C) \leftarrow g \ B) \leftarrow \varphi((\Omega \leftarrow C \times \emptyset \times \emptyset \times \emptyset \times \emptyset \times \emptyset))\) is defined by
\[
e - > \text{call} r::i::m ps = (\dark \cap \text{spec}_{p}(\text{modify}_{IC} \cdot \{\text{this, parameters}\})) \cap \text{spec}_{p}(\text{modify}_{IC} \cdot \{\text{this, parameters, result}\} \cap \text{setThis.e} \cap \text{setParameters.ps})
\]
where (the \(\_\_\_\_\_\_\_\_\_\_\) are just brackets)
\[
\text{setThis.e} = \{[\text{this} = e]\}
\text{setParameters.ps} = \{[\text{ps.par} = \text{ps.p}] | p \in \text{ps.p} | p\}
\]

We explain this definition by describing in an operational manner what happens
during a method call.

First, by means of “setThis.e”, carrier this is assigned the value that expression e has in the old state and by means of “setParameters.ps”, carrier parameters is modified such that each parameter whose name p is in the domain of ps, is assigned the value that its corresponding expression ps.p has in the old state. To prevent complications, we require e and all ps.p to be total functions. Notice that the function setParameters is defined in such a way that it is possible to add parameters in a refinement of a call. Carrier result is part of “modifyIC::this, parameters, result” to avoid unwanted result dependencies.

The following step in a method call is a (visible) call to method m of interface i of role r. In section 11.2.17 we mentioned that we only have one type to represent objects: object. In ISpec, the type of an object is represented by means of a role name. In our formal ISpec language, all role names are replaced by object. A type checker should be able to determine the role name that corresponds to the object expression e of a call. This can then be used to determine the role name r of a call if this role name is not mentioned explicitly, or to check the role name if it is mentioned explicitly.

In the final step of a method call, the carriers this and parameters are changed back to their original values by “dark ∩ spéc_p(modifyIC::this, parameters)” and “∩ spéc_p(modifyIC::globalState, result)”. After execution of the call, carrier result contains the result of the call.

In ISpec it is allowed to refine a call by adding modifications of undeclared attributes and calls on undeclared interfaces before and after the call. This can be modeled by putting a dmodifyA, I.{} before and one after each call. For a set of attribute names A and a set of interface names I, we define the ISpec-call := ∈ (C ← − g B) ← (Q ← − C) × R × I × M × ((I ← − C) ← − P) by

\[
e \rightarrow_{\text{call}A, I :: :: m} ps
\]

\[
\hat{p} e \rightarrow_{\text{call}A, I :: :: m} ps
\]

\[
\hat{p} \text{ dmodifyA, I.{} }
\]

Notice that the result of the call is now lost. It would be possible to store the result in the carrier result by adding “∩ spéc_p[result = result]” to the top-most dmodify. This can however lead to undesirable result dependencies. We therefore choose to introduce a call that is combined with an assignment of the result to some attribute of the ‘this’ object, defining the operator := := ∈ (C ← − g B) ← A × (Q ← − C) × R × I × M × ((I ← − C) ← − P) by

\[
e \rightarrow_{\text{call}A, I :: :: m} ps
\]

\[
\hat{p} e \rightarrow_{\text{call}A, I :: :: m} ps
\]

\[
\hat{p} \text{ dmodifyA, I.{} }
\]

Notice that the result of the call is now lost. It would be possible to store the result in the carrier result by adding “∩ spéc_p[result = result]” to the top-most dmodify. This can however lead to undesirable result dependencies. We therefore choose to introduce a call that is combined with an assignment of the result to some attribute of the ‘this’ object, defining the operator := := ∈ (C ← − g B) ← A × (Q ← − C) × R × I × M × ((I ← − C) ← − P) by

\[
e \rightarrow_{\text{call}A, I :: :: m} ps
\]

\[
\hat{p} e \rightarrow_{\text{call}A, I :: :: m} ps
\]

\[
\hat{p} \text{ dmodifyA, I.{} }
\]
11.3 Plug pattern

We are now able to formalise the plug pattern and observer pattern that were introduced in chapter 2. We start with the plug pattern:

\[
a := e \Rightarrow \text{call} A, I r::\text{m} ps
\]

\[
d\text{modify}_A, I, \{\langle \text{this}, a \rangle \} \cap \text{spéc}_A, \{\text{this} \Rightarrow \text{c} a = \text{result}\}
\]

\[
\text{fillWith, } \text{insertInto, remove, } \text{plugPattern, hasPlug, plug, inHole, hole, Hole, Plug}
\]

11.3.1 Signature

The attributes and interfaces that are declared by the plug pattern’s class diagram, are represented by the sets \( PPA \) and \( PPI \) respectively. Together they constitute the signature of the suite:

\[
PPA = \{\text{hasPlug, plug, inHole, hole}\}
\]

\[
PPI = \{\text{IFill, IIInsert}\}
\]

These sets are used for the “\( A \)” and “\( I \)” of modify statements and call statements.

11.3.2 Hierarchy

The role-interface-method hierarchy that is defined by the class diagram, is constructed with the ISpec operators suite, role and interface. Role Main is not part of our formalisation as explained in section 2.5.6.
The sets $A$ and $I$ represent the declared attributes and declared interfaces respectively (the signature). They are passed as arguments, starting from $\text{PlugPattern}_{A,I}$. The formula $\text{PlugPattern}_{PPA, PPI}$ thus represents the plug pattern. The reason why we pass arguments $A$ and $I$, and not simply use $PPA$ and $PPI$ at places where they are needed, is that in the construction of refinements of a suite, we often use the same statements, except for the fact that they are interpreted in an extended signature. This is similar to inheritance of method implementations in object-oriented programming. See sections 2.4.4 and 11.4.2 for further explanation.

### 11.3.3 Invariants

Next to the role-interface-method hierarchy, the class diagram also defines attribute types for the roles Hole and Plug. In our formal ISpec language, these are part of the role invariants:

\[
HRInv_{A,I} = \text{roleInvariant}. (\{\},  \\
(\{HSInv_{1},\{\}\},(HSInv_{2},\{\}\)),  \\
(\{HHasPlugT,\{\}\},(HPlugT,\{\}\)))
\]

\[
PRInv_{A,I} = \text{roleInvariant}. (\{\},  \\
(\{PSInv_{1},\{\}\},(PSInv_{2},\{\}\)),  \\
(\{PinHoleT,\{\}\},(PHoleT,\{\}\)))
\]

The four attribute-type statements $HHasPlugT$, $HPlugT$, $PinHoleT$ and $PHoleT$ are defined by

\[
HHasPlugT = \text{attributeType}. (\text{bool, hasPlug})
\]

\[
HPlugT = \text{attributeType}. (\text{object, plug})
\]

\[
PinHoleT = \text{attributeType}. (\text{bool, inHole})
\]

\[
PHoleT = \text{attributeType}. (\text{object, hole})
\]

Notice the replacement of a role name by object, as explained in section 11.2.20.

The state-invariant statements $HSInv_{1}$, $HSInv_{2}$, $PSInv_{1}$ and $PSInv_{2}$, in section
2.2.2 represented by

\[ HSI_{\text{inv}} 1 = \text{stateInvariant hasPlug} \Rightarrow \text{plug.inHole} \]

\[ HSI_{\text{inv}} 2 = \text{stateInvariant hasPlug} \Rightarrow \text{plug.hole = this} \]

\[ PS_{\text{inv}} 1 = \text{stateInvariant inHole} \Rightarrow \text{hole.hasPlug} \]

\[ PS_{\text{inv}} 2 = \text{stateInvariant inHole} \Rightarrow \text{hole.plug = this} \]

are in our formal ISpec language defined by

\[ HSI_{\text{inv}} 1 = \text{stateInvariant.} \]
\[ \{ \text{this } \rightarrow_{c} \text{ hasPlug} \Rightarrow \text{(this } \rightarrow_{c} \text{ plug) } \rightarrow_{c} \text{ inHole} \} \]

\[ HSI_{\text{inv}} 2 = \text{stateInvariant.} \]
\[ \{ \text{this } \rightarrow_{c} \text{ hasPlug} \Rightarrow \text{(this } \rightarrow_{c} \text{ plug) } \rightarrow_{c} \text{ hole = this} \} \]

\[ PS_{\text{inv}} 1 = \text{stateInvariant.} \]
\[ \{ \text{this } \rightarrow_{c} \text{ inHole} \Rightarrow \text{(this } \rightarrow_{c} \text{ hole) } \rightarrow_{c} \text{ hasPlug} \} \]

\[ PS_{\text{inv}} 2 = \text{stateInvariant.} \]
\[ \{ \text{this } \rightarrow_{c} \text{ inHole} \Rightarrow \text{(this } \rightarrow_{c} \text{ hole) } \rightarrow_{c} \text{ plug = this} \} \]

11.3.4 Methods

What is left, is the definition of the individual methods. For each method of the plug pattern we give its description from section 2.2.3, followed by its formalisation:
HFillWith = method $fillWith$
  $<HFiParameter>$
  $HFiResult$
  $<HFiEffect>$

HFiParam = parameter $p$
  objectType Plug

HFiResult = result
  valueType void

HFiEffect = effect
  HFiPreCondition
  HFiActionClause
  HFiPostCondition

HFiPreCondition = preCondition
  $\neg$hasPlug
  $\wedge$ p.inHole
  $\wedge$ p.hole = this

HFiActionClause = actionClause
  modify{hasPlug,plug}

HFiPostCondition = postCondition
  hasPlug
  $\wedge$ plug = p

HFilWith_{A,I} = method.
  $\{HFiPost_{A,I}$
  , $HFiPre\}$

HFiPost_{A,I} = post.
  $\{HFiActionClause_{A,I}$
  , $HFiPostCondition$
  , $HFiResultType\}$

HFiPre = pre.
  $\{HFiPreCondition$
  , $HFiParametersType\}$

HFiParamType = parametersType.{\{object.p\}}

HFiResultType = resultType. void

HFiPreCondition = preCondition.
  $\{\neg$hasPlug
  $\wedge$ p.inHole
  $\wedge$ p.hole = this\}$

HFiActionClause_{A,I} = actionClause.
  dmodify_{A,I}.{\{hasPlug
  ,\{this,plug\}\}}

HFiPostCondition = postCondition.
  $\{\neg$hasPlug
  $\wedge$ this $\rightarrow c$ plug = p.par\}$
HEmpty = method empty
<>
HEmResult
<HEmEffect>

HEmResult = result valueType void

HEmEffect = effect HEmPreCondition
HEmActionClause
HEmPostCondition

HEmPreCondition = preCondition hasPlug

HEmActionClause = actionClause modify(hasPlug, plug)

HEmPostCondition = postCondition ¬hasPlug

HEmpty_{A, I} = method⟨HEmPost_{A, I}, HEmPre⟩

HEmPost_{A, I} = post⟨HEmActionClause_{A, I}, HEmPostCondition, HEmResultType⟩

HEmPre = pre⟨HEmPreCondition, HEmParametersType⟩

HEmParametersType = parametersType.\

HEmResultType = resultType. void

HEmPreCondition = preCondition. [this \rightarrow_{c} hasPlug]

HEmActionClause_{A, I} = actionClause. dmodify_{A, I}.{(this, hasPlug), (this, plug)}

HEmPostCondition = postCondition. [\neg (this \rightarrow_{c} hasPlug)]
$PI_{\text{nto}}$ = method
insertInto
<$PI_{\text{InParameter}}$>
$PI_{\text{nResult}}$
<$PI_{\text{InEffect}}$>

$PI_{\text{InParameter}}$ = parameter
$PI_{\text{InParameter}}$

$PI_{\text{nResult}}$ = result
$PI_{\text{nResult}}$

$PI_{\text{nEffect}}$ = effect
$PI_{\text{nEffect}}$

$PI_{\text{nPreCondition}}$ = preCondition
$PI_{\text{nPreCondition}}$

$PI_{\text{nActionClause}}$ = actionClause
$PI_{\text{nActionClause}}$

$PI_{\text{nPostCondition}}$ = postCondition
$PI_{\text{nPostCondition}}$

$PI_{\text{nto}, A, I}$ = method.
$PI_{\text{nto}, A, I}$

$PI_{\text{Post}, A, I}$ = post.
$PI_{\text{Post}, A, I}$

$PI_{\text{Pre}}$ = pre.
$PI_{\text{Pre}}$

$PI_{\text{ParametersType}}$ = parametersType.{(object, h)}
$PI_{\text{ParametersType}}$

$PI_{\text{ResultType}}$ = resultType.
$PI_{\text{ResultType}}$

$PI_{\text{PreCondition}}$ = preCondition.
$PI_{\text{PreCondition}}$

$PI_{\text{ActionClause}, A, I}$ = actionClause.
$PI_{\text{ActionClause}, A, I}$

$PI_{\text{PostCondition}}$ = postCondition.
$PI_{\text{PostCondition}}$

267
11.3.5 A trace of insertInto

In this section we explain the relationship between the description of a suite in terms of I$\text{S}$pec operators (syntax) and its behaviour (semantics). We do this by means of
a trace for method \texttt{insertInto} of the plug-pattern suite. Actually we should talk about a \textit{family} of traces, because the trace contains mathematical variables. We start with a graphical representation of the trace that we consider:

\begin{center}
\begin{tikzpicture}
  \node (h) at (0,0) {\texttt{h}};
  \node (p) at (2,0) {\texttt{p}};
  \draw[-latex] (h) -- node[above] {\texttt{insertInto}} (p);
  \draw[-latex] (h) -- (p.north west) node[above] {\texttt{fillWith}};
  \draw[-latex] (h) -- (p.north east) node[above] {\texttt{fillWith}};
\end{tikzpicture}
\end{center}

The trace represents the interaction between objects \texttt{h} (for hole) and \texttt{p} (for plug). It is possible that \texttt{h} equals \texttt{p}, in which case we are looking at an object talking to itself.

The trace starts with a call of method \texttt{insertInto} on object \texttt{p} by ‘someone from the outside’ (the main program). The parameter value that is passed, is \texttt{h}. The rectangle that follows, shows the values of the global state’s significant fields. Attribute \texttt{hasPlug} of object \texttt{h} and attribute \texttt{inHole} of object \texttt{p} both have value \texttt{false}.

The rounded rectangle that follows, represents the fields that are possibly modified: the attributes \texttt{inHole} and \texttt{hole} of object \texttt{p}. After this, method \texttt{fillWith} of object \texttt{h} is called with parameter value \texttt{p}. The rectangle that follows, again shows the values of the significant fields. The attributes \texttt{hasPlug} and \texttt{plug} of object \texttt{h} are then possibly modified, after which method \texttt{fillWith} returns and finally \texttt{insertInto} returns.

We now show how to formally write down that the behaviour of the plug-pattern suite contains this particular (family of) trace(s):
\[
\begin{align*}
( & \langle \langle \langle c_0, \text{insertInto}, \text{IInsert}, \text{Plug} \rangle \rangle, \langle \langle c_1, \text{fillWith}, \text{IFill} \rangle \rangle, \langle \langle c_2, \text{fillWith}, \text{IFill} \rangle \rangle, \langle \langle c_3, \text{insertInto}, \text{IInsert}, \text{Plug} \rangle \rangle, ) \\
\epsilon & \mu(\text{PlugPattern}_{PPA, PPI})
\end{align*}
\]

for all carrier records \(c_0, \ldots, c_3\) and object names \(p, h\) such that

\[
\begin{align*}
\text{this} & .c_3 = p \\
\text{h}_{par} & .c_3 = h \\
(h \rightarrow_c \text{hasPlug}).c_3 = & \ false \\
(p \rightarrow_c \text{inHole}) .c_3 = & \ false \\
\text{this} & .c_2 = h \\
\text{p}_{par} & .c_2 = p \\
(h \rightarrow_c \text{hasPlug}).c_2 = & \ false \\
(p \rightarrow_c \text{inHole}) .c_2 = & \ true \\
(p \rightarrow_c \text{hole}) .c_2 = & \ h \\
\text{this} & .c_1 = h \\
\text{p}_{par} & .c_1 = p \\
(h \rightarrow_c \text{hasPlug}).c_1 = & \ true \\
(h \rightarrow_c \text{plug}) .c_1 = & \ p \\
(p \rightarrow_c \text{inHole}) .c_1 = & \ true \\
(p \rightarrow_c \text{hole}) .c_1 = & \ h \\
\text{this} & .c_0 = p \\
\text{h}_{par} & .c_0 = h \\
(h \rightarrow_c \text{hasPlug}).c_0 = & \ true \\
(h \rightarrow_c \text{plug}) .c_0 = & \ p \\
(p \rightarrow_c \text{inHole}) .c_0 = & \ true \\
(p \rightarrow_c \text{hole}) .c_0 = & \ h \\
c_2(\text{modify}_{I_{PPA}}.\langle \langle \hat{p}, \text{inHole } \rangle \rangle, \langle \hat{p}, \text{hole} \rangle) c_3 \\
c_1(\text{modify}_{I_{PPA}}.\langle \langle \hat{h}, \text{hasPlug} \rangle \rangle, \langle \hat{h}, \text{plug} \rangle) c_2 \\
c_0(\text{modify}_{I_{PPA}}.\langle \rangle) c_1 \\
c_0(\text{modify}_{I_{C}}.\langle \text{globalState.result} \rangle) c_3 \\
c_1(\text{modify}_{I_{C}}.\langle \text{globalState.result} \rangle) c_2 \\
\end{align*}
\]

The first four blocks of formulas correspond (in the same order, from top to bottom) to the four rectangles in the graphical representation of the trace. The fifth block is concerned with equality of certain states with respect to certain fields. Its first formula enforces for example that in state \(c_2\), the values of the attributes \text{hasPlug}, \text{plug}, \text{inHole} and \text{hole} of every object are equal to their values in state \(c_3\), except
for the attributes \texttt{inHole} and \texttt{hole} of object \texttt{p}. The last block enforces equality of certain states with respect to the carriers \texttt{this} and \texttt{parameters}.

We tried to construct a complete formal proof of the above theorem, but quickly learned that a brute-force proof is infeasible by hand. A high-level proof theory and/or automatic theorem prover is required for this, the construction of which we leave for future research.

Despite the fact that we do not have a formal proof theory, we can still show the main proof obligations that we expect to obtain from such a theory. We only consider proofs about state invariants, pre-conditions, post-conditions and action clauses. Proofs about attribute types, parameter types and result types are omitted. We leave it to the reader to verify that the proof obligations that we formulate, indeed hold.

We start with proof obligations for the state invariants. In our formalism, the state invariants of a role hold at the beginning and end of each of its methods. We therefore expect to obtain the following proof obligations:

\begin{align*}
  & c_3 \in \text{PSInv1}' \\
  & c_3 \in \text{PSInv2}' \\
  & c_0 \in \text{PSInv1}' \\
  & c_0 \in \text{PSInv2}' \\
  & c_2 \in \text{HSInv1}' \\
  & c_2 \in \text{HSInv2}' \\
  & c_1 \in \text{HSInv1}' \\
  & c_1 \in \text{HSInv2}'
\end{align*}

where $\text{PSInv1}'$, $\text{PSInv2}'$, $\text{HSInv1}'$ and $\text{HSInv2}'$ are parts of the plug-pattern formalisation. Their names suggest their origin:

\begin{align*}
  & \text{PSInv1}' = \{ \text{this} \rightarrow_d \text{inHole} \Rightarrow (\text{this} \rightarrow_d \text{hole}) \rightarrow_d \text{hasPlug} \} \\
  & \text{PSInv2}' = \{ \text{this} \rightarrow_d \text{inHole} \Rightarrow (\text{this} \rightarrow_d \text{hole}) \rightarrow_d \text{plug} = \text{this} \} \\
  & \text{HSInv1}' = \{ \text{this} \rightarrow_d \text{hasPlug} \Rightarrow (\text{this} \rightarrow_d \text{plug}) \rightarrow_d \text{inHole} \} \\
  & \text{HSInv2}' = \{ \text{this} \rightarrow_d \text{hasPlug} \Rightarrow (\text{this} \rightarrow_d \text{plug}) \rightarrow_d \text{hole} = \text{this} \}
\end{align*}

A pre-condition is something that results in proof obligations if it holds (if the pre holds, the post holds). We expect to have to check if the following formulas hold:

\begin{align*}
  & c_3 \in \text{PInPreCondition}' \\
  & c_2 \in \text{HFiPreCondition}'
\end{align*}

where

\begin{align*}
  & \text{PInPreCondition}' = \{ \neg (\text{this} \rightarrow_d \text{inHole}) \land \neg (\text{h}_{\text{par}} \rightarrow_d \text{hasPlug}) \} \\
  & \text{HFiPreCondition}' = \{ \neg (\text{this} \rightarrow_d \text{hasPlug}) \land \text{p}_{\text{par}} \rightarrow_d \text{inHole} \land \text{p}_{\text{par}} \rightarrow_d \text{hole} = \text{this} \}
\end{align*}
The reader can verify that the above two formulas indeed hold, which means that we obtain proof obligations for the corresponding post-conditions and action clauses.

With respect to post-conditions, we expect the following proof obligations:

\[ c_0 \left( P_{inPostCondition}' \right) c_3 \]
\[ c_1 \left( H_{fiPostCondition}' \right) c_2 \]

where

\[ P_{inPostCondition}' = \left[ \text{this} \leadsto_c \text{inHole} \land \text{this} \leadsto_c \text{hole} = h_{par} \right] \]
\[ H_{fiPostCondition}' = \left[ \text{this} \leadsto_c \text{hasPlug} \land \text{this} \leadsto_c \text{plug} = p_{par} \right] \]

What is left, are the action clauses. We expect the following proof obligations:

\[ c_3' \left( \text{modify}_{cPPA}.\{\text{this, inHole}, \text{this, hole}\} \right) c_3 \]
\[ c_3' \left( \text{modify}_{IC}.\{\text{globalState, result}\} \right) c_3 \]
\[ c_2 \left( \text{modify}_{cPPA}.\{\} \right) c_3' \]
\[ p_{par}.c_2 = \text{this}.c_3' \]
\[ \text{this}.c_2 = \left( \text{this} \leadsto_c \text{hole} \right).c_3' \]
\[ c_0 \left( \text{modify}_{IC}.\{\text{globalState, result}\} \right) c_3' \]
\[ c_1 \left( \text{modify}_{cPPA}.\{\text{this, hasPlug}, \text{this, plug}\} \right) c_2 \]
\[ c_1 \left( \text{modify}_{IC}.\{\text{globalState, result}\} \right) c_2 \]

for some \( c_3' \).

The first two formulas are related to the modify statement in method \textit{insertInto}, the next four to the call statement in this method and the last two to the modify statement in method \textit{fillWith}.

The first formula of both modify-statement blocks expresses the core meaning of a modify statement: that with respect to some attribute signature \( PPA \) in this case), only certain attributes of certain objects possibly change.

The last formula of each block expresses that modify statements and call statements do not change the value of the carriers \textit{this} and \textit{parameters}.

The first formula of the call-statement (middle) block expresses that, with respect to attribute signature \( PPA \), the global state that the called method starts in, is equal to the global state at call time (the moment the call is performed). The next formula in this block expresses that for the called method, the value of parameter \( p \) is equal to the value of the parameter expression of the call statement at call time.

The third formula in this block expresses that for a called method, the value of carrier \textit{this} is equal to the value of the object expression of the call statement at call time.
11.4 Observer pattern

In this section we formalise the observer pattern. We again start with its class diagram:

![Diagram](image)

The observer pattern consists of two parts that are both suites. One part is the plug pattern and the other is an extension that contains observer-pattern specifics. The observer pattern itself is defined as the behavioural intersection of these two suites:

\[
\text{ObserverPattern}_{A, I} = \text{PlugPattern}_{PPA, PPI} \cap \text{Extension}_{A, I}
\]

In the following subsections we define the extension.

11.4.1 Signature

The signature of the observer pattern extends the signature of the plug pattern with attributes \(\text{orig}\) and \(\text{copy}\) and interface \(\text{IUpdate}\):

\[
\begin{align*}
\text{OPA} &= PPA \cup \{\text{orig}, \text{copy}\} \\
\text{OPI} &= PPI \cup \{\text{IUpdate}\}
\end{align*}
\]
11.4.2 Hierarchy

The role-interface-method hierarchy of the extension is defined by

\[
\begin{align*}
\text{Extension}_{A, I} & = \text{suite.} \{ \langle \text{Subject}_{A, I}, \text{Hole} \rangle, \langle \text{Observer}_{A, I}, \text{Plug} \rangle \} \\
\text{Subject}_{A, I} & = \text{role.} \{ \langle \text{SIFill}_{A, I}, \text{IFill} \rangle, \langle \text{SRInv}_{A, I} \rangle \} \\
\text{Observer}_{A, I} & = \text{role.} \{ \langle \text{PIInsert}_{A, I}, \text{IInsert} \rangle, \langle \text{OIUpdate}_{A, I}, \text{IUpdate} \rangle, \langle \text{ORInv}_{A, I} \rangle \} \\
\text{SIFill}_{A, I} & = \text{interface.} \{ \langle \text{SFillWith}_{A, I}, \text{fillWith} \rangle, \langle \text{HEmpty}_{A, I}, \text{empty} \rangle \} \\
\text{OIUpdate}_{A, I} & = \text{interface.} \{ \langle \text{OUpdate}_{A, I}, \text{update} \rangle \}
\end{align*}
\]

We use the names Hole and Plug and not Subject and Observer because our theory does not support renaming (see section 2.5.4).

For interface IInsert and method empty we reuse parts of the plug pattern, which are now interpreted in the extended signature. Where a call to empty in the observer pattern does not change the attributes orig and copy of any object, these attributes may change arbitrarily when the same call is performed in the plug pattern. Furthermore, a call to empty in the observer pattern does not result in any call on interface IUpdate, whereas in the plug pattern these calls can occur.

11.4.3 Invariants

Because we inherit the invariants from the plug pattern, we do not need to worry much about how to plug observers into subjects. We can focus on the extension that the observer pattern provides: if an observer is plugged into a subject, the value of its attribute copy should be kept up to date with the value of the subject’s attribute orig. The role invariants of subjects and observers are formalised by

\[
\begin{align*}
\text{SRInv}_{A, I} & = \text{roleInvariant.} \{ \langle \}, \langle \}, \langle \text{SSInv}_{A, I} \rangle, \langle \text{SOrigT} \rangle \} \} \\
\text{ORInv}_{A, I} & = \text{roleInvariant.} \{ \langle \}, \langle \}, \langle \text{OSInv}_{A, I} \rangle, \langle \text{OCopyT} \rangle \} \} \\
\text{SOrigT} & = \text{attributeType.} \langle \text{void}, \text{orig} \rangle \\
\text{OCopyT} & = \text{attributeType.} \langle \text{void}, \text{copy} \rangle \\
\end{align*}
\]

The two attribute-type statements SOrigT and OCopyT are defined by

The state-invariant statements SSInv and OSInv, in 2.2.4 represented by

\[
\begin{align*}
\text{SSInv} & = \text{stateInvariant.} \langle \text{hasPlug} \Rightarrow \text{plug.copy} = \text{orig} \rangle \\
\text{OSInv} & = \text{stateInvariant.} \langle \text{inHole} \Rightarrow \text{copy} = \text{hole.orig} \rangle \\
\end{align*}
\]
are formalised by

\[ SSInv = \text{stateInvariant.} \{ (\text{this} \rightarrow_c \text{hasPlug} \Rightarrow (\text{this} \rightarrow_c \text{plug}) \rightarrow_c \text{copy} = \text{this} \rightarrow_c \text{orig}) \} \]

\[ OSInv = \text{stateInvariant.} \{ (\text{this} \rightarrow_c \text{inHole} \Rightarrow \text{this} \rightarrow_c \text{copy} = (\text{this} \rightarrow_c \text{hole}) \rightarrow_c \text{orig}) \} \]

### 11.4.4 Methods

What is left, is the definition of the methods `fillWith` and `update`. For each of both methods, we give its description from section 2.2.4, followed by its formalisation:

\[ SFillWith = \text{method} \]
\[ \langle HFiParameter \rangle \]
\[ HFiResult \]
\[ <SFiEffect> \]

\[ SFiEffect = \text{effect} \]
\[ HFiPreCondition \]
\[ SFiActionClause \]
\[ HFiPostCondition \]

\[ SFiActionClause = \text{actionClause} \]
\[ \text{modify} \{ \text{hasPlug, plug} \} \]
\[ ; \text{plug.update()} \]

\[ SFillWith_{A,I} = \text{method.} \]
\[ \{ SFiPost_{A,I} \]
\[ , HFiPre \} \]

\[ SFiPost_{A,I} = \text{post.} \]
\[ \{ SFiActionClause_{A,I} \]
\[ , HFiPostCondition \]
\[ , HFiResultType \} \]

\[ SFiActionClause_{A,I} = \text{actionClause.} \]
\[ \text{dmodify}_{A,I} \{ \text{this, hasPlug} \}
\[ , \{ \text{this, plug} \} \]
\[ ; (\text{this} \rightarrow_c \text{plug}) \rightarrow_{\text{call}_{A,I}} \]
\[ \text{Plug::IUpdate::update} \]} \]
Although we reuse several parts of the plug pattern, we do not behaviourally inherit much. Every method is redefined, although we could have chosen ‘true’ for the post-condition of `fillWith` and at least have behaviourally inherited that part of behaviour. We do however behaviourally inherit the invariants of the plug pattern.

In general it is not beneficial to inherit the behaviour of a method without imposing extra constraints on the attributes and interfaces that extend the signature. It would however be beneficial to behaviourally inherit methods for which the “A”
and \( I \) of modify statements and call statements are equal to the universal set \( I \). This is similar to implementation inheritance of generic methods in object-oriented programming where dynamic binding (modeled by behavioural intersection in our case) is exploited. We leave investigation of this topic for future research.

### 11.4.5 A trace of insertInto

Also for the observer pattern we examine a (family of) trace(s) for `insertInto`. The trace shows the extra call to `update`. We again start with a graphical representation of the trace:

```
\[
\begin{array}{c}
s \\
\end{array}
\begin{array}{c}
\text{insertInto} \\
\end{array}
\begin{array}{c}
\text{fillWith} \\
\end{array}
\begin{array}{c}
\text{update} \\
\end{array}
\begin{array}{c}
\text{copy} \\
\end{array}
\begin{array}{c}
\text{hasPlug} \\
\end{array}
\begin{array}{c}
\text{plug} \\
\end{array}
\begin{array}{c}
\text{orig} \\
\end{array}
\begin{array}{c}
inHole \\
\end{array}
\begin{array}{c}
\text{hasPlug} \\
\end{array}
\begin{array}{c}
\text{plug} \\
\end{array}
\begin{array}{c}
\text{orig} \\
\end{array}
\begin{array}{c}
inHole \\
\end{array}
\begin{array}{c}
\text{hasPlug} \\
\end{array}
\begin{array}{c}
\text{plug} \\
\end{array}
\begin{array}{c}
\text{orig} \\
\end{array}
\begin{array}{c}
inHole \\
\end{array}
\begin{array}{c}
\text{hasPlug} \\
\end{array}
\begin{array}{c}
\text{plug} \\
\end{array}
\begin{array}{c}
\text{orig} \\
\end{array}
\begin{array}{c}
inHole \\
\end{array}
\begin{array}{c}
\text{hasPlug} \\
\end{array}
\begin{array}{c}
\text{plug} \\
\end{array}
\begin{array}{c}
\text{orig} \\
\end{array}
\begin{array}{c}
inHole \\
\end{array}
\begin{array}{c}
\text{hasPlug} \\
\end{array}
\begin{array}{c}
\text{plug} \\
\end{array}
\begin{array}{c}
\text{orig} \\
\end{array}
\end{array}
```

The fact that the observer-pattern suite contains this trace, is formalised by
$$\mu(\text{ObserverPattern}_{OPT, OPF})$$

where

this $c_5 = o$

$h_{par} .c_5 = s$

$(s \rightarrow_c \text{hasPlug}).c_5 = false$

$(s \rightarrow_c \text{orig}) .c_5 = x$

$(\overline{o} \rightarrow_c \text{inHole}) .c_5 = false$

this $c_4 = s$

$p_{par} .c_4 = o$

$(s \rightarrow_c \text{hasPlug}).c_4 = false$

$(s \rightarrow_c \text{orig}) .c_4 = x$

$(\overline{o} \rightarrow_c \text{inHole}) .c_4 = true$

$(\overline{o} \rightarrow_c \text{hole}) .c_4 = s$

this $c_3 = o$

$(s \rightarrow_c \text{hasPlug}).c_3 = true$

$(s \rightarrow_c \text{plug}) .c_3 = o$

$(s \rightarrow_c \text{orig}) .c_3 = x$

$(\overline{o} \rightarrow_c \text{inHole}) .c_3 = true$

$(\overline{o} \rightarrow_c \text{hole}) .c_3 = s$

this $c_2 = o$

$(s \rightarrow_c \text{hasPlug}).c_2 = true$

$(s \rightarrow_c \text{plug}) .c_2 = o$

$(s \rightarrow_c \text{orig}) .c_2 = x$

$(\overline{o} \rightarrow_c \text{inHole}) .c_2 = true$

$(\overline{o} \rightarrow_c \text{hole}) .c_2 = s$

$(\overline{o} \rightarrow_c \text{copy}) .c_2 = x$

this $c_1 = s$

$p_{par} .c_1 = o$

$(s \rightarrow_c \text{hasPlug}).c_1 = true$

$(s \rightarrow_c \text{plug}) .c_1 = o$

$(s \rightarrow_c \text{orig}) .c_1 = x$

$(\overline{o} \rightarrow_c \text{inHole}) .c_1 = true$

$(\overline{o} \rightarrow_c \text{hole}) .c_1 = s$

$(\overline{o} \rightarrow_c \text{copy}) .c_1 = x$
We again present the proof obligations that we expect to obtain from a formal proof theory, starting with the state invariants:

\begin{itemize}

\item \(c_5 \in P_{\text{Inv}'}\)
\item \(c_5 \in P_{\text{Inv}'}\)
\item \(c_5 \in O_{\text{Inv}'}\)
\item \(c_0 \in P_{\text{Inv}'}\)
\item \(c_0 \in P_{\text{Inv}'}\)
\item \(c_0 \in O_{\text{Inv}'}\)
\item \(c_4 \in H_{\text{Inv}'}\)
\item \(c_4 \in H_{\text{Inv}'}\)
\item \(c_4 \in S_{\text{Inv}'}\)
\item \(c_1 \in H_{\text{Inv}'}\)
\item \(c_1 \in H_{\text{Inv}'}\)
\item \(c_1 \in S_{\text{Inv}'}\)
\item \(c_3 \in P_{\text{Inv}'}\)
\item \(c_3 \in P_{\text{Inv}'}\)
\item \(c_7 \in O_{\text{Inv}'}\)
\item \(c_2 \in P_{\text{Inv}'}\)
\item \(c_2 \in P_{\text{Inv}'}\)
\item \(c_2 \in O_{\text{Inv}'}\)

\end{itemize}

where
\[\text{PSInv}^1 = \{ \text{this} \rightarrow_c \text{inHole} \Rightarrow (\text{this} \rightarrow_c \text{hole}) \rightarrow_c \text{hasPlug} \}\]

\[\text{PSInv}^2 = \{ \text{this} \rightarrow_c \text{inHole} \Rightarrow (\text{this} \rightarrow_c \text{hole}) \rightarrow_c \text{plug} = \text{this} \}\]

\[\text{OSInv}^\prime = \{ \text{this} \rightarrow_c \text{inHole} \Rightarrow (\text{this} \rightarrow_c \text{hole}) \rightarrow_c \text{copy} = \text{this} \}\]

\[\text{HSInv}^1 = \{ \text{this} \rightarrow_c \text{hasPlug} \Rightarrow (\text{this} \rightarrow_c \text{plug}) \rightarrow_c \text{inHole} \}\]

\[\text{HSInv}^2 = \{ \text{this} \rightarrow_c \text{hasPlug} \Rightarrow (\text{this} \rightarrow_c \text{plug}) \rightarrow_c \text{hole} = \text{this} \}\]

\[\text{SSInv}^\prime = \{ \text{this} \rightarrow_c \text{hasPlug} \Rightarrow (\text{this} \rightarrow_c \text{plug}) \rightarrow_c \text{copy} = \text{this} \rightarrow_c \text{orig} \}\]

The two formulas that have been blotted out, are not proof obligations because interface IUpdate is excluded from invariant OSInv. The first one of these two formulas indeed does not hold in general for our family of traces. Traces are helpful in determining which interfaces should be excluded from which invariants. If an interface is excluded from a certain invariant, this invariant then usually ends up in the post-condition of the interface’s methods.

For the pre-conditions we expect to have to check:

\(c_5 \in \text{PInPreCondition}^\prime\)
\(c_4 \in \text{HFiPreCondition}^\prime\)
\(c_3 \in \text{OUpPreCondition}^\prime\)

where

\[\text{PInPreCondition}^\prime = \{ \neg (\text{this} \rightarrow_c \text{inHole}) \land \neg (\text{h}_{\text{par}} \rightarrow_c \text{hasPlug}) \}\]

\[\text{HFiPreCondition}^\prime = \{ \neg (\text{this} \rightarrow_c \text{hasPlug}) \land \text{p}_{\text{par}} \rightarrow_c \text{inHole} \land \text{p}_{\text{par}} \rightarrow_c \text{hole} = \text{this} \}\]

\[\text{OUpPreCondition}^\prime = \{ \text{this} \rightarrow_c \text{inHole} \}\]

Remember that these are not proof obligations, but can (and again do in this case) result into proof obligations.

For the post-conditions we expect the following proof obligations:

\(c_0 (\text{PInPostCondition}^\prime) c_5\)
\(c_1 (\text{HFiPostCondition}^\prime) c_4\)
\(c_2 (\text{OUpPostCondition}^\prime) c_3\)

where

\[\text{PInPostCondition}^\prime = [\text{this} \rightarrow_c \text{inHole} \land \text{this} \rightarrow_c \text{hole} = \text{h}_{\text{par}}]\]

\[\text{HFiPostCondition}^\prime = [\text{this} \rightarrow_c \text{hasPlug} \land \text{this} \rightarrow_c \text{plug} = \text{p}_{\text{par}}]\]

\[\text{OUpPostCondition}^\prime = [\text{this} \rightarrow_c \text{copy} = \text{this} \rightarrow_c \text{hole} \rightarrow_c \text{orig}]\]

Finally for the action clauses we expect
\[
c_5' \left( \text{modify}_{c, OPA}.\{\langle \text{this, inHole} \rangle, \langle \text{this, hole} \rangle \} \right) c_5
\]
\[
c_5' \left( \text{modify}_{IC}.\{\text{globalState, result} \} \right) c_5
\]
\[
c_4 \left( \text{modify}_{c, OPA}.\} c_5' \right)
\]
\[
p_{par}.c_4 = \text{this}.c_5'
\]
\[
\text{this}.c_4 = (\text{this} \rightarrow_c \text{hole}).c_5'
\]
\[
c_0 \left( \text{modify}_{IC}.\{\text{globalState, result} \} \right) c_5'
\]
\[
c_4' \left( \text{modify}_{c, OPA}.\{\langle \text{this, hasPlug} \rangle, \langle \text{this, plug} \rangle \} \right) c_4
\]
\[
c_4' \left( \text{modify}_{IC}.\{\text{globalState, result} \} \right) c_4
\]
\[
c_3 \left( \text{modify}_{c, OPA}.\{\right) c_4'
\]
\[
\text{this}.c_3 = (\text{this} \rightarrow_c \text{plug}).c_4'
\]
\[
c_1 \left( \text{modify}_{IC}.\{\text{globalState, result} \} \right) c_4'
\]
\[
c_2 \left( \text{modify}_{c, OPA}.\{\right) c_3
\]
\[
c_2 \left( \text{modify}_{IC}.\{\text{globalState, result} \} \right) c_3
\]

for some $c_5'$ and $c_4'$.

### 11.4.6 Property

Traces are a nice way to illustrate behaviour of a specification or to test that a specification exhibits intended behaviour. In this section we show how our formal ISpec language can be used to describe more general properties of a suite. The following property tells that, within the chosen signature, a call to method \text{insertInto} on a plug only possibly changes the attributes \text{inHole}, \text{hole} and \text{copy} of this plug and the attributes \text{hasPlug} and \text{plug} of the hole it is being plugged into:

\[
\begin{align*}
Modify_{A,I} & = \text{suite.} \ \{\langle Mod_{A,I} , \text{Plug } \rangle \} \\
Mod_{A,I} & = \text{role.} \langle \{\langle MIInsert_{A,I} , \text{Insert } \rangle \}, MRInv \rangle \\
MIInsert_{A,I} & = \text{interface.} \{\{MIInsert_{A,I}, \text{insertInto}\} \\
MRInv & = \text{roleInvariant.} \{\}, \{\}, \{\}, \{\}
\end{align*}
\]
\[ M_{\text{InsertInto}}_{A,I} = \text{method}. \langle M_{\text{InPost}}_{A,I} , P_{\text{InPre}} \rangle \]

\[ M_{\text{InPost}}_{A,I} = \text{post}. \langle M_{\text{ActionClause}}_{A,I} , M_{\text{PostCondition}} , M_{\text{ResultType}} \rangle \]

\[ M_{\text{ActionClause}}_{A,I} = \text{actionClause. smodify}_A. \{(\text{this, inHole}) \}, \{\text{this, hole}\}, \{(\text{this} \rightarrow \text{ hole, hasPlug}) \}, \{(\text{this} \rightarrow \text{ hole, plug}) \}, \{(\text{this, copy}) \} \]

\[ M_{\text{PostCondition}} = \text{postCondition.}[\text{true}] \]

\[ M_{\text{ResultType}} = \text{resultType. void} \]

The fact that the observer-pattern suite satisfies this property, is formalised by

\[ \text{ObserverPattern}_{OPA, OPI} \subseteq \text{Modify}_{OPA, OPI} \]

A proof system that enables one to prove these kinds of facts, is left for future research.

### 11.5 Conclusions

In this chapter we used the theory of the previous chapters to construct a formal language for a subset of ISpec. We formalised the plug pattern and observer pattern that were introduced in chapter 2 and discussed two (families of) traces, one for the plug pattern and one for the observer pattern. We also showed how properties of suites can be formulated as suites. A proof system that enables us to prove these properties is beyond the scope of this thesis.
Chapter 12

Conclusions

The primary goal of our work was to investigate and formalise ISpec, a specification approach that is used in industry for the specification of interfaces of component-based systems. We strived to “make things as simple as possible, but not simpler”. For us, “as simple as possible” meant that we tried to construct mathematical structures that are simple and regular. We tried to achieve “but not simpler” by means of many discussions with the designer of ISpec, Hans Jonkers. Small, simple examples appeared to be a powerful communication means between the formal theory and the ideas behind ISpec.

12.1 Our work

Apart from formalising ISpec, a contribution that we wanted to make with this thesis is to provide a new and hopefully inspiring view on many well-known concepts, integrated into a single formal framework. We present a short list of what, in our view, are new ideas that this thesis presents:

- The research into extensible typing and uniform notation for type operators, presented in chapter 4.
- The unifying view on the relational product and sum and several related new products like the conjoint sum, presented in chapter 5.
- A novel treatment of non-strict and non-deterministic expressions in chapter 6, based on the products that are introduced in chapter 5.
- The partial- and total-correctness operators presented in sections 7.3 and 7.8.3 that use binary relations for pres and posts. We found [48] to use a notion of correctness that is somewhat similar to our notion of total correctness. This is further discussed in the next section.
• The pack operator that corresponds to the conjoint sum (section 5.6), enabling the creation of hierarchically structured specifications that can be straightforwardly extended by means of refinement (section 7.5).

• The aspect operator, providing a simple and powerful means to add aspects and invariants to hierarchically structured specifications (section 7.7).

• The investigation in section 8.5 of the subtle differences between behavioural intersection and overriding we thought to be novel. The same goes for the notion of conservative users presented in section 8.6.4. As the next section shows, these notions turned out to be already investigated in [41]. Our particular way of formalising these notions and the way we link them to other concepts, can be considered a contribution though.

• The generalisation of relations to protocols and in particular the notion of continuous genuine statements that captures the concept of visible calls (chapter 9). This provides a model that, like process-algebras, is explicit about communication between components, but in a way that matches the relational method-call semantics that is conventional for object-oriented systems.

The definition of the formal ISpec language and the treatment of the notorious observer-pattern example illustrate the use of many of these theoretical concepts.

12.2 Related work

As can be expected in the context of well-established computer-science concepts like ‘object-oriented’ and ‘component-based’, much research has been performed that is related to our work.

Closely related to our work is of course the research carried out on the predecessor of ISpec, called COLD (Common Object-oriented Language for Design) [19]. COLD is actually not a single language, but a family of languages, defined as syntactic extensions of a formally defined kernel language, called COLD-K. COLD-K is a rich design language that allows for the definition of abstract datatypes, state-based generalisations called classes, specifications built with these, called schemes, and designs consisting of components (specifications with an optional implementation). Our thesis can be seen as supplementary research where we disregard algebraic specification of datatypes, only informally introducing some basic types, and focus on inter-component communication. We also try to provide some more insight into other notions that play a role in COLD and ISpec, like non-strict expression evaluation, signature extension, invariants, conservative extension and field modification.

A language that is like COLD based on the notion of evolving algebras is the language of Abstract State Machines (ASMs) [12]. An executable language that is based on ASMs is the Abstract State Machine Language (AsmL) [23]. There does however not seem to exist much research on ASMs that is related to our work.
Another well-known example of a specification approach for complex software systems is Z [29]. Two derivatives are the B-method [2], focusing mainly on software implementation, and Object-Z [47], that extends Z with several object-oriented concepts. Object-Z again has a spin-off called CSP-OZ [46]. CSP-OZ combines the relational world of Object-Z with the process-algebraic world of communicating sequential processes [25] (CSP). This approach is however different from the common object-oriented approach where communication in a system consists of method calls between objects. Our notion of protocol was designed to reflect this common object-oriented style.

Comparable to Z is the Vienna Development Method (VDM) [30]. VDM uses the Logical of Partial Functions (LPF) to deal with partiality in specifications. The truth tables of VDM correspond to the ones that we use (section 6.2) which strengthens our belief that we made proper choices. Like Z, VDM also has an object-oriented version VDM++ [18] and a related language RSL (Rigorous Approach to Industrial Software Engineering (RAISE) Specification Language [21]) that combines VDM with CSP-like process-algebraic concepts. Similar to the CSP-OZ case, RSL also does not reflect the common object-oriented style.

Research into object-oriented programming has also been performed in the context of the refinement calculus [3]. Just before the finishing of this thesis, we discovered a paper [41] that contains theory that is strikingly similar to the sections 8.5 and 8.6 of this thesis. What the authors of that paper call the “flexibility property”, corresponds to the “???” in the formula

\[
(s_0 Z_0 \cup Z_1 \ s_1) \subseteq Z_0 \cup Z_1, s
\]

at the beginning of section 8.5.7 of this thesis. Furthermore, their “no cycles requirement” corresponds to the notion of “independence” presented in section 8.5.8. The paper also describes the notions “no revision self-calling assumptions” and “no base class down-calling assumptions” that correspond to the notion of “conservative users” presented in section 8.6.4. The fact that the same concepts were discovered in complete separation, using different formalisms, is a clear indication of the fact that these are fundamental notions in the context of object-oriented programming and probably deserve more attention than is apparent from the number of citations to [41].

As mentioned at the end of section 8.5.8 of this thesis, the restrictions imposed there (and also in [41]) might be too severe for certain practical situations. This view seems to be supported by the fact that several researchers are working on proof systems for non-hierarchical object-oriented systems. For a comprehensive overview we refer to [37] and [40]. This particular research area is focused on powerful proof systems for object-oriented programming languages like Java and C#. From an abstract point of view, a key goal of all these proof-system developers is to find an appropriate model that abstracts from the object-oriented language in such a
way that it enables one to prove the correctness of intricate practical examples in a feasible manner. Although our primary concern was not the construction of a powerful proof system for an object-oriented programming language, enabling formal proofs was an important incentive for several design decisions and can be seen as an attempt to approximate the kind of model we just mentioned. We hope that our notion of “visible calls” will inspire people from the proof-system community and can help them in their quest for the ultimate proof system for object-oriented programming.

As mentioned in the previous section, the paper [48] presents a notion of correctness that is somewhat similar to our notion of total correctness. Formally, the notion of correctness that is presented in [48] is

\[ P \subseteq S^\sim \circ R \]

whereas our notion of total correctness is

\[ S \circ P \subseteq R \land P^\prec \subseteq S^\prec \]

Although these notions are equivalent for functional relations \( S \), they are not for relations \( S \) in general. To simplify things a little, assume that \( \text{pre} P \) is equal to the identity relation \( I \) and that \( \text{post} R \) is total (\( P = R^\succ = I \)). Our notion of total correctness of body \( S \) then means that \( S \) may reduce the non-determinism that is present in \( Q \). The notion of [48] is equivalent to

\[ \forall \langle z \mid \exists \langle y \mid y (S) z \land y (R) z \rangle \rangle \]

In other words, \( S \) is correct if it is somehow able to produce a value that adheres to the post. This is a rather unorthodox notion of correctness that is at least not suitable for doing successive refinement, whereas our notion is suitable for that purpose.

12.3 Future work

This thesis only captures a small part of ISpec. There are still many concepts of ISpec missing as section 2.5 shows. These include required interfaces, parallelism, object existence, renaming, black-box semantics, activation specifications, dynamic binding and probably many others. Next to this, there are many questions in this thesis for which we do not give a proper answer or only hint at a possible answer, for example how to deal with non-strictness in expressions, how to construct Hoare-like proofs for ISpec specifications, how to deal with infinite behaviour, how to prove invariants, how to deal with total refinement in general, how to deal with mutual dependencies between components or how to deal with pointers. Many of these topics have been extensively investigated by others, but for most of them there still does not seem to exist a ‘definitive answer’. We hope that the formal theory that is presented in this thesis can help researchers to obtain new insights in these areas.
In this thesis we formalise several concepts that can be found in practical approaches to the specification of large software systems. Our focus is ISpec, an industrial specification approach developed within Philips Research, that is used to specify interfaces of component-based systems.

Our primary mathematical formalism is the calculus of relations. We develop several new relation-algebraic constructs that suit our needs. For example, to be able to be very detailed in typing relations, we introduce a rich collection of type operators. A key role is fulfilled by the cylindric-type operator that generalises the co/contravariant way in which functions are usually typed in functional programming.

The cylindric-type operator is used to define a new construct that we call the cylindric product. We show that the product and sum of allegory theory are constrained forms of this cylindric product. A detailed analysis of several other constrained forms of the cylindric product is presented. One of these constructs is the conjoint sum, a construct that closely resembles the (disjoint) sum of allegory theory. However, adding a relation to a conjoint sum results in a smaller relation (subrelation), just like the addition of a method to a class results in a subclass in object-oriented programming. It is shown how the different products can be used to model different kinds of expression evaluation, like strict evaluation, non-strict evaluation and even several kinds of non-deterministic evaluation.

All this forms the basis for the formalisation of concepts that are more well-known. Refinement plays a crucial role. It serves as basis of a Hoare-like proof system that is purely based on (binary) relations. We show how to construct specifications that consist of a declarative as well as an operational part and show how common behaviour of specifications can be isolated in an aspect. This enables not only the formulation of more clear specifications, but also simplifies refinement. Aspects actually appear to be closely related to invariants. These are also investigated in detail.

Another topic of this thesis is components and their composition. We show how these concepts can be formalised and investigate modular refinement of components. Fixed-point theory is used to enable communication between components. The impact of this on modular refinement of components is investigated. The use of
relations as semantic model has as a consequence that communication between components is invisible. We introduce a generalisation of relations, called protocols, to enable visible communication.

In object-oriented programming, several variable-concepts like attributes, a ‘this’ object, parameters and results are used. These concepts are also modeled and investigated in this thesis.

The relation between theory and practice is made explicit by a formal language for a subset of ISpec.
Samenvatting

In dit proefschrift formaliseren we verscheidene concepten die te vinden zijn in praktische aanpakken voor de specificatie van grote softwaresystemen. Onze focus is ISpec, een industriële specificatie-aanpak ontwikkeld binnen Philips Research, die gebruikt wordt om interfaces van componentgebaseerde systemen te specificeren.

Ons primaire wiskundige formalisme is de relatiecalculus. We ontwikkelen verscheidene nieuwe relatie-algebraïsche constructies die nuttig zijn voor ons doel. Bijvoorbeeld, om heel gedetailleerd te kunnen zijn bij het typen van relaties, introduceren we een rijke collectie type-operatoren. Een sleutelrol wordt vervuld door de cylindrisch-type operator. Deze generaliseert de co/contra-variante wijze waarop functies gewoonlijk worden getypeerd in functioneel programmeren.

De cylindrisch-type operator wordt gebruikt om een nieuwe constructie te definieren die we het cylindrisch produkt noemen. We tonen aan dat het produkt en de som van allegorie theorie ingeperkte vormen zijn van dit cylindrisch produkt. Tevens wordt een gedetailleerde analyse gepresenteerd van verscheidene andere ingeperkte vormen van het cylindrisch produkt. Een van deze constructies is de conjuncte som, een constructie die veel lijkt op de (disjuncte) som van allegorie theorie. Echter, het toevoegen van een relatie aan een conjuncte som resulteert in een kleinere relatie (subrelatie), net zoals het toevoegen van een methode aan een klasse resulteert in een subklasse in object-georiënteerd programmeren. We laten zien hoe de verschillende produkten gebruikt kunnen worden om verschillende soorten expressie-evaluatie te modelleren, zoals strikte evaluatie, niet-strikte evaluatie en zelfs verscheidene vormen van niet-deterministische evaluatie.

Dit alles vormt de basis voor de formalisatie van concepten die meer bekend zijn. Verfijning speelt hierbij een cruciale rol. Het dient als basis voor een Hoare-achtig bewijsysteem dat puur gebaseerd is op (binaire) relaties. We laten zien hoe specificaties kunnen worden geconstrueerd die bestaan uit zowel een declaratief als een operationeel gedeelte en hoe gemeenschappelijk gedrag van specificaties geïsoleerd kan worden in een aspect. Dit maakt niet alleen de formulering van duidelijker specificaties mogelijk, maar vereenvoudigt ook verfijning. Aspecten blijken nauw verwant te zijn met invarianten. Deze worden ook in detail bestudeerd.

Een ander onderwerp van dit proefschrift is componenten en hun compositie. We
laten zien hoe deze concepten geformaliseerd kunnen worden en onderzoeken modulaire verfijning van componenten. De theorie van vaste punten wordt gebruikt om communicatie tussen componenten mogelijk te maken. We onderzoeken de impact die dit heeft op modulaire verfijning van componenten. Het gebruik van relaties als semantisch model heeft tot gevolg dat communicatie tussen componenten onzichtbaar is. We introduceren een generalisatie van relaties, genaamd protocollen, om zichtbare communicatie mogelijk te maken.

In object-georiënteerd programmeren worden verscheidene variabele-concepten gebruikt zoals attributen, een 'this' object, parameters en resultaten. Deze concepten worden ook gemodelleerd en onderzocht in dit proefschrift.

De relatie tussen theorie en praktijk wordt expliciet gemaakt door middel van een formele taal voor een deel van ISpec.
Curriculum vitae

Louis van Gool was born in Venlo, the Netherlands, on the 1st of July 1976. In 1994 he received his Gymnasium diploma at the Thomas College in Venlo. After that he started a combined study in Mathematics and Computer Science at Technische Universiteit Eindhoven. After receiving a first-year’s degree in each of both studies, he continued with Computer Science, receiving his Master’s degree on “Cylindrische Componenten Calculus” in 2000 (cum laude) under the supervision of dr. Jaap van der Woude.

After receiving his Master’s degree, Louis started as a Ph.D. student at Technische Universiteit Eindhoven. Under the supervision of promotor prof. dr. Jos Baeten and copromotor dr. Ruurd Kuiper and in close cooperation with dr. Hans Jonkers from Philips Research, he worked on an analysis and formalisation of the specification approach iSpec.

After his Ph.D. period, Louis started as a ‘postdoctoral’ researcher on the IDEALS project where he has worked on software-maintainability analysis and model-driven development for ASML Lithography.
Bibliography


Index

(y, z), see connection
(o, a), see field name
(x, v), see trace
(y : v : z), see connection trace
[x₀, ..., x₀⁻¹], see col. enumeration
{x₀, ..., x₀⁻¹}, see set enumeration
{x₀, ..., x₀⁻¹}, see tuple
(x₀, ..., x₀⁻¹), see sequence

[f.x | p.x | x], see col. comprehension
{f.x | p.x | x}, see set comprehension
(f.x | p.x | x), see fun. comprehension
(x₀ : x : v), see mirrored fun. compreh.
[E], see col.-comprehension operator
{E}, see set-comprehension operator
y (R) z, see connect output with input

R ⊆ Z, see partial-correctness operator
R | S | P, see total-correctness operator
false, see boolean

true, see boolean
∀, see universal quantification
∃, see existential quantification
∧, see conjunction
∨, see disjunction
⇒, see implication
⇐, see consequence
≡, see equivalence
¬, see negation
•, see assumption
., see application
see fix notation
∉, see negated binary predicate
∈, see element-of
∋, see owner-of
ε, see is-contained-by
⊆, see all-contained-by
⊆, see subcollection-of
⊂, see statement-inclusion

⊂, see behavioural refinement
⊂ Z, see refinement
⊂ Z, see statement-refinement
⊂ Z, see behavioural refinement
="; see assignment
I, see identity relation
|, see identity statement
Ip, see identity protocol
Ip, see identity protocol statement
K, see constant function
∅, see empty collection
∅, see empty statement
Π, see full collection
Π, see universal statement
Πp, see universal genuine statement
∩, see arbitrary intersection
∩, see arbitrary statement-intersection
∩p, see arb. genuine-stat.–intersection
∪, see arbitrary union
∪, see binary union
∪, see binary statement-union
\tilde{\gamma}, \textit{see} genuine-stat.-aspect-operator
\Delta, \textit{see} doubling function
\mathcal{A}, \textit{see} cylindric-doubling function
\odot, \textit{see} cylindric product
\oplus_M, \textit{see} constrained cylindric product
\times, \textit{see} cartesian product
+: \textit{see} cartesian sum
\cap, \textit{see} conjoint sum
\circ, \textit{see} loose product
\ast, \textit{see} connectional product
\ast, \textit{see} loose product
\ast, \textit{see} connectional product
\nu, \textit{see} greatest fixed point
\circ, \textit{see} default tag
\circ, \textit{see} default tag
\circ, \textit{see} default tag
\circ, \textit{see} component composition
\circ, \textit{see} stat.-component composition
\circ, \textit{see} override operator
\circ, \textit{see} statement-override operator
A, \textit{see} attribute name
B, \textit{see} suite behaviour
C, \textit{see} carrier record
G, \textit{see} global state
I, \textit{see} interface name
M, \textit{see} method name
O, \textit{see} object name
P, \textit{see} parameter name
R, \textit{see} role name
2, \textit{see} default tag
B, \textit{see} boolean
I, \textit{see} identifier
N, \textit{see} natural number
R, \textit{see} real number
Z, \textit{see} integer number
\nu, \textit{see} set
F, \textit{see} field name
P, \textit{see} power relator
PT, \textit{see} introspective
T, \textit{see} trace
V, \textit{see} sequence
ActionClause, \textit{see} action-clause stat.
actionClause, \textit{see} action-clause stat.
ActionInvariant, \textit{see} action-invariant stat.
actionInvariant, \textit{see} action-invariant stat.
AttributeType, \textit{see} attribute-type stat.
attributeType, \textit{see} attribute-type stat.
bool, \textit{see} primitive type
call, \textit{see} call
const, \textit{see} constrain
dark, \textit{see} universal dark protocol
d\textit{dark}, \textit{see} universal stat.-dark stat.
dark_{\textit{d}}, \textit{see} dark modify
d\textit{modify}, \textit{see} dark modify
f\textit{modify}, \textit{see} field-modify
f\textit{modify}_{C}, \textit{see} carrierised field-modify
f\textit{modify}_{C}, \textit{see} carrierised field-modify
g\textit{lobalState}, \textit{see} carrier
hide, \textit{see} hide
if \textit{B} then \textit{S}_0 \textit{else} \textit{S}_1, \textit{see} if-then-else

in, \textit{see} occurs-in
Interface, \textit{see} interface statement
interface, \textit{see} interface statement
int, \textit{see} primitive type
Method, \textit{see} method statement
method, \textit{see} method statement
modify, \textit{see} modify
modify_{D}, \textit{see} declared variable
object, \textit{see} primitive type
parameters, \textit{see} carrier
ParametersType, \textit{see} param.-type stat.
parametersType, \textit{see} param.-type stat.
p\textit{modify}, \textit{see} pointer-modify
Post, \textit{see} post statement
post, \textit{see} post statement
PostCondition, \textit{see} post-condition stat.
postCondition, \textit{see} post-condition stat.
PostInvariant, \textit{see} post-invariant stat.
postInvariant, \textit{see} post-invariant stat.
Pre, \textit{see} pre statement
pre, \textit{see} pre statement
PreCondition, \textit{see} pre-condition stat.
preCondition, \textit{see} pre-condition stat.
result, \textit{see} carrier
resultType, \textit{see} result-type statement
resultType, \textit{see} result-type statement
Role, \textit{see} role statement
role, \textit{see} role statement
roleInvariant, \textit{see} role-invariant stat.
roleInvariant, \textit{see} role-invariant stat.
show, \textit{see} show
show, \textit{see} statement-show
smodify, \textit{see} spec-modify

300
canonical behaviour, 175
carrier, 243
carrier name, 243
carrier record, 243, 252
carrier value, 243
carrierised field selection, 244
carrierised field-modify, 244
cartesian product
  binary, 91
cartesian sum
  arbitrary, 111
  binary, 93
chain, 60
child, 12
choice function, 44
class, 155
closed
  subcollection, 57
  supercollection, 57
c/a/contra-variance rule, 73
course specification, 137
collection, 32
collection comprehension, 41
collection enumeration, 41
collection-comprehension operator, 130
combined type operator, 77
combining suites, 12
communication, 17
commutation, 59
commutative, 58
comparability of collections, 60
complement, 48
completeness, 139
component, 158
component composition, 158, 188
component-oriented programming, 158
comprehension
  collection, 41
  function, 42
    mirrored, 42
  relation, 42
  set, 42
comprehension operator
  collection-, 130
  set-, 131
computation, 64
concatenation, 210
concrete metaclass, 4
confrontation, 68
conjoint, 58
conjoint sum, 111
conjunction, 31
connect output with input, 33
connection, 33
connection pattern, 212
connection protocol, 213
connection trace, 211
connection-protocol picture, 213
connection-trace picture, 212
connectional expression, 132
connectional product, 66
consequence, 31
conservative extension, 27, 204
conservative user, 201
conservativity rule, 204
constant expression, 119
constant function, 52, 53
constant lift, 119
constrain, 177
constrained cylindric pack, 96
constrained cylindric product
  arbitrary, 110
  binary, 94
constraint, 94
constructor, 165
contain, 33
context, 117, 178
context expression, 119
context preserving, 179
continuity preserving, 226
continuous, 60
contravariant, 75
converse, 51
correct
  partially, 139
  totally, 168
covariant, 75
cylindric product
  binary, 92
  constrained
    arbitrary, 110
    binary, 94
cylindric typing, 74
cylindric view, 20
cylindric-doubling function, 125
cylindric-type operator, 75
cylindric-type rule, 74
dark modify, 259, 260
dark protocol, 215
datatype-induction principle, 165
declarative, 150
declared attribute, 242
declared interface, 259
declared variable, 237
decomposition, 160
Dedekind, 69
default tag, 32, 39
define, 34
defined, 121
definition, 34
denotational semantics, 23, 118
dependency, 161
dependency relation, 198
dependent typing, 39
depth of a trace, 224
Desargues, 69
diagonal preserving, 87
direct instance, 4
directed graph, 34
disjoint, 58
disjoint sum, 111
disjoint-type rule, 127
disjunction, 31
distribution, 59
half-, 69
domain, 51, 77
  lower, 51
  upper, 51
domain equal, 58
domain part, 42
domain-equality relation, 58
domain-inclusion, 167
domain-split type rule, 126
dot matrix, 34
doubling function, 53
dummy part, 42
dynamic binding, 30
element, 32
element-of, 38
element-of relation, 39
empty, 54
empty collection, 48
empty in, 34
empty on, 82
empty relation, 48
empty sequence, 210
empty set, 48
empty statement, 176
enumeration
  collection, 41
  relation, 41
  set, 41
equality, 32, 46
equality relation, 38
equivalence, 31
everywhere, 53
excluded interface, 22, 257
existence of a variable, 236
existence of an object, 19
existential quantification, 31
expression, 117
  connectional, 132
  semantic, 117
  syntactic, 117
expression operator
  loose binary, 120
  loose unary, 121
  strict unary, 119
extensibility, 73, 78
extension of a suite, 25
external type, 39
external type system, 39
factory, 165
Feijen notation, 44
field, 18, 240
field expression, 24
field name, 240
field selection, 241
field value, 240
field-modify, 242
final global state, 18
fine specification, 137
fix notation, 36
fixed point, 63
  greatest, 63
  least, 63
fixed-point-of relation, 72
flat, 62
full, 54

303
full collection, 48
full in, 35
full on, 82
full relation, 48
function, 35
function comprehension, 42
mirrored, 42
functional, 35, 55
functional in, 35
functional on, 76
fundamental theorem, 62
fusion, 64
Galois connection, 61
genuine, 224
genuine-statement–aspect-operator, 250
genuine-statement–over, 248
genuine-statement–pack, 225
genuine-statement–spec, 225
genuine-statement–under, 248
genuineness preserving, 224
global state, 18, 240, 243
greatest fixed point, 63
greatest–fixed-point operator, 63
Guard, 143
half-distribution, 69
hide, 233
history invariant, 163
Hoare logic, 139
idempotent, 58
identifier, 32, 39
identity protocol, 214
identity protocol statement, 219
identity relation, 50
identity statement, 176
if-then-else, 143
implementation inheritance, 207
implication, 31
implicit typing, 39
inclusion, 46
independent part, 193
indirect equality, 46
indirect inequality, 46
induction, 64
infinite loop, 144
infix notation, 36
information preserving, 100, 110
inheritance, 13, 164
syntactic, 16
initial global state, 18
initialiser, 165
injection function, 93
injective, 55
inner class, 155
input, 33, 211
input condition, 56
input conditioner, 133
input type, 75
instance of a metaclass, 4
integer number, 32, 39
interaction, 3
interaction pattern, 3
interface, 18
interface name, 252
interface statement, 253, 254
internal type, 39
intersection
  arbitrary, 49
  binary, 47
introspective, 223
introspectivity preserving, 223
invariant, 163
invisible call, 185
irreflexive, 56
is-behavioural-refinement-of, 183, 190
is-contained-by, 33
is-element-of, 38
is-equal-to, 38, 46
is-override-of, 159
is-owner-of, 38
is-refinement-of, 160
is-statement-override-of, 188
is-statement-refinement-of, 190
is-subcollection-of, 46
is-supercollection-of, 46
is-total-behavioural-refinement-of, 199
is-total-refinement-of, 167, 171
is-total-statement-refinement-of, 199
isolation lemma, 195
ISpec carrier, 252
ISpec metamodel, 4
ISpec operator, 251
ISpec–sequential-composition, 259
ISpec-call, 261
iteration, 51

304
leapfrog, 52
least fixed point, 63
least–fixed-point operator, 63
left-unit, 59
left-zero, 59
length of a sequence, 209
lift
  constant, 119
  loose binary, 120
  loose unary, 121
  strict unary, 119
lift operator, 178
loop invariant, 142
loose binary expression operator, 120
loose binary lift, 120
loose product, 113
loose unary expression operator, 121
loose unary lift, 121
lower adjoint, 61
lower domain, 51
lower range, 51
main, 29, 165
map, 235
meaning of a suite, 25
metaclass, 4
metamodel, 4
method, 18, 155
method name, 252
method statement, 253, 254
mirrored function comprehension, 42
modify, 236
modify statement, 24
monomorph, 57
  anti-, 57
  in an argument, 57
monotonic, 57, 64
  anti-, 57
  in an argument, 57
monotonicity preserving, 177, 179
multiple-argument relation, 36

name, 155
natural number, 32, 39
negated binary predicate, 38
negation, 31
new state, 236
new value, 236
node, 211
node element, 210
node trace, 210
non-determinism, 118
non-empty, 54
non-full, 54
non-strict, 43
null, 236
number
  integer, 32
  natural, 32
  real, 32
number-of, 210
object, 3, 18, 240
object name, 240
observation sequence, 212
observer, 3
observer pattern, 3, 13
occurs-in, 210
old state, 236
old value, 236
operational, 151
order
  partial, 56
other, 48
output, 33, 211
output condition, 56
output conditioner, 133
output type, 75
over, 50
override, 159
override operator, 189
overriding, 26
owner-of, 38
owner-of relation, 39
pack, 155
package, 96, 155
pair, 36
pair algebra, 72
parameter, 18, 243
parameter name, 245
parameter record, 245
parameter value, 245
parameters-type statement, 253, 257
parent, 12
part
removed-from, 47
restricted to, 76
result, 18, 243
result-type statement, 253, 256
return, 17
right-unit, 59
right-zero, 59
role, 3, 18
role invariant, 7
role name, 252
role statement, 253, 254
role-invariant statement, 253, 257
root element, 210
same, 48
section, 37
selection, 235
self, 19, 243
semantic expression, 117
semantic inheritance, 26
semantics, 117
denotational, 118
semantics function, 117
semi-commutative diagram, 81
sequence, 209
sequential composition, 24, 50
sequential prot.-stat. composition, 219
sequential protocol composition, 214
sequential statement-composition, 176
set, 38, 40, 55
set comprehension, 42
set enumeration, 41
set-comprehension operator, 131
set-conjoint, 58
set-forcer, 78
set-preserving, 77
shape, 118
sharp, 62
show, 215
shunting, 67, 82, 133
signature, 20, 262
simple prot.-stat.-refinement rule, 229
simple protocol-refinement rule, 218
simple refinement rule, 183
single connection, 55
single element, 55
single input, 55
single output, 55
single-valued, 35
single-valued in, 35
single-valued on, 76
single-valued semantics, 43
single-valuedness type rules, 126
singleton set, 96
somewhere, 53
spack, 52
spec, 216
spec-modify, 260
specialising suites, 12, 25
specification, 137
spick, 52
square, 56
square rule, 181
state, 163, 236
state invariant, 7, 163
state-invariant condition, 166
state-invariant statement, 253, 258
state-invariant type, 166
statement, 175
statement operator, 176
statement–aspect-operator, 250
statement–visual-indep. extraction, 220
statement–visual-independent, 220
statement-component composition, 188
statement-dark, 220
statement-hide, 234
statement-inclusion, 176
statement-intersection
  arbitrary, 249
  binary, 176
statement-monotonic, 179
statement-over, 247
statement-override, 188
statement-override operator, 189
statement-pack, 177
statement-pick, 177
statement-refinement, 182, 190
statement-show, 220
statement-spec, 220
statement-under, 247
statement-union
  binary, 176
strict unary expression operator, 119
strict unary lift, 119
structural hierarchy, 155
structure of a suite, 20
visible call, 185
visual, 210
visual independent, 216
visual-independent extraction, 217
well-founded, 144
when, 47
while, 143
white-box semantics, 29
without, 47
zero, 59
Titles in the IPA Dissertation Series


A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computer Science, TUE. 1996-04


D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09


N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12


B.L.E. de Fluitert. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01


F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01


J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty
of Mathematics and Computer Science, UU. 1998-03


E. Voermans. Inductive Datatypes with Laws and Subtyping – A Relational Model. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. Towards Probabilistic Unification-based Parsing. Faculty of Computer Science, UT. 1999-02


C.H.M. van Kemenade. Recombinative Evolutionary Search. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. Learning Reliability: a Study on Indecisiveness in Sample Selection. Faculty of Mathematics and Natural Sciences, RUG. 1999-05


J.P. Warners. Nonlinear approaches to satisfiability problems. Faculty of Mathematics and Computing Science, TUE. 1999-08


P.R. D’Argenio. Algebras and Automata for Timed and Stochastic Systems. Faculty of Computer Science, UT. 1999-10

G. Fábián. A Language and Simulator for Hybrid Systems. Faculty of Mechanical Engineering, TUE. 1999-11


R. Schiefer. Viper, A Visualisation Tool for Parallel Program Construction. Faculty of Mathematics and Computing Science, TUE. 1999-15


T.E.J. Vos. UNITY in Diversity. A stratified approach to the verification of distributed algorithms. Faculty of Mathematics and Computer Science, UU. 2000-02


P.H.F.M. Verhoeven. The Design of the MathSpad Editor. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Pey. Design of a Fruit Juice Blending and Packaging Plant. Faculty of Mechanical Engineering, TUE. 2000-06

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language.* Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure.* Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication.* Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle.* Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection.* Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06


J. Hage. *Structural Aspects of Switching Classes.* Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes.* Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking.* Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols.* Faculty of Mathematics and Computing Science, TU/e. 2001-11


D. Bošnački. *Enhancing state space reduction techniques for model checking.* Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04


M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric
Systems. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. Models of Molecular Computing. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Feinher. Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. On-line Scheduling and Bin Packing. Faculty of Mathematics and Natural Sciences, UL. 2002-09


M.B. van der Zwaag. Models and Logics for Process Algebra. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. Probabilistic Extensions of Semantical Models. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12


J.I. van Hemert. Applying Evolutionary Computation to Constraint Satisfaction and Data Mining. Faculty of Mathematics and Natural Sciences, UL. 2002-14


Y.S. Usenko. Linearization in μCRL. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. Random Redundant Storage for Video on Demand. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. To Reuse or To Be Reused: Techniques for component composition and construction. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02


S.M. Bohte. Spiking Neural Networks. Faculty of Mathematics and Natural Sciences, UL. 2003-04


S.V. Nedea. Analysis and Simulations of Catalytic Reactions. Faculty of Mathematics and Computer Science, TU/e. 2003-06


H.P. Benz. Casual Multimedia Process Annotation – CoMPAs. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08


M.H. ter Beek. Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. The λ Abroad – A Functional Approach to Software Components. Faculty of Mathematics and Computer Science, UU. 2003-11

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02


P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

P. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16


S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07


E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08


P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20


N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21


R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02
C.N. Chong. Experiments in Rights Control - Expression and Enforcement. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. Design and Verification of Lock-free Parallel Algorithms. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. Specification and Analysis of Internet Applications. Faculty of Mathematics and Computer Science, TU/e. 2005-05


I. Kurtev. Adaptability of Model Transformations. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. Computational Aspects of Treewidth - Lower Bounds and Network Reliability. Faculty of Science, UU. 2005-09

O. Tveretina. Decision Procedures for Equality Logic with Uninterpreted Functions. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. Evolution of Finite Populations in Dynamic Environments. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. Data Mining using Genetic Programming: Classification and Symbolic Regression. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heerens. Top Quality Type Error Messages. Faculty of Science, UU. 2005-13

G.F. Frehse. Compositional Verification of Hybrid Systems using Simulation Relations. Faculty of Science, Mathematics and Computer Science, RU. 2005-14


T. Gelsema. Effective Models for the Structure of pi-Calculus Processes with Replication. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. Composing Constraint Solvers. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18


M. Valero Espada. Modal Abstraction and Replication of Processes with Data. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. Stepping through Haskell. Faculty of Science, UU. 2005-21


E. Dolstra. The Purely Functional Software Deployment Model. Faculty of Science, UU. 2006-01


P.R.A. Verbaan. The Computational Complexity of Evolving Systems. Faculty of Science, UU. 2006-03

L. Cheung. Reconciling Nondeterministic and Probabilistic Choices. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

M. Kyas. Verifying OCL Specifications of UML Models: Tool Support and Compositionality. Faculty of Mathematics and Natural Sciences, UL. 2006-05

B. Badban. Verification techniques for Extensions of Equality Logic. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

M. Hendriks. Model Checking Timed Automata - Techniques and Applications. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

A.J. Mooij. Constructive formal methods and protocol standardization. Faculty of Mathematics and Computer Science, TU/e. 2006-14

J. Ketema. Böhm-Like Trees for Rewriting. Faculty of Sciences, VUA. 2006-07


C.-B. Breunesse. On JML: topics in tool-assisted verification of JML programs. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

M.E. Warnier. Language Based Security for Java and JML. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

B. Markvoort. Towards Hybrid Molecular Simulations. Faculty of Biomedical Engineering, TU/e. 2006-09

B. Gebremichael. Expressivity of Timed Automata Models. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

S.G.R. Nijsen. Mining Structured Data. Faculty of Mathematics and Natural Sciences, UL. 2006-10

V. Sundramoorthy. At Home In Service Discovery. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

G. Russellos. Separation and Adaptation of Concerns in a Shared Data Space. Faculty of Mathematics and Computer Science, TU/e. 2006-11

B. Gebremichael. Expressivity of Timed Automata Models. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. Formalising Interface Specifications. Faculty of Mathematics and Computer Science, TU/e. 2006-19

L. Cheung. Reconciling Nondeterministic and Probabilistic Choices. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

M. Kyas. Verifying OCL Specifications of UML Models: Tool Support and Compositionality. Faculty of Mathematics and Natural Sciences, UL. 2006-05

B. Badban. Verification techniques for Extensions of Equality Logic. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

M. Hendriks. Model Checking Timed Automata - Techniques and Applications. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

A.J. Mooij. Constructive formal methods and protocol standardization. Faculty of Mathematics and Computer Science, TU/e. 2006-14

J. Ketema. Böhm-Like Trees for Rewriting. Faculty of Sciences, VUA. 2006-07


C.-B. Breunesse. On JML: topics in tool-assisted verification of JML programs. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

M.E. Warnier. Language Based Security for Java and JML. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

B. Markvoort. Towards Hybrid Molecular Simulations. Faculty of Biomedical Engineering, TU/e. 2006-09

B. Gebremichael. Expressivity of Timed Automata Models. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

S.G.R. Nijsen. Mining Structured Data. Faculty of Mathematics and Natural Sciences, UL. 2006-10

V. Sundramoorthy. At Home In Service Discovery. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

G. Russellos. Separation and Adaptation of Concerns in a Shared Data Space. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L.C.M. van Gool. Formalising Interface Specifications. Faculty of Mathematics and Computer Science, TU/e. 2006-19