Connecting Informal and Formal Mathematics

Hugo Johannes Elbers
Connecting Informal and Formal Mathematics

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op donderdag 28 mei 1998 om 16.00 uur

door

Hugo Johannes Elbers

geboren te Ubbergen
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. A.M. Cohen
en
prof.dr. J.W. Klop

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Elbers, Hugo Johannes

NUGI 852
Subject headings: formal languages / computer mathematics /
programming languages : data types / lambda calculus /
functional programming languages / term rewriting systems /
mathematical logic
CR Subject Classification (1991): F.4

druk: Universiteitsdrukkerij TU Eindhoven
## Contents

1 Introduction ................................................................. 1  
  1.1 Formal mathematics .................................................. 1  
  1.2 Computer assistance ................................................. 3  
  1.3 Related Work .......................................................... 7  
  1.4 Overview .............................................................. 9  

2 Rewriting ................................................................. 11  
  2.1 Abstract Reduction Systems ......................................... 11  
  2.2 Term Rewriting Systems ............................................ 13  

3 Logic in Type Theory .................................................... 20  
  3.1 Higher Order Logic .................................................. 20  
  3.2 Representation of Mathematics ..................................... 25  
  3.3 Automated Verification ............................................. 27  

4 Computations in Formal Proofs ......................................... 29  
  4.1 External Computations ............................................... 30  
  4.2 Inductive Types ..................................................... 35  
  4.3 Oracle types .......................................................... 45  

5 Representing Functions in Term Rewriting ............................ 53  
  5.1 Functional Programming ............................................ 54  
  5.2 Constructor Systems ................................................ 56  
  5.3 Priority Constructor Systems ....................................... 61  

6 Priority Rewriting in a Type System .................................. 77  
  6.1 Higher Order Logic with Pattern Matching ....................... 77  
  6.2 Properties of the Extended System ............................... 89  
  6.3 Encoding of Pattern Matching by Inductive Types .............. 102  
  6.4 Reduction Behaviour of the Encoding .............................. 118  
  6.5 Typability ............................................................ 134
7  Proof Development using Priority Rewriting  
  7.1  LEGO ................................................................. 140  
  7.2  LEGO with Pattern Matching .................................. 142  
  7.3  Proving Primality by Computations ......................... 148  

Bibliography  
  157  

Index  
  160  

Notation  
  162  

Samenvatting  
  164  

Curriculum Vitae  
  165
Preface

In the beginning, when I started working at the CWI in Amsterdam four years ago, I had only a few vague ideas of the daily work of a Ph.D. student. All of them turned out to be (more or less) true, except for one. I was expecting a strict plan for the research I was supposed to perform from my supervisors so I could only work on its detailed implementation. Instead, they gave me impulses for possible research, and did not prescribe everything I had to do. It took me quite some time to get used with this freedom. It is remarkable how my work gradually developed from doing trivial experiments to designing and implementing formal mathematical languages.

I would like to thank my supervisors Henk Barendregt, Arjeh Cohen and Jan Willem Klop, who shared their inspiring ideas with me, and gave me the time and the freedom to find a good subject for this thesis. Their support has significantly improved the quality of this manuscript I also thank Gilles Barthe, who stimulated me to write down my ideas and to plan my work. I appreciate the discussions with Erik Barendsen and Herman Geuvers, that helped me to state my ideas more precisely. Finally I want to thank Rob Nederpelt, Martijn Oostdijk, and Milena Stefanova for proofreading preliminary versions of this thesis and making suggestions for the improvement of the text. The diagrams in this thesis have been drawn using Paul Taylor's 'Commutative Diagrams' package.

I appreciate the financial support from the Netherlands Organization for Scientific Research (NWO) for the WINST project (SION 612-16-607).

Hugo Elbers

Eindhoven, March 31, 1998
Chapter 1

Introduction

A scientific discussion between two experts in some mathematical field can only be fully understood by experts in that field. First of all these experts use all kinds of keywords that have been defined in this particular field and are only known by mathematicians familiar with it. If one is familiar with the vocabulary of the field, one can understand what they talk about. But in order to be able to verify the correctness of what they say, one has to be an expert in the field, because they use large reasoning steps that are based on knowledge of the field.

An expert can make such an informal proof accessible for mathematicians that are familiar with the field by replacing each high level reasoning step by a number of reasoning steps of a lower level of abstraction. A mathematician who is familiar with the field can give definitions of its keywords and provide proofs of their fundamental properties. By use of this extra information the size of the reasoning steps in the proof can be decreased. If we repeat this process of making the proof more detailed, we obtain a proof with low level reasoning steps that can be understood by any mathematician.

At the other extreme, there is formal mathematics, that has been developed to give a precise meaning to informal mathematics. In formal mathematics one fixes a language of mathematical objects, statements, and reasoning rules that specify how a statement (the conclusion) can be derived from a number of statements (the assumptions). A mathematical theory can be formalized by specifying its primitive objects and axioms. A proof of a statement s in a theory is a derivation with conclusion s and with only axioms of the theory as assumptions. If a statement has a proof it is a true statement. The concept of proof allows us to verify the validity of statements.

We will now discuss the latter approach towards mathematics in more detail.

1.1 Formal mathematics

We will describe two formalisms for representing mathematics, namely set theory and type theory.
Set theory

A well-known example of a formal mathematical language is the axiomatized version of Cantor’s set theory by Zermelo ([35]) and Fraenkel ([15]) in first order predicate logic, shortly denoted as ZF. In set theory all mathematical objects can be coded as sets. For instance, a function is represented as a set of pairs that specify the (unique) result for each argument. Because everything is (coded as) a set, one can express meaningless statements such as $1 = +$ in ZF. This is not desirable, because a formal language should only represent informal mathematics and nothing else. Another disadvantage of ZF is that it is not possible to give operational definitions of functions, because they have to be coded as sets using axiomatic, descriptive definitions. Thus the role of computations in mathematics is ignored by this formal language.

Type theory

Type theory is a powerful formalism for representing formal mathematics, because this formalism allows us to filter out meaningless expressions of a language. In type theory a formal language is extended with types and rules for assigning types to expressions in the language. Instead of considering all expressions of the language, only typable expressions are considered. For instance, in the functional programming language ML [29] the number 1 has type int and + has type int→int→int. Since = demands two arguments of the same type, the expression 1 = + is not accepted by the ML type checker.

In a typed language one can represent statements as types by the so-called propositions-as-types principle (see [13]). A term with a proposition as type is considered a proof of that proposition. Thus the typing rules of such a language formalize the reasoning rules of the system.

Simply typed lambda calculus

By way of example, we present a formalization of first order proposition logic with logical connective $\rightarrow$ (implication) by simply typed lambda calculus, shortly denoted as $\lambda \rightarrow$.

We represent the propositions by types; these propositions have syntax

$$\text{Type} = P \mid \text{Type} \rightarrow \text{Type},$$

where $P$ is an infinite set of proposition variables.

We represent proofs by terms; they have syntax

$$\text{Term} = V \mid \text{Term \ Term} \mid \lambda V : P. \text{Term},$$

where $V$ is an infinite set of term variables.

A statement is a pair consisting of a Term $p$ and Type $t$, and is denoted as $p : t$ and is interpreted as ‘$p$ is a proof of $t$’. An assumption is a statement with a variable as first element. For instance, the sequence $x_1 : P_1, x_2 : P_1 \rightarrow P_2$ represents the assumption of the propositions $P_1$ and $P_1 \rightarrow P_2$. We will now describe the typing rules that represent the logical rules for deriving a conclusion from a sequence of assumptions in first order proposition logic. A typing rule of the form $\Delta \vdash A\alpha$ should be interpreted as ‘if $A$ and
... and $A_n$ then $C'$. The following typing rules formalize the notion $\Gamma \vdash p : t$, where $\Gamma$ is a sequence of assumptions, and $p \in \text{Term}$, and $t \in \text{Type}$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>(assumption)</td>
<td>$x : P \in \Gamma$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash x : P$</td>
</tr>
<tr>
<td>($\rightarrow$-elimination)</td>
<td>$\Gamma \vdash M : P \rightarrow Q$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash N : P$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash (MN) : Q$</td>
</tr>
<tr>
<td>($\rightarrow$-introduction)</td>
<td>$\Gamma, x : P \vdash M : Q$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash (\lambda x : P.M) : P \rightarrow Q$</td>
</tr>
</tbody>
</table>

If $\Gamma \vdash t : p$ we say that $t$ has type $p$ (or $t$ is an inhabitant of $p$) in $\Gamma$. Using the propositions-as-types principle we can formalize propositional proofs in $\lambda \rightarrow$. For instance, the following derivation represents a proof of the proposition $P_2$ from the assumptions $\Gamma = x_1 : P_1, x_2 : P_1 \rightarrow P_2$. Note the similarity with a logical derivation.

\[
\frac{x_2 : P_1 \rightarrow P_3 \in \Gamma \quad x_1 : P_1 \in \Gamma}{\Gamma \vdash x_2 : P_1 \rightarrow P_2} \quad \frac{\Gamma \vdash x_1 : P_1}{\Gamma \vdash x_2 \ x_1 : P_2}
\]

In this type system the structure of a Term determines which rule must be used to obtain a correct derivation of its type. The expression $x_2 \ x_1$ is called a proof-object, as it contains sufficient information to reconstruct the derivation above (given $\Gamma$). Thus one can use a proof-object with its assumptions to represent a propositional proof.

In simple typed lambda calculus all types represent propositions and all typable terms represent proofs. Later on we will present more complicated type systems for lambda calculus which allow us to represent mathematical objects such as the number 0 and the natural numbers, and to reason about these objects.

### 1.2 Computer assistance

The introduction of the computer has influenced mathematics deeply. Methods for solving problems of the form 'given $x$ find $y$ such that $P(x, y)$ holds', could often be brought to life as executable procedures (computer programs). Thus, instead of obtaining the desired result by writing out the defining equations of the method for a certain $x$, one could obtain this result automatically by running the computer program with input $x$. Moreover, computations that could not be done because of their length, can now efficiently be performed by an algorithm. We will discuss several aspects of the use of computers in formal mathematics.

**Proof checking**

The reasoning rules of most formal mathematical languages are so simple, that their applications can be verified by a computer program. To allow automated verification of formal
proofs the formal language must be decidable: an effective algorithm must be able to determine for each input whether it is a correct proof or not. Such an algorithm should not only recognize the syntax of the expressions and statements, but also verify the mathematical correctness. Unlike human beings, computers cannot forget to verify a certain condition in the application of a reasoning rule (if the rule is implemented correctly). Therefore the verification of a formal proof by a computer program (proof checking) contributes to its reliability.

**Demands imposed on formal mathematical languages**

First of all a formal mathematical language should be able to represent mathematical objects such as variables, functions, statements, proofs and work with these objects (by means of application, instantiation). Of course, any formal object or proof should be a representation of a mathematical object or proof in informal mathematics.

An important aspect of mathematics is the ability to increase the level of abstraction by using definitions, theorems and algorithms (due to the availability of computers). These abstraction mechanisms allow us to express complex notions, smart reasoning, or large computations by a short expression in the mathematical language. Clearly, these mechanisms should be available in a formal mathematical language to obtain a constant ratio between the length of a proof that can be understood by any mathematician, and the length of its formal representation (that can be verified by a proof checker). Formalizing mathematics would not be feasible if the length of a computer verifiable proof would be exponentially larger than the size of the equivalent human proof.

To allow the efficient verification by computer the mathematical objects of the formal language should be easily representable in a computer and its rules should have easily verifiable (decidable) conditions. Clearly, a large computer program is more likely to contain errors than a small program (written in the same language by the same programmer). Therefore a formal system should have as few constructs and rules as possible in order to allow a small implementation. Consequently, a small proof checker is more reliable than a large proof checker. Thus a good formal mathematical language is very expressive and requires only a small implementation. In particular such a language should not contain a predefined mathematical theory, but should be able to represent it.

**Automath**

One of the first systems for automated verification of mathematics is the Automath system ([13]), developed by N.G. de Bruijn. The Automath language has one general syntax for proofs, mathematical objects, statements, and types. The type of an expression determines what kind of object it is. One mechanism is used for introducing primitive objects and axioms (introduction of a typed variable). The definition mechanism enables us to give a name to a (parametrized) typed expression (for instance a proof). One can use definitions via instantiation. For instance, if we have defined double(x:nat):=x+x:nat then double(1) denotes 1+1. Using both mechanisms one can formalize a mathematical theory.
The features described above, make Automath a simple, general formal mathematical language that can be verified efficiently by a small proof checker. Unfortunately, one cannot define (and execute) algorithms in Automath. For instance, a statement such as 107 + 242 = 349 requires a long equational proof in Automath. If one could give an algorithmic definition of + (and execute it), then giving a formal proof of this statement would be trivial.

### Semantic levels in mathematics

In ordinary life we use language to refer to objects in the real world. For instance, when someone says the word chair we think about a certain piece of furniture. When we say the word ‘chair’ has five letters, we use it on a syntactic level. In the last case we talk about the language as part of our reality.

For some people, called Platonists, mathematics is just a description of a real world of mathematical objects. For mathematical work we do not need this Platonic reality. We can represent mathematics on a computer by a formalization of the mathematical language. We can also code the words of this formal language in order to be able to reason about the words. Via an evaluation function we can interpret coded words as formal mathematical expressions. In Figure 1.1 we visualize how natural numbers are represented in the four semantic levels. On the left-hand side we have mathematics in real life, and on the right-hand side we have the representation of mathematics on a computer.

### Interactive proof development

Formalizing mathematics in a general framework can be a tedious job. Therefore one usually builds a proof development system on top of the proof checker of the formal system. Such a tool provides assistance for developing formal mathematical theories in the proof checker. In this approach the reliability of the formal proofs depends on the proof checker. This is visualized in Figure 1.2. Thus the related proof development system can be made as user-friendly as possible by providing high level facilities for proving, defining, computing, using libraries of mathematical theories, and pretty printing. In practice, the proof checking
and proof development facilities are integrated in one system with a small core system that implements the formal language.

If we compare proof systems with implementations of programming languages then a proof checker is on the level of an assembler and we would like to have a proof development system on the level of an implementation of a modern programming language. A proof development system is just a user interface, it cannot increase the intrinsic mathematical strength of its formal system. For instance, if a formal language does not have constructs for defining and executing algorithms one can only simulate computations of an algorithm via equational reasoning on an axiomatically introduced function symbol.

Computations

The availability of computational power can be used in several ways. First one can give an algorithmic definition of a mathematical function instead of an axiomatic definition. In this way the result of the application of such a function to an argument can be automatically obtained. A more advanced use of computations is the implementation of methods that can solve certain mathematical problems as executable procedures.

Computer Algebra

Both approaches for using computations are implemented by Computer Algebra Systems. For instance, one can multiply matrices or solve an equation efficiently in a CAS. Unfortunately, a CAS provides hardly any facilities for reasoning and has no formal concept for proofs. As a consequence, it is not possible to prove the correctness of an algorithm of a CAS in the mathematical language provided by it. Most CASs only provide informal descriptions of the intended meaning of their algorithms in their manuals. Several designers of Computer Algebra Systems have realized this shortcoming and have started to pay more attention to the specification of the mathematical meaning of their expressions and algorithms. For instance, the system Axiom [22] has very complicated typing rules for specifying the meaning of its expressions.

Altogether, Computer Algebra Systems do not satisfy the demands that we formulated for (implementations of) formal mathematical languages. Still these efficient tools can support the process of finding proofs; we will illustrate this approach in this thesis. We will also describe how we can use computations of a CAS algorithm by formalizing the relation that is computed by the algorithm.
Related Work

Inductive types

Proof development systems such as Coq [10] and LEGO [27] provide computational power via structural induction on inductive terms. In theory this formalism is powerful enough to define and execute algorithms. One can even prove relations between the input and output of these algorithms in the formal language. Defining an algorithm by structural induction is not easy, because this formalism operates on a low level and imposes severe restrictions on recursive calls. Moreover algorithms defined in this way are hard to understand and have an inefficient computational behaviour. Thus inductive types are not the ideal formalism for introducing computations in type theory.

Functional programming

As we remarked before, executable procedures provide an essential abstraction mechanism for formalizing mathematics. Therefore we want to have a general, expressive formalism in which one can easily define executable functions. A powerful, user-friendly method for defining functions is provided by the pattern matching formalism of so-called functional programming languages. In pure functional programming languages the result of a function only depends on the value of its arguments, due to the absence of side-effects. Thus one cannot assign values to global variables in these languages. This makes reasoning about the result of a mathematical function easier than in imperative programming languages such as C ([23]). Unfortunately, reasoning is not possible in functional programming languages, because these languages do not have a type for statements. In order to overcome this problem we extend a typed formal language with the pattern matching formalism, in such a way that the computational meaning of functions can be used in formal proofs.

1.3 Related Work

We will describe several proof development systems that implement typed formal languages in which algorithms can be defined.

Nuprl

Besides the standard constructs of typed lambda calculus, the formal language of the Nuprl proof development system ([9]) has several basic type constructors such as list, disjoint union, cartesian product, and operators for specifying algorithms on their members, that are also available in programming languages. The type theory of Nuprl is so complex that type checking is in general undecidable, which makes verification less efficient. Thus the typing rules needed to derive a type for a term must be explicitly given. As the formal proof of a statement requires a well-formedness proof of the statement, the number of typing rules needed to derive a type for it is much larger than the length of the formal proof. Fortunately, the system can solve most well-formedness obligations automatically.
One can define functions by a recursive equation using a tactic that transforms the equation in an application of a fixed-point combinator (the \( \text{Y} \) combinator of the lambda calculus). The typing rules for this combinator do not demand a termination proof, thus functions with an infinite computational behaviour can be defined using this term.

In his thesis ([21]) P. Jackson describes how several methods for solving mathematical problems used in Computer Algebra Systems can be formalized in the Nuprl proof system. For example, he presents a collection of tactics for partially automating equational reasoning based on rewriting.

**Coq**

The proof development system Coq ([10]) implements a version of the typed lambda calculus with inductive types. In the language of the Coq proof assistant one can write case expressions using patterns in a syntax close to that of the functional programming language ML [29]. The patterns in the left-hand sides of these case expressions are built using variables and constructors and must be linear and exhaustive. These case expressions are compiled into primitive constructions that can only inspect the head constructor of an inductive term. In this compilation the patterns of the rules are inspected from top to bottom and from left to right in order to avoid ambiguity. When we combine this facility for writing case expressions with the fixed-point operator named \texttt{Fix} we can define recursive functions. The recursive calls of such a definition should be \textit{guarded by destructors}, in order to allow the definition of terminating functions only. Roughly speaking, this means that one argument (of an inductive type), say the \( k \)th argument, is selected for iteration: this argument may be inspected arbitrarily deep (using case analysis), but all recursive calls of the function being defined should have a proper subterm of this term as \( k \)th argument. For instance, the induction principle of each inductive type satisfies this criterion. For a precise definition of the notion ‘guarded by destructors’ we refer to [17].

This formalism is more flexible than primitive recursion, as recursive calls are also allowed on deep subterms. For instance, the function \texttt{Half} defined by the equations \( \text{Half}(0) = 0, \text{Half}(S(0)) = 0, \text{Half}(S(S(n))) = S(\text{Half}(n)) \), can be defined using this formalism, but it cannot be coded directly using primitive recursion, because the argument of the recursive call of the last equation is \( n \) and not \( S(n) \).

Each function defined using \texttt{Fix} can be codified as a function that is provably equal to it and is defined using elimination principles of inductive types. But the codified version does not always have the same reduction behaviour as the original function.

**ALF**

In ALF [28] one can may define functions by pattern matching. The patterns in such a definition should be exhaustive and mutually disjoint and may be non-linear. Moreover, a restriction is imposed on recursive calls that should guarantee that only terminating functions can be defined. One can interactively build an exhaustive, mutually disjoint set of patterns for a function definition using an algorithm written by Coquand ([11]).
Overview

After the user has selected a variable argument of an inductive type \( I \) the algorithm tries to determine all possible, most general patterns with a constructor of \( I \) as head. For instance, the most general patterns for a variable of the inductive type for natural numbers (with constructors 0 and \( S \)) are 0 and \( S \, y \). If the type \( I \) of the selected argument is a parameterized inductive type with constructors with dependent types, the algorithm tries to eliminate impossible cases. For instance, we can define the binary relation on the natural numbers \( \leq \) as a parameterized inductive type \( \text{Leq} \) with two constructors of type \( \text{Leq} \, 0 \, x \) and \( (\text{Leq} \, x \, y) \rightarrow (\text{Leq} \, (S(x)) \, (S(y))) \) respectively. Now, we can prove \( \neg (\text{Leq} \, (S(x)) \, 0) \) by defining a function \( f \) of this type without any rules by pattern matching. Notice that \( \neg P \) is represented as \( P \rightarrow \bot \), where \( \bot \) denotes the false proposition. Thus this function \( f \) takes an argument of type \( \text{Leq} \, (S(x)) \, 0 \) and yields a proof of the false proposition \( (\bot) \). The definition of \( f \) contains no rules, because all possible cases for this argument are treated, as no constructor of \( \text{Leq} \) can yield a term of type \( \text{Leq} \, (S(x)) \, 0 \). As this function definition has no rules, no restrictions are imposed on its range (except for type correctness). Thus reasoning by cases can be represented by pattern matching.

In this example we can rule out both cases, because one of their arguments begins with a different constructor and \( \text{Leq} \) itself is a constructor. But in general, it is not always possible to determine which cases can be ruled out. Thus this formalism of pattern matching is very powerful, but verification of its rules is an undecidable problem.

1.4 Overview

The main contribution of this thesis is the description of a formal mathematical language, that we call \( \lambda \text{HOL}_\nu \), based on type theory in which executable functions can be defined by pattern matching. The pattern matching formalism of this formal language can be used for reasoning by cases, and enables the development of certified proving procedures. We implement a program, called LEGO with Pattern Matching, that can verify the correctness of expressions in this formal language. In Figure 1.3 we visualize the relations between the several languages and computer programs that occur in this thesis. New implementations and languages are indicated by fat boxes, and bold numbers refer to sections.

In Chapter 2 we describe how we can formalize computations. First we introduce a very general model for computations on objects whose structure is unknown, called Abstract Reduction System. Then we present Term Rewriting Systems as a general model for computations in functional programming languages. A Term Rewriting System is an Abstract Reduction System that rewrites objects with a term structure.

In Chapter 3 we describe how we can represent logic in type theory. First we introduce Higher Order Logic, a typed version of \( \lambda \)-calculus, and illustrate how mathematics can be formalized in this language. Then we describe the fundamental properties of this formal language that make it suited for automated verification.

In Chapter 4 we discuss the role of computations in formalizing mathematics. First we present several methods for using external computations (that are not done in the formal language) to make the construction of formal proofs easier. Then we discuss two formalisms
for representing computations in the formal language.

In Chapter 5 we illustrate the elegance of the pattern matching formalism of functional programming languages for defining functions. Then we introduce Constructors Systems as Term Rewriting Systems with two kinds of function symbols: constructor symbols (without a computational meaning) and defined symbols (with a computational meaning). This separation of defined symbols and constructor symbols gives Constructor Systems a clear semantics that is simpler than the semantics of (the more general) Term Rewriting Systems. Next we introduce Priority Constructor Systems that use an order on rules to determine which rule may be applied. Priority rewriting can be used to disambiguate Term Rewrite Systems with overlapping rules and to specify an exhaustive function by a few rules. Finally we present an algorithm that transforms Priority Constructor Systems to equivalent Constructor Systems.

In Chapter 6 we describe the problems that are involved in the extension of the formal language of Higher Order Logic with pattern matching and how these problems can be solved. After we have given a formal definition of this extension we prove several fundamental properties of this type system. For instance, we show that the result of the application of a function defined by pattern matching to its arguments is unique. Next we analyse the relation between pattern matching and inductive types and present an algorithm that maps functions defined by pattern matching to their representations based on inductive types. Using this algorithm we show that (a sequential version of) priority rewriting is terminating for (legal) types. Finally we present a type synthesis algorithm for the extended type system and show that it has decidable type checking.

In Chapter 7 we give a global description of our extension of the proof development system LEGO with priority rewriting. We explain how the new typing rules are verified, and we describe its facilities for defining functions by pattern matching. Finally we describe the formalization of a serious example to test our prototype.
Chapter 2

Rewriting

In this chapter we will introduce basic concepts concerning computations that will be used in the remainder of this thesis. First Abstract Reduction Systems, which form a very abstract model for computations, are discussed. In this model no assumptions are made on the objects on which the computations are performed. But we can formulate fundamental questions such as 'does the computation terminate?', and 'is the result of the computation unique?' in this framework. In programming languages and mathematics, objects with a computational meaning are expressions that have a general structure. A Term Rewriting System (TRS) is an Abstract Reduction System that rewrites expressions, that are objects with a term-structure (of the form \( f(a_1, \ldots, a_n) \)). The computations in a TRS are specified by a set of rules, that are directed equations between terms. We can take advantage of the structure of the expressions and the computation rules in order to specify conditions that guarantee fundamental properties such as termination of computations or uniqueness of the result of a computation.

2.1 Abstract Reduction Systems

An abstract model for computations is given by Abstract Reduction Systems. We will define some basic notions and properties. For more information we refer to [24].

**Notation 2.1.1** In this section we assume \( I \) to be a finite set.

**Definition 2.1.2** An Abstract Reduction System (ARS) is a structure \( \langle A, (\rightarrow_i)_{i \in I} \rangle \) consisting of a set \( A \) and a set of binary relations \( \rightarrow_i \) on \( A \), also called reduction or rewrite relations. If \( I = \{ i \} \), we just write \( \rightarrow \) instead of \( \rightarrow_i \).

**Definition 2.1.3** If, for \( a, b \in A \), we have \((a, b) \in \rightarrow_i\), we write \( a \rightarrow_i b \), and call \( a \rightarrow_i b \) a reduction step and \( b \) a one-step \((i-)\) reduct of \( a \) in the ARS \( \langle A, (\rightarrow_i)_{i \in I} \rangle \).

The reflexive closure of \( \rightarrow_i \) is written as \( \rightarrow_i^\ast \). Thus \( a \rightarrow_i^\ast a \), for all \( a \in A \). The transitive closure of \( \rightarrow_i \) is written as \( \rightarrow_i^+ \). Thus \( a \rightarrow_i^+ b \), if there exists a non-empty sequence
of reduction steps \( a_1 \rightarrow_i \ldots \rightarrow_i a_n \), such that \( a = a_1 \) and \( b = a_n \) (1 < n). We call \( a_1 \rightarrow_i \ldots \rightarrow_i a_n \) a reduction sequence. The transitive reflexive closure of \( \rightarrow_i \) is written as \( \\rightarrow_i^= \). The equivalence relation generated by \( \rightarrow_i \) is denoted by \( \equiv_i \). The union \( \rightarrow_i \cup \rightarrow_j \) is denoted by \( \rightarrow_{i,j} \).

Now that we know what an ARS is, we can describe some fundamental properties. An example of a fundamental property of ARSs is whether any two finite reduction sequences with the same initial element can be extended such that they have the same resulting element.

**Definition 2.1.4** Let \( \langle A, (\rightarrow_i)_{i \in I} \rangle \) be an ARS.

1. \( \rightarrow_i \) is confluent, if for any two reductions \( a \rightarrow_i b, a \rightarrow_i c \), there exists an element \( d \in A \) such that \( b \rightarrow_i d \) and \( c \rightarrow_i d \).

2. \( \rightarrow_i \) commutes with \( \rightarrow_j \), if for any two reductions \( a \rightarrow_i b, a \rightarrow_j c \), there exists an element \( d \in A \) such that \( b \rightarrow_j d \) and \( c \rightarrow_i d \).

3. \( \rightarrow_i \) is subcommutative if for any two one step reductions \( a \rightarrow_i b, a \rightarrow_i c \), there exists an element \( d \in A \) such that \( b \rightarrow_i^= d \) and \( c \rightarrow_i^= d \).

**Lemma 2.1.5** If \( \langle A, \rightarrow_i, \rightarrow_j \rangle \) is an ARS such that \( \rightarrow_i = \rightarrow_j \) and \( \rightarrow_i \) is subcommutative then \( \rightarrow_j \) is confluent.

**Lemma 2.1.6** (Hindley-Rosen) Let \( \langle A, (\rightarrow_i)_{i \in I} \rangle \) be an ARS such that for all \( i, j \in I, \rightarrow_i \) commutes with \( \rightarrow_j \). Then the union \( \bigcup_{i \in I} \rightarrow_i \) is confluent.

**Lemma 2.1.7** Let \( \langle A, \rightarrow_i, \rightarrow_j \rangle \) be an ARS. Assume that for any two reductions \( a \rightarrow_i b \) and \( a \rightarrow_j c \) there exists \( d \in A \) such that \( b \rightarrow_j d \) and \( c \rightarrow_i^= d \). Then \( \rightarrow_i \) and \( \rightarrow_j \) commute.

Another fundamental property of ARSs is whether reduction sequences are finite, and if this is the case what the resulting terms are.

**Definition 2.1.8** Let \( \langle A, \rightarrow \rangle \) be an ARS.

1. We say that \( a \in A \) is a normal form if there is no \( b \in A \) such that \( a \rightarrow b \). Further \( b \in A \) has a normal form, if \( b \rightarrow a \) for some normal form \( a \in A \).

2. We say that \( a \in A \) is weakly normalizing if \( a \) has a normal form. \( \langle A, \rightarrow \rangle \) is called weakly normalizing if all \( a \in A \) are weakly normalizing.

3. We say \( a \in A \) is strongly normalizing if there is no infinite reduction sequence starting with \( a \). \( \langle A, \rightarrow \rangle \) is called strongly normalizing if all \( a \in A \) are strongly normalizing.

**Remark 2.1.9** In a confluent ARS \( \langle A, \rightarrow \rangle \) each \( a \in A \) has at most one normal form.
2.2 Term Rewriting Systems

We now introduce the concept of Term Rewriting System (TRS); it can be regarded as a refinement of the notion ‘Abstract Reduction System’. For a general introduction we refer to [24]. Since we intend to use functions in a typed environment, we will describe many-sorted term rewriting systems.

**Notation 2.2.1** Let $A$ be a set. The set of finite sequences of elements of $A$ is denoted by $A^*$. The empty sequence is denoted by $\epsilon$. A typical element of $A^*$ is $a_1, \ldots, a_n$, where $a_i \in A$ for all $i \leq n$. An arbitrary sequence is denoted as $\vec{a}$. The length of sequence $\vec{a} \in A^*$ is denoted as $|\vec{a}|$. If $\vec{a} \in A^*$ then $a_i$ denotes the $i$th element of $\vec{a}$, where $i \leq |\vec{a}|$. If $\vec{a} = a_1, \ldots, a_m \in A^*$ and $\vec{b} = b_1, \ldots, b_n \in A^*$ then $\vec{a}, \vec{b}$ denotes the sequence $a_1, \ldots, a_m, b_1, \ldots, b_n \in A^*$. The set of non-empty sequences of elements of $A$ is denoted by $A^+$. In some situations we will write $(a_1, \ldots, a_n)$ to denote the sequence $a_1, \ldots, a_n \in A^*$ in order to improve the readability.

In this section we assume $S$ to be a finite set. In the following definition we use the notation introduced in ([20]).

**Definition 2.2.2** An $S$-sorted signature $\Sigma$ is a set of function symbols $\mathcal{F}$ together with a typing function $\tau : \mathcal{F} \rightarrow S^+$. We call $S$ the sorts of $\Sigma$. For ease of notation, we just write $G \in \Sigma$ instead of $G \in \mathcal{F}$.

The typing function $\tau$ of a signature $\Sigma$ specifies the number of arguments and their types (the arity) and the type of the result (the sort) for each function symbol.

**Notation 2.2.3** To indicate that we use sequences in $S^+$ as types, we denote $s_1, \ldots, s_n, s \in S^+$ as $s_1 \times \ldots \times s_n \rightarrow s$. If $n = 0$ we just write $s$. If $\tau(G) = s_1 \times \ldots \times s_n \rightarrow s$, then $G$ is said to have arity $s_1 \times \ldots \times s_n$ and sort $s$. If $\tau(G) = s$ we call $G$ a constant (of sort $s$).

Notice that ‘arity’ is not a number, but a sequence of sorts.

**Example 2.2.4** We can specify a signature for lists of natural numbers as follows.

<table>
<thead>
<tr>
<th>sort</th>
<th>nat,list</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td>0:</td>
</tr>
<tr>
<td></td>
<td>$S$: nat $\rightarrow$ nat</td>
</tr>
<tr>
<td></td>
<td>Nil: list</td>
</tr>
<tr>
<td></td>
<td>Cons: nat $\times$ list $\rightarrow$ list</td>
</tr>
<tr>
<td></td>
<td>Length: list $\rightarrow$ nat</td>
</tr>
</tbody>
</table>

Such a specification should be interpreted as follows: the sorts of the signature are listed behind the keyword sort; the function symbols with their arity and sort are listed behind the keyword func. Thus in the specification above the sorts are $\{\text{nat}, \text{list}\}$, the function symbols are $\{0, S, \text{Nil}, \text{Cons}, \text{Length}\}$ and we have $\tau(0) = \text{nat}$, $\tau(S) = \text{nat} \rightarrow \text{nat}$, ... etc.
We want to formalize the notion of 'algebraic data type' as used in functional programming languages. Therefore we introduce the notion 'strict', following the definition in [20], that guarantees that a sort is not empty. Furthermore, we introduce the notion of 'dependency' between sorts, that indicates, that for constructing a terms of one sort, terms of another sort are needed. We will forbid mutual dependency of sorts to allow the representation of algebraic data types by inductive types.

**Definition 2.2.5** Let \( \Sigma \) be an \( S \)-sorted signature.

1. A sort \( s \in S \) is strict in \( \Sigma \), if for some \( F \in \Sigma \)
   
   (a) \( \tau(F) = s \), or
   
   (b) \( \tau(F) = s_1 \times \ldots \times s_n \rightarrow s \), and \( s_i \) is strict in \( \Sigma \), for all \( i \leq n \).

   We say that \( \Sigma \) is strict if all sorts are strict in \( \Sigma \).

2. A sort \( s \) depends on \( t \in S \) in \( \Sigma \), if \( s \neq t \), \( \tau(F) = s_1 \times \ldots \times s_n \rightarrow s \) for some \( F \in \Sigma \), and

   (a) \( t = s_i \), or
   
   (b) \( s_i \) depends on \( t \) in \( \Sigma \), for some \( i \leq n \).

3. We call \( \Sigma \) an algebraic data type-signature if \( \Sigma \) is strict and has no mutually dependent sorts.

**Example 2.2.6** The signature of Example 2.2.4 is an algebraic data type-signature. The following strict signature is *not* an algebraic data type signature:

<table>
<thead>
<tr>
<th>sort</th>
<th>nat, tree, forest</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td>S: nat \rightarrow nat</td>
</tr>
<tr>
<td></td>
<td>Node: nat \times forest \rightarrow tree</td>
</tr>
<tr>
<td></td>
<td>Nil: forest</td>
</tr>
<tr>
<td></td>
<td>Cons: tree \times forest \rightarrow forest</td>
</tr>
</tbody>
</table>

In Abstract Reduction Systems no assumptions are made on the objects on which the computations are performed. We will now introduce objects, called terms, that have a certain structure. A term is built from function symbols of a signature, and variables that represent arbitrary expressions.

**Definition 2.2.7** Let \( I \) be a set. An \( I \)-indexed set \( A \) is a family of component sets \( A_i \) for each index \( i \in I \). If \( A \) and \( B \) are \( I \)-indexed sets, then a \((I\text{-indexed})\) mapping of \( I \)-indexed sets \( f : A \rightarrow B \) is an \( I \)-indexed set of functions \( \langle f_i : A_i \rightarrow B_i \mid i \in I \rangle \).
Notation 2.2.8 We assume $V$ to be an $S$-indexed set of variables, such that $V_s$ is infinite for each sort $s \in S$. If we want to emphasize the sort of a variable we will write $x$ to denote a variable $x$ of sort $s$.

Now we can define inductively what a ‘term’ of ‘sort’ $s$ is.

Definition 2.2.9 The $S$-indexed set $T(\Sigma, V)$ of terms over an $S$-sorted signature $\Sigma$ is the smallest set satisfying the following clauses:

1. $x \in T(\Sigma, V)_s$, for $x \in V_s$.

2. $F(t_1, \ldots, t_n) \in T(\Sigma, V)_s$, for $F \in \Sigma$ with $\tau(F) = s_1 \times \ldots \times s_n \rightarrow s$, and $t_i \in T(\Sigma, V)_{s_i}$ (for all $i \leq n$).

For $t \in T(\Sigma, V)$, we write $\text{var}(t)$ for the set containing all variables that occur in $t$. If $\text{var}(t) = \emptyset$, we call $t$ a closed term, and we write $t \in T(\Sigma)$.

By convention names of variables will begin with a lower-case letter, and names of function symbols will not begin with such a character. Instead of $C()$ we will just write $C$.

Example 2.2.10 Examples of terms over the signature of Example 2.2.4 are:

$0, S(0), \text{Length}(\text{head})$ and $\text{Cons}(S(\text{nat} x), \text{Nil})$.

The expressions $\text{Nil}(0), S(\text{Nil})$ and $\text{Cons}(\text{Nil}, 0)$ are not terms.

Now that we have defined terms, we can define ‘rewrite rules’. Later we will show how rewrite rules give a computational meaning to certain terms.

Definition 2.2.11 The $S$-indexed set $R(\Sigma, V)$ of rewrite rules over an $S$-sorted signature $\Sigma$ is the smallest set such that: if $t_1, t_2 \in T(\Sigma, V)_s$, and $t_1 \notin V$ and $\text{var}(t_2) \subseteq \text{var}(t_1)$ then $(t_1, t_2) \in R(\Sigma, V)_s$. We will denote a rewrite rule $(t_1, t_2) \in R(\Sigma, V)$ as $t_1 \rightarrow t_2$.

If we want to give a rule a name, e.g., $r$, we will write $r : t_1 \rightarrow t_2$.

Example 2.2.12 Examples of rewrite rules for the function symbol Length of the signature specified in Example 2.2.4 are:

$$\text{Length}(\text{Nil}) \rightarrow 0$$

$$\text{Length}(\text{Cons}(x, y)) \rightarrow S(\text{Length}(y))$$

Notice that $S(\text{Length}(y)) \rightarrow \text{Length}(\text{Cons}(x, y))$ is not a rewrite rule.

Definition 2.2.13 An $S$-sorted Term Rewriting System is a pair consisting of an $S$-sorted signature $\Sigma$ and an $S$-indexed set of rewrite rules over $\Sigma$.

The pair consisting of the signature of Example 2.2.4 and the rewrite rules of Example 2.2.12 is a Term Rewriting System. From now on we will use the abbreviation TRS for Term Rewriting System.

Before we can define how a rewrite rule induces a rewrite relation, we need to know what substitutions and subterms are. Substitution is the replacement of a number of variables in a term by terms.
Definition 2.2.14 Let $\Sigma$ be an $S$-sorted signature.

1. A substitution $\sigma$ is a mapping of $S$-indexed sets $\sigma : V \rightarrow T(\Sigma, V)$

2. Instantiating a term $t$ with substitution $\sigma$, notation $t^\sigma$, is inductively defined as:
   - $(a)$ $x^\sigma = \sigma(x)$, for $x \in V$.
   - $(b)$ $F(t_1, \ldots, t_n)^\sigma = F(t_1^\sigma, \ldots, t_n^\sigma)$, for $F \in \Sigma$ and terms $t_i \in T(\Sigma, V)$.

3. The composition of two substitutions $\sigma$ and $\tau$, notation $\tau \circ \sigma$, is defined as $(\tau \circ \sigma)(x) = (\sigma(x))^\tau$, for $x \in V$.

Example 2.2.15 Let $\sigma$ be a substitution with $\sigma(\text{nat}(x)) = 0$ and $\sigma(\text{list}(y)) = \text{Nil}$. We have:

- $\text{Cons}(x, y)^\sigma = \text{Cons}(0, \text{Nil})$
- $\text{S}(\text{Length}(y))^\sigma = \text{S}(\text{Length}(\text{Nil}))$

The intended meaning of a rewrite rule $l \rightarrow r$ is that any instance of its left-hand side $l^\sigma$ may be reduced to $r^\sigma$. In order to allow reduction inside a term we introduce the notion of ‘term with a hole’, that is usually called context.

Definition 2.2.16 The $S \times S$-indexed set $C(\Sigma, V)$ of terms with a hole over an $S$-sorted signature $\Sigma$ is the smallest set satisfying the following requirements:

1. $[ ] \in C(\Sigma, V)_{s_1, s_1}$, for all $s \in S$.

2. $F(t_1, \ldots, t_i-1, [ ], t_{i+1}, \ldots, t_n) \in C(\Sigma, V)_{s_0, s_{n+1}}$, for all $[ ] \in C(\Sigma, V)_{s_0, s_1}$, all $F \in \Sigma$ with $\tau(F) = s_1 \times \ldots \times s_n \rightarrow s_{n+1}$, and all $t_j \in T(\Sigma, V)_{s_j}$ ($1 \leq i, j \leq n, i \neq j$).

A term with a hole $C[ ] \in C(\Sigma, V)_{s_1, s_2}$ is said to have arity $s_1$ and sort $s_2$. The result of replacing the hole in $C[ ]$ with a term $t \in T(\Sigma, V)_{s_1}$, denotation $C[t]$, is a term in $T(\Sigma, V)_{s_2}$. We say that $t$ is a subterm of $C[t]$. If $C[ ] \neq [ ]$, then $t$ is a proper subterm of $C[t]$.

Example 2.2.17 Examples of terms with a hole over the signature of Example 2.2.4 are:

- $\text{nat}( [ ]), \text{S}(\text{nat}( [ ]))$, and $\text{Length}(\text{Cons}(0, \text{list}( [ ])))$.

We have $\text{nat}(0) = 0$, and $\text{Cons}(x, \text{list}(\text{Cons}(0, \text{Nil}))) = \text{Cons}(x, \text{Cons}(\text{Cons}(0, \text{Nil})))$.

The expressions $0, \text{Cons}(\text{nat}( [ ]), \text{list}( [ ]))$, and $\text{S}(\text{list}( [ ]))$ are not terms with a hole.

Now we can define the notion of ‘reduction step’.

Definition 2.2.18 Let $s_1, s_2 \in S$. Let $r : t_1 \rightarrow t_2 \in R(\Sigma, V)_{s_1}$ be a rewrite rule. Let $\sigma : V \rightarrow T(\Sigma, V)$ be a substitution, and $C[ ] \in C(\Sigma, V)_{s_1, s_2}$ a context. Then $C[t_1^\sigma] \rightarrow_r C[t_2^\sigma]$ is a reduction step of sort $s_2$. We say $C[t_1^\sigma]$ is a redex, and $t_1^\sigma$ is a redex occurrence in $C[t_1^\sigma]$.

We call $\rightarrow_r$ the one-step reduction relation generated by $r$.

Example 2.2.19 Using the rewrite rules for $\text{Length}$ of Example 2.2.12 and the substitutions of Example 2.2.15 we obtain the following reduction steps:
Length(Cons(0,Nil)) → S(Length(Nil)),
S(Length(Nil)) → S(0).

Remark 2.2.20 For each TRS(Σ, R) there is a corresponding ARS, namely
⟨T(Σ, V), (→r)⟩ ∈ R. Via the associated ARS, all notions and properties defined in the
previous section carry over to TRSs.

Normalization

A well-known technique for proving that a TRS is strongly normalizing, is constructing a
total, well-founded order on terms, such that every rule reduces a term to a smaller term.
We will define a variant of the lexicographical path order (see [14]), that is used for this
purpose.

Definition 2.2.21 Let Σ be an S-sorted signature. Let ∇ be a strict partial order on Σ
(thus G ∇ G for G ∈ Σ). The relation ∇po on terms is defined as follows:

1. F(t₁, ..., tₙ) ∇po tᵢ, for i ≤ n.
2. F(t₁, ..., tₙ) ∇po G(u₁, ..., uₙ), if F ∇ G ∧ ∀i ≤ m F(t₁, ..., tₙ) ∇po uᵢ.
3. F(t₁, ..., tₙ) ∇po F(t₁, ..., tᵢ₋₁, uᵢ, ..., uₙ), if tᵢ ∇po uᵢ ∧ ∀j > i F(t₁, ..., tₙ) ∇po uⱼ.
4. t₁ ∇po t₂, if t₁ ∇po t₂ ∧ t₂ ∇po t₃.

Example 2.2.22 Let Σ be the signature of Example 2.2.4. We can define a strict partial
order ∇ on Σ by: Length ∇ S and Length ∇ 0. Now we have S(0) ∇po 0, and
Length(Nil) ∇po 0, and Length(Cons(x, y)) ∇po S(Length(y)).
We do not have S(x) ∇po 0, because S and 0 are not ordered.

Theorem 2.2.23 Let (Σ, R) be an S-sorted Term Rewriting System. Assume ∇ is a well-
founded strict partial order on Σ. If l ∇po r, for every rule l → r ∈ R, then (Σ, R) is
strongly normalizing.

Proof
To be found, for instance, in [14].

Example 2.2.24 Using the theorem with the partial order defined in Example 2.2.22 we
can prove that the TRS for Length of Example 2.2.12 is strongly normalizing.
Confluence

If a rule interferes with another rule, a reduction step of the first rule can eliminate a redex occurrence for the second rule. Thus a TRS with interfering rules might not be confluent.

Example 2.2.25 The following TRS for the function Choice is not confluent, because Choice(False,True) has normal forms True and False.

```
sort  bool
func True: bool
        False: bool
        Choice: bool x bool -> bool
rule  Choice(False,x) -> False
        Choice(x,True) -> True
```

We will now formulate in detail how such harmful interference arises, and next introduce the notion of ‘weak orthogonality’ that forbids such interference and guarantees confluence.

Definition 2.2.26 Let $\Sigma$ be an $S$-sorted signature. Terms $t, u \in \mathcal{T}(\Sigma, V)$ are unifiable, if there exists a substitution $\sigma : V \rightarrow \mathcal{T}(\Sigma, V)$ such that $t^\sigma = u^\sigma$. We say that $\sigma$ is a unifier for $t$ and $u$. A substitution $\sigma$ is a most general unifier for $t$ and $u$, notation $\sigma = \mgu(t, u)$, if $\sigma$ is a unifier for $t$ and $u$, and for all unifiers $\tau$ for $t$ and $u$ there exists a substitution $\sigma'$ with $\sigma' \circ \sigma = \tau$.

If two terms $t$ and $u$ are unifiable, then there exists a $\sigma$ such that $\sigma = \mgu(t, u)$; notice that $\sigma$ is unique modulo renaming of variables.

Definition 2.2.27 Let $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ be two rewrite rules such that $l_1$ is unifiable with a non-variable subterm of $l_2$. Thus there exists a term with a hole $C[\ ]$, a term $t$ and a substitution $\sigma$, such that $l_2 = C[t]$ and $\sigma = \mgu(t, l_1)$. The term $l_2'$ can be reduced in two possible ways: $C[t]^\sigma \rightarrow C[r_1]^\sigma$ and $l_2' \rightarrow r_2^\sigma$. Now the pair of reducts $(C[r_1]^\sigma, r_2^\sigma)$ is called a critical pair obtained by superposition of $l_1 \rightarrow r_1$ on $l_2 \rightarrow r_2$. If $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ are the same rewrite rule, we furthermore require that $l_1$ is unifiable with a proper (i.e., $\neq l_1$) subterm of $l_2 = l_1$. A critical pair $(t_1, t_2)$ is trivial if $t_1 = t_2$.

Notice that a TRS can only be confluent if the terms of each critical pair obtained by superposition of two of its rules have a common reduct.

Example 2.2.28 The TRS for Choice of Example 2.2.25 has a critical pair (False,True). As False and True are two different normal forms this TRS is not confluent. The interference of the first two rules of the following TRS is harmless:
This TRS has one critical pair \( (0, 0) \).

From the previous example we learnt that a TRS with critical pairs needs not always be non-confluent. By inspecting the critical pairs we can detect cases in which a TRS with interfering rules is confluent. Before we can define a property which guarantees confluence, we need to define the notion 'left-linearity'.

**Definition 2.2.29** Let \( \Sigma \) an \( S \)-sorted signature. A term \( t \in T(\Sigma, V) \) is **linear** if no variable occurs more than once in \( t \). A rule \( t_1 \rightarrow t_2 \in R(\Sigma, V) \) is **left-linear** if \( t_1 \) is linear. An \( S \)-sorted TRS \( (\Sigma, R) \) is **left-linear** if each rule \( r \in R \) is left-linear.

**Definition 2.2.30** Let \( (\Sigma, R) \) be an \( S \)-sorted Term Rewriting System. \( (\Sigma, R) \) is **weakly orthogonal** if \( (\Sigma, R) \) is left-linear and contains only trivial critical pairs.

**Theorem 2.2.31** Every weakly orthogonal Term Rewriting System is confluent.

**Proof**
Can be found in [24].

Using this theorem we can prove that the TRSs of Examples 2.2.12 and 2.2.28 are confluent.
Chapter 3

Logic in Type Theory

In this chapter we will describe how we can represent logic in type theory. First we introduce the type system of Higher Order Logic ($\lambda$HOL), a language for representing formal mathematics. This framework has rules for specifying the structure of its objects, and a computation rule on them that formalizes the computation of the result of an application of a function $x \mapsto f_a$ to an argument $a$ yielding $f_a$. Mathematical theories can be represented in $\lambda$HOL by contexts that contain their primitive notions and axioms. This framework has also typing rules, for determining the meaningful objects in a context. Thus an object is meaningful if it can be given an type, and the type of an object indicates what kind of object it is. Roughly speaking, the typing rules for proofs formalize the reasoning rules of intuitionistic predicate logic. This framework is suited for an axiomatic formalization of mathematical theories, but has no ‘calculation-power’: it is not expressive enough for executing algorithms. Finally we show that Higher Order Logic is suited for automated verification. Thus the correctness of its formal proofs is decidable.

3.1 Higher Order Logic

In the previous chapter we have described sorted Term Rewriting Systems as a formalism for specifying functions with a computational meaning. Unfortunately, TRSs are not expressive enough to serve as a formal mathematical language. For instance, it is not possible to represent propositions such as $(\exists x \neg (P(x))) \rightarrow \neg \forall x P(x)$ as a term (or a sort) of a TRS. In this section we will describe a formal language in which one can represent mathematical notions, propositions and proofs.

Mathematics is too complicated to specify all its notions and rules at once. In order to overcome this problem we will specify a formal mathematical language, that is introduced by Church in [8], in two stages: first we introduce a grammar for pseudo terms that describes the structure of the formal mathematical expressions, and second we present typing rules that determine which pseudo terms represent mathematical objects.

Definition 3.1.1 Let $V$ be an infinite set of variables, and $C$ an infinite set of constants. The sets $V$ and $C$ are disjoint. The set of pseudo terms $T$ is defined as follows:
\[ T = V \mid C \mid (TT) \mid \lambda V : T . T \mid \Pi V : T . T \]

A pseudo-term \((fa)\) represents the syntactic expression of an application of the function \(f\) to argument \(a\) (thus the result is not computed), a pseudo-term \(\lambda x : t . b\) represents the function \(x \mapsto b\), and a pseudo-term \(\Pi x : t . u\) represents the type of a function, which maps an argument \(x\) of type \(t\) to a result of type \(u\). Some of the constants represent type universes. For instance, the constant \(*\) represents the type universe of propositions.

**Notation 3.1.2** Members of \(V\) are denoted by \(x, y, \ldots\). If \(x\) does not occur in \(u\), then we will write \(t \rightarrow u\) instead of \(\Pi x : t . u\). We will use the convention that application associates to the left and \(\rightarrow\) associates to the right; thus we may write \(Plus \times Zero\) instead of \(((Plus \times Zero))\), and \(Nat \rightarrow Nat \rightarrow Bool\) instead of \(Nat \rightarrow (Nat \rightarrow Bool)\). We will use the abbreviation \(\lambda x, y, z : t . b\) for \(\lambda x : t . \lambda y : t . \lambda z : t . b\), and \(\Pi x, y : t . u\) for \(\Pi x : t . \Pi y : t . u\).

We will give a computational meaning to pseudo terms by \(\beta\)-reduction. Before we can define this notion we need to define ‘substitution’. First we describe the ‘variable convention’, that specifies how names of variables should be chosen in order to prevent name-clashes.

**Definition 3.1.3** 1. The set of bound variables of a pseudo term \(t\), notation \(BV(t)\) is defined as:

(a) \(BV(x) = \emptyset\), for \(x \in V\).

(b) \(BV(C) = \emptyset\), for \(C \in C\).

(c) \(BV(fa) = BV(f) \cup BV(a)\).

(d) \(BV(\lambda x : t . b) = \{x\} \cup BV(t) \cup BV(b)\).

(e) \(BV(\Pi x : t . b) = \{x\} \cup BV(t) \cup BV(b)\).

2. The set of free variables of a pseudo term \(t\), notation \(FV(t)\) is defined as:

(a) \(FV(x) = \{x\}\), for \(x \in V\).

(b) \(FV(C) = \emptyset\), for \(C \in C\).

(c) \(FV(fa) = FV(f) \cup FV(a)\).

(d) \(FV(\lambda x : t . b) = FV(t) \cup (FV(b) \setminus \{x\})\).

(e) \(FV(\Pi x : t . b) = FV(t) \cup (FV(b) \setminus \{x\})\).

If \(x \in FV(t)\) then we say that the variable \(x\) *occurs free* in the pseudo term \(t\).

**Convention 3.1.4** We *identify* terms that can be obtained from each other by renaming bound variables. Names of the bound variables in a pseudo term \(t\) will always be chosen such that for each subterm \(u\) of \(t\) we have \(BV(u) \cap FV(u) = \emptyset\).

**Example 3.1.5** The pseudo terms \(\lambda x : Nat. \lambda y : Nat . y\) and \(\lambda x : Nat. \lambda z : Nat . z\) are identified. The pseudo term \(\lambda x : Nat. \lambda y : Nat . x\) denotes another pseudo term than the previous ones.
Definition 3.1.6 Substituting a pseudo term $t$ for free occurrences of a variable $x$ in a pseudo term $u$, notation $u[x := t]$, is defined as follows:

1. $u[x := t] = \begin{cases} t & \text{if } x = y \\ y & \text{otherwise} \end{cases}$ for $y \in V$.

2. $C[x := t] = C$, for $C \in C$.

3. $(fa)[x := t] = (f[x := t] a[x := t])$.

4. $(\lambda y:u.b)[x := t] = \begin{cases} \lambda y:u.b & \text{if } x = y \\ \lambda y:u[v := t].b[x := t] & \text{otherwise} \end{cases}$.

5. $(\Pi y:u.b)[x := t] = \begin{cases} \Pi y:u.b & \text{if } x = y \\ \Pi y:u[x := t].b[x := t] & \text{otherwise} \end{cases}$.

Example 3.1.7 We have $(\lambda y:\text{Nat.Plus } x y)[x := O] = \lambda y:\text{Nat.Plus } O y$, and $(\lambda y:\text{Nat.Plus } x y)[y := O] = \lambda y:\text{Nat.Plus } x y$.

Following ([32]) we use the notion of 'pseudo term with a hole', that allows us to specify the notion of 'subterm' and to define reduction relations on pseudo terms in a similar way as for terms of Term Rewriting Systems.

Definition 3.1.8 The set $\mathcal{H}$ of pseudo terms with a hole is defined as:

$$\mathcal{H} = [ ] | \mathcal{H} T | T \mathcal{H} | \lambda V: \mathcal{H}.T | \lambda V: \mathcal{T}.\mathcal{H} | \Pi V: \mathcal{H}.T | \Pi V: \mathcal{T}.\mathcal{H}$$

An element in $\mathcal{H}$ is denoted by $C[ ]$. Replacing the $[ ]$ in $C[ ]$ by a pseudo term $t$ is denoted by $C[t]$. We call $t$ a subterm of $C[t]$.

The following definition is needed for specifying which pseudo terms are identified.

Definition 3.1.9 A pseudo term $u$ is the result of renaming a bound variable in $t$, if $t = C[\lambda x:t_1.t_2]$ and $u = C[\lambda y:t_1.t_2[x := y]]$ or $t = C[\Pi x:t_1.t_2]$ and $u = C[\Pi y:t_1.t_2[x := y]]$, where $y \notin \text{FV}(t_2)$.

Definition 3.1.10 The notion of $\beta$-reduction is defined as follows:

$$C[(\lambda v:t.b)a] \rightarrow_\beta C[b[v := a]],$$

for $C[ ] \in \mathcal{H}$.

Notice that because of the variable convention we have $\text{FV}((\lambda v:t.b)a) \cap \text{BV}((\lambda v:t.b)a) = \emptyset$. As $\text{FV}(a) \subseteq \text{FV}((\lambda v:t.b)a)$, and $\text{BV}(b) \subseteq \text{BV}((\lambda v:t.b)a)$, no free variable of $a$ gets bound by a binder of $b$ in the pseudo term $b[v := a]$.

Example 3.1.11 We have $S((\lambda x:\text{Nat.Plus } x x) (S \ O)) \rightarrow_\beta S(\text{Plus } (S \ O) \ (S \ O))$. 
Higher Order Logic

Not all pseudo terms represent mathematical objects. For instance, \( Eq(SO) \) \( \text{Plus} \) is a meaningless expression. In order to be able to filter out these meaningless expressions, we introduce typing rules. Only pseudo terms that can be given a type will be considered as meaningful expressions. Before we introduce the typing rules we need to define the notion of 'pseudo context', in which primitive notions and axioms are collected.

**Definition 3.1.12** 1. A *statement* is a pair \( (a, t) \) of pseudo terms \( a, t \in T \). It will be written as \( a : t \). We call \( a \) the *subject* and \( t \) the *predicate* of \( a : t \).

2. A *variable declaration* is a statement with a variable as subject.

3. A *pseudo context* is a finite ordered sequence of variable declarations. The set of pseudo contexts will be denoted by \( \mathcal{X} \). The set \( FV(\Gamma) \) contains the subjects of the variable declarations of a pseudo context \( \Gamma \in \mathcal{X} \).

A statement \( a : t \) should be interpreted as 'pseudo term \( a \) has type \( t \).

Before we give a definition of the typing rules we select a subset Universes of the set of constants \( C \) that contains the constants that will be used to represent type universes. Usually the set Universes is called Sorts (or \( S \)), but in order to avoid confusion with the notion 'sort' for signatures and TRSs we have chosen a different name. The constant that will be used for the type universe of propositions will be denoted as \( * \), and the constant that represents the type universe of sets will be denoted as \( \emptyset \), and the constant \( \triangle \) is the type of \( \emptyset \). The typing rules for these type universes are determined by a set Axioms, containing pairs of constants, that specifies the type universe of certain constants, and a set Rules, containing triples of constants, that determines the type universe of function types. For instance, the pair \( (\emptyset, \triangle) \in \text{Axioms} \) indicates that \( \emptyset \) has type \( \triangle \). A triple \( (s_1, s_2, s_3) \in \text{Rules} \) indicates that a function type \( \Pi x : d.r \) has type \( s_3 \) if its domain \( d \) has type \( s_1 \) and its range \( r \) has type \( s_2 \).

**Definition 3.1.13** Specification of the constants that will be used as type universes, the axioms for constants, and the rules for function types:

\[
\begin{align*}
\text{Universes} & = \{*, \emptyset, \triangle\} \\
\text{Axioms} & = \{(\emptyset, \emptyset, (\emptyset, \triangle))\} \\
\text{Rules} & = \{(\emptyset, *, *), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset, \emptyset)\}
\end{align*}
\]

**Notation 3.1.14** The set of variables \( V \) is divided into disjoint infinite subsets \( V_s \) for each \( s \in \text{Universes} \). If we want to emphasize that \( x \in V_s \) we also write \( *x \) for \( s \in \text{Universes} \).

**Definition 3.1.15** We will axiomatize the notion

\[ \Gamma \vdash a : t \]

stating that the statement \( a : t \) can be derived from pseudo context \( \Gamma \).
### Logic in Type Theory

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Rule</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(axiom)</td>
<td>$\Gamma \vdash C : s$</td>
<td>if $(C,s) \in \text{Axioms}$</td>
</tr>
<tr>
<td>(var-start)</td>
<td>$\frac{\Gamma \vdash t : s}{\Gamma, x : t \vdash x : t}$</td>
<td>if $x \in V$, $x \notin \text{FV} (\Gamma)$</td>
</tr>
<tr>
<td>(var-weak)</td>
<td>$\frac{\Gamma \vdash a : t \quad \Gamma \vdash u : s}{\Gamma, x : u \vdash a : t}$</td>
<td>if $x \in V$, $x \notin \text{FV} (\Gamma)$</td>
</tr>
<tr>
<td>(product)</td>
<td>$\frac{\Gamma \vdash t_1 : s_1 \quad \Gamma, x : t \vdash u : s_2}{\Gamma \vdash (\Pi x : t. u) : s_3}$</td>
<td>if $(s_1, s_2, s_3) \in \text{Rules}$</td>
</tr>
<tr>
<td>(abstraction)</td>
<td>$\frac{\Gamma, x : t \vdash b : u \quad \Gamma \vdash (\Pi x : t. u) : s}{\Gamma \vdash (\lambda x : t. b) : (\Pi x : t. u)}$</td>
<td></td>
</tr>
<tr>
<td>(application)</td>
<td>$\frac{\Gamma \vdash f : (\Pi x : t. u) \quad \Gamma \vdash a : t}{\Gamma \vdash (fa) : u [x := a]}$</td>
<td></td>
</tr>
<tr>
<td>($\beta$-conversion)</td>
<td>$\frac{\Gamma \vdash a : t \quad \Gamma \vdash t' : s \ \Rightarrow \ \beta t'}{\Gamma \vdash a : t'}$</td>
<td></td>
</tr>
</tbody>
</table>

In the rules above we use the following notation:

$C \in C \quad s, s_1, s_2, s_3 \in \text{Universes} \quad x \in V \quad \Gamma \in X \quad a, b, t, t', u \in T$

Recall from Section 2.1 that $\Rightarrow_{\beta}$, used in the rule ($\beta$-conversion), denotes the equivalence relation induced by $\rightarrow_{\beta}$ (see Definition 3.1.10).

If we can derive $\Gamma \vdash a : t$ then we call $a$ an **inhabitant** of $t$.

**Example 3.1.16** We can derive the statement $(\Pi p : p) : *$ in the empty context as follows:

\[
\begin{align*}
\vdash * : \Box \\
\vdash * : \Box & \quad p : * \vdash p : * \\
\vdash (\Pi p : p) : * & \quad \vdash (\Pi p : p) : * \\
\end{align*}
\]

In this derivation we used the rules (axiom) (twice), and (var-start) and (product).

**Fact 3.1.17** We will now state several properties that hold for the typing relation $\vdash$.

1. If $\Gamma \vdash t : \Box$ then $t = \Box$.

2. We do not have $\top \vdash t : \Box \Rightarrow t = *'$. For instance, we have $\vdash * \rightarrow * : \Box$.

3. Let $s \in \text{Universes}$ and $v \in V$. If $\Gamma \vdash v : t$ then $\Gamma \vdash t : s$.

Now we can combine the notions of pseudo term, pseudo context, $\beta$-reduction, and type assignment to define the language of ‘Higher Order Logic’.

**Definition 3.1.18** Higher Order Logic is the tuple $(T, X, \rightarrow_{\beta}, \vdash)$.
We will use the abbreviation λHOL for Higher Order Logic.

**Definition 3.1.19** We will now define when a pseudo term is 'legal' (meaningful).

1. A pseudo term \( a \in T \) is *legal* if we can derive \( \Gamma \vdash a : t \) or \( \Gamma \vdash t : a \), for some pseudo context \( \Gamma \) and pseudo term \( t \).

2. A pseudo context \( \Gamma \in \mathcal{X} \) is *legal* if we can derive \( \Gamma \vdash a : t \) for some pseudo terms \( a, t \).

We will call a legal pseudo term a *term* and a legal pseudo context a *context*.

### 3.2 Representation of Mathematics

Now that we have formalized \( \lambda \text{HOL} \) we will try to explain how we can represent mathematical objects as its terms, and mathematical theories as its contexts. First we will discuss which terms represent propositions, and how the typing rules represent reasoning.

**Example 3.2.1** By the *propositions-as-types* principle we interpret inhabitants of \( * \) as propositions in \( \lambda \text{HOL} \). For instance, a variable of type \( * \) is interpreted as an arbitrary proposition. If \( p \) and \( q \) have type \( * \) then \( \Pi x : p.q \) represents the proposition '\( p \) logically implies \( q \)'. This is formalized in the (product)-rule for the triple \((* , * , *)\). Recall that we may denote the term \( \Pi x : p.q \) as \( p \rightarrow q \) if \( x \) does not occur in \( q \).

Inhabitants of a proposition are interpreted as proofs of that proposition. For instance, a variable with a proposition as type is interpreted as an assumption of that proposition. As we represent logical implication by a function type, we represent a proof of '\( p \) implies \( q \)' as a function that maps a proof of \( p \) to a proof of \( q \). This is formalized in the (abstraction)-rule. The cut rule is formalized by the (application)-rule. Thus if \( f \) represents a proof of '\( p \) implies \( q \)', and \( a \) represents a proof of \( p \) then \((f \ a)\) represents a proof of \( q \).

**Example 3.2.2** Our formal language is called *Higher Order Logic*, because we can construct propositions by quantification over a proposition variable. This is formalized in the (product)-rule for the triple \((\Box, *, *)\). For instance, the proposition \( \Pi x : p \) of Example 3.1.16 represents the proposition 'all propositions are valid' and has no inhabitants (in the empty context). The true proposition '\( p \) implies \( p \)' is represented in \( \lambda \text{HOL} \) by \( \Pi x : p.p \rightarrow p \). This trivial proposition has a proof \( \lambda p.x.\lambda x.p.x \), as we can derive \( \epsilon \Gamma (\lambda p.x.\lambda x.p.x) : (\Pi x : p.p \rightarrow p) \).

We can represent the logical connectives \( \neg, \land, \lor \) as:

| \neg p       | \rightarrow (\Pi r : r \Gamma) |
| p \land q   | \Pi r : [(p \rightarrow q \rightarrow r) \rightarrow r] |
| p \lor q    | \Pi r : [(p \rightarrow r) \rightarrow (q \rightarrow r) \rightarrow r] |

Notice that the usual logical rules are valid for the notions defined above. For instance, we can prove for arbitrary propositions \( p \) and \( q \) that \( p \) follows from the assumption \( p \land q \). This is formalized by the derivation in Figure 3.1. Thus we can construct a term of type \( p \) in a context that contains a variable declaration with \( p \land q \) as predicate.
\[
\Gamma \vdash i : p \land q \\
\Gamma \vdash p : * \\
\Gamma \vdash i \in p : (p \rightarrow q \rightarrow p) \\
\Gamma \vdash \lambda x : p.(\lambda y : q. x) : (q \rightarrow p) \\
\Gamma \vdash (p \rightarrow q \rightarrow p) : * \\
\Gamma \vdash (\lambda x : p. \lambda y : q. x) : (p \rightarrow q \rightarrow p) \\
\Gamma \vdash (i \in p. (\lambda x : p. \lambda y : q. x)) : p
\]

Figure 3.1: Derivation of \( p \) from the assumption \( p \land q \) in context \( \Gamma = p : *, q : *, i : p \land q \)

We interpret inhabitants of \( \Box \) as sets, and inhabitants of sets are elements of that set.

**Example 3.2.3** We can represent predicate logic as follows. If we have an element \( x \) of a set \( T \), and a proposition \( p_x \) in which \( x \) occurs, then \( \Pi x : T. p_x \) represents \( \forall x \in T. p_x \).

We can represent existential quantification and equality on \( T \) as:

\[
\begin{align*}
\forall x \in T. p_x & = \Pi x : T. p_x \\
\exists x \in T. p_x & = \Pi r : (\Pi x : T. p_x \rightarrow r) \rightarrow r \\
x =_T y & = \Pi x : (T \rightarrow *). (p x) \rightarrow (p y)
\end{align*}
\]

The relation \( =_T \) is called *Leibniz equality* on \( T \). If \( x =_T y \) then all predicates on \( T \) that hold for \( x \) also hold for \( y \). The usual logical rules for \( \forall \), \( \exists \) and \( =_T \) are valid. For instance, we can prove the symmetry of \( =_T \) by the following derivable statement:

\[
\begin{align*}
T : \Box \vdash & \lambda x : T. \lambda y : T. \lambda i : (x =_T y). (i \in (\lambda z : T. (z =_T x)). (\lambda p : T \rightarrow *. \lambda j : (p x). j)) : \\
& (\forall x, y \in T. (x =_T y) \rightarrow (y =_T x))
\end{align*}
\]

The trick of this proof is that the predicate \( \lambda z : T. z =_T x \), that holds for \( x \), also holds for \( y \) if \( x =_T y \). The crucial step in the derivation of this statement is the use of the \( (\beta\text{-conversion}) \) rule that allows us to convert the type \( ((\lambda x : T. (z =_T x)) x) \rightarrow ((\lambda x : T. (z =_T x)) y) \) of the subterm \( i \) \( (\lambda x : T. (z =_T x)) \) to the type \( (x =_T x) \rightarrow (y =_T x) \) in the context \( T : \Box, x : T, y : T, i : (x =_T y) \).

Using the propositions-as-types principle we can represent a mathematical theory by a context containing representations of the primitive notions and axioms of this theory.

**Example 3.2.4** The theory of monoids can be represented by:

\[
\Delta_{\text{Mon}} =
\begin{align*}
M & : \Box, \\
op & : M \rightarrow M \rightarrow M, \\
e & : M, \\
\text{assoc} & : \forall x, y, z \in M. \text{op} x (\text{op} y z) =_M \text{op} (\text{op} x y) z, \\
\text{neutL} & : \forall x \in M. \text{op} e x =_M x, \\
\text{neutR} & : \forall x \in M. \text{op} x e =_M x
\end{align*}
\]

We can represent a proof of a proposition in a mathematical theory by an inhabitant of the representation of this proposition in the context for this theory.
Example 3.2.5 For instance, the property that the neutral element $e$ is unique can be represented in $\Delta_{\text{Mon}}$ by:

$$\forall f \in M.(\forall x \in M.\text{op } f x =_M x) \rightarrow (\forall x \in M.\text{op } x f =_M x) \rightarrow (f =_M e)$$

It is inhabited by the term:

$$\lambda f : M. \lambda x (\forall x \in M.\text{op } f x =_M x). \lambda y : (\forall y \in M.\text{op } x f =_M x).$$

$$=_{\text{trans}} f (\text{op } f e) (\text{=}_{\text{sym}} (\text{op } f e) f (\text{neutR } f)) (l e)$$

Where $=_{\text{sym}}$ is an inhabitant of $\forall x, y \in M. (x =_M y) \rightarrow (y =_M x)$, and $=_{\text{trans}}$ is an inhabitant of $\forall x, y, z \in M. (x =_M y) \rightarrow (y =_M z) \rightarrow (x =_M z)$.

The examples illustrate that we can formalize mathematics in Higher Order Logic using the axiomatic approach. In particular we can formalize first order predicate logic in it. As ZF ([35],[15]) is axiomatized in this logic, we can represent this version of Cantor's set theory in $\lambda$HOL.

### 3.3 Automated Verification

In the previous section we have illustrated how we can formalize mathematics in Higher Order Logic. In this section we will present the fundamental properties of Higher Order Logic that make this type system suited for automated verification. Some of these properties are inherited from a more general framework, in which $\lambda$HOL can be described.

The typing rules presented in Definition 3.1.15 are parameterized by a set Universes, a set Axioms and a set Rules. We can define other type systems by choosing other sets of Universes, Axioms and Rules.

Example 3.3.1 The Calculus of Constructions ([12]) can be specified by choosing:

- Universes = \{*, \Box\}
- Axioms = \{(*, \Box)\}
- Rules = \{(*, *, *, (\Box, *, *), (\Box, \Box, \Box), (\Box, \Box, \Box, \Box, \Box))\}

Each type system that can be defined in this way is called a Pure Type System (PTS). For a general introduction to PTSs we refer to [4]. We will now describe several fundamental properties of Pure Type Systems.

Theorem 3.3.2 The relation $\rightarrow_{\beta}$ is confluent.

Proof

To be found in [4].

A general property that holds for all Pure Type Systems is that reducing a legal term does not change its type. This property is called subject reduction. More precisely:
Theorem 3.3.3 If $\Gamma \vdash a : t$ and $a \rightarrow_\beta a'$ then $\Gamma \vdash a' : t$.

Proof
See [4]. $\square$

Remark 3.3.4 Because of this property, each PTS has a corresponding ARS consisting of the set of legal terms and reduction relation $\rightarrow_\beta$. Via the associated ARS, all notions defined for Abstract Reduction Systems carry over to PTSs.

Theorem 3.3.5 $\lambda$HOL is strongly normalizing.

Proof
The Calculus of Constructions ($\lambda$CC) is strongly normalizing (see [4]). We can embed $\lambda$HOL in $\lambda$CC via a reduction preserving transformation (see [16]). This transformation is done via a mapping $E$ that is defined as follows. Let $t \in T$. Then $E(t)$ denotes the pseudo term obtained from $t$ by replacing each $\Box$ with $\ast$, and each $\Delta$ with $\Box$. Let $\Gamma \in \lambda$ be a pseudo context. Then $E(\Gamma)$ denotes the pseudo context obtained from $\Gamma$ by replacing each predicate $p$ by $E(p)$. If $\Gamma \vdash a : t$ in $\lambda$HOL, then we have $E(\Gamma) \vdash E(a) : t'$ in $\lambda$CC, where $t' = \{ \begin{array}{ll} \Box & (\text{if } a = t_1 \rightarrow \ldots \rightarrow t_n \rightarrow * \ (0 \leq n)) \\ E(t) & (\text{otherwise}) \end{array}$. As $\lambda$CC is strongly normalizing, the system $\lambda$HOL must be strongly normalizing. $\square$

Definition 3.3.6 The following notions are important for (Pure) Type Systems:

1. Type checking is the problem: Given $\Gamma, a, t$. Does $\Gamma \vdash a : t$ hold?

2. Typability is the problem: Given $\Gamma, a$. Is there a $t$ such that $\Gamma \vdash a : t$ holds?

3. Inhabitation is the problem: Given $\Gamma, t$. Is there an $a$ such that $\Gamma \vdash a : t$ holds?

Proposition 3.3.7 Type checking and typability are decidable for (weakly or strongly) normalizing Pure Type Systems.

Proof
See [4]. $\square$

Remark 3.3.8 A consequence of this proposition and Theorem 3.3.5 is that type checking and typability are decidable for $\lambda$HOL.

By the propositions-as-types principle, verifying the correctness of a formal proof $f$ of a proposition $p$ in a mathematical theory $t$ corresponds to type checking $[t] \vdash [f] : [p]$, where $[t], [f], [p]$ denote the respective representations of $t, f, p$ in $\lambda$HOL. The provability of a property $p$ (in $t$) corresponds to the inhabitation of $[p]$ (in $[t]$). As type checking is decidable in $\lambda$HOL, this type system is suited for automated verification.
Chapter 4

Computations in Formal Proofs

Since the availability of computers, the role of computations role in mathematics has increased. The power of algorithms for solving mathematical problems is very high. This is illustrated by Computer Algebra Systems (CASs) such as Axiom [22], Maple [7], Mathematica [34], and Reduce [19] that can do large computations efficiently. In general Computer Algebra Systems can solve problems of the form: given $x$, find $y$ such that $P(x, y)$ holds. Thus a CAS implements an algorithm $F_P$, such that $F_P(x)$ computes some $y$ for which $P(x, y)$ holds.

Clearly, a good formal mathematical language should provide constructs for specifying computations in order to be able to represent such algorithms. Moreover the computational meaning of functions should be available in formal proofs, because it allows us to give formal correctness proofs of algorithms. Notice that one cannot give formal correctness proofs of algorithms in the language of a CAS, because a CAS does not have a formal representation for proofs. Computations can play a role in the formalization of mathematics in two ways.

The first option is to use the computational power that is provided by a formal mathematical language. We will call computations that are done in a formal mathematical language ‘internal computations’. We will discuss several formalisms for specifying and doing computations in a formal mathematical language, and how these formalisms provide a way to increase the level of abstraction of formal proofs.

The other option is to use the results of computations of mathematical tools in the development of formal proofs. These algorithms may be written in any language, because the correctness of the actual calculations does not have to be proved formally: only the formal proofs that are based on these external computations are verified. A good example of the use of external computations are tactics of proof development systems, that are algorithms that assist the interactive construction of formal proofs. The reasoning steps provided by tactics have a higher level of abstraction than a rule of its formal language, because a tactic can invoke several calls to formal rules. Another example is the use of Computer Algebra Systems for computing witnesses that can be used to give short formal proofs of certain properties. We will describe several mathematical problems for which this approach can be used.
4.1 External Computations

Without any restriction one can use external tools that produce formal proofs for certain problems, as long as the proof checker verifies the results. Not the algorithm of the tool, but the result of its application to some input is verified by the proof checker. The reliability of a formally verified proof only depends on the reliability of the proof checker and not on the way the proof is constructed. A proof checker does not ‘know’ whether a formal proof is the result of human thinking or is generated by an algorithm.

First we discuss tactics that are algorithms that assist the interactive construction of formal proofs in a proof checker. Then we describe how we can solve certain mathematical problems using results of computations in a CAS.

Tactics

In general the rules of (an implementation of) a formal system are on a low level of abstraction in order to keep the size of the implementation small. Thus developing a mathematical theory directly in a proof checker is not convenient, since one informal reasoning step is equivalent to a large number of formal reasoning steps. Moreover the construction of a formal proof in a proof checker is a bottom-up process, whereas developing a proof is merely a top-down process. To support the interactive construction of a mathematical theory in a proof checker one normally builds a proof development system on top of it. In a proof development system one can try to prove a proposition, called the goal, using tactics that are algorithms that can automate parts of a proof. Either a tactic transforms a goal into a number of new goals or it fails. If the tactic succeeds and the new goals are proved (for instance when there are no new goals) the proof development system constructs a proof-object of the original goal and has it verified by the proof checker.

Example 4.1.1 A commonly used tactic is the Intros tactic which removes all quantifications of the goal and puts the related variable declarations in the context. If the new goal is solved, the (abstraction) rule (see Definition 3.1.15) is used repeatedly to obtain a proof of the original goal. For instance, consider the goal of proving the true proposition in \( \text{HOL} \) (see Example 3.2.2):

\[ \Gamma \vdash (\Pi x. p \to p) \]

The Intros tactic transforms it into a goal

\[ p : *, h : p \vdash ? : p \]

Clearly the assumption \( h \) solves this new goal. Now the proof development system applies the abstraction rule twice on this term to construct a proof of the original goal:

\[ \Gamma \vdash (\lambda p x. \lambda h p h) : (\Pi x. p \to p) \]
A tactic does not only transform a goal into a number of new goals, it also provides a function that constructs a proof of the original goal from proofs of the new goals using the rules of the proof checker. Thus if all new goals are solved (proved) the proof development system applies this function to their proofs and obtains a proof of the original goal.

The tactics provided by a general proof development system provide a higher level of reasoning for using the rules of a formal system. When doing mathematics in a particular field one can have certain proof situations that cannot be handled automatically by the standard tactics. If these situations occur frequently, it would be nice to have a tactic that can fill in the needed proof-objects. For instance, in formal languages with hardly any computational power calculations can only be simulated by equational reasoning. Thus large calculations require large equational proofs. Therefore we will describe a tactic for partially automating equational reasoning by term rewriting.

**Term rewriting with tracing**

For some equational theories it is possible to define complete (that is, strongly normalizing and confluent) Term Rewriting Systems (see Section 2.2) that can decide equality for terms in such theories (by comparing the normal forms).

**Example 4.1.2** Equality in the theory of monoids (see Example 3.2.4) can be decided by the following complete TRS:

$$
\begin{array}{ll}
\text{sort} & M \\
\text{func} & e :: M \\
 & \cdot :: M \times M \rightarrow M \\
\text{rule} & e \cdot x \rightarrow x \\
 & x \cdot e \rightarrow x \\
 & x \cdot (y \cdot z) \rightarrow (x \cdot y) \cdot z
\end{array}
$$

For instance, \((e \cdot x) \cdot (y \cdot z)\) and \((x \cdot y) \cdot (z \cdot e)\) have the same normal form \((x \cdot y) \cdot z\).

We will describe a tactic that tries to solve equational problems in models of first order equational theories by rewriting. The idea is to construct a proof in a model of an equational theory from a reduction sequence in a related TRS. The natural numbers with neutral element 0 and binary operator + are a monoid.

**Example 4.1.3** We can axiomatize the natural numbers by the following context:

$$
\Delta_N = \begin{align*}
N : & \quad \square, \\
0 : & \quad N, \\
S : & \quad N \rightarrow N, \\
S\text{not}0 : & \quad \forall x \in N. \neg (S(x) =_N 0), \\
S\text{injective} : & \quad \forall x, y \in N. S(x) =_N S(y) \rightarrow x =_N y, \\
in\text{duction} : & \quad \forall P \in N \rightarrow \ast. (P0) \rightarrow (\forall n \in N. (Pn) \rightarrow (P(Sn))) \rightarrow \forall n \in N. Pn
\end{align*}
$$
Notice that \( \equiv_N \) is Leibniz equality on \( N \) (see Example 3.2.3).

We can represent the binary operator \( + \) as follows.

**Example 4.1.4** We extend the context of the previous example with an axiomatization of the addition:

\[
\Delta_{\text{plus}} = \begin{align*}
\text{plus} : & \quad N \to N \to N, \\
\text{plus0} : & \quad \forall y \in N.\text{plus 0} y =_N y, \\
\text{plusS} : & \quad \forall x, y \in N.\text{plus} (Sx) y =_N S(\text{plus} x y)
\end{align*}
\]

The type \( N \) with binary operator \( \text{plus} \) and neutral element \( 0 \) is a monoid.

In the TRS of Example 4.1.2 we have the following reduction sequence:

\((e \cdot x) \cdot (y \cdot z) \to x \cdot (y \cdot z) \to (x \cdot y) \cdot z\). This reduction sequence can be interpreted as the equation \( \forall x, y, z \in N.\text{plus} (\text{plus} 0 x) (\text{plus} y z) =_N \text{plus} (\text{plus} x y) z \). For constructing a proof of the interpreted equation of a reduction sequence in a model we need proofs of the fact that all rewrite rules (interpreted as universally quantified equations) are valid in the model.

**Example 4.1.5** To show that the natural numbers are a model of \( M \) we need proofs for the equations obtained from the rules for \( M \) in Example 4.1.2. Thus we must prove:

\[
\begin{align*}
\forall x \in N.\text{plus} 0 x & =_N x & (\text{plus0}) \\
\forall x \in N.\text{plus} x 0 & =_N x & (\text{0identR}) \\
\forall x, y, z \in N.\text{plus} x (\text{plus} y z) & =_N \text{plus} (\text{plus} x y) z & (\text{plus-assoc})
\end{align*}
\]

The names behind these equations refer to formal proofs of these equations. The first equation is proved by the axiom \( \text{plus0} \); the other equations require more complicated proofs. But we will not give these proofs.

Furthermore we need proofs of the compatibility of the equivalence relation of the model with respect to all operators, because rewrite rules may also be applied on subterms.

**Example 4.1.6** For the natural numbers we need a proof of

\[
\forall x_1, y_1, x_2, y_2 \in N. (x_1 =_N y_1) \to (x_2 =_N y_2) \to (\text{plus} x_1 x_2 =_N \text{plus} y_1 y_2) \quad (\text{plus-resp})
\]

A trace of a reduction sequence \( t_1 \to t_2 \to \ldots \to t_n \) specifies which rules and substitutions are used in each reduction step. By inspecting the trace we can construct a formal proof of the equation, that is associated with its reduction sequence, based on the proofs of the equations, that are associated with the rewrite rules, and the compatibility proofs of the function symbols. In the next example we will show how.

**Example 4.1.7** The first reduction step \((e \cdot x) \cdot (y \cdot z) \to x \cdot (y \cdot z)\) uses the first rule of \( M \). Using \( \text{plus0} \) and \( \text{plus-resp} \) we obtain a proof of
\[ \text{plus} \ (\text{plus} \ 0 \ x) \ (\text{plus} \ y \ z) =_N \text{plus} \ x \ (\text{plus} \ y \ z) \]

In a similar way the reduction step \( x \cdot (y \cdot z) \rightarrow (x \cdot y) \cdot z \) induces a proof of

\[ \text{plus} \ x \ (\text{plus} \ y \ z) =_N \text{plus} \ (\text{plus} \ x \ y) \ z \]

Using the transitivity of \( =_N \) we can link the obtained proofs to construct a proof of

\[ \text{plus} \ (\text{plus} \ 0 \ x) \ (\text{plus} \ y \ z) =_N \text{plus} \ (\text{plus} \ x \ y) \ z \]

In a similar way we can construct a proof of

\[ \text{plus} \ (\text{plus} \ x \ y) \ (\text{plus} \ 0 \ z) =_N \text{plus} \ (\text{plus} \ x \ y) \ z \]

We can combine the obtained proofs to construct the proof

\[ =_N \text{-trans} \ (\text{plus} \ (\text{plus} \ 0 \ x) \ (\text{plus} \ y \ z)) \ (\text{plus} \ (\text{plus} \ x \ y) \ (\text{plus} \ z \ 0)) \]
\[ (=_N \text{-trans} \ (\text{plus} \ (\text{plus} \ 0 \ x) \ (\text{plus} \ y \ z)) \ (\text{plus} \ x \ (\text{plus} \ y \ z)) \ (\text{plus} \ (\text{plus} \ x \ y) \ z) \ (\text{plus}-\text{resp}) \]
\[ \text{ (plus} \ 0 \ x) \ x \ (\text{plus} \ y \ z) \ (\text{plus} \ y \ z)) \ (\text{plus} \ 0 \ x) \ (\text{plus}-\text{resp}) \] \( (\text{plus} \ x \ y) \ (\text{plus} \ x \ z) \) \( (\text{plus}-\text{resp}) \)
\[ \text{ (plus} \ (\text{plus} \ x \ y) \ (\text{plus} \ z \ 0)) \ (\text{plus} \ (\text{plus} \ x \ y) \ (\text{plus} \ x \ y) \ z) \ (\text{plus}-\text{resp}) \]
\[ \text{ (plus} \ (\text{plus} \ x \ y) \ (\text{plus} \ y \ z)) \ (\text{plus}-\text{resp}) \ (\text{plus} \ x \ y) \ (\text{plus} \ x \ y) \]
\[ \text{ (plus} \ x \ y) \ (\text{plus} \ x \ y) \ z) \]

of the equality

\[ \text{plus} \ (\text{plus} \ 0 \ x) \ (\text{plus} \ y \ z) =_N \text{plus} \ (\text{plus} \ x \ y) \ (\text{plus} \ z \ 0) \]

Using this method we can automatically construct a proof of the equality of the interpretations of two terms in a model of the TRS, if these terms have the same normal form. We just simulate a reduction sequence

of the external TRS by equational reasoning in the model. The proof obtained from the trace (that indicates the used rules and substitutions) of a reduction sequence is verified by the proof checker. A disadvantage of this method is that each generated formal proof has a low level of abstraction (only basic equational reasoning rules are used) and its length is proportionate to the length of the reduction sequence used to construct it.

**Efficiency**

Inspecting a term \( t_1 \) to determine whether it is a redex or not can be done in time \( O(|t_1|) \) (if testing syntactic equality of function symbols costs a constant amount of time); if \( t_1 \) is a redex then its reduct \( t_2 \) can be computed in time \( O(|t_2|) \) (if the costs for substitution are linear in the number of symbols of the obtained term). Thus the computation of a reduction step \( t_1 \rightarrow t_2 \) can be done in time \( O(|t_1| + |t_2|) \). The time needed to compute a reduction sequence is linear in the number of symbols of the terms of that reduction sequence.

The verification costs of an equational proof obtained from a reduction step \( t_1 \rightarrow t_2 \) are \( O(|t_1|^2 + |t_2|^2) \). We will show how these costs are obtained. Let \( f_1, \ldots, f_n \) be unary function symbols. The equational proof obtained from a reduction step

\[ f_1(\ldots(f_n(t^r))) \rightarrow f_1(\ldots(f_n(r^s))) \]

requires a proof of \( l^r = r^s \), which has length \( O(|l^r| + |r^s|) \),
and $n$ compatibility proofs of the form "if $f_i(f_{i+1}(\ldots(f_n(t^n)))) = f_i(f_{i+1}(\ldots((f_n(r^n))))))$ then $f_i(f_{i+1}(\ldots(f_n(t^n)))) = f_i(f_{i+1}(\ldots((f_n(r^n))))))$. The $i$th compatibility proof adds length $1 + |t^n| + (n - i) + |r^n| + (n - i)$ to the proof. Thus the length of the compatibility part of the proof is $\sum_{i=1}^{n} |t^n| + |r^n| + 2 \ast (n - i) + 1 = n \ast (|t^n| + |r^n| + n)$. Therefore the total length of the proof has length $O((\sum_{i=1}^{n-1} |t_i|)^2)$. The lengths of these proofs may be better in special cases, e.g. if $t_1$ is the contracted redex occurrence ($n = 0$). Thus verification of an equational proof obtained from a reduction sequence $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n$ costs $O(\sum_{i=1}^{n} |t_i|)$ (the verification of the applications of the transitivity rule costs $O(\sum_{i=1}^{n} |t_i|))$. The verification time of a proof obtained from a reduction sequence is polynomial (quadratic) in the number of symbols of the terms of that reduction sequence. Thus computing the reduction sequence costs less time than verifying the equational proof obtained from it.

Although verifying equational proofs obtained by this method is time consuming, the use of external rewriting has the advantage that it allows us to experiment with rewrite rules, strategies and different forms of rewriting (conditional rewriting, priority rewriting) to construct proofs of equations in models of the TRS, without bothering about the formal correctness.

**Example 4.1.8** A rule for commutativity such as $x \cdot y \rightarrow y \cdot x$ can cause infinite reductions. This can be prevented by imposing a restriction on the use of this rule. For instance, one can define a suitable order $< on terms and allow the use of the rule for rewriting $t + u$ only if $t < u$.

The example with the monoid illustrates how term rewriting with tracing works. We will now shortly describe a more complicated example.

**Example 4.1.9** Let $R$ be a ring. We define a binary operator $[ , ]$ on $R$ as follows: $[x, y] = x \cdot y - y \cdot x$. Now, the following equality holds: $[x, [y, z]] + [y, [x, z]] + [z, [x, y]] = 0$. This equality is called the Jacobi identity.

We defined a conditional TRS (consisting of 19 rules) that can decide equality of elements of a ring (using the trick of the last example to handle commutativity of $+)$. Using our tactic we automatically obtained a large LEGO proof (45 kilobytes) of the Jacobi identity based on a reduction sequence of 44 steps.

**CAS as oracle**

For some problems $P$, it is much harder to find a $y$ such that $P(x, y)$ holds for a given $x$ than to verify $P(x, y)$ for a given $x$ and $y$. For these problems one can use a CAS as an oracle (to find $y$), and verify the obtained result ($P(x, y)$) in a proof checker. For this approach the way the result is obtained is irrelevant for its verification in the proof checker. Thus the proof is not a simulation of the computation of the result.

**Example 4.1.10** For instance the problem whether a number $n$ is composite or not is much harder to solve than to verify whether $n = n_1 \cdot n_2$ holds for given numbers $n_1, n_2$. 

In Section 7.3 we will show that we can use a CAS to obtain a relatively short formal proof of the primality of a number. Another example of the use of a CAS as a guide for a theorem prover is given in [18] where a CAS is used to compute the integral of a polynomial over the reals and this result is verified in a theorem prover by computing its derivative. In the last example the computed $y$ is the desired result, and in the other examples mentioned above $y$ serves as a witness to prove some property of $x$. In the next example both result and witnesses are computed.

**Example 4.1.11** Assume we want to find the greatest common divisor of two numbers $m$ and $n$. Using a CAS we can compute $\gcd(m, n) = g$. Verifying that $g$ is a divisor of $m$ and $n$ is easy, if we are given $c$ and $d$ such that $c \cdot g = m$ and $d \cdot g = n$. We can verify that a divisor $g$ of $m$ and $n$ is their greatest common divisor, if we are given $a$ and $b$ such that $a \cdot m + b \cdot n = g$. These witnesses ($a$, $b$, $c$, and $d$) can be easily computed by a CAS. For instance, $\gcd(36, 56) = 4$ can be certified by testing $2 \cdot 56 + -3 \cdot 36 = 4$ (, and $36 = 9 \cdot 4$ and $56 = 14 \cdot 4$).

We can use this method for obtaining the greatest common divisor in a proof checker if we formally prove its correctness once. Thus we must prove that testing these equalities is sufficient to prove that some number is the greatest common divisor of two numbers. After the method has been formally justified, we can obtain the greatest common divisor of any two numbers by computing it and its witnesses in the CAS and verifying their equalities in the proof checker.

### 4.2 Inductive Types

The type system λHOL of the Section 3.1 does not have much computational power. Therefore we can only axiomatize functions, but we cannot define executable procedures. In the previous section we have seen that we can simulate external computations by deductions in the formal system. The disadvantage of this approach is that large low level proofs are produced. If we could do the computations in the formal system we could obtain shorter proofs, as the length of an internal computation does not influence the length of the proof. Moreover these proofs would have a higher level of abstraction, as the used method is part of the proof, and not its result for a special case. Thus it can be useful to do computational steps inside the formal system. In order to achieve this we can extend this type system with so-called inductive types. Roughly speaking an inductive type consists of a number...
of constructors and a recursor for defining functions by structural induction (primitive recursion). In this section we will present an extension of λHOL with a simplified version of inductive types. For this purpose we must extend our syntax of pseudo terms with extra constructs \textbf{Ind}, \textbf{Constr}, \textbf{Elim} for representing inductive types, their constructors and their elimination principles, respectively. For more information on inductive types we refer to [30].

**Definition 4.2.1** The set of inductive pseudo terms \( T_i \) is defined as follows:

\[
T_i = V \ | \ C \ | \ (T_i \ T_i) \ | \ \lambda V : T_i . T_i \ | \ \Pi V : T_i . T_i \ | \ \text{Ind}(V : T_i \{ T_i \}) \ | \ \text{Constr}(\mathbb{N}, T_i) \ | \ \text{Elim}(T_i, T_i, T_i)\{ T_i \}
\]

Recall that \( T_i \) denotes the set of sequences of elements in \( T_i \).

A pseudo term of the form \text{Ind}(x : u \{ t_1, \ldots, t_n \}) represents a type with \( n \) constructors; the type of the \( i \)th constructor is specified by \( t_i \), for \( 1 \leq i \leq n \); the type (universe) of this pseudo term is \( u \). In our restricted version only \( \square \) will be allowed as type universe of pseudo terms of this form, as allowing to represent propositions by inductive types (of type \( * \)) would be in conflict with the idea of ‘irrelevance of (the structure of) proofs’.

**Notation 4.2.2** In this section \( I_0 \) denotes the pseudo term \text{Ind}(x : \square \{ t_1, \ldots, t_n \}).

The pseudo term \text{Constr}(i, I_0) represents the \( i \)th constructor of \( I_0 \), for \( 1 \leq i \leq n \). A constructor does not have a computational meaning. This properties enables the definition of executable algorithms, that operate on inhabitants of \( I_0 \), by specifying the result for pseudo terms of the form \text{Constr}(i, I_0) \bar{a}. A pseudo term of the form \text{Elim}(I_0, Q, b \{ f_1, \ldots, f_n \}) represents an algorithm that operates on a pseudo term \( b \) that should have type \( I_0 \), and yields a pseudo term of type \( Q \). Each pseudo term \( f_i \) specifies the result for pseudo terms of the form \text{Constr}(i, I_0) d_1 \ldots d_m, for \( 1 \leq i \leq n \).

Before we can describe these new constructs in more detail, we must adapt several notions that are defined for pseudo terms in \( T \) for pseudo terms in \( T_i \).

**Definition 4.2.3** We define bound and free variables and substitution for pseudo terms \(( \in T_i \) as in definition 3.1.3 and 3.1.6 by adding cases for the new constructs.

1. The new rules for the set of bound variables of a pseudo term are:
   \[
   \begin{align*}
   (a) & \quad BV(\text{Ind}(x : u \{ t_1, \ldots, t_n \})) = \{ x \} \cup BV(u) \cup BV(t_1) \cup \ldots \cup BV(t_n). \\
   (b) & \quad BV(\text{Constr}(i, t)) = BV(t). \\
   (c) & \quad BV(\text{Elim}(I, Q, a \{ t_1, \ldots, t_n \})) = BV(I) \cup BV(Q) \cup BV(a) \cup BV(t_1) \cup \ldots \cup BV(t_n).
   \end{align*}
   \]

2. The new rules for the set of free variables of a pseudo term are:
   \[
   \begin{align*}
   (a) & \quad FV(\text{Ind}(x : u \{ t_1, \ldots, t_n \})) = FV(u) \cup (FV(t_1) \cup \ldots \cup FV(t_n)) \setminus \{ x \}). \\
   (b) & \quad FV(\text{Constr}(i, t)) = FV(t). \\
   (c) & \quad FV(\text{Elim}(I, Q, a \{ t_1, \ldots, t_n \})) = FV(I) \cup FV(Q) \cup FV(a) \cup FV(t_1) \cup \ldots \cup (t_n).
   \end{align*}
   \]
3. The new rules for substitution are as follows:

\[
\begin{align*}
(a) \quad \text{Ind}(y:u)\{t_1, \ldots, t_n\}[x := b] &= \\
& \left\{ \begin{array}{ll}
\text{Ind}(y:u)\{t_1, \ldots, t_n\} & \text{(if } x = y) \\
\text{Ind}(y:u)[x := b]\{t_1[x := b], \ldots, t_n[x := b]\} & \text{(otherwise)}
\end{array} \right. \\
(b) \quad \text{Constr} (i, t)[x := b] &= \text{Constr} (i, t[x := b]) \\
(c) \quad \text{Elim}(I, Q, a)\{t_1, \ldots, t_n\}[x := b] &= \\
& \text{Elim}(I[x := b], Q[x := b], a[x := b])\{t_1[x := b], \ldots, t_n[x := b]\}.
\end{align*}
\]

For being able to define reduction inside a pseudo term we introduce the notion of 'pseudo term with a hole'.

**Definition 4.2.4** We define the set \( \mathcal{H}_i \) of pseudo terms with a hole as follows:

\[ \mathcal{H}_i = [ ] \mid \mathcal{H}_i \mid T \mid \mathcal{T}_i \mathcal{H}_i \mid \lambda V: \mathcal{H}_i.T \mid \Pi V: \mathcal{H}_i.T \mid \Pi \mathcal{H}_i.T \mid \text{Ind}(V: \mathcal{H}_i)\{ \mathcal{T}_i \} \mid \text{Ind}(V: \mathcal{T}_i)\{ \mathcal{T}_i \mathcal{H}_i \} \mid \text{Ind}(V: \mathcal{H}_i, \mathcal{T}_i)\{ \mathcal{T}_i \} \mid \text{Constr}(N, \mathcal{H}_i) \mid \text{Elim}(\mathcal{H}_i, \mathcal{T}_i, \mathcal{T}_i)\{ \mathcal{T}_i \} \mid \text{Elim}(\mathcal{T}_i, \mathcal{H}_i, \mathcal{T}_i)\{ \mathcal{T}_i \mathcal{H}_i \mathcal{T}_i \} \]

**Definition 4.2.5** The notion of \( \beta \)-reduction for pseudo terms \( (\in \mathcal{T}_i) \) is defined as follows:

\[ C[(\lambda u. t) a] \rightarrow_\beta C[b[v := a]], \text{ for } C[ ] \in \mathcal{H}_i. \]

Before we describe the reduction relation induced by the recursor (Elim), and the typing rules for the new constructs, we specify how the natural numbers can be represented using Ind.

**Example 4.2.6** We can represent the natural numbers by the following pseudo term:

\[ \text{Nat} = \text{Ind}(x: \Box)\{x, x \rightarrow x\}. \]

This pseudo term has two constructors that we will give a name to make their intended meaning clear. The constant zero is represented by \( O = \text{Constr}(1, \text{Nat}) \), and the successor function by \( S = \text{Constr}(2, \text{Nat}) \).

Recall the axiomatic definition of the natural numbers by \( \Delta_N \) in Example 4.1.3. If we interpret \( \text{Nat} \) as \( N \), \( O \) as 0, and \( S \) as \( S \), then the axioms in \( \Delta_N \) specify the intended meaning of \( \text{Nat} \). In Example 4.2.6 the definition of \( \text{Nat} \) contains a sequence of pseudo terms \( x, x \rightarrow x \), that specifies the types of the constructors of \( \text{Nat} \). The variable \( x \) serves as a dummy for \( \text{Nat} \). Thus the intended meaning of this definition is that \( O \) has type \( \text{Nat} \) and \( S \) has type \( \text{Nat} \rightarrow \text{Nat} \). The correct syntax for the pseudo terms \( t_1, \ldots, t_n \) that occur in \( \text{Ind}(x: \Box)\{t_1, \ldots, t_n\} \) is specified by the notion 'type of constructor in \( x \).

**Definition 4.2.7** Let \( x \in V_\Delta \). A pseudo term \( t \) is a (simple) type of constructor in \( x \), notation \( \text{constr}_x(t) \), if:

1. \( t = x \); or
2. \( t = u \rightarrow c \), and \( \text{constr}_x(c) \), and either
   
   \( (a) \quad u = x, \) or
(b) $\text{FV}(u) = \emptyset$, $*$ $\not\in u$.

The last condition prevents that a constructor can have a proposition as argument and is needed in Theorem 4.2.27. The other criteria are restrictive simplifications of the criteria in the original definition of the notion 'type of constructor' (see [30]). The only reason for this simplification is that it makes the definition easier to understand.

**Example 4.2.8** The pseudo terms $x$, and $x \rightarrow x$, and $\text{Ind}(y:\square\{y,y\}) \rightarrow x$ are types of constructor in $x$. If $y \in V_{\Delta}$ and $x \neq y$ then $y \rightarrow x$ is not a type of constructor in $x$.

A (pseudo) inductive type is a pseudo term of the form $\text{Ind}(x:\square\{t_1,\ldots,t_n\})$ such that $\text{constructor}(t_i)$ for all $1 \leq i \leq n$.

**Remark 4.2.9** From now on we assume that the pseudo terms $t_i$ that occur in $I_0$ are type of constructors in $x$. Thus $I_0$ is assumed to be an inductive type.

The terms $\text{Constr}(i, I_0)$ are called constructors of the inductive type $I_0$, for $1 \leq i \leq n$. An inductive type can be based on other inductive types. For instance, we can define an inductive type that represents the integers, based on the inductive type $\text{Nat}$.

**Example 4.2.10** We define an inductive type $\text{Int}$ as follows:

$$\text{Int} = \text{Ind}(y:\square\{\text{Nat} \rightarrow y, y, \text{Nat} \rightarrow y\})$$

Let $\text{Neg} = \text{Constr}(1, \text{Int})$, and $\text{Zero} = \text{Constr}(2, \text{Int})$, and $\text{Pos} = \text{Constr}(3, \text{Int})$.

We will interpret the constructors of this inductive types as follows. The term $\text{Zero}$ represents $0 \in \mathbb{Z}$. If $n'$ represents $n \in \mathbb{N}$ then $\text{Neg } n'$ represents $-(n + 1)$ and $\text{Pos } n'$ represents $n + 1 \in \mathbb{Z}$.

We will show how we can represent functions on inductive types using $\text{Elim}$.

**Example 4.2.11** Assume we want to represent the negation function $-$ : $\mathbb{Z} \rightarrow \mathbb{Z}$ as a pseudo term that operates on $\text{Int}$. Then we must specify the domain and range of the function, and the desired result for the cases $\text{Neg } n$, and $\text{Zero}$, and $\text{Pos } n$.

We want a pseudo term $f$ such that:

$$f(\text{Neg } n) \rightarrow \text{Pos } n$$

$$f(\text{Zero}) \rightarrow \text{Zero}$$

$$f(\text{Pos } n) \rightarrow \text{Neg } n$$

Let $f_1 = \lambda n: \text{Nat}. \text{Pos } n$, $f_2 = \text{Zero}$, and $f_3 = \lambda n: \text{Nat}. \text{Neg } n$. Now the operator $-$ is represented by:

$$\text{Negation}(j) = \text{Elim}(\text{Int}, \text{Int}, j)\{f_1, f_2, f_3\}$$

The first argument of $\text{Elim}$ specifies the domain, and the second argument specifies the range and the terms $f_i$ specify the result for $\text{Constr}(i, \text{Int})$, for $i \in \{1, 2, 3\}$.

The desired reduction behaviour of $\text{Negation}$ is:
Inductive Types

\[
\begin{align*}
\text{Negation}(\text{Neg } n) \quad &\longrightarrow \quad f_1 \ n \ \rightarrow_\beta \ \text{Pos } n \\
\text{Negation}(\text{Zero}) \quad &\longrightarrow \quad f_2 \quad = \quad \text{Zero} \\
\text{Negation}(\text{Pos } n) \quad &\longrightarrow \quad f_3 \ n \ \rightarrow_\beta \ \text{Neg } n
\end{align*}
\]

In Example 4.2.11 we have specified a non-recursive function. We will now show how we can represent functions with recursive calls.

**Example 4.2.12** Recall the definition of the inductive type \( \text{Nat} \) of Example 4.2.6. We want to represent the operator \( + : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \) by a pseudo term \( g \) such that

\[
g \ \text{O } y \quad \longrightarrow \quad y \\
g \ (\text{S } x) \ y \quad \longrightarrow \quad \text{S } \ (g \ x \ y)
\]

The pseudo term \( g_1 = y \) specifies the result for \( \text{O} \). For specifying the result in the second case we need the result of \( g \ x \ y \). This result is given as an extra argument \( (g_2, g) \) to the function \( g_2 = \lambda z: \text{Nat} . \lambda g_2 : \text{Nat} . \text{S}(g_2, y) \), that specifies the result for pseudo terms of the form \( \text{S } z \). Now define

\[
\text{Plus}(x, y) = \text{Elim}(\text{Nat}, \text{Nat}, x)\{g_1, g_2\}
\]

The desired reduction behaviour of the function \( \text{Plus} \) is as follows:

\[
\begin{align*}
\text{Plus}(\text{O}, y) \quad &\longrightarrow \quad g_1 \quad = \quad y \\
\text{Plus}(\text{S } x, y) \quad &\longrightarrow \quad g_2 \ x \ (\text{Plus}(x, y)) \quad \rightarrow_\beta \quad \text{S}(\text{Plus}(x, y))
\end{align*}
\]

Before we can specify the computational behaviour of pseudo terms of the form \( \text{Elim}(I_0, Q, \alpha)\{h_1, \ldots, h_n\} \), we need auxiliary functions \( \text{ElimFun}_x(t_i, h_i, F, I_0) \) that take care that recursive calls of the function being defined are distributed over the functions \( h_i \) that specify the result for each constructor \( \text{Constr}(i, I_0) \) with type of constructor \( t_i \). The third argument of \( \text{ElimFun}_x \), \( F \), is a dummy function, that is supposed to compute the result for recursive calls. The last argument of \( \text{ElimFun}_x \), \( I_0 \), should be an inductive type of which \( t_i \) is a type of constructor in \( x \), and is used to obtain type correct recursive calls (by replacing \( x \) with \( I \)).

**Definition 4.2.13** The elimination function \( \text{ElimFun}_x \) for a type of constructor in \( x \) and three terms is defined as follows:

1. \( \text{ElimFun}_x(x, f, F, I) = f \)
2. \( \text{ElimFun}_x(u \rightarrow c, f, F, I) = \begin{cases} \lambda p: I . \text{ElimFun}_x(c, f, p (F \ p), F, I) & \text{(if } u = x \text{)} \\ \lambda p u. \text{ElimFun}_x(c, f, p, F, I) & \text{(otherwise)} \end{cases} \)

Notice that we have a recursive call in the second case with \( u = x \); this is indicated by the extra argument \( F \ p \).

**Definition 4.2.14** We define a reduction relation \( \rightarrow_\iota \) as follows:

\[
C[\text{Elim}(I, Q, \text{Constr}(i, I') m)] \rightarrow_\iota C[\text{ElimFun}_x(T_i, f_i, \lambda z: I . \text{Elim}(I, Q, z)\{f\}, I) \ m]
\]
with \( C[\_] \in \mathcal{H}_i \), and \( 1 \leq i \leq |\tilde{f}| \), and \( I = \text{ind}(x:A)(\tilde{T}) \), such that \( |\tilde{f}| = |\tilde{T}| \) and \( \text{const}_{x}(T_j) \) for all \( j \leq |\tilde{T}| \).

We will now demonstrate how this reduction relation \( \rightarrow_i \) gives a computational meaning to the functions we defined above.

**Example 4.2.15** Recall the definition of the operator *Negation* of Example 4.2.11. It has the following reduction behaviour:

\[
\begin{align*}
\text{Negation}(\text{Neg } n) & \rightarrow_i (\lambda x: \text{Nat}. f_1 \; p) \; n \quad \rightarrow_{\beta} \text{Pos } n \\
\text{Negation}(\text{Zero}) & \rightarrow_i f_2 = \text{Zero} \\
\text{Negation}(\text{Pos } n) & \rightarrow_i (\lambda x: \text{Nat}. f_3 \; p) \; n \quad \rightarrow_{\beta} \text{Neg } n
\end{align*}
\]

The operator *Plus* of Example 4.2.12 has the following reduction behaviour:

\[
\begin{align*}
\text{Plus}(O, y) & \rightarrow_i g_1 \\
\text{Plus}(Sx, y) & \rightarrow_i (\lambda x: \text{Nat}. g_2 \; p \; ((\lambda x: \text{Nat}. \text{Plus}(x, y)) \; p)) \; x = y \\
& \rightarrow_{\beta} S(\text{Plus}(x, y))
\end{align*}
\]

Notice that *Plus* is an *operational* version of the function *plus*, that is axiomatized in Example 4.1.4. We use the abbreviation \( \text{Plus}(x, y) \) in order to improve the readability of the expressions in the reduction sequence.

Now that we have described the intended meaning of the new constructs we will specify their typing rules. First we adapt the notions of Definition 3.1.12 for pseudo terms in \( \mathcal{T}_i \).

**Definition 4.2.16** A *statement* is a pair of pseudo terms in \( \mathcal{T}_i \), and a *variable declaration* is a statement with a variable as subject. The set of pseudo contexts \( \mathcal{X}_i \) is a finite sequence of variable declarations.

We will comment on the typing rules for the relation \( \vdash_i \) that are presented in Definition 4.2.25.

The first typing rule (\( \text{ind} \)) specifies that inductive types \( \text{ind}(x: \Box)(t_1, \ldots, t_n) \) are inhabitants of \( \Box \) if all type of constructors \( t_i \) are inhabitants of \( \Box \), for \( 1 \leq i \leq n \).

**Example 4.2.17** Recall the definition of *Nat* in Example 4.2.6. We can derive \( \varepsilon \vdash_i \text{Nat}: \Box \) as follows:

\[
\begin{align*}
\varepsilon \vdash_i \Box: \Delta & \quad \frac{}{x: \Box \vdash_i x: \Box} \\
& \quad \frac{\varepsilon \vdash_i x \rightarrow x: \Box}{\varepsilon \vdash_i \text{ind}(x: \Box)(x, x \rightarrow x): \Box}
\end{align*}
\]

The second typing rule (\( \text{intro} \)) specifies that the \( i \)th constructor \( \text{Constr}(i, I_0) \) has type \( t_i[x := I_0] \) if \( I_0 \) is a legal inductive type.

**Example 4.2.18** Using the derivation of Example 4.2.17 we can derive \( \varepsilon \vdash_i S: \text{Nat} \rightarrow \text{Nat} \).

Before we can specify typing rules for pseudo terms of the form \( \text{Elim}(I_0, Q, a)(h_1, \ldots, h_n) \), we need auxiliary functions *NodepElimType*\(_2\)(\( t_i, Q, I_0 \)) that specify the types of the functions \( h_i \), that determine the result for the \( i \)th constructor \( \text{Constr}(i, I_0) \). Recall that \( I_0 \) is the domain and \( Q \) is the range of the function specified by \( \text{Elim}(I_0, Q, a)(h_1, \ldots, h_n) \).
Remark 4.2.19 Recall the definition of \(\text{Negation}(j)\) that has specified domain \(\text{Int}\) and range \(\text{Int}\) (see Example 4.2.11). The functions \(f_1, f_2,\) and \(f_3\) that specify the result for pseudo terms of the form \(\text{Neg} \ n, \text{Zero},\) and \(\text{Pos} \ n\) resp., can be typed as follows:

\[
\begin{align*}
  j : \text{Int} & \vdash f_1 : \text{Nat} \rightarrow \text{Int} \\
  j : \text{Int} & \vdash f_2 : \text{Int} \\
  j : \text{Int} & \vdash f_3 : \text{Nat} \rightarrow \text{Int}
\end{align*}
\]

The type of \(f_i\) is determined by the type of \(\text{Constr}(i, \text{Int})\) and the specified range (\(\text{Int}\)).

Recall that in the definition of \(\text{Plus}(x, y)\) the function \(g_2\), that specifies the result if \(x = S \ x'\), has an extra argument that gives us the result for \(\text{Plus}(x', y)\). We can derive the following type for this function:

\[
y : \text{Nat} \vdash g_2 : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}
\]

Definition 4.2.20 The non-dependent elimination type \(\text{NodepElimType}_x\) for a type of constructor in \(x\) and two pseudo terms \(Q, I\) is defined as:

1. \(\text{NodepElimType}_x(x, Q, I) = Q\)
2. \(\text{NodepElimType}_x(u \rightarrow c, Q, I) =\)
   \[
   \begin{cases}
   I \rightarrow (Q \rightarrow \text{NodepElimType}_x(c, Q, I)) & \text{(if } u = x) \\
   u \rightarrow \text{NodepElimType}_x(c, Q, I) & \text{(otherwise)}
   \end{cases}
   \]

The third typing rule (nodep-elim) of Definition 4.2.25 specifies that a pseudo term of the form \(\text{Elim}(I_0, Q, a)\{h_1, \ldots, h_n\}\) has type \(Q\) if \(Q\) has type \(\Box\), and \(a\) has type \(I_0\), and each \(h_i\) has type \(\text{NodepElimType}_x(t_i, Q, I_0)\). Recall that the pseudo term \(a\) serves as argument.

Example 4.2.21 Let \(\Gamma\) be the pseudo context \(x : \text{Nat}, y : \text{Nat}\). Using the (nodep-elim) rule we can derive \(\Gamma \vdash_1 \text{Plus}(x, y) : \text{Nat}\) (see Example 4.2.12) as follows:

\[
\begin{array}{c}
\ldots \\
\Gamma \vdash_1 \text{Nat} : \Box \\
\Gamma \vdash_1 g_1 : \text{Nat} \\
\Gamma \vdash_1 g_2 : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\Gamma \vdash_1 x : \text{Nat} \\
\Gamma \vdash_1 \text{Elim}(\text{Nat}, \text{Nat}, x)\{g_1, g_2\} : \text{Nat}
\end{array}
\]

Besides functions of type \(I_0 \rightarrow Q\) we can also represent reasoning by cases using the \(\text{Elim}-\text{construct}\). If \(P\) is a predicate on \(I_0\) (thus \(P\) has type \(I_0 \rightarrow \star\)) then a pseudo term \(\text{Elim}(I_0, P, b)\{h_1, \ldots, h_n\}\) represents a proof of \(P \ b\). The pseudo term \(b\) should have type \(I_0\) and each pseudo term \(h_i\) should prove \(P(\text{Constr}(i, I_0) \ a_1 \ldots a_m)\), for all typable terms of the form \(\text{Constr}(i, I_0) \ a_1 \ldots a_m\), for \(1 \leq i \leq n\).

Example 4.2.22 Recall the definition of Leibniz equality (=\(\text{Int}\)) of Example 3.2.3. Let \(\text{IntSplit}\) be the predicate on \(\text{Int}\) defined by:

\[
\text{IntSplit} = \lambda j : \text{Int}. (j =_{\text{Int}} \text{Zero} \lor (\exists n \in \text{Nat}. j =_{\text{Int}} \text{Neg} \ n \lor j =_{\text{Int}} \text{Pos} \ n)).
\]
A proof of $\forall j \in \text{Int. IntSplit } j$ by reasoning by cases requires pseudo terms $q_1$, $q_2$, and $q_3$ such that:

$$
\begin{align*}
\Gamma \vdash q_1 : \forall n \in \text{Nat. IntSplit (Neg } n) \\
\Gamma \vdash q_2 : \text{IntSplit Zero} \\
\Gamma \vdash q_3 : \forall n \in \text{Nat. IntSplit (Pos } n)
\end{align*}
$$

Now the desired type of $\text{Elim}(I_0, \text{IntSplit, } b)\{q_1, q_2, q_3\}$ is $\text{IntSplit } b$.

In the same way as $\text{NodepElimType}_x(t_i, Q, I_0)$ specifies the type of functions $h_i$ in pseudo terms of the form $\text{Elim}(I_0, Q, a)\{h_1, \ldots, h_n\}$ such that $Q$ is an inhabitant of $\Box$,

$\text{DepElimType}_x(t_i, P, \text{Constr}(i, I_0), I_0)$ specifies the type of the proofs $q_i$ in pseudo terms of the form $\text{Elim}(I_0, P, b)\{q_1, \ldots, q_n\}$ such that $P$ is an inhabitant of $I_0 \to *$.

**Definition 4.2.23** The dependent elimination type $\text{DepElimType}_x$ for a type of constructor in $x$ and three terms is defined as:

1. $\text{DepElimType}_x(x, P, a, I) = Pa$

2. $\text{DepElimType}_x(u \to c, P, a, I) = \begin{cases} 
\text{Ily}.I.(P \ y) \to \text{DepElimType}_x(c, P, a \ y, I) & (\text{if } u = x) \\
\text{Ily}.u.\text{DepElimType}_x(c, P, a \ y, I) & (\text{otherwise})
\end{cases}$

The fourth typing rule (dep-eim) of Definition 4.2.25 specifies that a pseudo term of the form $\text{Elim}(I_0, P, b)\{q_1, \ldots, q_n\}$ has type $P \ b$ if $P$ has type $I_0 \to *$, and $b$ has type $I_0$, and each $q_i$ has type $\text{DepElimType}_x(t_i, P, \text{Constr}(i, I_0), I_0)$.

The construction rule replaces the $\lambda$-conversion rule of $\lambda$HOL. The usage of the last two typing rules will be discussed in Example 4.2.24. We will illustrate how the new conversion rule enables us to use the computational meaning of functions, that are defined by the Elim constructor, in formal proofs.

**Example 4.2.24** Consider the definition of the addition on natural numbers by $\text{Plus}$ in Example 4.2.12. Recall the definition of Leibniz equality of Example 3.2.3. Via the conversion rule we can use the computational behaviour of $\text{Plus}$ to give a short proof of its associativity $(\text{Plus}(x, \text{Plus}(y, z))) =_{\text{Nat}} \text{Plus}(\text{Plus}(x, y), z)$ by induction on $x$.

Let $\text{Assoc} = \lambda n: \text{Nat}. \text{Plus}(n, \text{Plus}(y, z)) =_{\text{Nat}} \text{Plus}(\text{Plus}(n, y), z)$;

let $\Gamma = (x: \text{Nat}, y: \text{Nat}, z: \text{Nat})$. We want to solve the problem:

$$
\Gamma \vdash ? : \text{Assoc } x.
$$

For doing induction on $x$ we need to apply the (dep-eim)-rule. Thus we fill in $\text{Elim}(\text{Nat}, \text{Assoc}, x)\{?, ?\}$ for $?$. Now we need to solve the problems:

$$
\Gamma \vdash ?_1 : \text{Assoc } O \\
\Gamma \vdash ?_2 : (\forall x' \in \text{Nat}. (\text{Assoc } x') \to (\text{Assoc}(S x'))) \tag{1}
$$
The instantiation of the proof of the reflexivity of \( =_{\text{Nat}} \) (\( =_{\text{Nat} \text{-refl}} \)) with \( \text{Plus}(y, z) \) has type \( \text{Plus}(y, z) =_{\text{Nat}} \text{Plus}(y, z) \), that is convertible (modulo \( =_{\beta, \delta} \)) with \( \text{Assoc O} \). Now we can use the (conv)-rule to derive that this term \( (=_{\text{Nat} \text{-refl}} (\text{Plus}(y, z))) \) has type \( \text{Assoc O} \). Thus we can fill in this term for \( \gamma_1 \). If we fill in \( \lambda x'.\text{Nat} . \lambda i : (\text{Assoc } x').\gamma_3 \) for \( \gamma_2 \) we have to prove:

\[
\Gamma, x' : \text{Nat}, i : (\text{Assoc } x') \vdash_3 (\text{Assoc}(S \ x'))
\]

For \( \gamma_3 \) we can use a proof of the equivalent (modulo \( =_{\beta, \delta} \)) proposition

\[
\text{S}(\text{Plus}(x', \text{Plus}(y, z))) =_{\text{Nat}} \text{S}(\text{Plus}(\text{Plus}(x', y), z)).
\]

Using the compatibility of \( =_{\text{Nat}} \) with respect to \( S \) we need to prove:

\[
\Gamma, x' : \text{Nat}, i : (\text{Assoc } x') \vdash_4 (\text{Plus}(x', \text{Plus}(y, z))) =_{\text{Nat}} (\text{Plus}(\text{Plus}(x', y), z))
\]

We can fill in the assumption \( i \) (the induction hypothesis) for \( \gamma_4 \), because its type \( (\text{Assoc } x') \) is convertible with the desired type \( (\text{Plus}(x', \text{Plus}(y, z))) =_{\text{Nat}} (\text{Plus}(\text{Plus}(x', y), z)) \).

Here we present the typing rules for inductive types.

**Definition 4.2.25** Typing rules for \( \vdash_1 \) are the rules for \( \vdash \) of Definition 3.1.15 together with following rules for inductive types:

\[
\begin{array}{ll}
\text{(ind)} & \frac{\Gamma, x : \Box \vdash C_j : \Box | \vec{C}| \quad \text{if constr}_x(C_j), \text{ all } j \leq |\vec{C}|}{\Gamma_1 \vdash \text{Ind}(x : \Box)\{\vec{C}\} : \Box} \\
\text{(intro)} & \frac{\Gamma_1 \vdash \text{Ind}(x : A)\{\vec{C}\} : T}{\Gamma_1 \vdash \text{Constr}(j, \text{Ind}(x : A)\{\vec{C}\}) : C_j[x := \text{Ind}(x : A)\{\vec{C}\}]} \quad \text{if } j \leq |\vec{C}| \\
\text{(nodep-elem)} & \frac{\Gamma_1 \vdash Q : \Box \quad (\Gamma_1 \vdash f_j : \text{NodepElimType}_x(C_j, Q, I))_{j=1}^n \quad \Gamma_1 \vdash t : I}{\Gamma_1 \vdash \text{Elim}(I, Q, t)\{f_1, \ldots, f_n\} : Q} \\
\text{(dep-elem)} & \frac{\Gamma_1 \vdash P : I \rightarrow \star \quad (\Gamma_1 \vdash f_j : \text{DepElimType}_x(C_j, P, \text{Constr}(j, I), I))_{j=1}^{|\vec{C}|} \quad \Gamma_1 \vdash t : I}{\Gamma_1 \vdash \text{Elim}(I, P, t)\{f_1, \ldots, f_n\} : P t} \quad \text{with } I = \text{Ind}(x : A)\{C_1, \ldots, C_n\} \\
\text{(conv)} & \frac{\Gamma_1 \vdash a : T \quad \Gamma_1 \vdash t : s \quad \Gamma_1 \vdash U : s \quad T =_{\beta, \delta} U}{\Gamma_1 \vdash a : U} \quad s \in \text{Universes}
\end{array}
\]

Now we can define the language of Higher Order Logic with inductive types:

**Definition 4.2.26** Higher Order Logic with inductive types is the tuple

\( (\mathcal{T}_1, \mathcal{X}_1, \{\rightarrow_{\beta, \delta}, \rightarrow_1\}, \vdash_1) \).
We will use the abbreviation $\lambda$HOL$_4$ for Higher Order Logic with inductive types.

We now state several fundamental properties of the type system $\lambda$HOL$_4$, that show its use as a formal mathematical language.

**Theorem 4.2.27** The system $\lambda$HOL$_4$ has the following properties:

1. $\rightarrow_{\beta, \Delta}$ is confluent.
2. If $\Gamma \vdash t: u$ and $t \rightarrow_{\beta, \Delta} t'$ then $\Gamma \vdash t': u$.
3. $\rightarrow_{\beta, \Delta}$ is strongly normalizing for legal terms.

**Proof**

In [33] these properties are proved for the Calculus of Constructions with inductive types.

1. The pseudo terms and reduction relation $\rightarrow_{\beta, \Delta}$ of $\lambda$HOL$_4$ and $\lambda$CC with inductive types are the same.

2. Proof with a similar structure as the proof for subject reduction of Proposition 6.2.38.

3. The reduction preserving embedding of $\lambda$HOL in $\lambda$CC (see proof of Theorem 3.3.5) by the mapping $E_i$ that replaces each $\Box$ with $*$ and each $\Delta$ with $\Box$, is extended to a map $E_i$ on pseudo terms in $T_i$ by adding the following rules:

   (a) $E_i(\text{Elim}(I, Q, t)\{f_1, \ldots, f_n\}) =$

   $\begin{cases} 
   \text{Elim}(E_i(I), \lambda E_i(I), E_i(Q), E_i(t)) & \text{(if } Q = u_1 \rightarrow \ldots \rightarrow u_{n+1}, \text{ and } u_{n+1} \neq *) \\
   \text{Elim}(E_i(I), E_i(Q), E_i(t))\{E_i(f_1), \ldots, E_i(f_n)\} & \text{(otherwise)}
   \end{cases}$

   (b) $E_i(\text{Constr}(j, I)) = \text{Constr}(j, E_i(I))$

   (c) $E_i(\text{Ind}(x : u)\{t_1, \ldots, t_n\}) = \text{Ind}(x : E_i(u))\{E_i(t_1), \ldots, E_i(t_n)\}$

For instance, we have $E_i(\text{Nat}) = \text{Ind}(x : *)\{x, x \rightarrow x\}$, and $E_i(\text{Plus}(x, y)) = \text{Elim}(E_i(\text{Nat}), \lambda E_i(\text{Nat}), E_i(\text{Nat}), x)\{y, \lambda z: E_i(\text{Nat}), \lambda g_{x,y}: E_i(\text{Nat}), E_i(\text{S}) \ g_{x,y}\}$.

By $E_i(\Gamma)$ we denote the pseudo context obtained from $\Gamma$ by replacing each predicate $p$ by $E_i(p)$. If $\Gamma \vdash a : t$ then we have $E_i(\Gamma) \vdash E_i(a) : t'$ in $\lambda$CC with inductive types, where $t' = \begin{cases} 
\Box & \text{(if } a = t_1 \rightarrow \ldots \rightarrow t_n \rightarrow * (0 \leq n))
\end{cases}$. Our definition of the notion of ‘simple type of constructor’ in Definition 4.2.7 guarantees that any legal inductive type in $\lambda$HOL$_4$ is mapped to a small inductive type in $\lambda$CC with inductive types. The (nodep-elim) rule of $\lambda$CC may only be used for small inductive types. Notice that only typable terms of the form $\text{Elim}(I, u_1 \rightarrow \ldots \rightarrow u_n \rightarrow *, t)\{f\}$ are transformed by $E_i$ to terms that require this rule.

An inductive type is small if all its type of constructors are small. A type of constructor $t_1 \rightarrow \ldots \rightarrow t_n \rightarrow t_{n+1}$ is small if all $t_i$ have type $\ast$. \qed
Using an operational instead of an axiomatic definition can make proofs shorter, as an internal computation can represent several deduction steps.

**Example 4.2.28** Recall the axiomatic definition of *plus* in Example 4.1.4. A formal proof of the associativity for this operator in \( \lambda \text{HOL} \) is much larger than the proof for the associativity of \( \text{Plus} \), that we described in Example 4.2.24. For instance, for proving the base case \((\text{plus } 0 (\text{plus } y z) =_{N} \text{plus } (\text{plus } 0 y) z)\) we need \text{plus}0 twice, and compatibility, symmetry, and transitivity of \( =_{N} \) once. As a comparison, the base case in the proof of associativity for the operator \( \text{Plus} \) requires only the reflexivity of \( =_{\text{Nat}} \), since the conversion rule allows us to use the computational behaviour of \( \text{Plus} \).

An operational definition of a notion using inductive types does not always help to make proofs of properties of this notion shorter. For instance, the expression \( \text{Plus}(\text{Plus}(x, y), \text{Plus}(z, 0)) \) does not reduce, because \( \text{Plus} \) only inspects the left argument. Thus this expression is not convertible with \( \text{Plus}(\text{Plus}(x, y), z) \). But we know that we can prove \( \text{Plus}(\text{Plus}(x, y), \text{Plus}(z, 0)) =_{\text{Nat}} \text{Plus}(\text{Plus}(x, y), z) \), as \( \text{Nat} \) is a monoid. Therefore it would be convenient if we could use the TRS of Example 4.1.2 in the formal system to solve this equational problem automatically. This is possible if we extend the type theory with so-called oracle types which allow to introduce complete TRSs in type theory. We will present joint work with Gilles Barthe on the extension of the proof development system LEGO with this formalism.

### 4.3 Oracle types

[Joint work with Gilles Barthe]

We would like to be able to obtain formal proofs for valid instances of decidable relations. This can be formalized by adding a new syntactical construct \( \text{Axiom} \) with a typing rule of the following form:

\[
\vdash \text{Axiom}(R\tilde{a}) : R\tilde{a}, \text{ if } R(\tilde{a}).
\]

In this rule \( R \) denotes a decidable relation, and an underlined mathematical expression denotes its formal representation. Thus, if we have a decision procedure for a relation we can use it to obtain formal proofs for valid instances of this relation (see Figure 4.2). The rule described above is too general to be useful in automated verification, because only a mathematician can provide the link between (the decision procedure for) a mathematical relation and its formal representation. The best we can achieve is to restrict the use of this rule for some fixed set of decision procedures, or for relations for which the formal representations can be automatically obtained. We opt for the last solution, by considering the equivalence relations induced by rewrite relations of confluent, terminating Term Rewriting Systems with finitely many function symbols and rules.

For this purpose we will extend the formal language \( \lambda \text{HOL}_4 \) of Section 4.2 with oracle types. Roughly speaking an oracle type consists of a complete TRS, and a representation
Figure 4.2: Extension of a Proof Assistant with an Oracle

(in λHOL with inductive types) of its equational theory. Using a new construct Rewrite we can link computable equality in the TRS of an oracle type to derivable equality in its equational theory. In this approach, TRSs are part of the formal language and therefore their reduction relations are considered as internal reductions.

In order to be able to use the derivable equality in the equational theory in its models we need executable interpretation functions that map terms in an equational theory to elements in its models. Therefore the terms in the equational theory will be represented as canonical inhabitants of an inductive type. For representing variables the inductive type has an extra construct that constructs a term from a natural number; in this way we can represent infinitely many variables and we can distinguish between different variables (by a function in the formal language).

An oracle type for a given complete one-sorted TRS(Σ, R) consists of this TRS with an extended signature Σ♯ for representing variables by natural numbers, and an inductive type representing the Σ-terms with an equivalence relation representing provable equality in the equational theory of (Σ, R). Furthermore we have an axiom relating computable equality in (Σ♯, R) with provable equality in the inductive type. In the next example we describe an extension of the formal language of λHOL with inductive types (see Section 4.2) with an oracle type for monoids.

**Example 4.3.1** Recall the representation of the natural number by an inductive type Nat with constructors O and S (see Example 4.2.6). An oracle type for the TRS for monoids of Example 4.1.2 consists of:

1. The extension (Σ♯, R) of the TRS for M with the signature:

   sort N

   func O: N
   S: N → N
   var_M: N → M

2. An inductive type M = Ind(x : □) {x → x → x, Nat → x} with constructors :
   = Constr(M, 1), e = Constr(M, 2), var_M = Constr(M, 3), representing Π(Σ, V). The constructors : and e represent the function symbols · and e of Σ. Applications of the constructor var_M to canonical pseudo terms of type Nat, such as var_M(O), represent variables.
Oracle types

3. A map \([-]\): \(T(\Sigma^\#)^\# \to T_1\) mapping closed terms, i.e. terms not containing variables, to their representations, such that

\[
\begin{align*}
[x \cdot y] &= \varepsilon([x]_\#) ([y]_\#) \\
[a]_\# &= \varepsilon \\
[\text{var}_\#(n)]_\# &= \text{var}_\#([n]_\#)
\end{align*}
\]

The map \([-]:\) maps closed terms of sort \(\mathbb{N}\) to their representations in \(\text{Nat}\).

4. A construct \text{Rewrite}, with a typing rule (rewrite) that relates computable equality in \((\Sigma^\#, R)\) to derivable equality in \(M\):

\[
\text{(rewrite)} \quad \epsilon \vdash \text{Rewrite}([a]_\#, [b]_\#) : [a]_\# \sim_M [b]_\# \quad \text{if} \ a =_R b, \ a, b \in T(\Sigma^\#)_\#
\]

Where \(\sim_M\) is the binary relation on \(M\) defined as:

\[
m_1 \sim_M m_2 = \forall r \in M \to M \to \ast. (\text{eqrel}(r) \land \text{monoidrel}(r)) \to r \ m_1 \ m_2.
\]

In this definition \(\text{eqrel}(r)\) denotes equivalence of \(r\), and we have

\[
\begin{align*}
\forall x \in M, r (\varepsilon x) x & \land \\
\text{monoidrel}(r) &= \\
\forall x, y, z \in M, r (\varepsilon (x \cdot y z)) & (\varepsilon (x y z)) & \land \\
\forall x_1, y_1, x_2, y_2 \in M, (r \ x_1 y_1) & (r \ x_2 y_2) & \to r (\cdot x_1 x_2) (\cdot y_1 y_2)
\end{align*}
\]

The typing rule for the new construct \text{Rewrite} allows us to use computational equality in a TRS to prove equality in the representation of the related equational theory in the type system. This is illustrated in Figure 4.3.

<table>
<thead>
<tr>
<th>Representation of the TRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varepsilon \cdot (\text{var}_#(0)) =<em>R \text{var}</em>#(0)))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mathematical language</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varepsilon \cdot x_0 = x_0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coding of the equational theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{Rewrite}([\varepsilon \cdot \text{var}<em>#(0)) \sim_M [\text{var}</em>#(0))])</td>
</tr>
</tbody>
</table>

Figure 4.3: Representation of a monoid by an oracle type

Example 4.3.2 Let \(v_0 = \text{var}_\#(0), \ v_1 = \text{var}_\#(S(0)),\) and \(v_2 = \text{var}_\#(S(S(0)))\) be closed terms \((v_0, v_1, v_2 \in T(\Sigma^\#)_\#)\). We have \((\varepsilon \cdot v_0) \cdot (v_1 \cdot v_2) =_R (v_0 \cdot v_1) \cdot (v_2 \cdot v_0)\).

Let \(v_0 = \text{var}_\#(O), \ v_1 = \text{var}_\#(S(O)),\) and \(v_2 = \text{var}_\#(S(S(O)))\) be the inductive representations \((v_0, v_1, v_2 \in T_1)\) of the closed terms \(v_0, v_1,\) and \(v_2\). Thus \([v_0]_\# = v_0, [v_1]_\# = v_1,\) and \([v_2]_\# = v_2\).

By \(\text{rewrite}\) we obtain a formal proof of \(\varepsilon \vdash \text{Rewrite}((\cdot \ v_0) (\cdot \ v_1 \ v_2) \sim_M (\cdot \ v_0 \ v_1) (\cdot \ v_2 \ v_0))\) as follows:

\[
\varepsilon \vdash \text{Rewrite}((\cdot \ v_0) (\cdot \ v_1 \ v_2) \sim_M (\cdot \ v_0 \ v_1) (\cdot \ v_2 \ v_0))
\]

\[
(\cdot \ v_0) (\cdot \ v_1 \ v_2) \sim_M (\cdot \ v_0 \ v_1) (\cdot \ v_2 \ v_0)
\]
Proving equalities in models of oracle types

We will now show how we can use derivable equality in an oracle type to prove equality in its models by the two-level approach ([6]). For this purpose we define an interpretation function mapping syntactic terms to elements of an algebra, and the notion model of a theory.

Example 4.3.3 Let type $A$ with binary operator $\cdot^A : A \to A \to A$ and constant $e^A : A$ be the formal representation of a $\Sigma$-algebra. We define $\text{int}^A_M : (\text{Nat} \to A) \to (M \to A)$ by structural induction on $M$, such that

$$\text{int}^A_M \rho (\cdot x y) \xrightarrow{\beta,\varepsilon} \cdot^A (\text{int}^A_M \rho x) (\text{int}^A_M \rho y)$$

$$\text{int}^A_M \rho \in \xrightarrow{\beta,\varepsilon} e^A$$

$$\text{int}^A_M \rho (\text{var}_M n) \xrightarrow{\beta,\varepsilon} v n$$

By $\text{model}^A_M$ we denote the property:

$$\forall \rho \in \text{Nat} \to A. \text{monoidrel}(\lambda m_1, m_2 : M. \text{int}^A_M \rho m_1 =_A \text{int}^A_M \rho m_2)$$

Notice that one of the consequences of $\text{model}^A_M$ is:

$$\forall \rho \in \text{Nat} \to A. \forall x \in M. (\lambda m_1, m_2 : M. \text{int}^A_M \rho m_1 =_A \text{int}^A_M \rho m_2) (\cdot x e) x$$

The previous statement is equivalent (modulo $=_{\beta,\varepsilon}$) with:

$$\forall \rho \in \text{Nat} \to A. \forall x \in M. \cdot^A (\text{int}^A_M \rho x) e^A =_A \text{int}^A_M \rho x$$

and it implies: $\forall a \in A. \cdot^A a e^A =_A a$.

We can use derivable equality in the equational theory of monoids to prove equalities in its models. This is illustrated in Figure 4.4 and Example 4.3.4.

<table>
<thead>
<tr>
<th>Representation of the natural numbers</th>
<th>plus 0 y =N y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{int}^N_M$</td>
</tr>
<tr>
<td>0 + y = y</td>
<td>$\varepsilon (\text{var}_M(S(O))) \sim_M (\text{var}_M(S(O)))$</td>
</tr>
<tr>
<td>Mathematical language</td>
<td>Coding of the equational theory</td>
</tr>
</tbody>
</table>

Figure 4.4: Representation of the monoid of the natural numbers

In the following example we will derive a proof for the equation described in Example 4.1.7 using the oracle type for the theory of monoids. In this way we can compare the two methods for obtaining formal proofs of valid equations. Notice that we have three
different representations of the natural numbers. First we have the axiomatic version $N$
with constant 0 and operator plus of Example 4.1.4 that plays the rôle of monoid; second we
have the inductive type $\text{Nat}$ with constructors $O$ and $S$ that is used in the representation
of variables by $\text{var}_M$; third we have the sort $\mathbb{N}$ with function symbols $0$ and $S$ that is used
in the representation of variables by $\text{var}_N$ in the TRS $(\Sigma, R)$.

**Example 4.3.4** Using the definition of $\sim_M$ we can prove that derivable equality is valid
in each model:

$$\text{mode}_{\text{M}}^A \rightarrow (\forall m_1, m_2 \in \mathbb{M}. m_1 \sim_M m_2 \rightarrow (\forall \rho \in \text{Nat} \rightarrow A.\text{int}_M^A \rho m_1 =_A \text{int}_M^A \rho m_2))$$

Using the properties of Examples 4.1.5, 4.1.6 we can show that we have $\text{mode}_{\text{M}}^N$ (with
$\text{N} = \text{plus}$ and $\text{N} = 0$). Using this proof and the derivation of Example 4.3.2 we can
specialize the previous result to obtain a proof of:

$$\forall \rho \in \text{Nat} \rightarrow N.\text{int}_M^N \rho \left(\cdot (\cdot v_0) (\cdot v_1 v_2)\right) =_N \text{int}_M^N \rho \left(\cdot (\cdot v_0 v_1) (\cdot v_2 \varepsilon)\right)$$

If we specialize this for a valuation function $\rho : \text{Nat} \rightarrow N$ such that $\rho (O) \rightarrow_{\beta,v} x$, $\rho (S(O)) \rightarrow_{\beta,v} y$, and $\rho (S(S(O))) \rightarrow_{\beta,v} z$ we obtain a proof of:

$$\text{plus} (\text{plus} 0 x) (\text{plus} x y) =_N \text{plus} (\text{plus} x y) (\text{plus} z 0)$$

Using this method for partially automating equational reasoning we can obtain short proofs
of valid equations in models of first order theories. A formal proof of a valid equation
$t_1 = t_2$, that is obtained in this way, has a length of $\mathcal{O}(|t_1| + |t_2|)$. The obtained proofs
have a higher level of abstraction than proofs obtained by term rewriting with tracing
(see Example 4.1.7), since they are based on the soundness of computable equality in a
TRS with respect to provable equality in the related equational theory. Once the relation
between provable equality in the equational theory $T$ of a complete TRS and equality in a
model $A$ has been established, all interpretations in $A$ of derivable equations in $T$ can be
proved automatically. Notice that we need external computational power for computing
the $\Sigma$-terms and the valuation function for given members of a $\Sigma$-algebra, because no
internally defined function can recognize the structure of all terms (of a certain type).
However, the price we have to pay for automatically solving certain equational problems
is the extension of the formal system with oracle types.

**Differences with the original definition**

We have presented a simplification of the original definition of oracle types in [5]. The
main motivation for this simplification is that it captures the essence of oracle types as
a formalism relating computations in a TRS with deductions in the related equational
theory. In the original definition the TRS itself is represented in the language by a so-
called algebraic type whose rewrite rules are formalized by a new reduction relation that
is used in an extended conversion rule. As a consequence, the map that relates terms of
the TRS to their representations, is also part of the formal language. In our approach
the computational power of the TRS of an oracle type is used only in the condition of the typing rule for the new construct Rewrite, and therefore the conversion rule remains unchanged.

In the original version of oracle types, the definition of the oracle type for a monoid would differ from the definition in Example 4.3.1 in the following points:

1. The set of pseudo terms is extended with the symbols \( \cdot \), \( \cdot e \), and \( \text{var}_M \). The following typing rules are added for typing these constants:

\[
\varepsilon \vdash M : \square \quad \varepsilon \vdash \cdot : M \to M \to M \quad \varepsilon \vdash e : M \quad \varepsilon \vdash \text{var}_M : \text{Nat} \to M
\]

and the following reduction rules:

\[
\cdot e x \to \cdot x \\
\cdot x e \to \cdot x \\
\cdot x (\cdot y z) \to (\cdot x y) z
\]

2. The representation of \( M \) is the same.

3. The set of pseudo terms is extended with a construct \([-]\), that represents the 'inverse' of the map \([-]_M\). It has the following typing rule:

\[
\Gamma \vdash a : M \\
\Gamma \vdash [a] : M
\]

and the following reduction rules:

\[
[\cdot x y] \to x \cdot [x] [y] \\
[e] \to x e \\
[\text{var}_M(n)] \to x \text{var}_M(n)
\]

4. The construct Rewrite is replaced by a construct noconf with the following typing rule:

\[
\Gamma \vdash p : [a] =_M [b] \\
\Gamma \vdash \text{noconf} p : a \sim_M b
\]

5. The conversion rule is extended as follows:

\[
\Gamma \vdash a : A \\
\Gamma \vdash B : s \\
A =_{\beta, \eta, \delta} B \\
\Gamma \vdash a : B
\]

In the original version the conversion rule is extended in order to implement the rewrite rules of the monoid. In order to guarantee the uniqueness of the results, we must define \([-]\) as the 'inverse' of \([-]_M\). Therefore the executable map \([-]\) transforms arguments of type \( M \) to their representations in \( M \), and not the other way around.
Implementation of oracle types

We have combined the LEGO proof system [27] with the symbolic computation system Reduce [19] to obtain an implementation of oracle types. First we describe the proof checking facilities that we need for oracle types.

The use of computations of an oracle type in a formal proof is made explicit by the presence of the Rewrite construct in the proof-object. The verification of this new rule consists of determining the oracle type from the inductive type of the inductive terms in the Rewrite construct, computing the closed terms that are mapped to these inductive terms, and verifying whether the normal forms of the closed terms are syntactically equal. The result of mapping a closed term of the TRS of an oracle to an inductive term consists of applications of constructors of inductive types only. Thus determining the oracle type from the inductive terms is easy, as the inductive type is part of the constructor. Computing the closed term from an inductive term built from applications of constructors is easy. Thus verifying the application of the (rewrite) rule can be done efficiently.

We have implemented facilities that support the development of proofs using oracle types. In our tool one can:

Define a new oracle type as a Term Rewriting System.
This TRS is defined in Reduce, and the related inductive type is defined in LEGO.

Bind an ‘algebra’ A to an oracle type O by proving model_A^O.
The notions int_A^O and model_A^O are automatically generated by a tactic. The proof of model_A^O is used to generate a proof that each provable equality in O is valid in A.

Prove an equality in a model A of an oracle type O.
First an equality a =_A b is transformed into an equivalent int_A^O \rho [t]_O =_A int_A^O \rho [u]_O, by computing terms t, u and a valuation function \rho, such that int_A^O \rho [t]_O \rightarrow_{\beta, \delta} a and int_A^O \rho [u]_O \rightarrow_{\beta, \delta} b. If t and u have the same normal form, then the Rewrite axiom is used to produce a proof of a =_A b (based on the proof generated by the Bind command).

The commands described above can fail, for instance if one tries to prove an invalid equality.

Discussion

The extension of LEGO with oracle types provides a way to partially automate equational reasoning. Although this extension makes proof development easier, it does not increase the mathematical strength of the formal system (we can neither express more mathematical notions, nor prove more theorems). In principle we could define a function that computes the normal form of (the inductive representation of) an expression in the equational theory of a complete TRS in LEGO. The 'only' problem is that it is not feasible to define such a
function based on structural induction. Moreover proving the correctness of such a function is even less feasible. We learnt this when we formalized such functions for equational theories with associative, commutative, distributive operators, and with neutral elements in LEGO. In this formalization we used an inductive type for representing terms. For defining functions that compute normal forms of terms we needed approximately 200 lines, and for proving their correctness we needed about 1450 lines of LEGO code. These functions should be able to recognize the structure of their arguments in order to compute their normal forms. Due to the fact that the recursor of inductive types allows us to inspect only the head constructor of an inductive term, recognizing a term built from \( n \) constructors requires \( n \) applications of the recursor. Therefore these functions are defined as combinations of subfunctions that perform certain subtasks. First the tasks performed by these subfunctions had to be specified and the correctness of these subfunctions had to be proved. Finally the correctness of the normal form functions was proved. Most of these correctness proofs are based on structural induction, because the functions are defined using the recursor of inductive types.

Extending the formal system with special proving procedures is not a good solution for solving this problem, since it threatens the reliability of the proof checker (a larger program is more likely to contain errors). Moreover such specific algorithms do not belong to a general language for expressing mathematics, but should be definable in it. What we need is a general formalism for defining algorithms and proving their correctness in a feasible way. We feel that \textit{pattern matching} is a good candidate. Before we give a formal description of an extension of \( \lambda \)HOL with pattern matching, we will try to explain the ideas behind this extension in the framework of Term Rewriting Systems.
Chapter 5

Representing Functions in Term Rewriting

Algorithms that are proven correct (certified algorithms) provide an important abstraction mechanism in formal mathematics. Although inductive types are powerful enough to formalize certified algorithms, it is quite hard to specify functions and prove them correct using this formalism in practice. In this chapter we will describe how functions can be defined in functional programming languages by pattern matching in an elegant way. We will compare the definition of functions in the functional programming language ML [29] with definitions of these functions in Higher Order Logic with inductive types. Interesting aspects of the pattern matching formalism of ML are that the patterns are built from constructors of algebraic data types and variables only, and that no variable may occur more than once in a pattern. These restrictions are formalized in Constructor Systems, that are Term Rewriting Systems with two kinds of function symbols: defined symbols (with a computational meaning) and constructor symbols (without a computational meaning). The semantics of Constructor Systems is easy to understand and allows us to guarantee fundamental properties such as confluence and termination by easily verifiable conditions.

The textual order of the rules of a function definition in ML determines the order in which they are tried. This rewrite strategy gives the function a deterministic computational behaviour, and allows us to focus on the interesting cases. The use of an order on rules will be formalized in Priority Constructor Systems, that are Constructor Systems with a strict partial order on rules. We restrict the applicability of the rules by imposing a decidable condition on them, that guarantees that no rule with higher priority can become applicable. We will present an algorithm that transforms Priority Constructor Systems into equivalent Constructor Systems. Based on the criteria developed for Constructor Systems we will determine conditions that guarantee fundamental properties of Priority Constructor Systems and prove their correctness using the transformation algorithm.
5.1 Functional Programming

Functions can be elegantly defined in functional programming languages. Such a function definition consists of some (recursive) equations. Function definitions in functional programming languages have a computational meaning. The equations of a function definition are regarded as rewrite rules, by orienting them from left to right, and they are tried in textual order from top to bottom.

Example 5.1.1 Definition of the factorial function in a functional programming language:

```haskell
fun Fac 0 = 1 |
    Fac n = n * Fac(n-1);
```

The alternative equations are separated by |. The second rule may only be used for applications of Fac to nonzero arguments. Thus Fac 1 reduces in one step to 1 * Fac(1-1). To ensure that the second rule is not applied wrongly, the argument of Fac is evaluated before a rule of Fac is applied. Thus the expression Fac(1-1) will be rewritten to Fac 0 and not to (1-1) * Fac((1-1)-1). Eventually Fac 1 reduces to 1, that can not be rewritten. Thus 1 is the normal form of Fac 1 (see Section 2.1).

The meaning of a function defined in this way is intuitively clear. Moreover using the textual order to decide which rule should be applied is natural and results in a deterministic computational behavior.

In a type system with inductive types we could define such a function using the elimination constant (recursor) of the inductively defined natural numbers.

Example 5.1.2 Recall the definition of the inductive type Nat of Example 4.2.6. Assume we have defined multiplication by a function Mul. We can define the factorial function as:

```haskell
Fac(n) = Elim(Nat,Nat,n)(S O,\lambda m: Nat.\lambda f: Nat. Mul(Sm, f))
```

This function has the following reduction behaviour:

```haskell
Fac(S O) \rightarrow \lambda p: Nat. (\lambda m: Nat. \lambda f: Nat. Mul(Sm, f)) p ((\lambda n': Nat. Fac(n')) p) O
\rightarrow beta Mul(S O, Fac(O))
```

If we compare this definition with the previous one, we notice that it is less readable. Furthermore the reduction behavior of the first function is more natural than that of the second function.

The pattern matching formalism of functional programming languages allows us to inspect all arguments at any level of nesting at the same time. But recursors of inductive types only allow us to inspect one argument at the same time and only one level deep. Therefore functions, that require inspecting several arguments or inspecting one argument at several levels, have nested definitions in type systems with inductive types. The pattern matching formalism does not impose restrictions on recursive calls, whereas recursors of inductive types only allow to use one argument for recursion and require that the recursive calls operate on proper subterms of that argument. Definitions with nested occurrences of recursors are hard to understand. This is illustrated by the following example:
Example 5.1.3 Definition of ≤ on natural numbers using recursors:

\[
\begin{align*}
Leq(m, n) &= \text{Elim}(\text{Nat}, \text{Nat} \to \text{Bool}, m)\{\lambda n'.\text{Nat}.\text{True}, \\
&\quad \lambda n_1:\text{Nat}.\lambda l_1:\text{Nat} \to \text{Bool}.\lambda n_2:\text{Nat} \cdot \text{Elim}(\text{Nat}, \text{Bool}, n_1)\{\text{False}, \\
&\quad \lambda n_2:\text{Nat}.\lambda l_2:\text{Bool}.l_1\} n_2\} n
\end{align*}
\]

An equivalent definition using pattern matching:

```ml
fun Leq 0 n = True |
  Leq (S m) 0 = False |
  Leq (S m) (S n) = Leq m n;
```

Obviously the latter definition is more readable than the former definition.

In the functional programming language ML ([29]) only total functions are allowed. This requirement could lead to long function definitions, if we would not have a priority on the rules of function definitions. As the rules of a function in ML have a priority according to their textual order, one can focus on the interesting cases and use one default rule, that handles the remaining cases.

Example 5.1.4 For instance, a function that tests whether its argument is a list containing only a zero can be specified in ML as follows:

```ml
fun iszeronil [0] = True |
    iszeronil _ = False;
```

The argument _ denotes an arbitrary variable and is used to indicate that the argument is not needed for the result of the computation.

A definition of this function as a Term Rewriting System would require four rules, as the three cases that are not treated by the first rule, must be specified.

Remark 5.1.5 The demands imposed on the left-hand sides of the rules of ML function definitions, and the use of a rewrite strategy that tries the rules according to their textual order, enables an efficient implementation for computing the results of applications of functions to their arguments.

As the rules are linear, no equality tests are needed to compare subterms for determining whether a rule may be applied are not. Only the constructors in the left-hand sides of the rules of a function impose restrictions on the arguments for applications of that function.

Therefore it is sufficient to step-wise evaluate the arguments on positions, that have a non-variable pattern in the left-hand side, until a term with starting with constructor symbol is obtained. If this symbol is different from the head symbol of the pattern on the same position in the left-hand side of the rule, then this rule is not applicable and the next rule can be tried. Otherwise there are two options. If all non-variable patterns have been tried then the term is a redex for the rule. Otherwise try the next non-variable pattern.
5.2 Constructor Systems

In this section we will describe a class of Term Rewriting Systems in which we have two kinds of function symbols: constructors without a computational meaning, and defined symbols with a computational meaning. In Constructor Systems the left-hand side of each rewrite rule is an application of a defined symbol to patterns. A pattern is a term built from constructors and variables only. A similar restriction is imposed on the rules of a function definition in ML. The separation of defined symbols and constructors gives Constructor Systems a simple semantics.

Definition 5.2.1 An $S$-sorted Constructor System is an $S$-sorted TRS($\Sigma, R$) with the property that $\Sigma$ can be divided into disjoint sets $D$ and $C$ such that every left-hand side $F(t_1, \ldots, t_n)$ of a rewrite rule of $R$ satisfies $F \in D$ and $t_1, \ldots, t_n \in T(C, V)$.

A pattern is an element of $T(C, V)$. A value is a pattern in which no variables occur. We will call a rule with left-hand side $F(t_1, \ldots, t_n)$ an $F$-rule. To emphasize the distinction between constructors and defined symbols we will write CS($D, C, R$).

Example 5.2.2 The TRS for Choice in Example 2.2.25 is a Constructor System. It has one defined symbol Choice; all other function symbols are constructors. The arguments in the left-hand sides of the rules for the defined symbol Choice are patterns. The following TRS is not a constructor system.

```
sort M
func 0: M x M -> M
      Plus: M x M -> M
rule Plus(x,0) -> x
      Plus(x,Plus(y,z)) -> Plus(Plus(x,y),z)
```

If we could divide the signature into defined symbols $D$ and constructors $C$, then certainly Plus should be in $D$ because every left-hand side of the rules is an application of Plus. But in the last rule Plus occurs inside an argument on the left-hand side, which is not allowed in a constructor system.

Each function definition in ML should be exhaustive, this means that such a function should be defined for all possible arguments. We can define this notion for defined symbols of Constructor Systems as follows:

Definition 5.2.3 Consider an $S$-sorted CS($D, C, R$). A defined symbol $F \in D$ exhaustively defined, if for each application to values $v_1, \ldots, v_n$, some rule $r \in R$ is applicable for $F(v_1, \ldots, v_n)$.

Example 5.2.4 The defined symbol Length of Example 2.2.12 is exhaustively defined. We can add a function Nth, that computes the n-th element of a list, to the signature of lists as follows:
\begin{tabular}{|l|}
\hline
\textbf{def} \text{Nth}: \text{nat} \times \text{list} \rightarrow \text{nat} \\
\hline
\textbf{rule} \text{Nth}(0,\text{Cons}(x,y)) \rightarrow x \\
\text{Nth}(S(n),\text{Cons}(x,y)) \rightarrow \text{Nth}(n,y) \\
\hline
\end{tabular}

In a specification of a constructor system we specify the defined symbols behind the keyword \textbf{def} and the constructors behind the keyword \textbf{cons}. The symbol \texttt{Nth} is \textit{not} exhaustively defined, because none of the rules is applicable for \texttt{Nth}(0,\texttt{Nil}).

\textbf{Remark 5.2.5} Notice that for \textit{finite} Constructor Systems, that are specified using the method described above, \textit{exhaustively definedness} is \textit{decidable}.

In Constructor Systems rules can interfere only at the root, that is the left hand side of a rule can only be unifiable with the lhs of a rule and not with a proper subterm of it, because a pattern is not unifiable with a left-hand side of a rule. Therefore a simpler condition than weak orthogonality is sufficient to guarantee that a Constructor System is confluent.

\textbf{Definition 5.2.6} Let $\Sigma$ be an $S$-sorted signature. An $S$-sorted TRS($\Sigma, R$) is \textit{weakly head-orthogonal}, if $(\Sigma, R)$ is left-linear and all critical pairs obtained from unification of the left-hand sides of two rules are trivial.

\textbf{Example 5.2.7} The TRS of Example 5.2.2 is weakly head-orthogonal, because no critical pair is obtained from unification of the left-hand sides of its rules. Notice that this TRS is not weakly orthogonal, because it has a non-trivial critical pair $(\text{Plus}(\text{Plus}(x,y), 0), \text{Plus}(x,y))$ (obtained by superposition of the first rule on the last one). The TRS of Example 2.2.25 is not weakly head-orthogonal.

The notion of ‘weak head-orthogonality’ is less restrictive than weak orthogonality. In the next definition we will introduce the notion of ‘\textit{almost orthogonality’}, that is more restrictive than weak orthogonality. However, these notions are equivalent for Constructor Systems. Before we prove this, we first establish some simple properties of Constructor Systems.

\textbf{Definition 5.2.8} Let $\Sigma$ be an $S$-sorted signature. A TRS($\Sigma, R$) is \textit{almost orthogonal}, if $(\Sigma, R)$ is left-linear and all critical pairs are trivial and are obtained from unification of the left-hand sides of two rules.

\textbf{Lemma 5.2.9} Consider an $S$-sorted CS($\mathcal{D}, \mathcal{C}, R$). Let $F \in \mathcal{D}$, $q, p_1, \ldots, p_n \in \mathcal{T}(\mathcal{C}, V)$. If $q \notin V$, then $F(p_1, \ldots, p_n)$ is not unifiable with $q$.

\textbf{Proof}
Assume $q \notin V$. We must have $q = C(q_1, \ldots, q_m)$ for some $C \in \mathcal{C}$, and $q_1, \ldots, q_m \in \mathcal{T}(\mathcal{C}, V)$. For each substitution $\sigma$, we have $F(p_1, \ldots, p_n)_{\sigma} = F(p'_{1}, \ldots, p'_n) \neq C(q'_{1}, \ldots, q'_m) = C(q_1, \ldots, q_m)_{\sigma}$, because $C$ and $\mathcal{D}$ are disjoint. Thus $F(p_1, \ldots, p_n)$ is not unifiable with $q$. \Box

\textbf{Lemma 5.2.10} Consider an $S$-sorted CS($\mathcal{D}, \mathcal{C}, R$).
1. No critical pair is obtained from superposition of a rule on itself.

2. Let $F, G \in \mathcal{D}$, $p_1, \ldots, p_n, q_1, \ldots, q_m \in \mathcal{T}(\mathcal{C}, V)$. If $F(p_1, \ldots, p_n)$ is unifiable with a non-variable subterm of $G(q_1, \ldots, q_m)$, then $F(p_1, \ldots, p_n)$ and $G(q_1, \ldots, q_m)$ are unifiable.

3. All critical pairs are obtained from unification of the left-hand sides of two rules.

Proof

1. We have $l = F(p_1, \ldots, p_n)$ with $F \in \mathcal{D}$ and $p_1, \ldots, p_n \in \mathcal{T}(\mathcal{C}, V)$, because $l \rightarrow r \in R$. Using the previous lemma we have that $l$ is not unifiable with some $p_i$ (after renaming variables).

2. Assume $F(p_1, \ldots, p_n)$ is unifiable with a subterm $t$ of $G(q_1, \ldots, q_m)$ with $t \notin V$. The term $t$ can not be a proper subterm of $G(q_1, \ldots, q_m)$, by the previous lemma. Thus $t = G(q_1, \ldots, q_m)$.

3. Assume we have obtained a critical pair by superposition of $l_1 \rightarrow r_1 \in R$ on $l_2 \rightarrow r_2 \in R$. We have $l_1 = F(p_1, \ldots, p_n)$ and $l_2 = G(q_1, \ldots, q_m)$ for some $F, G \in \mathcal{D}$, $p_1, \ldots, p_n, q_1, \ldots, q_m \in \mathcal{T}(\mathcal{C}, V)$, because $(\mathcal{C}, \mathcal{C}, R)$ is a constructor system. By 2. we obtain that $F(p_1, \ldots, p_n)$ is not unifiable with a non-variable proper subterm of $G(q_1, \ldots, q_m)$. Thus $l_1$ and $r_1$ must be unifiable.

Proposition 5.2.11 For Constructor Systems the following notions are equivalent:

1. almost orthogonal.

2. weakly orthogonal.

3. weakly head-orthogonal.

Proof

1. $\Rightarrow$ 2. Let $\text{CS}(D, C, R)$ be an almost orthogonal Constructor System. Then every rule $r \in R$ is left-linear, and all critical pairs are trivial.

2. $\Rightarrow$ 3. Let $\text{CS}(D, C, R)$ be a weakly orthogonal Constructor System. Then every rule $r \in R$ is left-linear, and all critical pairs obtained from unification of the left-hand sides of two rules are trivial.

3. $\Rightarrow$ 1. Let $\text{CS}(D, C, R)$ be weakly head-orthogonal. Then every rule $r \in R$ is left-linear, and all critical pairs obtained from unification of the left-hand sides of two rules are trivial. Using Lemma 5.2.10 we know that each critical pair is obtained from unification of the left-hand sides of two rules.

We will present a simplified version of the lexicographical path order for proving that a Constructor System is strongly normalizing.
Definition 5.2.12 Let \( > \) be a binary relation on \( \Sigma \)-terms. The *lexicographical extension* of \( > \) over finite sequences of terms of length \( n \), denoted by \( >_{\text{lex}} \), is defined as follows: Let \( k < n \). if \( t_{k+1} > u_1 \) then \( t_1, \ldots, t_n >_{\text{lex}} t_1, \ldots, t_k, u_1, \ldots, u_{n-k} \).

We introduce relations 'structurally smaller' on two terms and a 'argument decreasing' on rules. A term \( t \) is structurally smaller than a term \( u \), if it can be obtained from \( u \) by repeatedly replacing a subterm by one of its proper subterms. Roughly speaking, a rule is argument decreasing if all its recursive calls are on structurally smaller arguments (regarded as a sequence of terms). More precisely:

Definition 5.2.13 Let \( \Sigma \) be an \( S \)-sorted signature.

1. \( t \) is *structurally smaller* than \( u \), notation \( t <_s u \), is defined by:
   
   (a) \( t_i <_s F(t_1, \ldots, t_n) \), for \( i \leq n \).
   
   (b) If \( t_i <_s u \), then \( F(t_1, \ldots, t_n) <_s F(t_1, \ldots, t_{i-1}, u, t_{i+1}, \ldots, t_n) \) for \( 1 \leq i \leq n \).
   
   (c) If \( t <_s u \) and \( u <_s w \) then \( t <_s w \).

2. Let \( F \in \Sigma \) be an \( n \)-ary function symbol. A term \( u \) is *argument decreasing* for \( F(t_1, \ldots, t_n) \), if \( u_1, \ldots, u_n(\leq)_\text{lex} t_1, \ldots, t_n \) for each occurrence \( F(u_1, \ldots, u_n) \) of \( F \) in \( u \). A rule \( F(t_1, \ldots, t_n) \rightarrow r \) is *argument decreasing*, if \( r \) is argument decreasing for \( F(t_1, \ldots, t_n) \). A TRS(\( \Sigma, R \)) is *argument decreasing* if all rules \( r \in R \) are argument decreasing.

3. Let \( \triangleright \) be a strict partial order on \( \Sigma \). A term \( t \) is *bounded by* \( F \), if for each function symbol \( G \neq F \) in \( t \) we have \( F \triangleright G \). A rule \( F(t_1, \ldots, t_n) \rightarrow r \) is *bounded*, if \( r \) is bounded by \( F \). A TRS(\( \Sigma, R \)) is *bounded* if all rules are bounded.

4. Let \( \mathcal{D}, \mathcal{C} \) be disjoint \( S \)-sorted signatures. Let \( \triangleright \) be a binary relation on \( \mathcal{D} \). We extend it to a binary relation \( \triangleright_C \) on \( \mathcal{D} \cup \mathcal{C} \) as follows:

   (a) \( F \triangleright_C C \), for \( F \in \mathcal{D}, C \in \mathcal{C} \).

   (b) \( F \triangleright_C G \), if \( F \triangleright G \) for \( F, G \in \mathcal{D} \).

In the following lemma we show that we can simplify the verification of \( >_{\text{ipo}} \) (see Definition 2.2.21).

Lemma 5.2.14 Let \( \Sigma \) be an \( S \)-sorted signature. Let \( \triangleright \) be strict partial order on \( \Sigma \).

1. If \( t <_s u \) then \( t <_{\text{ipo}} u \), for \( t, u \in \mathcal{T}(\Sigma, \mathcal{V}) \).

2. If \( v \in \text{var}(t) \) then \( t >_{\text{ipo}} v \) or \( t = v \).

3. Let \( F \in \Sigma \). If \( u \) is bounded by \( F \), \( \text{var}(u) \subseteq \text{var}(F(t_1, \ldots, t_n)) \), and \( u \) is argument decreasing for \( F(t_1, \ldots, t_n) \), then \( F(t_1, \ldots, t_n) >_{\text{ipo}} u \).
Proof

1. Induction on the proof of \( t <_s u \).

   **Case 1:** \( t_i <_s F(t_1, \ldots, t_n) \) \( (i \leq n) \). By definition \( t_i <_{lpo} F(t_1, \ldots, t_n) \).

   **Case 2:** \( F(t_1, \ldots, t_n) <_s F(t_1, \ldots, t_{i-1}, u, t_{i+1}, \ldots, t_n) \), because \( t_i <_s u \) for \( 1 \leq i \leq n \).

      By the induction hypothesis we have \( t_i <_{lpo} u \).

      Thus we have \( F(t_1, \ldots, t_n) <_{lpo} F(t_1, \ldots, t_{i-1}, u, t_{i+1}, \ldots, t_n) \).

   **Case 3:** \( t <_s w \), because \( t <_s u \) and \( u <_s w \). By the induction hypothesis we have \( t <_{lpo} u \) and \( u <_{lpo} w \). As \( <_{lpo} \) is transitive, we have \( t <_{lpo} w \).

2. Structural induction on \( t \).

   **Base case:** \( t = w \in V \). Assume \( v \in \text{var}(t) \). Then \( t = v \).

   **Induction step:** \( t = F(t_1, \ldots, t_n) \). Assume \( v \in \text{var}(F(t_1, \ldots, t_n)) \). Then for some \( i \leq n : v \in \text{var}(t_i) \). By the induction hypothesis we have \( t_i >_{lpo} v \) or \( t_i = v \).

   Thus by definition \( F(t_1, \ldots, t_n) >_{lpo} u \).

3. Structural induction on \( u \).

   **Base case:** \( u = v \in V \). Assume \( \text{var}(u) \subseteq \text{var}(F(t_1, \ldots, t_n)) \). Then by the previous we have \( F(t_1, \ldots, t_n) >_{lpo} u \).

   **Induction step:** \( u = G(u_1, \ldots, u_m) \). Assume \( u \) is bounded by \( F \) and \( u \) is argument decreasing for \( F(t_1, \ldots, t_n) \). We have \( u_i \) is bounded by \( F \), \( \text{var}(u_i) \subseteq \text{var}(F(t_1, \ldots, t_n)) \) and \( u_i \) is argument decreasing for \( F(t_1, \ldots, t_n) \) \( (i \leq n) \). By the induction hypothesis we obtain \( F(t_1, \ldots, t_n) >_{lpo} u_i \) \( (i \leq n) \). Either \( F = G \) or \( F > G \).

   **Case** \( F = G \). We have \( u_1, \ldots, u_m( <_s \) \( \text{lex} t_1, \ldots, t_n \) and thus \( t_1, \ldots, t_n( >_{lpo} \) \( \text{lex} u_1, \ldots, u_n \). By definition \( F(t_1, \ldots, t_n) >_{lpo} u \).

   **Case** \( F > G \). By definition \( F(t_1, \ldots, t_n) >_{lpo} u \).

\[ \square \]

The first part of the following lemma is needed in the next proposition. Combining the second part of this lemma and the next proposition gives a simple method for using the lexicographical path order for proving that a Constructor System is strongly normalizing.

**Lemma 5.2.15** Let \( D, C \) be disjoint \( S \)-sorted signatures. Let \( \triangleright \) be a binary relation on \( D \).

1. If \( \triangleright \) is a strict partial order then \( \triangleright_C \) is a strict partial order.

2. If \( \triangleright \) is well-founded, then \( \triangleright_C \) is well-founded.

**Proof**
1. Assume $\triangleright$ is a strict partial order. We have $C \not\subset C$, for each $C \in C$. For each $F \in D$ we have $F \not\subset F$, because $\triangleright$ is irreflexive. Thus $\triangleright_C$ is irreflexive.

Assume $F_1 \triangleright_C F_2$. Then $F_1 \in D$. If $F_2 \in C$, then $F_2 \not\subset F_1$. If $F_2 \in D$, then $F_2 \not\subset F_1$ because $\triangleright$ is anti-symmetric. Thus $\triangleright_C$ is anti-symmetric.

Assume $F_1 \triangleright_C F_2$ and $F_2 \triangleright_C F_3$. Then $F_1, F_2 \in D$. If $F_3 \in C$, then $F_1 \triangleright_C F_3$. If $F_3 \in D$, then $F_1 \triangleright_C F_3$, because $\triangleright$ is transitive. Thus $\triangleright_C$ is transitive.

Thus $\triangleright_C$ is a strict partial order.

2. Assume we have an infinite sequence $F_1 \triangleright_C F_2 \triangleright_C F_3 \triangleright_C \ldots$. We must have $F_i \in D$ for all $i$, as $C \not\subset F$, for $C \in C$. But then we have $F_1 \triangleright F_2 \triangleright F_3 \triangleright \ldots$. □

Now we can prove a proposition, that simplifies the use of the lexicographical path order for proving strong normalization for Constructor Systems (see Theorem 2.2.23).

**Proposition 5.2.16** Let $(D, C, R)$ be an $S$-sorted Constructor System. Assume $\triangleright$ is a strict partial order on $D$. If a rule $l \rightarrow r \in R$ is argument decreasing and is $\triangleright_C$-bounded then $l \triangleright_{lpo} r$.

**Proof**

Assume $F(t_1, \ldots, t_n) \rightarrow r \in R$ is argument decreasing and is $\triangleright_C$-bounded by $F$. Then $r$ is argument decreasing for $F(t_1, \ldots, t_n)$, and $r$ is $\triangleright_C$-bounded by $F$. By Lemma 5.2.14 we obtain $F(t_1, \ldots, t_n) \triangleright_{lpo} r$, as $\text{var}(r) \subseteq F(t_1, \ldots, t_n)$. □

**Corollary 5.2.17 (termination)** Let $(D, C, R)$ be an argument decreasing $S$-sorted Constructor System. Assume $\triangleright$ is a well-founded strict partial order on $D$. If $(D, C, R)$ is $\triangleright_C$-bounded then $(D, C, R)$ is strongly normalizing.

**Proof**

Assume $\triangleright$ is a well-founded strict partial order on $D$. By Lemma 5.2.15 $\triangleright_C$ is a well-founded strict partial order. If $(D, C, R)$ is $\triangleright_C$-bounded then by Proposition 5.2.16 we have $l \triangleright_{lpo} r$ for all rules $r \in R$. Thus by Theorem 2.2.23 we obtain that $(D, C, R)$ is strongly normalizing. □

### 5.3 Priority Constructor Systems

The last step for obtaining a formalization of first order functions in ML is to let the textual order of the rules determine which rule is applied when several rules are applicable. In this way a precise semantics can be given to ambiguous rule systems. In general this can be done by adding a priority to the rules. The semantics of a priority rewrite system can be problematic, because a rule may only be applied if no other rule with higher priority can become applicable. A general description of priority rewrite systems can be found in [2] and [3].

**Definition 5.3.1** An $S$-sorted Priority Constructor System is a pair consisting of an $S$-sorted left-linear CS$(D, C, R)$ and a strict partial order $>$ on $R$. We will write PCS$(D, C, R, >)$. 
Example 5.3.2 Priority Constructor Systems are well suited for defining equality:

<table>
<thead>
<tr>
<th>sort</th>
<th>bool, nat</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td>True: bool</td>
</tr>
<tr>
<td></td>
<td>False: bool</td>
</tr>
<tr>
<td></td>
<td>0: nat</td>
</tr>
<tr>
<td></td>
<td>S: nat → nat</td>
</tr>
<tr>
<td>def</td>
<td>Eq: nat × nat → bool</td>
</tr>
<tr>
<td>rule</td>
<td>Eq(0, 0) → True</td>
</tr>
<tr>
<td></td>
<td>&gt; Eq(S(w), S(x)) → Eq(w, x)</td>
</tr>
<tr>
<td></td>
<td>&gt; Eq(y, z) → False</td>
</tr>
</tbody>
</table>

By separating rules with > we emphasize that the rule in front of > has priority over the rule(s) behind >.

The intended meaning of the order on the rules is, that a rule may only be applied if no rule with higher priority could ever become applicable after rewriting the arguments. Unfortunately, this is not decidable in general. Therefore we introduce a decidable condition, that guarantees that no rule is ever applied incorrectly. It is based on the fact that there are no rewrite rules for constructors, and thus rewriting an application of a constructor will yield an application of the same constructor.

Definition 5.3.3 Let \((D, C, R)\) be an \(S\)-sorted Constructor System.

1. Two terms \(t, u\) are strongly incompatible, notation \(t \#_S u\), if \(t = C_1(t_1, \ldots , t_m), u = C_2(u_1, \ldots , u_n)\), for \(C_1, C_2 ∈ C\) and \(m, n ≥ 0\), and either
   
   (a) \(C_1 \neq C_2\), or
   
   (b) i. \(C_1 = C_2\), and
      
      ii. \(∃i ≤ m t_i \#_S u_i\).

2. Two terms \(t, u\) have a strongly incompatible argument, denotation \(t \#_S u\), if \(t = F(t_1, \ldots , t_n), u = F(u_1, \ldots , u_n)\), and \(∃i ≤ n t_i \#_S u_i\), for some \(F ∈ D\).

Example 5.3.4 In the PCS of Example 5.3.2 we have: \(0 \#_S S(w)\), and \(S(0) \#_S S(S(w))\), but not \(0 \#_S w\); also we have \(Eq(S(y), 0) \#_S Eq(0, 0)\), and \(Eq(S(y), 0) \#_S Eq(S(w), S(x))\), but not \(Eq(y, 0) \#_S Eq(0, 0)\).

Definition 5.3.5 Let PCS\((D, C, R, >)\) be an \(S\)-sorted Priority Constructor System.

1. Let \(F ∈ D\). Let \(l_1 → r_1 ∈ R\) be an \(F\)-rule. A redex \(C[l]_1\) is enabled, if \(l_2 \#_S l_1\), for each \(F\)-rule \(l_2 → r_2 ∈ R\), such that \(l_2 → r_2 > l_1 → r_1\).

2. A reduction step \(C[l] → C[r]\) is enabled, if the redex \(C[l]\) is enabled. We denote this by \(C[l] →_c C[r]\). The reflexive transitive closure of \(→_c\) is denoted by \(→_c\).
Example 5.3.6 Enabled reduction steps of the PCS of Example 5.3.2 are:

\[
\begin{align*}
\text{Eq}(0,0) & \rightarrow e \quad \text{True}, \\
\text{Eq}(S(x),0) & \rightarrow e \quad \text{False}, \\
\text{Eq}(S(0),S(0)) & \rightarrow e \quad \text{Eq}(0,0).
\end{align*}
\]

The reduction step \(\text{Eq}(x,0) \rightarrow \text{False}\) is not enabled, because \(x\) and 0 are not strongly incompatible.

Lemma 5.3.7 Let \(\text{CS}(D,C,R)\) be an \(S\)-sorted Constructor System.

1. If \(t\#_\sigma u\) then \(t'\#_\sigma u\), for each substitution \(\sigma\).

2. If \(t\#_\sigma u\) then \(t\) and \(u\) are not unifiable.

Proof

1. Induction on the proof of \(t\#_\sigma u\).

   **Base case:** \(C(t_1,\ldots,t_m)\#_\sigma C'(u_1,\ldots,u_n)\), because \(C \neq C'\), for \(C, C' \in C\). Then also
   \[C(t_1,\ldots,t_m)^\sigma \#_s C'(u_1,\ldots,u_n)\].

   **Induction step:** \(C(t_1,\ldots,t_n)\#_\sigma C(u_1,\ldots,u_n)\), because \(t_i\#_\sigma u_i\) for some \(i \leq n\) and
   \(C \in C\). By the induction hypothesis we have \(t_i^\sigma \#_s u_i\).
   Thus we obtain \(C(t_1,\ldots,t_n)^\sigma \#_s C(u_1,\ldots,u_n)\).

2. Induction on the proof of \(t\#_\sigma u\).

   **Base case:** \(C(t_1,\ldots,t_m)\#_\sigma C'(u_1,\ldots,u_n)\), because \(C \neq C'\), for \(C, C' \in C\). We have
   \[C(t_1,\ldots,t_m)^\sigma = C(t_1^\sigma,\ldots,t_m^\sigma) \neq C'(u_1^\sigma,\ldots,u_n^\sigma) = C(u_1,\ldots,u_n)^\sigma\].

   **Induction step:** \(C(t_1,\ldots,t_n)\#_\sigma C(u_1,\ldots,u_n)\), because \(t_i\#_\sigma u_i\) for some \(i \leq n\) and
   \(C \in C\). By the induction hypothesis \(t_i\) and \(u_i\) are not unifiable, and thus
   \(C(t_1,\ldots,t_n)\) and \(C(u_1,\ldots,u_n)\) are not unifiable. \(\square\)

In general fewer (and never more) rules are needed to define a function in a Priority Constructor System, than for a definition of the same (in mathematical sense) function in a Constructor System.

Example 5.3.8 The definition of equality on natural numbers in a Constructor System requires four rules:

<table>
<thead>
<tr>
<th>rule</th>
<th>Eq(0,0) → True</th>
<th>Eq(S(w),S(x)) → Eq(w,x)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Eq(0,S(y)) → False</td>
<td>Eq(S(z),0) → False</td>
</tr>
</tbody>
</table>
The cases that are not treated by the first two rules are made explicit by the last two rules. In the PCS of Example 5.3.2 these remaining cases are treated by one default rule.

More in general, the definition of the equality function on a sort with \( n (n > 1) \) constructors requires \( n + 1 \) rules in a PCS, and \( n^2 \) rules in a CS. Both definitions require a rule for each constructor \( C \) stating that two terms \( C(t_1, \ldots, t_n) \) and \( C(u_1, \ldots, u_n) \) are equal if \( t_1 \) and \( u_1 \) are equal, \ldots, and \( t_n \) and \( u_n \) are equal. This costs \( n \) rules (one rule for each constructor). In a PCS we can handle all other cases by one default rule, since all terms that do not match the other rules are not equal. But in a CS, we must explicitly mention all the remaining cases. This costs \( n - 1 \) rules for each constructor, thus \( n \cdot (n - 1) \) rules in total.

In fact, for each finite Priority Constructor System there exists a Constructor System whose reduction relation is the same as the enabled reduction relation of the PCS. This theoretical result has been shown in [25]. But it is not sufficient to know the existence of an equivalent CS for a given PCS. We want an efficient method (algorithm) that has as input a PCS and as output an equivalent CS. The main reason is that such an algorithm would allow us to formulate criteria that ensure that a PCS is confluent (and strongly normalizing and exhaustive), and prove their correctness. As we have described such criteria for Constructor Systems in the previous section, we would only have to prove that the result of applying the transformation algorithm for a PCS that satisfies certain conditions is a CS that meets those criteria. Moreover such a transformation algorithm would allow us to use implementations of Term Rewriting Systems for Priority Constructor Systems.

**Transformation**

We will present an algorithm for transforming any PCS \((\mathcal{D}, \mathcal{C}, R, >)\) with a finite signature and finitely many rules into a CS \((\mathcal{D}, \mathcal{C}, R')\) such that the rewrite relation \( \rightarrow \) defined by \( R' \) is the same as the rewrite relation \( \rightarrow_e \) defined by \( R \) and \( > \). The idea behind the transformation is that a rule is replaced by its most general instances that do not interfere with any of the preceding rules. Since interference only depends on the left-hand sides of the rules, we do not need the right-hand sides of the rules to compute these instances. Because all left-hand sides of the rules are linear, the names of the variables play no role in determining the (enabled) reduction steps. Therefore the transformation algorithm will operate on terms in which all variables are replaced by the same new constant. The transformation algorithm is based on several algorithms that perform subtasks.

\[
\text{IncPat} \rightarrow \text{IncLhs} \rightarrow \text{todo}^F \rightarrow \text{new} \rightarrow \text{subs} \rightarrow \text{newrules} \rightarrow \text{transform} \rightarrow \text{precrules} \rightarrow \]

Figure 5.1: Dependency graph of the algorithms needed for the transformation

In the following definition we use definitions given in [24].
Definition 5.3.9 Let $\Sigma$ be an $S$-sorted signature.

1. Consider an extra constant $\Omega \notin \Sigma$, that has any sort $s \in S$ as sort. The set $\mathcal{T}(\Sigma \cup \{\Omega\}, V)$, also denoted by $\mathcal{T}_\Omega$, is called the set of $\Omega$-terms. By $t_0$ we indicate the $\Omega$-term obtained from a term $t$ by replacing each variable with $\Omega$. Let $W$ be a finite subset of $V$ and $u$ an $\Omega$-term. $\text{fresh}(W,u)$ indicates the term obtained from $u$ by replacing each $\Omega$ with a different variable not in $W$. Let $F \in \Sigma$, $\tau(F) = s_1 \times \ldots \times s_n \rightarrow s$, $t_i \in \mathcal{T}(\Sigma, V)_{s_i}$. $\Omega(F, i, t_i)$ denotes the $\Omega$-term $F(t_1, \ldots, t_n)$, such that $t_j = \Omega$ ($1 \leq j \neq i \leq n$).

2. The preorder $\succeq$ on $\mathcal{T}_\Omega$ is defined as follows:
   
   (a) $t \succeq \Omega$ for all $t \in \mathcal{T}_\Omega$,
   (b) $F(t_1, \ldots, t_n) \succeq F(u_1, \ldots, u_n)$ ($n \geq 0$) if $t_i \succeq u_i$ for $i = 1, \ldots, n$.

   We write $t \succ u$ if $t \succeq u$ and $t \neq u$.

3. Let $s \in S$. Let $t, u \in \mathcal{T}(\Sigma \cup \{\Omega\}, V)_s$ be $\Omega$-terms. $t$ and $u$ are compatible, denoted by $t \uparrow u$, if there exists some $\Omega$-term $r$ such that $r \succeq t$ and $r \succeq u$; otherwise $t$ and $u$ are incompatible, which is indicated by $t \# u$.

   The least upper bound for two compatible $\Omega$-terms $t$ and $u$ is denoted by $t \sqcup u$.

Remark 5.3.10 For each $\Omega$-term $t$ without variables we have $(\text{fresh}(W,t))_\Omega = t$.

Example 5.3.11 Take the signature of the Example 5.3.2. The $\Omega$-term $\text{Eq}(0, 0)$ is incompatible with $\text{Eq}(S(\Omega), S(\Omega))$. The $\Omega$-terms $\text{Eq}(0, \Omega)$ and $\text{Eq}(\Omega, S(\Omega))$ are compatible and have least upper bound $\text{Eq}(0, S(\Omega))$.

Lemma 5.3.12 Let $\Sigma$ be an $S$-sorted signature.

1. If $t \# u$ and $w \succeq u$, then $t \# w$.
2. If $t \# u$ and $t \uparrow u$, then $t \# w \sqcup u$.
3. $\exists i \leq n \ t_i \# u_i \iff F(t_1, \ldots, t_n) \# F(u_1, \ldots, u_n)$, for $1 \leq n$.

Proof

1. Assume $t \# u$ and $w \succeq u$. Now $t \uparrow w$ would contradict $t \# u$, thus $t \# w$.

2. $w \sqcup u \succeq u$, thus $t \# u$ implies $t \# w \sqcup u$.

3. By the definition of $\succeq$ and $\uparrow$ we have: $F(t_1, \ldots, t_n) \uparrow F(u_1, \ldots, u_n)$ if and only if $t_i \uparrow u_i$ ($i \leq n$). Thus $F(t_1, \ldots, t_n) \# F(u_1, \ldots, u_n)$ if and only if $t_i \# u_i$ for some $i \leq n$. $\square$
In Constructor Systems the notions 'strongly incompatible' for patterns and 'incompatible' for \(\Omega\)-terms of patterns are equivalent. We will use this fact to work with \(\Omega\)-patterns in the algorithm for transforming a PCS into an equivalent CS.

**Lemma 5.3.13** Let \(CS(D,C,R)\) be an \(S\)-sorted Constructor System. \(p_\Omega \n# q_\Omega \iff p \n# q\), for \(p,q \in T(C,V)\).

**Proof** Structural induction on \(p\) and \(q\).
We will only treat the case \(p = C(p_1,\ldots,p_n)\) and \(q = C'(q_1,\ldots,q_m)\).

- **Case** \(C \neq C'\). We have \(C(p_1,\ldots,p_n) \n# C(q_1,\ldots,q_m)\) and \(C(p_1,\ldots,p_n) \Omega \n# C'(q_1,\ldots,q_m)\).

- **Case** \(C = C'\). By definition \(C(p_1,\ldots,p_n) \n# C(q_1,\ldots,q_n)\) is equivalent to \(p_i \n# q_i\) for some \(i \leq n\). By the induction induction hypothesis we have \((p_i)\Omega \n# (q_i)\Omega\) if, and only if \(p_i \n# q_i\) for each \(i \leq n\). By Lemma 5.3.12 we have \(C(p_1,\ldots,p_n) \Omega \n# C(q_1,\ldots,q_n)\Omega\) if and only if \((p_i)\Omega \n# (q_i)\Omega\) for some \(i \leq n\). Thus we have \(C(p_1,\ldots,p_n) \n# C(q_1,\ldots,q_n)\Omega\) if and only if \(C(p_1,\ldots,p_n) \n# C(q_1,\ldots,q_n)\).

A term is a redex of a left-linear rule, if and only if it is an upper bound of the \(\Omega\)-term of the left-hand side of that rule. This observation will be used to show the correctness of our transformation algorithm.

**Lemma 5.3.14** Let \(\Sigma\) be an \(S\)-sorted signature.

1. If \(u\) is linear and \(t \geq u_\Omega\) then \(u^\sigma = t\), for some substitution \(\sigma\).

2. If \(u^\sigma = t\) then \(t \geq u_\Omega\).

**Proof**

1. Proof with structural induction on \(u\).

- **First case**: \(u = v \in V\). Take \(\sigma\) with \(\sigma(v) = t\).

- **Last case**: \(u = F(u_1,\ldots,u_n)\). If \(t \geq u_\Omega\) then \(t = F(t_1,\ldots,t_n)\) with \(t_i \geq (u_i)_\Omega\) \((i \leq n)\). By the induction hypothesis we have \(u_i^\sigma = t_i\) for substitutions \(\sigma_i\) \((i \leq n)\). As \(u\) is linear we can define \(\sigma\) such that \(\sigma(v) = \sigma_i(v)\), for all \(i \leq n\) and all \(v \in \text{var}(u_i)\). Then \(u^\sigma = t\).

2. Proof with structural induction on \(u\).

We will define a function 'IncPat', that for a given pattern, computes \(\Omega\)-terms that are incompatible with its \(\Omega\)-term. This function will be used to define a function 'IncLhs' that, for a given application of a defined symbol to patterns, computes \(\Omega\)-terms that are incompatible with its \(\Omega\)-term.

**Definition 5.3.15** Let \(CS(D,C,R)\) be an \(S\)-sorted Constructor System.
1. (a) \( \text{IncPat}(C(t_1, \ldots, t_n)) = \{ C'(\Omega, \ldots, \Omega) | C' \in C, \text{sort}(C') = s, C \neq C' \} \cup \{ (\Omega, (i, u)) | 1 \leq i \leq n, u \in \text{IncPat}(t_i) \} \), for \( C \in C \), with \( \tau(C) = s_1 \times \ldots \times s_n \rightarrow s \).

(b) \( \text{IncPat}(v) = \emptyset \), for \( v \in V \).

2. Let \( F \in D \), \( \tau(F) = s_1 \times \ldots \times s_n \rightarrow s \). Let \( p_1, \ldots, p_n \in T(C, V) \) (\( 1 \leq n \)),
\[
\text{IncLhs}(F(p_1, \ldots, p_n)) = \bigcup_{i=1}^{n} \{ \Omega(F, i, u) | u \in \text{IncPat}(p_i) \}.
\]

**Example 5.3.16** In the CS of Example 5.3.2 we have for instance:
\( \text{IncPat}(0) = \{ S(\Omega) \} \).
Every pattern that is strongly incompatible with 0 is of the form \( S(t) \), because \( \text{nat} \) has two constructors 0, \( S \), and moreover \( S(t) \geq S(\Omega) \).

Another example: \( \text{IncPat}(S(S(x))) = \{ 0, S(0) \} \).

We can compute \( \text{IncLhs} \) for the left-hand sides of the rules for \( \text{Eq} \):
\[
\begin{align*}
\text{IncLhs}(\text{Eq}(0, 0)) &= \{ \text{Eq}(S(\Omega), \Omega), \text{Eq}(\Omega, S(\Omega)) \} \\
\text{IncLhs}(\text{Eq}(S(w), S(x))) &= \{ \text{Eq}(0, \Omega), \text{Eq}(\Omega, 0) \} \\
\text{IncLhs}(\text{Eq}(y, z)) &= \emptyset.
\end{align*}
\]

Every term \( \text{Eq}(t, u) \) with \( t \ #_s 0 \) or \( u \ #_s 0 \) has a lower bound in \( \text{IncLhs}(\text{Eq}(0, 0)) \). Thus every enabled redex of the second or the third \( \text{Eq} \)-rule has a lower bound in \( \text{IncLhs}(\text{Eq}(0, 0)) \).

In the next lemma we show that \( \text{IncPat} \) and \( \text{IncLhs} \) compute \( \Omega \)-terms that are incompatible with the \( \Omega \)-terms of their arguments.

**Lemma 5.3.17** Let \( CS(D, C, R) \) be an \( S \)-sorted Constructor System.

1. \( \forall t \in \text{IncPat}(p) \ p_0 \#_s t \), for \( p \in T(C, V) \).
2. \( \forall t \in \text{IncLhs}(F(p_1, \ldots, p_n)) \ F(p_1, \ldots, p_n) \Omega \#_s t \).

**Proof**

1. Structural induction on \( p \).

   **Case** \( p = v \in V \). Trivial.

   **Case** \( p = C(p_1, \ldots, p_n) \). Let \( t \in \text{IncPat}(C(p_1, \ldots, p_n)) \).

   Either \( t = C'(\Omega, \ldots, \Omega) \) and \( C' \in C, C \neq C' \). Then \( C(p_1, \ldots, p_n) \#_s C'(\Omega, \ldots, \Omega) \).

   Or \( t = \Omega(C, i, u) \) where \( u \in \text{IncPat}(p_i) \). By the induction hypothesis \( p_i \#_s u \) holds, thus by Lemma 5.3.12 we have \( C(p_1, \ldots, p_n) \#_s \Omega(C, i, u) \).

2. By definition \( t = \Omega(F, i, u) \) for some \( u \in \text{IncPat}(p_i) \). Thus \( (p_i)_{\Omega} \#_s u \) and therefore by Lemma 5.3.12 \( F(p_1, \ldots, p_n) \Omega \#_s \Omega(F, i, u) \). \( \square \)
In the following lemma we show that all terms that IncPat and IncLhs compute the most general $\Omega$-terms that are incompatible with the $\Omega$-terms of their arguments.

**Lemma 5.3.18** Let $CS(\mathcal{D}, \mathcal{C}, R)$ be an $S$-sorted Constructor System. Let $t \in T(\mathcal{D} \cup \mathcal{C}, V)$.

1. If $p \#_s t$ then $t \succeq u$ for some $u \in \text{IncPat}(p)$, for $p \in T(\mathcal{C}, V)$.

2. If $F(p_1, \ldots, p_n) \#_s \text{t} \text{g} t$ then $t \succeq u$ for some $u \in \text{IncLhs}(F(p_1, \ldots, p_n))$.

**Proof**


   Case $p = v \in V$. Trivial.

   Case $p = C(p_1, \ldots, p_n)$. Because $C(p_1, \ldots, p_n) \#_s t$, we have $t = C'(q_1, \ldots, q_m)$ for some $C' \in C$.

   Case $C \neq C'$. We have $C'(\Omega_1, \ldots, \Omega) \in \text{IncPat}(C(p_1, \ldots, p_n))$ and $C'(q_1, \ldots, q_m) \succeq C'(q_i, \ldots, q_m)$.

   Case $C = C'$. We have $p_i \#_s q_i$, for some $i$. By the induction hypothesis there exists $u \in \text{IncPat}(p_i)$ that $q_i \succeq u$. Thus $C(q_1, \ldots, q_m) \succeq C(q_i, \ldots, q_m)$.

   By definition $\Omega(C, i, u) \in \text{IncPat}(C(p_1, \ldots, p_n))$.

2. Assume $F(p_1, \ldots, p_n) \#_s \text{t} \text{g} t$. Then we must have $t = F(t_1, \ldots, t_n)$ and $p_i \#_s t_i$, for some $i \leq n$ and terms $t_1, \ldots, t_n$. Thus for some $u \in \text{IncPat}(p_i)$ we have $t_i \succeq u$.

   Thus $\Omega(F, i, u) \in \text{IncLhs}(F(p_1, \ldots, p_n))$ and $F(t_1, \ldots, t_n) \succeq F(t_1, \ldots, t_n)$.

Now we will define a function ‘$\text{t} \text{o} \text{d} \text{o} \text{F}$’ that, given a sequence $\tilde{I}$ of application of $F$ to patterns, computes $\Omega$-terms that are incompatible with all $\Omega$-terms $(l_i)_\Omega$. This definition is based on ‘IncLhs’.

Using the definition of ‘$\text{t} \text{o} \text{d} \text{o} \text{F}$’ we will define a function ‘new’ that, given a sequence of applications $\tilde{I}$ of a defined symbol $F$ to patterns, and an application $m$ of $F$ to patterns, computes upper bounds of $m_\Omega$ that are incompatible with all $\Omega$-terms $(l_i)_\Omega$.

Finally we will define a function ‘subs’ that, given a sequence of applications $\tilde{I}$ of a defined symbol $F$ to patterns, and an application $m$ of $F$ to pattern, computes substitutions $\sigma$, such that $l_i \#_s \text{t} \text{o} \text{d} \text{o} \text{F} \sigma$.

**Definition 5.3.19** Let $(\mathcal{D}, \mathcal{C}, R)$ be an $S$-sorted Constructor System. Let $F \in \mathcal{D}$.

1. Let $l_1, \ldots, l_{k+1}$ be applications of $F$ to patterns.

   (a) $\text{t} \text{o} \text{d} \text{o} \text{F}(\epsilon) = \{ F(\Omega_1, \ldots, \Omega) \}$.

   (b) i. $\text{t} \text{d} \text{o} \text{F}(l_1, \ldots, l_{k+1}) = \{ t \in \text{t} \text{d} \text{o} \text{F}(l_1, \ldots, l_k) | (l_{k+1})_0 \#_s t \} \cup \{ u \cup t | u \in \text{IncLhs}(l_{k+1}), t \in \text{t} \text{d} \text{o} \text{F}(l_1, \ldots, l_k), (l_{k+1})_0 \uparrow t, u \uparrow t \}$

   ii. $\text{t} \text{o} \text{d} \text{o} \text{F}(l_1, \ldots, l_{k+1}) = \{ t \in \text{t} \text{d} \text{o} \text{F}(l_1, \ldots, l_{k+1}) | t \neq \text{t} \text{d} \text{o} \text{F}(l_1, \ldots, l_{k+1}) \}$. 

2. Let \( \vec{t} \) be a sequence of applications of \( F \) to patterns. Let \( m \) be an application of \( F \) to patterns.

   (a) \( \text{nw}(\vec{t}, m) = \{ m_{\Omega} \cup t | t \in \text{todo}^F(\vec{t}, m_{\Omega} \uparrow t) \} \); 

   (b) \( \text{new}(\vec{t}, m) = \{ t \in \text{nw}(\vec{t}, m) | t \not\in \text{nw}(\vec{t}, m) \} \).

3. Let \( \vec{t} \) be a sequence of applications of \( F \) to patterns. Let \( m \) be an application of \( F \) to patterns.

   \( \text{subs}(\vec{t}, m) = \{ \text{mgu}(m, \text{fresh}(\text{var}(m), u)) | u \in \text{new}(\vec{t}, m) \} \).

**Example 5.3.20** Let \((D, C, R, >)\) be the PCS of Example 5.3.2. Computation of todo for the left-hand sides of the rules for \( \text{Eq} \):

\[
\begin{align*}
\text{todo}^E(\text{Eq}(0, 0)) &= \{ \text{Eq}(S(\Omega), 0), \text{Eq}(0, S(\Omega)) \}; \\
\text{todo}^E(\text{Eq}(0, 0), \text{Eq}(S(w), S(x))) &= \{ \text{Eq}(S(\Omega), 0), \text{Eq}(0, S(\Omega)) \}; \\
\text{todo}^E(\text{Eq}(0, 0), \text{Eq}(S(w), S(x)), \text{Eq}(y, z)) &= \emptyset.
\end{align*}
\]

Computation of new for the left-hand sides of the rules for \( \text{Eq} \):

\[
\begin{align*}
\text{new}(\epsilon, \text{Eq}(0, 0)) &= \{ \text{Eq}(0, 0) \}; \\
\text{new}(\{ \text{Eq}(0, 0), \text{Eq}(S(w), S(x)) \}) &= \{ \text{Eq}(S(\Omega), S(\Omega)) \}; \\
\text{new}(\{ \text{Eq}(0, 0), \text{Eq}(S(w), S(x)), \text{Eq}(y, z) \}) &= \{ \text{Eq}(S(\Omega), 0), \text{Eq}(0, S(\Omega))y \}.
\end{align*}
\]

The following technical lemma is necessary to show that applying a substitution of \( \text{subs} \) to a linear application of a defined symbol \( F \) to patterns yields a linear application of \( F \) to patterns.

**Lemma 5.3.21** Let \((D, C, R)\) be an \( S \)-sorted Constructor System. Let \( F \in D \). Let \( \vec{t} \) be a sequence of linear applications of \( F \) to patterns. Let \( m \) be a linear application of \( F \) to patterns.

1. \( \forall t \in \text{todo}^F(\vec{t}) \exists t_1, \ldots, t_n \in C \cup \{ \Omega \} \) \( t = F(t_1, \ldots, t_n) \).

2. \( \forall t \in \text{new}(\vec{t}, m) \exists t_1, \ldots, t_n \in C \cup \{ \Omega \} \) \( t = F(t_1, \ldots, t_n) \).

3. \( \forall \sigma \in \text{subs}(\vec{t}, m) \) linear \( (m^\sigma) \land \exists p_1, \ldots, p_n \in C, V \) \( m^\sigma = F(p_1, \ldots, p_n) \).

**Proof**

1. Induction on length of \( \vec{t} \).

   *Case \(|\vec{t}| = 0\).* Trivial.

   *Case \(|\vec{t}| = i + 1\).* We will prove \( \forall t \in \text{td}^F(\vec{t}) \) \( t = F(t_1, \ldots, t_n) \) for some \( t_1, \ldots, t_n \in C \cup \{ \Omega \} \).

   Either \( t \in \text{todo}^F(t_1, \ldots, t_i) \) and \( (l_{i+1})_{\Omega} \# t \). By the induction hypothesis we have \( t = F(t_1, \ldots, t_n) \) for some \( t_1, \ldots, t_n \in C \cup \{ \Omega \} \).
Or \( t = s \cup u \) with \( s \in \text{IncLhs}((l_{i+1})_n), u \in \text{todo}^F(l_1, \ldots, l_i) \) such that \((l_{i+1})_n \uparrow u\) and \( s \uparrow u \). By the induction hypothesis we have \( u = F(u_1, \ldots, u_n) \) for some \( u_1, \ldots, u_n \in \mathcal{T}(\mathcal{C} \cup \{\Omega\}) \). We also have \( s = F(s_1, \ldots, s_n) \) for some \( s_1, \ldots, s_n \in \mathcal{T}(\mathcal{C} \cup \{\Omega\}) \). As \( s \cup u = F(s_1 \cup u_1, \ldots, s_n \cup u_n) \), and \( s_i \cup u_i \in \mathcal{T}(\mathcal{C} \cup \{\Omega\}) \) \( (i \leq n) \). Thus \( t = F(t_1, \ldots, t_n) \) for some \( t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C} \cup \{\Omega\}) \).

2. Let \( t \in \text{new}(\vec{l}, m) \). Then \( t = m_\Omega \cup t' \) with \( t' \in \text{todo}^F(\vec{l}) \). Thus \( t' = F(t_1, \ldots, t_n) \) for some \( t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C} \cup \{\Omega\}) \). Also we have \( m = F(p_1, \ldots, p_n) \) for some \( p_1 \ldots p_n \in \mathcal{T}(\mathcal{C}, V) \). Thus \((p_i)_\Omega \in \mathcal{T}(\mathcal{C} \cup \{\Omega\}) \) \( (i \leq n) \). The result follows, because \( m_\Omega \cup t' = F((p_1)_\Omega \cup t_1, \ldots, (p_n)_\Omega \cup t_n) \) and \((p_i)_\Omega \cup t_i \in \mathcal{T}(\mathcal{C} \cup \{\Omega\}) \) \( (i \leq n) \).

3. Let \( \sigma \in \text{subs}(\vec{l}, m) \). Then \( \sigma = \text{mgu}(m, \text{fresh}(\text{var}(m), u)) \) for some \( u \in \text{new}(\vec{l}, m) \). Let \( u' = \text{fresh}(\text{var}(m), u) \). Let \( \sigma = \text{mgu}(m, u') \). Because \( m \) and \( u' \) are linear and \( \text{var}(m) \cap \text{var}(u') = \emptyset \), \( m^\sigma \) must be linear. We have \( u' = F(u_1, \ldots, u_n) \) for patterns \( u_1, \ldots, u_n \). Using this and the definition of \( \sigma \) we obtain that \( m^\sigma = F(q_1, \ldots, q_n) \), for some patterns \( q_1, \ldots, q_n \).

The following lemma shows that \( \text{subs} \) can be used to compute instances of the left-hand side of a rule that must have an strongly incompatible argument with the left-hand sides of some other rules.

**Lemma 5.3.22** Let \((\mathcal{D}, \mathcal{C}, R)\) be an \( S \)-sorted Constructor System. Let \( F \in \mathcal{D} \). Let \( \vec{l} \) be a sequence of applications of \( F \) to patterns. Let \( m \) be an application of \( F \) to patterns.

1. \( \forall t \in \text{todo}^F(\vec{l}) \) \( \forall j \leq |\vec{l}| \) \((l_j)_\Omega \# \# t\).

2. \( \forall t \in \text{new}(\vec{l}, m) \) \( \forall j \leq |\vec{l}| \) \((l_j)_\Omega \# \# t\).

3. \( \forall \sigma \in \text{subs}(\vec{l}, m) \) \( \forall j \leq |\vec{l}| \) \( l_j \#_{\sigma} m^\sigma \).

**Proof**

1. Induction on \(|\vec{l}|\).

   **Case** \(|\vec{l}| = 0\). Trivial.

   **Case** \(|\vec{l}| = i + 1\). We will prove the property for \( \text{td}^F(\vec{l}) \). Assume \( t \in \text{td}^F(\vec{l}) \).

   Either \( t \in \text{todo}^F(l_1, \ldots, l_i) \) and \((l_{i+1})_\Omega \# t\). By induction \( \forall j \leq i \) \((l_j)_\Omega \# t\).

   Or \( t = s \cup u \) with \( s \in \text{IncLhs}((l_{i+1})_n), u \in \text{todo}^F(l_1, \ldots, l_i) \) such that \((l_{i+1})_n \uparrow u\) and \( s \uparrow u \). Because \((l_{i+1})_n \# s\) and \( \forall j \leq i \) \((l_j)_\Omega \# u\) we have by Lemma 5.3.12 \((l_j)_\Omega \# s \cup u\), for all \( j \leq i + 1 \). Thus \( \forall j \leq i + 1 \) \((l_j)_\Omega \# t\).

   Thus \( \forall t \in \text{todo}^F(l_1, \ldots, l_{i+1}) \) \( \forall j \leq i + 1 \) \((l_j)_\Omega \# t\).

2. Let \( t \in \text{new}(\vec{l}, m) \). Then \( t = (m)_\Omega \cup t' \) with \( t' \in \text{todo}^F(\vec{l}) \). Thus we have \((l_j)_\Omega \# t'\) \( (j \leq |\vec{l}|) \). The result follows with properties of \( \cup \) in Lemma 5.3.12.
3. Let $\sigma \in \text{subs}(\tilde{l}, m)$. Then $\sigma = \text{mgu}(m, \text{fresh}(\text{var}(m), u))$ for some $u \in \text{new}(\tilde{l}, m)$.

Thus $(l_j)_{\Omega} \sigma = u$ ($j \leq |\tilde{l}|$). Therefore $l_j^{\text{Arg}} u^\sigma$, and $l_j^{\text{Arg}} u^\sigma = m^\sigma$ ($j \leq |\tilde{l}|$). \hfill \Box

The following lemma shows that \text{subs} computes the most general substitutions for the left-hand side of a rule, such that the its instances have a strongly incompatible argument with the left-hand sides of some other rules.

**Lemma 5.3.23** Let $(D, C, R)$ be an $S$-sorted Constructor System. Let $F \in D$. Let $\tilde{l}$ be a sequence of applications of $F$ to patterns. Let $m$ be an application of $F$ to patterns. Let $t$ be an application of $F$ to terms.

1. $(\forall j \leq |\tilde{l}|) \ l_j^{\text{Arg}} \Rightarrow \exists u \in \text{todo}^F(\tilde{l}) \ t \geq u$.

2. $(\forall j \leq |\tilde{l}|) \ l_j^{\text{Arg}} m^\sigma \Rightarrow \exists u \in \text{new}(\tilde{l}, m) \ m^\sigma \geq u$, for any substitution $\sigma$.

3. $(\forall j \leq |\tilde{l}|) \ l_j^{\text{Arg}} \land (\exists \tau \ m^\tau = t) \Rightarrow \exists \sigma \in \text{subs}(\tilde{l}, m) \ \exists \sigma' \ (m^\sigma)^{\sigma'} = t$.

**Proof**

1. Induction on $|\tilde{l}|$.

   Case $|\tilde{l}| = 0$. Follows from $t \geq F(\Omega, \ldots, \Omega)$.

   Case $|\tilde{l}| = i + 1$. Assume $\forall j \leq i + 1 \ l_j^{\text{Arg}}$. By induction we have $t \geq u$ for some $u \in \text{todo}^F(l_1, \ldots, l_i)$.

   Either $(l_{i+1})_{\Omega} \# u$ and $u \in \text{td}^F(\tilde{l})$. Then $u \geq u'$ for some $u' \in \text{todo}^F(l_1, \ldots, l_{i+1})$.

   Thus also $t \geq u'$.

   Or $(l_{i+1})_{\Omega} \uparrow u$. As $l_{i+1}^{\text{Arg}}$ we have $t \geq w$ for some $w \in \text{IncLhs}(l_{i+1})$.

   Therefore we have $w \uparrow u$ and thus $w \cup u \in \text{td}^F(\tilde{l})$. Thus we have $w \cup u \geq t'$ for some $t' \in \text{todo}^F(l_1, \ldots, l_{i+1})$. Moreover we have $t \geq w \cup u$ and thus $t \geq t'$.

2. Assume $\forall j \leq |\tilde{l}| \ l_j^{\text{Arg}} m^\sigma$. Then we have $m^\sigma \geq t'$ for some $t' \in \text{todo}^F(\tilde{l})$. Thus $m_{\Omega} \geq t'$. Therefore $m_{\Omega} \cup t' \in \text{nw}(\tilde{l}, m)$. Thus we have $m_{\Omega} \cup t' \geq u$ for some $u \in \text{new}(\tilde{l}, m)$. The result follows, because $m^\sigma \geq u$.

3. Assume $l_j^{\text{Arg}}$ ($j \leq |\tilde{l}|$) and $m^t = t$. Then $m^\tau \geq u$ for some $u \in \text{new}(\tilde{l})$. Let $w = \text{fresh}(\text{var}(m), u)$. Then we have $m^\tau = t = w^\tau$ for some substitution $\tau'$, because $\text{var}(m) \cap \text{var}(w) = \emptyset$. By definition we have $(m^{\text{mgu}(m, w)})^{\sigma'} = m^\tau = t$ for some $\sigma'$ and $\text{mgu}(m, w) \in \text{subs}(\tilde{l}, m)$. \hfill \Box

We will use the definition of \text{subs} to define a function 'newrules', that given a sequence $\tilde{l}$ of $F$-rules and an $F$-rule, computes instances of this rule, such that their left-hand sides and all the left-hand sides of $l_i$ have strongly incompatible arguments.
Definition 5.3.24 Let \((D,C,R)\) be an \(S\)-sorted Constructor System. Let \(F \in D\). Let \(l_1 \rightarrow r_1, \ldots, l_k \rightarrow r_k, m \rightarrow t\) be \(F\)-rules.
newrules((\(l_1 \rightarrow r_1, \ldots, l_k \rightarrow r_k\), \(m \rightarrow t\)) = \{m^\sigma \rightarrow t^\sigma | \sigma \in \text{subs}((l_1, \ldots, l_k, m))\}.

Example 5.3.25 Let \((D,C,R,>)\) be the PCS of Example 5.3.2. We can compute newrules for the rules of Eq with respect to their preceding rules as follows:

newrules(\(\epsilon, \text{Eq}(0,0) \rightarrow \text{True}\)) = \{\text{Eq}(0,0) \rightarrow \text{True}\};
newrules((\text{Eq}(0,0) \rightarrow \text{True}), \text{Eq}(S(w),S(x)) \rightarrow \text{Eq}(w,x)) =
\{\text{Eq}(S(a),S(b)) \rightarrow \text{Eq}(a,b)\};
newrules((\text{Eq}(0,0) \rightarrow \text{True}), \text{Eq}(S(w),S(x)) \rightarrow \text{Eq}(v,x), \text{Eq}(y,z) \rightarrow \text{False}) =
\{\text{Eq}(S(c),0) \rightarrow \text{False}, \text{Eq}(0,S(d)) \rightarrow \text{False}\}.

Notice that although the third rule for Eq in the PCS interferes with its preceding rules, its instances computed by newrules do not interfere with these rules.

In the following lemma we show that for a given sequence \(\overline{t}\) of \(F\)-rules and an \(F\)-rule, newrules computes the most general instances of this rule, such that their left-hand sides and all the left-hand sides of \(l_i\) have strongly incompatible arguments.

Lemma 5.3.26 Let \((D,C,R)\) be an \(S\)-sorted Priority Constructor System. Let \(F \in D\). Let \(l_1 \rightarrow r_1, \ldots, l_{k+1} \rightarrow r_{k+1}\) be \(F\)-rules.

1. \(\forall l \rightarrow r \in \text{newrules}(l_1 \rightarrow r_1, \ldots, l_k \rightarrow r_k, l_{k+1} \rightarrow r_{k+1})\) \(\forall j \leq k\) \(l_j \not\rightarrow^{S}_l\).

2. If \(l_j \not\rightarrow^{S}_l\), for all \(j \leq k\), and \(l_{k+1}' = t\) then \(t\) is an \(r\)-redex for some \(r \in \text{newrules}(l_1 \rightarrow r_1, \ldots, l_k \rightarrow r_k, l_{k+1} \rightarrow r_{k+1})\).

Proof

1. Let \(r \in \text{newrules}(l_1 \rightarrow r_1, \ldots, l_k \rightarrow r_k, l_{k+1} \rightarrow r_{k+1})\). By definition, for some \(\sigma \in \text{subs}((l_1, \ldots, l_k, l_{k+1}))\) \(r = l_{k+1}' \rightarrow r_{k+1}'\). Thus \(\forall j \leq k\) \(l_j \not\rightarrow^{S}_l\) by Lemma 5.3.22.

2. Assume \(\forall j \leq k\) \(l_j \not\rightarrow^{S}_l\) and \(\exists \sigma \ l_{k+1}' = F(t_1, \ldots, t_n)\). By Lemma 5.3.23 we have \((l_{k+1}')^r = F(t_1, \ldots, t_n)\), for some \(r \in \text{subs}((l_1, \ldots, l_k, l_{k+1}))\), and \(\text{substitution} r'\). By definition we have \(l_{k+1}' \rightarrow r_{k+1}' \in \text{newrules}(l_1, \ldots, l_k, l_{k+1})\) and \(F(t_1, \ldots, t_n)\) is an \(l_{k+1}' \rightarrow r_{k+1}'\)-redex.

Now we will define a function 'transform' that transforms a PCS to an equivalent CS by computing for each rule instances that do not interfere with any of the preceding rules.

Definition 5.3.27 Let \((D,C,R,>)\) be an \(S\)-sorted Priority Constructor System with finitely many rules.

1. Let \(F \in D\). Let \(l \rightarrow r\) be an \(F\)-rule. The sequence \(\text{prerules}(R,>, l \rightarrow r)\) contains all \(F\)-rules \(l' \rightarrow r' \in R\), such that \(l' \rightarrow r' > l \rightarrow r\).
2. transform\((\mathcal{D}, \mathcal{C}, R, >) = (\mathcal{D}, \mathcal{C}, \cup_{r \in R} \text{newrules}(\text{precrules}(R, >, r)), r))\).

**Example 5.3.28** Let \((\mathcal{D}, \mathcal{C}, R, >)\) be the PCS of Example 5.3.2. We will show its transformation to an equivalent CS. Computation of precrules for the rules of Eq:

- \text{precrules}(R, >, \text{Eq}(0, 0) \rightarrow \text{True}) = e;
- \text{precrules}(R, >, \text{Eq}(S(w), S(x)) \rightarrow \text{Eq}(w, x)) = \text{Eq}(0, 0) \rightarrow \text{True};
- \text{precrules}(R, >, \text{Eq}(y, z) \rightarrow \text{False}) = \text{Eq}(0, 0) \rightarrow \text{True}, \text{Eq}(S(w), S(x)) \rightarrow \text{Eq}(w, x).

Using the results of the previous example we get:

\[
\text{transform}(\mathcal{D}, \mathcal{C}, R, >) = (\mathcal{D}, \mathcal{C},
\{\text{Eq}(0, 0) \rightarrow \text{True}, \text{Eq}(S(a), S(b)) \rightarrow \text{Eq}(a, b),
\text{Eq}(S(c), 0) \rightarrow \text{False}, \text{Eq}(0, S(d)) \rightarrow \text{False}\}).
\]

In the next proposition we will show that an \(l \rightarrow r\)-reduction step is *enabled* if and only if it is an instance of a rule in the *newrules* of \(l \rightarrow r\) for the preceding rules of \(l \rightarrow r\).

**Proposition 5.3.29** Let \((\mathcal{D}, \mathcal{C}, R, >)\) an \(S\)-sorted Priority Constructor System with finitely many rules. Let \(l_1 \rightarrow r_1 \in R\). Let \(\sigma\) be a substitution.

\(l_1^\pi \rightarrow e r_1^\pi \Leftrightarrow \exists l_2 \rightarrow r_2 \in \text{newrules}(\text{precrules}(R, >, l_1 \rightarrow r_1), l_1 \rightarrow r_1) \exists r \ l_1^\pi = l_2^r \land r_1^\pi = r_2^r.

**Proof** Let \(l_1 \rightarrow r_2 \in R\). Let \(\sigma\) be a substitution.

\(\Rightarrow\). Assume \(l_1^\pi \rightarrow e r_1^\pi\). Then we have \(l'^\pi \rightarrow r'^\pi\) for all \(l' \rightarrow r' \in R\) with \(l' \rightarrow r' > l_1 \rightarrow r_1\).

By Lemma 5.3.26 we obtain that \(l_1^\pi\) is an \(l_2 \rightarrow r_2\)-reduct for some rule \(l_2 \rightarrow r_2 \in \text{newrules}(\text{precrules}(R, >, l_1 \rightarrow r_1), l_1 \rightarrow r_1)\). By definition of newrules we have \(l_2 \rightarrow r_2 = l_2^r \rightarrow r_1^r\) for some substitution \(r\).

\(\Leftarrow\). Assume \(l_1^\pi = l_2^r\) and \(r_1^\pi = r_2^r\) for \(l_2 \rightarrow r_2 \in \text{newrules}(\text{precrules}(R, >, l_1 \rightarrow r_1), l_1 \rightarrow r_1)\). If \(l' \rightarrow r' > l_1 \rightarrow r_1\) then by Lemma 5.3.26 we have \(l'^\#_\pi \text{Arg} l_2\) and thus \(l'^\#_\pi \text{Arg} l_2^r\), for all \(l' \rightarrow r' \in R\). By the definition of newrules we have \(l_2 \rightarrow r_2 = l_2^r \rightarrow r_1^r\) for some substitution \(r\). Thus we have \(l_1^\pi = (l_1^\pi)^r \rightarrow e (r_1^\pi)^r = r_1^\pi\). \(\Box\)

**Theorem 5.3.30 (reduction preservation)** Let \((\mathcal{D}, \mathcal{C}, R, >)\) be an \(S\)-sorted Priority Constructor System with finitely many rules. Then transform\((\mathcal{D}, \mathcal{C}, R, >)\) is a left-linear Constructor System with a reduction relation that is the same as the enabled reduction relation defined by \(R\) and \(>\).

**Proof**

Let PCS\((\mathcal{D}, \mathcal{C}, R, >)\) be an \(S\)-sorted Priority Constructor System with finitely many rules. According to Lemma 5.3.21 the rules of the transformed PCS are left-linear and have applications of defined symbols to patterns as left-hand sides. Thus the transformed PCS is a left-linear Constructor System. A consequence of Proposition 5.3.29 is: \(t_1 \rightarrow_e t_2 \Leftrightarrow t_1 \rightarrow t_2\) in transform\((\mathcal{D}, \mathcal{C}, R, >)\). \(\Box\)
Now that we have treated the transformation of a finite Priority Constructor System to an equivalent Constructor System, it is interesting to investigate which Priority Constructor Systems are transformed to weakly orthogonal Constructor Systems. Thus all critical pairs of the transformed PCS should be trivial. In the transformation each rule is replaced by several instances with left-hand sides that are not unifiable with the left-hand sides of its preceding rules (and their instances). If all conflicting rules in a PCS are ordered, then its transformation is weakly orthogonal.

**Definition 5.3.31** Let $\Sigma$ an $S$-sorted signature. Let $R$ be a set rewrite rules over $\Sigma$ with $V$. A strict partial order $>$ on $R$ prevents conflicts, if each critical pair is either trivial or obtained from two comparable (by $>$) rules.

**Example 5.3.32** The PCS of Example 5.3.2 has two critical pairs, namely $\langle \text{True, False} \rangle$ (as $\text{Eq}(0,0)$ is a redex for the first and third rule), and $\langle \text{Eq}(x,y), \text{False} \rangle$ (as $\text{Eq}(x(x),y(y))$ is a redex for the second and the third rule). As the order on this PCS is total all rules are comparable and thus it prevents conflicts.

Because no rules are ordered, the order of the following PCS does not prevent conflicts.

<table>
<thead>
<tr>
<th>sort</th>
<th>bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td>True: bool</td>
</tr>
<tr>
<td></td>
<td>False: bool</td>
</tr>
<tr>
<td>def</td>
<td>Choice: bool x bool -&gt; bool</td>
</tr>
<tr>
<td>rule</td>
<td>Choice(False,x) -&gt; False</td>
</tr>
<tr>
<td></td>
<td>Choice(y,True) -&gt; True</td>
</tr>
</tbody>
</table>

In the following lemma we will show how instantiation effects unifiability. We need this lemma in the next proposition in which we will show that the enabled reduction relation of a PCS with a conflict preventing order is confluent.

**Lemma 5.3.33** Let $\Sigma$ be an $S$-sorted signature. Let $l_1, l_2 \in T(\Sigma, V)$. Let $\sigma$ be a substitution.

1. If $l_1^\sigma$ and $l_2$ are unifiable and $\text{var}(l_1) \cap \text{var}(l_2) = \emptyset$, then $l_1$ and $l_2$ are unifiable.

2. If the critical pair obtained by unification of the left-hand sides of $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ is trivial, and $l_1^\sigma$ and $l_2$ are unifiable, then the critical pair obtained by unification of $l_1^\sigma$ and $l_2$ is trivial.

**Proof**

1. Assume $\tau$ is a unifier for $l_1^\sigma$ and $l_2$. We define a substitution $\sigma'$ as follows:

   $$\sigma'(v) = \begin{cases} v & \text{if } v \in \text{var}(l_2) \\ \sigma(v) & \text{otherwise} \end{cases}$$

   We have $l_1^\sigma = l_1'$ and $l_2' = l_2$.

   Thus $\tau \circ \sigma'$ is a unifier for $l_1$ and $l_2$. 

...
2. Assume that (possibly after renaming variables in \( l_2 \rightarrow r_2 \)) \( \text{var}(l_2') \cap \text{var}(l_2) = \emptyset = \text{var}(l_1) \cap \text{var}(l_2) \). Assume \( l_2' \) and \( l_2 \) are unifiable. We define a substitution \( \sigma' \) as follows: 
\[
\sigma'(v) = \begin{cases} 
  v & \text{(if } v \in \text{var}(l_2)) \\
  \sigma(v) & \text{(otherwise)}
\end{cases}
\]
We have \( l_1' = l_1' \), \( r_1' = r_2' \), \( l_2' = l_2 \), and \( r_2'' = r_2 \). Let \( \tau \) be a unifier for \( l_1' \) and \( l_2' \). Then \( \tau \circ \sigma' \) is a unifier for \( l_1 \) and \( l_2 \), thus \( \tau \circ \sigma' \) is a unifier for \( r_1 \) and \( r_2 \). Thus \( \tau \) is a unifier for \( r_1' \) and \( r_2' \). \( \square \)

The next proposition gives a criterion for determining whether a Priority Constructor System has a confluent enabled reduction relation.

**Proposition 5.3.34 (confluence)** Let \((D, C, R, >)\) be an \( S \)-sorted PCS with finitely many rules. If \( > \) prevents conflicts then transform\((D, C, R, >)\) is weakly orthogonal, and hence confluent.

**Proof**
Assume PCS\((D, C, R, >)\) is finite, and \( > \) prevents conflicts. Let \((D, C, R') = \text{transform}(D, C, R, >)\). By the theorem \((D, C, R')\) is a left-linear constructor system. By Proposition 5.2.11 we just have to prove that all critical pairs obtained from unification of the left-hand sides of two rules are trivial. Assume \( \langle r_1' , r_2' \rangle \) is a critical pair obtained from unification of the left-hand sides of two rules \( l_1 \rightarrow r_1 , l_2 \rightarrow r_2 \in R' \). By definition there are rules \( l_3 \rightarrow r_3 , l_4 \rightarrow r_4 \in R \) such that \( l_1 \rightarrow r_1 , l_2 \rightarrow r_2 \in \text{newrules}(\text{precrules}(R, >) , l_3 \rightarrow r_3) \) and \( l_2 \rightarrow r_2 \in \text{newrules}(\text{precrules}(R, >) , l_4 \rightarrow r_4) \). By Lemma 5.3.33 \( l_3 \) and \( l_4 \) are unifiable. As \( > \) prevents conflicts we either have \( r_3'' = r_4' \) (\( \tau = \text{mgu}(l_3, l_4) \)), or \( l_3 \rightarrow r_3 \) and \( l_4 \rightarrow r_4 \) are comparable by \( > \). In the first case we have \( r_1'' = r_2' \) by Lemma 5.3.33. In the second case by Lemma 5.3.7 and 5.3.26 we obtain \( l_1 \downarrow \ast_{\mathcal{R}} l_2 \) which contradicts the unifiability of \( l_1 \) and \( l_2 \). \( \square \)

We will show that certain properties, that hold for the underlying Constructor System of a PCS, also hold for its transformed version.

First we will prove that if the underlying Constructor system of a PCS is exhaustively defined, then its transformed version is exhaustively defined. Therefore we will show that an application to values is either incompatible with the \( \Omega \)-term of the left-hand side of a left-linear rule or is a redex for that rule.

**Lemma 5.3.35** Let \( \Sigma \) be an \( S \)-sorted signature. Let \( t \in \mathcal{T}(\Sigma) \) be a closed term, and let \( u \in t_{\Omega} \).

1. Let \( W \) be a finite subset of \( V \). Then \( t_{\Omega} = t = \text{fresh}(W, t) \).
2. If \( u \geq t \) then \( u = t \).
3. If \( t \uparrow u \) then \( t \geq u \).
4. If \( u \) is linear and \( t \uparrow u_{\Omega} \) then \( u^\sigma = t \), for some substitution \( \sigma \).

**Proof**
1. Because \( t \) does not contain variables we have \( t_\Omega = t \), and as \( t \) does not contain \( \Omega \) we have fresh\((W, t) = t\).

2. Induction on the proof of \( u \succeq t \).

Base case: \( u \succeq \Omega \). Cannot be applied, because \( \Omega \not\in \mathcal{T}(\Sigma) \).

Second case \( F(u_1, \ldots, u_n) \succeq F(u_1, \ldots, u_n) \), because \( \forall i \leq n \ t_i \succeq u_i \). By the induction hypothesis we have \( t_i = u_i (i \leq n) \), thus \( F(t_1, \ldots, t_n) = F(u_1, \ldots, u_n) \).

3. If \( t \uparrow u \), then for some \( w \) we have \( w \succeq t \) and \( w \succeq u \). By 2. we have \( w = t \), and therefore \( t \succeq u \).

4. Assume \( u \) is linear and \( t \uparrow u_\Omega \). By 3. we have \( t \succeq u_\Omega \). Because \( u \) is linear, we have \( \exists \sigma \ u^\sigma = t \) by Lemma 5.3.14. \( \square \)

Proposition 5.3.36 (exhaustively definedness) Let PCS\((\mathcal{D}, \mathcal{C}, R, >)\) be an \( S \)-sorted Priority Constructor System. If the underlying CS\((\mathcal{D}, \mathcal{C}, R)\) is exhaustively defined, then transform\((\mathcal{D}, \mathcal{C}, R, >)\) is exhaustively defined.

Proof

Assume CS\((\mathcal{D}, \mathcal{C}, R)\) is exhaustively defined. Let \( F \in \mathcal{D}, t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}) \). Because CS\((\mathcal{D}, \mathcal{C}, R)\) is exhaustively defined, there exists an \( F \)-rule \( l \rightarrow r \in R \) and a substitution \( \sigma \), such that \( l^\sigma = F(t_1, \ldots, t_n) \) and for each \( F \)-rule \( l' \rightarrow r' \), such that \( l' \rightarrow r' > l \rightarrow r \) we have \( (l')^\sigma \neq F(t_1, \ldots, t_n) \) for all substitutions \( \tau \). Thus by Lemma 5.3.35 \( \sigma_0 \# F(t_1, \ldots, t_n) \) and by Lemma 5.3.12 and 5.3.13 we have \( l' \#^{\mathcal{A}} F(t_1, \ldots, t_n) \). Thus \( F(t_1, \ldots, t_n) \) is an enabled \( l \rightarrow r \)-redex. By Theorem 5.3.30 the term \( F(t_1, \ldots, t_n) \) is a redex in transform\((\mathcal{D}, \mathcal{C}, R, >)\). \( \square \)

Proposition 5.3.37 (termination) Let PCS\((\mathcal{D}, \mathcal{C}, R, >)\) be an \( S \)-sorted Priority Constructor System. If the underlying CS\((\mathcal{D}, \mathcal{C}, R)\) is strongly normalizing then transform\((\mathcal{D}, \mathcal{C}, R, >)\) is strongly normalizing.

Proof

Assume we have an infinite sequence \( t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots \) in transform\((\mathcal{D}, \mathcal{C}, R, >)\). By Theorem 5.3.30 we have \( t_1 \rightarrow_{e} t_2 \rightarrow_{e} t_3 \rightarrow_{e} \ldots \) in \((\mathcal{D}, \mathcal{C}, R, >)\). By definition we have \( t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots \) in \((\mathcal{D}, \mathcal{C}, R)\). \( \square \)
Chapter 6

Priority Rewriting in a Type System

In the previous chapter we have described Priority Constructor Systems as a formalism for defining functions and we have formulated easily verifiable criteria that guarantee fundamental properties such as confluence and termination. In this chapter we will present an extension of the type system of Higher Order Logic with function definitions based on this formalism. We discuss the restrictions that must be imposed on function definitions in order to guarantee that only mathematical functions can be defined. We show how this formalism for defining functions can be used for reasoning by cases, and how we can use the computational meaning of functions in formal proofs.

We establish some fundamental properties of our extended type system. We prove that the result of a computation and the type of a term are unique, and that the type of a term is preserved by computations. Because of these properties, it makes sense to consider the Abstract Reduction System consisting of the set of typable terms and the reduction relation determined by a context as a formal mathematical language.

Next, we discuss the relation between our extended type system with priority rewriting and the system λHOL with inductive types. We present a transformation function that maps terms of the former system to terms of the latter system. We analyse how the transformation function effects the type and the reduction relation of the encoded terms. Finally, we present a type synthesis algorithm for our type system. As we can use this algorithm to decide whether or not a pseudo term is typable (meaningful) in a certain context, our extended type system is suited for automated verification.

6.1 Higher Order Logic with Pattern Matching

In this section we will describe how we can extend the system Higher Order Logic of Section 3.1 with function definitions based on the ideas behind Priority Constructors Systems of Section 5.3. We will use the function symbols of an algebraic data type-signature as constructors, and the constants, that are not used as type universes, are used as defined symbols. The computational meaning of defined symbols is determined by function definitions that contain rewrite rules.
We extend the syntax of pseudo terms (of Definition 3.1.1) with sorts and algebraic data type constructors (see Definition 2.2.5).

**Definition 6.1.1** Let $S$ be a finite set of sorts. Let $\Sigma$ be a finite $S$-sorted algebraic data type-signature with function symbols $\mathcal{F}$ and typing function $\tau: \mathcal{F} \to S^+$, that is specified using the method of Example 2.2.4. The sets $V, C, S, \mathcal{F}$ are assumed to be pairwise disjoint. The syntax of pseudo terms $\mathcal{T}_\pi$ is defined as follows:

$$\mathcal{T}_\pi = V \mid C \mid S \mid \mathcal{F} \mid (\mathcal{T}_\pi, \mathcal{T}_\pi) \mid \lambda V.\mathcal{T}_\pi.\mathcal{T}_\pi \mid \Pi V.\mathcal{T}_\pi.\mathcal{T}_\pi$$

The constants in the set $C \setminus \text{Universes}$ will be used as names of function definitions. We assume that this set is split into disjoint infinite subsets $C_\pi$ for each type universe $\text{mbox{m}} \in \text{Universes}$. From now on we will assume that $S$ contains the sorts Bool, Nat, List, and that $\Sigma$ contains their well-known constructors (see Example 6.1.2). Recall that we mean $G \in \mathcal{F}$ if we write $G \in \Sigma$.

**Example 6.1.2** Algebraic data type-signature for booleans, natural numbers and lists:

<table>
<thead>
<tr>
<th>sort</th>
<th>Bool, Nat, List</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td></td>
</tr>
<tr>
<td>True:</td>
<td>Bool</td>
</tr>
<tr>
<td>False:</td>
<td>Bool</td>
</tr>
<tr>
<td>O:</td>
<td>Nat</td>
</tr>
<tr>
<td>S:</td>
<td>Nat $\to$ Nat</td>
</tr>
<tr>
<td>Nil:</td>
<td>List</td>
</tr>
<tr>
<td>Cons:</td>
<td>Nat $\times$ List $\to$ List</td>
</tr>
</tbody>
</table>

**Definition 6.1.3** We define bound and free variables and substitution for pseudo terms ($\in \mathcal{T}_\pi$) as in definition 3.1.6 by adding the cases:

1. $BV(b) = \emptyset$, for $b \in S \cup \Sigma$
2. $FV(b) = \emptyset$, for $b \in S \cup \Sigma$
3. $b[v := t] = b$, for $b \in S \cup \Sigma$

The set of pseudo terms with a hole in it is defined by the following grammar.

**Definition 6.1.4** The set $\mathcal{H}_\pi$ is defined as:

$$\mathcal{H}_\pi = [ ] \mid \mathcal{H}_\pi \mathcal{T}_\pi \mid \mathcal{T}_\pi \mathcal{H}_\pi \mid \lambda V.\mathcal{H}_\pi.\mathcal{T}_\pi \mid \lambda V.\mathcal{T}_\pi.\mathcal{H}_\pi \mid \Pi V.\mathcal{H}_\pi.\mathcal{T}_\pi \mid \Pi V.\mathcal{T}_\pi.\mathcal{H}_\pi$$

**Definition 6.1.5** The notion of $\beta$-reduction for pseudo terms ($\in \mathcal{T}_\pi$) is defined as follows:

$$C[(\lambda v.t.b)a] \rightarrow_\beta C[b[v := a]]$$

for $C[ ] \in \mathcal{H}_\pi$. 

Since we want to define a function by pattern matching, we must define 'patterns' and 'rules' first. A pattern is built from variables, and constructors of algebraic data types using applications.

Definition 6.1.6 1. The set of patterns $\mathcal{P}$ is the smallest set that satisfies:

(a) $v \in \mathcal{P}$, if $v \in V$.
(b) $C \in \mathcal{P}$, if $C \in \Sigma$.
(c) $(f,p) \in \mathcal{P}$, if $f,p \in \mathcal{P}$ and $f \not\in V$.

2. The set of rules $\mathcal{R}$ is the smallest set such that: if $F \in \mathcal{C}$, $p_1, \ldots, p_n \in \mathcal{P}$, $t \in T_v$, and $FV(t) \subseteq FV(Fp_1 \ldots p_n)$ then $(Fp_1 \ldots p_n, t) \in \mathcal{R}$. We will denote a rule $(Fp_1 \ldots p_n, t) \in \mathcal{R}$ as $Fp_1 \ldots p_n \Rightarrow t$. We call $Fp_1 \ldots p_n$ the left-hand side (lhs) and $t$ the right-hand side (rhs) of rule $Fp_1 \ldots p_n \Rightarrow t$. We say that $Fp_1 \ldots p_n \Rightarrow t \in \mathcal{R}$ is an $F$-rule with $n$ arguments.

Definition 6.1.7 A rule $l_1 \Rightarrow r_1$ is the result of renaming a free variable in $l_1 \Rightarrow r_1$ if $l_2 = l_1[x := y]$, $r_2 = r_1[x := y]$ and $y \not\in FV(l_1) \cup BV(r_1)$. A rule $l \Rightarrow r_2$ is the result of renaming a bound variable in $l \Rightarrow r_1$ if $r_2$ is the result of renaming a bound variable in $r_1$.

Convention 6.1.8 We identify rules that can be obtained from each other by renaming variables. Names of bound variables in the right-hand side of a rule $l \Rightarrow r$ will always be chosen such that $FV(l) \cap BV(r) = \emptyset$.

We will now formalize function definitions, that specify functions by a sequence of rules.

Definition 6.1.9 A function definition has form $F : t = \vec{r}$, with $F \in \mathcal{C}$, $t \in T_v$, $\vec{r}$ a non-empty sequence of $F$-rules all with the same number of arguments. We call $F$ the defined constant, $t$ the type, and $\vec{r}$ the rules of $F : t = \vec{r}$.

Example 6.1.10 Function definition of $\text{Leq}$:

\[
\begin{array}{l}
\text{Leq} : \text{Nat} \to \text{Nat} \to \text{Bool} = \\
\text{Leq} \; \text{O} \; y \quad \Rightarrow \text{True}, \\
\text{Leq} \; (S \; x) \; \text{O} \quad \Rightarrow \text{False}, \\
\text{Leq} \; (S \; x) \; (S \; y) \quad \Rightarrow \text{Leq} \; x \; y \\
\end{array}
\]

The meaning of a function defined in this way is intuitively clear, if we regard the rules as equations. If we instantiate a rule with canonical members of algebraic data types the result of the left-hand side is given by the right-hand side. For instance, $\text{Leq} \; (S \; \text{O}) \; \text{O} = \text{False}$. As we want to be able to use the meaning of defined constants, we extend the notion of 'pseudo context' with function definitions.
Definition 6.1.11  1. A context item is either a variable declaration or a function definition.

2. The set \( \lambda \) of pseudo contexts contains all finite sequence of context items.

3. Let \( \Gamma \in \lambda \) be a pseudo context. The set \( \text{fundef} \Gamma \) contains the function definitions of \( \Gamma \), and the set \( \text{consts} \Gamma \) contains the defined constants of the function definitions in \( \text{fundef} \Gamma \). By \( \text{FV} \Gamma \) we denote the set of subjects of the variable declarations of \( \Gamma \).

Before we can specify how the rules of a function definition give a computational meaning to its defined constant, we must introduce the notion of 'substitution sequence' that allows us to specify the instantiation of many variables in a pseudo term by one expression.

Definition 6.1.12  1. A substitution sequence is a sequence \( \vec{\sigma} = (v_1, t_1, \ldots, v_n, t_n) \), with \( v_i \in V, t_i \in T \). The substitution sequence \( \vec{\sigma} \) is standard if \( \text{FV}(t_i) \cap \{v_1, \ldots, v_n\} = \emptyset \) and \( v_i \neq v_j \) if \( i \neq j \), for all \( i, j \leq n \).

2. Instantiating a pseudo term \( t \) with a substitution sequence is defined as follows:

(a) \( \text{subst}(t, e) = t \).

(b) \( \text{subst}(t, (v_1, u_1, \vec{\sigma})) = \text{subst}(t[v_1 := u_1], \vec{\sigma}) \).

The rules in a function definition give its defined constant a computational meaning. We can formalize this by defining a rewrite relation on pseudo terms in a certain context. Notice the similarity with reduction for Term Rewriting Systems (see Definition 2.2.18).

Definition 6.1.13  1. Let \( l \Rightarrow t \in R \). A \( l \Rightarrow t \)-reduction step is a pair \( u_1 \Rightarrow u_2 \) such that for some substitution sequence \( \vec{\sigma} \) we have \( \text{subst}(l, \vec{\sigma}) = u_1 \) and \( \text{subst}(t, \vec{\sigma}) = u_2 \), for \( u_1, u_2 \in T \).

2. We define a ternary relation \( \rightarrow_{\infty} \) on a pseudo context and two pseudo terms as:
\[ \Gamma_1, F : t = \vec{r}, \Gamma_2 \vdash C[u_1] \rightarrow_{\infty} C[u_2], \text{ if } u_1 \Rightarrow u_2 \text{ is an } r_i \text{-reduction step, } C[ ] \in H. \]

Example 6.1.14 Let \( \Delta_{\text{Leq}} \) be the function definition of Example 6.1.10.
We have \( \Delta_{\text{Leq}} \vdash \text{Leq}(S O)(S(S O)) \rightarrow_{\infty} \text{Leq}(S O) \). But \( \text{not } e \vdash \text{Leq}(O) n \rightarrow_{\infty} \text{True} \), as the empty context does not contain a definition for \( \text{Leq} \).

Remark 6.1.15 If \( t_1 \Rightarrow t_2 \) is a \( l \Rightarrow r \)-reduction step and \( \text{FV}(t_1) \cap \text{FV}(l) = \emptyset \) then there exists a standard substitution sequence \( \sigma \) such that \( \text{subst}(l, \vec{\sigma}) = t_1 \). By renaming variables in \( l \Rightarrow r \) we can always obtain an equivalent rule \( l' \Rightarrow r' \) such that \( \text{FV}(t_1) \cap (\text{FV}(l') \cup \text{BV}(r')) = \emptyset \).
**Convention 6.1.16** The variables in a rule $l \Rightarrow r$ will always be chosen such that if $t_1 \Rightarrow t_2$ is a $l \Rightarrow r$-reduction step then $FV(t_1) \cap (FV(l) \cup BV(r)) = \emptyset$.

Now we can try to define a typing relation $\vdash_\pi$ for our new pseudo terms.

**Definition 6.1.17** The rules for $\vdash_\pi$ are the same as the rules for $\vdash$ of Definition 3.1.15. Furthermore we have the following typing rules for algebraic data types:

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(data-type)</strong></td>
<td>$\epsilon \vdash_\pi s : \square$</td>
</tr>
<tr>
<td><strong>(data-const)</strong></td>
<td>$\epsilon \vdash_\pi C : s$ if $\tau(C) = s$</td>
</tr>
<tr>
<td><strong>(data-constr)</strong></td>
<td>$\frac{(\Gamma \vdash t_i : s_i)<em>{i=1}^n}{\Gamma \vdash C</em>{t_1 \ldots t_n} : s}$ if $\tau(C) = s_1 \times \ldots \times s_n \rightarrow s$, $1 \leq n$</td>
</tr>
</tbody>
</table>

In the rules above we use the following notation: $s, s_1, \ldots, s_n \in S$, $C \in \Sigma$, $\Gamma \in \mathcal{X}_\pi$.

**Example 6.1.18** We can for instance derive $\epsilon \vdash_\pi \text{Boo} : \square$, and $x : \text{Nat} \vdash_\pi \text{Cons} x \text{ Nil} : \text{List}$. But we cannot derive $\epsilon \vdash_\pi S : \text{Nat} \rightarrow \text{Nat}$, since $S$ should be applied to an argument of type Nat.

We want to define a typing rule for function definitions. Such a rule should at least guarantee that the function has a legal type, and that the left-hand side and right-hand side of each rule have the same type. For typing a left-hand side $l$ of a rule we need to provide types for all variables in $l$. The pseudo context we need for a rule is fully determined by the type of the function definition and its left-hand side.

**Definition 6.1.19** 1. We define a relation ‘ctxtpat’ on a pseudo context, a pattern and a pseudo term as follows:

(a) $\text{ctxtpat}(\epsilon, C, s)$, if $\tau(C) = s$ for $C \in \Sigma$.
(b) $\text{ctxtpat}(x : t, x, t)$, for $x \in V$.
(c) $\text{ctxtpat}((\Gamma_1, \ldots, \Gamma_n), C p_1 \ldots p_n, s)$, if $\tau(C) = s_1 \times \ldots \times s_n \rightarrow s$, $\text{ctxtpat}(\Gamma_i, p_i, s_i)$, all $i \leq n$, for $C \in \Sigma$.

2. We define a relation ‘ctxtlhs’ on a pseudo context, and two pseudo terms as follows:

$\text{ctxtlhs}((\Gamma_1, \ldots, \Gamma_n), F p_1 \ldots p_n, \Pi x_1 : t_1, \ldots, \Pi x_n : t_{n+1})$ if $F \in C$, $p_1, \ldots, p_n \in P$, $1 \leq n$, and $\text{ctxtpat}(\Gamma_i, p_i, t_i[x_i := p_i] \ldots [x_{i-1} := p_{i-1}])$ all $i \leq n$.

**Example 6.1.20** We have $\text{ctxtpat}(x : \text{Nat}, S x, \text{Nat})$, and $\text{ctxtlhs}((x : \text{Nat}, y : \text{Nat}), \text{Leq}(S x)(S y), \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool})$. But we do not have $\text{ctxtlhs}((x : \text{Nat}, y : \text{Nat}), \text{Leq} \ O y, \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool})$, as $x$ does not occur in $\text{Leq} \ O y$.

We will make a first attempt of giving typing rules for function definitions. Later on we will illustrate by examples that these rules cause several problems. Finally this leads to a restricted version of the following typing rules.
Definition 6.1.21 Naive typing rules for function definitions:

\[
\frac{(\Gamma; x:t_i, \Gamma_i \vdash \pi(\text{lhs}(r_i)[F \leftarrow x]): t_i; \ldots; \Gamma; x:t, \Gamma_i \vdash \pi(\text{rhs}(r_i)[F \leftarrow x]): t_i)}{\Gamma; F := \overline{F_i} \pi F : t} \quad F \notin \text{consts}(\Gamma), \quad \text{ctxth}(\Gamma_i, \text{lhs}(r_i), t).
\]

\[
\frac{(\Gamma; x:t, \Gamma_i \vdash \pi(\text{lhs}(r_i)[F \leftarrow x]): t_i; \ldots; \Gamma; x:t, \Gamma_i \vdash \pi(\text{rhs}(r_i)[F \leftarrow x]): t_i)}{\Gamma; F := \overline{F_i} \pi a : u} \quad F \notin \text{consts}(\Gamma), \quad \text{ctxth}(\Gamma_i, \text{lhs}(r_i), t).
\]

Example 6.1.22 Let \( \Delta_{\text{Leq}} \) be the function definition of Example 6.1.10. Using the naive typing rules for function definitions of Definition 6.1.21 we can derive \( \Delta_{\text{Leq}} \vdash \pi \text{Leq} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool} \), because both sides of each rule have the same type:

\[
d_{\text{leq}, y : \text{Nat}} \vdash \pi \text{leq} \ O \ y : \text{Bool} \quad d_{\text{leq}, y : \text{Nat}} \vdash \pi \text{True} : \text{Bool}
\]

\[
d_{\text{leq}, x : \text{Nat}} \vdash \pi \text{leq} \ (S \ x) \ O : \text{Bool} \quad d_{\text{leq}, x : \text{Nat}} \vdash \pi \text{False} : \text{Bool}
\]

\[
d_{\text{leq}, x : \text{Nat}} \vdash \pi \text{leq} (S \ x) \ (S \ y) : \text{Bool} \quad d_{\text{leq}, x : \text{Nat}, y : \text{Nat}} \vdash \pi \text{leq} \ x \ y : \text{Bool}
\]

, where \( d_{\text{leq}} = \text{leq} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool} \) is the variable declaration for the function.

Example 6.1.23 We can also prove properties by a function definition. For instance, the induction principle for natural numbers can be proved by:

\[
\text{NatInd} : \forall p \in \text{Nat} \rightarrow *(p \ O) \rightarrow (\forall n \in \text{Nat}.(p \ n) \rightarrow (p(S \ n))) \rightarrow \forall n \in \text{Nat}. p \ n =
\]

\[
\text{NatInd} p \text{ base step O} \Rightarrow \text{base},
\]

\[
\text{NatInd} p \text{ base step (S n)} \Rightarrow \text{step n (NatInd p base step n)}
\]

Problem 6.1.24 In Pure Type Systems a typable function is a total function. Unfortunately, the naive typing rules for function definitions of Definition 6.1.21 allow us to derive a type for \text{partial} functions. For instance the following function definition of the predecessor is typable:

\[
P\text{red} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \Rightarrow (S \ x)
\]

We can also give a ‘proof’ by not considering all cases. For instance, we can ‘prove’ that all natural numbers are \( O \). Let \( =_{\text{Nat}} \) be Leibniz equality on natural numbers (see Example 3.2.3) and \( =_{\text{refl}} \) the proof of its reflexivity. Then the following definition is typable:

\[
\text{AllZero} : \forall n \in \text{Nat}. n =_{\text{Nat}} O =_{\text{AllZero}} O \Rightarrow =_{\text{refl}} O.
\]

To avoid this problem we must impose an extra condition on function definitions that guarantees that a function is defined for all possible arguments. As only the non-variable patterns in the left-hand sides of the rules of a function definition impose restrictions on arguments, such a condition only needs to consider these patterns.

Definition 6.1.25 A rule \( Fp_1 \ldots p_n \Rightarrow t \) has matching position \( i \) of sort \( s \) if \( p_i = Cq_1 \ldots q_k \) for some \( C \in \Sigma \) of sort \( s \). A function definition \( F : t = \overline{F} \) has matching position \( i \) of sort \( s \) if \( r_j \) has matching position \( i \) of sort \( s \), for some \( j \).
2. Let \( F; t = r \) be a function definition with \( k \) matching positions \( m_1 \) of sort \( s_1, \ldots, m_k \) of sort \( s_k \). \( F; t = r \) is exhaustive if for all patterns \( p_1, \ldots, p_n \in \mathcal{P} \) such that \( (\epsilon_{r_{m_i}; s_i})_{i=1}^k \) we have \( \text{subst(lhs}(r_j), \sigma) = Fp_1 \ldots p_n \), for some \( j, \sigma \).

Thus if a function definition is exhaustive, the function is defined for all (type correct, canonical) arguments. Recall that we have a similar notion for Constructor Systems (see Definition 5.2.3).

**Example 6.1.26** Let us consider the function definition of \( \text{Leq} \) in Example 6.1.10. This definition has two matching positions of sort \( \text{Nat} \). A pattern \( p \) of type \( \text{Nat} \) in context \( \epsilon \) is either \( O \) or of the form \( S\, p' \). Using this observation one can easily verify that the function definition of \( \text{Leq} \) is exhaustive.

The function definition for \( \text{Pred} \) is *not* exhaustive, because \( \text{Pred} \, O \) is not an instance of the left-hand side of the only rule \( \text{Pred} \, (S\, x) \Rightarrow x \).

**Problem 6.1.27** By definition the result of computing the value of a mathematical function for an argument is unique. Unfortunately, one can specify ‘functions’ that have more than one result for certain arguments. For instance, consider the following function definition:

\[
\Delta_{\text{Last}} =
\begin{align*}
\text{LastList} & \rightarrow \text{Nat} = \\
\text{Last} (\text{Cons} \, a \, \text{Nil}) & \Rightarrow a, \\
\text{Last} (\text{Cons} \, a \, l) & \Rightarrow \text{Last} \, l, \\
\text{Last} \, \text{Nil} & \Rightarrow O
\end{align*}
\]

The value of \( \text{Last}(\text{Cons} \, (S \, O) \, \text{Nil}) \) is *not* unique:

\[
\begin{align*}
\Delta_{\text{Last}} \vdash \text{Last}(\text{Cons} \, (S \, O) \, \text{Nil}) & \rightarrow_{\pi_0} (S \, O), \text{if we use the first rule;} \\
\Delta_{\text{Last}} \vdash \text{Last}(\text{Cons} \, (S \, O) \, \text{Nil}) & \rightarrow_{\pi_0} O, \text{if we use the last two rules.}
\end{align*}
\]

This problem is caused by overlapping rules. We have two options to solve this problem:

1. forbid overlapping rules in a function definition.
2. use priority rewriting.

The first solution is not very practical, because it can blow up the number of rules in a function definition. This is illustrated in the following example.

**Example 6.1.28** Consider the algebraic data type \( \text{Mon} \) for terms in a monoid:

\[
\begin{array}{|c|c|}
\hline
\text{sort} & \text{Mon} \\
\hline
\text{func} & \text{Unit: Mon} \\
\hline
\text{Op:} & \text{Mon} \times \text{Mon} \rightarrow \text{Mon} \\
\hline
\text{Var:} & \text{Nat} \rightarrow \text{Mon} \\
\hline
\end{array}
\]
A function which shifts a double application of \( \text{Op} \) to the right can be defined as follows:

\[
\begin{align*}
\text{ShiftOp}: \text{Mon} & \rightarrow \text{Mon} = \\
\text{ShiftOp} \ (\text{Op} \ (\text{Op} \ x \ y)) & \Rightarrow \text{Op} \ x \ (\text{Op} \ y \ z), \\
\text{ShiftOp} \ x & \Rightarrow x
\end{align*}
\]

We can disambiguate this function definition, if we use the second rule as a default rule which may only be applied if the first rule cannot become applicable. If we want to define this function by non-overlapping rules, we need four rules to specify the cases in which the first rule is not applicable.

The basic idea of priority rewriting is that the textual order of the rules determines which rule may be applied. To prevent unwanted applications of a rule we must be sure that textually preceding rules cannot become applicable. For instance, we may not apply the second rule of \( \text{Last} \) to rewrite \( \text{Last}((\text{Cons} \ (S \ O)) \ ((\lambda: \text{List}.l) \ \text{Nil})) \), because after one \( \rightarrow_b \)-step we obtain a redex for the first rule.

In general it is not decidable whether an \( F \)-rule will become applicable by reducing the arguments of the term \( Ft_1 \ldots t_n \). For a good definition of priority rewriting we need a decidable criterion which guarantees that a rule cannot become applicable by internal reduction. We can use the fact that constructors of data types \( C \in \Sigma \) have no computational meaning. Thus reducing a term \( Ct_1 \ldots t_n \) always yields a term of the form \( Ct_1' \ldots t_n' \).

**Definition 6.1.29**  
1. Two pseudo terms \( t, u \) are strongly incompatible, notation \( t \#_s u \), if \( t = C_1t_1 \ldots t_m \) and \( u = C_2u_1 \ldots u_n \), for \( C_1, C_2 \in \Sigma \), and either:

   a. \( C_1 \neq C_2 \), or
   
   b. \( C_1 = C_2 \) and
      i. \( t_i \#_s u_i \), for some \( i \).

2. Two pseudo terms \( t, u \) have an incompatible argument, notation \( t \#_{\text{arg}} u \), if \( t = Ft_1 \ldots t_n \) and \( u = Fu_1 \ldots u_n \), for some \( F \in C \), and \( t_i \#_s u_i \) for some \( i \).

Notice the similarity with strong incompatibility for Constructor Systems (see Definition 5.3.3).

**Example 6.1.30** We have \( \text{Cons} \ (S \ O) \ (\text{Cons} \ b \ m) \#_s \text{Cons} \ a \ \text{Nil} \), but not \( (\lambda: \text{List}.l) \ \text{Nil} \#_s \text{Nil} \).

Thus we have \( \text{Last}((\text{Cons} \ (S \ O) \ (\text{Cons} \ b \ m)) \#_{\text{arg}} \text{Last}((\text{Cons} \ a \ \text{Nil}) \), but not \( \text{Last}((\text{Cons} \ (S \ O) \ ((\lambda: \text{List}.l) \ \text{Nil})) \#_{\text{arg}} \text{Last}((\text{Cons} \ a \ \text{Nil})) \).

**Remark 6.1.31** Notice that if \( t_1 \#_s t_2 \) and either

- \( t_1' = \text{subst}(t_1, \bar{s}) \), for some substitution sequence \( \bar{s} \), or
- \( t_1 \rightarrow_b t_1' \), or
Higher Order Logic with Pattern Matching

- $\Gamma \vdash t_1 \rightarrow_{\pi} t'_1$, for some pseudo context $\Gamma$

then $t'_1 \#_s t_2$.

**Definition 6.1.32** Let $F : t \Rightarrow r$ be a function definition. An $r_i$-reduction step $u_1 \Rightarrow u_2$ is enabled for $F : t \Rightarrow r_i$ if $\text{lhs}(r_j) \#_{\pi}^{\Delta s} u_1$ for all $j < i$.

Notice the similarity with enabled reduction for Priority Constructor Systems (see Definition 5.3.5).

**Example 6.1.33** The reduction step $\text{Last}(\text{Cons} (S \ O) (\text{Cons} b m)) \Rightarrow \text{Last}(\text{Cons} b m)$ is enabled for the definition of $\text{Last}$.

But the reduction step $\text{Last}(\text{Cons} (S \ O) ((\lambda l : \text{List}.l) \text{Nil})) \Rightarrow \text{Last}((\lambda l : \text{List}.l) \text{Nil})$ is not enabled for this definition, because the first rule does not have an incompatible argument with this redex.

**Definition 6.1.34** The notion of $\pi$-reduction in a pseudo context is defined as follows:

If we write $\Gamma \vdash t \rightarrow_{\beta, \pi} u$, we mean $t \rightarrow_{\beta} u$ or $\Gamma \vdash t \rightarrow_{\pi} u$. Similarly we will write $\Gamma \vdash t \rightarrow_{\beta, \pi} u$ and $\Gamma \vdash t =_{\beta, \pi} u$.

**Example 6.1.35** Let $\Delta_{\text{Last}}$ be the function definition for $\text{Last}$.

Then $\Delta_{\text{Last}} \vdash \text{Last}(\text{Cons} (S \ O) (\text{Cons} b m)) \rightarrow_{\pi} \text{Last}(\text{Cons} b m)$.

But not $\Delta_{\text{Last}} \vdash \text{Last}(\text{Cons} (S \ O) ((\lambda l : \text{List}.l) \text{Nil})) \rightarrow_{\pi} \text{Last}((\lambda l : \text{List}.l) \text{Nil})$.

We would like to use the computational meaning of a function in a proof. This can be achieved if we allow type checking modulo $\pi$-reduction. For this purpose we replace the ($\beta$-conversion) rule with the following typing rule.

**Definition 6.1.36** Typing rule for $\beta, \pi$-conversion:

$$
\begin{array}{c}
(\beta, \pi\text{-conversion}) \\
\Gamma \vdash_{\pi} a : s \\
\Gamma \vdash_{\pi} t : b \\
\Gamma \vdash_{\pi} a =_{\beta, \pi} b \\
s \in \text{Universes}
\end{array}
$$

**Example 6.1.37** We can embed the algebraic data type $\text{Bool}$ in the propositions $\ast$ as follows:

$$
\Delta_{\text{IsTrue}} =
\begin{array}{c}
\text{IsTrue} \text{Bool} \rightarrow \ast = \\
\text{IsTrue True} \Rightarrow \Pi_{p \ast} \rightarrow p, \\
\text{IsTrue False} \Rightarrow \Pi_{p \ast} \rightarrow p
\end{array}
$$

We will illustrate the use of the ($\beta, \pi$-conversion) rule in a proof of the inequality of $\text{True}$ and $\text{False}$. Thus we will construct a proof-object of type $\neg(\text{True} =_{\text{Bool}} \text{False})$. The definition of $\neg$ can be found in Example 3.2.2, and for the definition $=_{\text{Bool}}$, which denotes the Leibniz equality on booleans, we refer to Example 3.2.3. Notice that $\text{True} =_{\text{Bool}} \text{False}$ implies $(\text{IsTrue True}) \rightarrow (\text{IsTrue False})$. We can reduce this statement as follows:
\[ \Delta_{\text{IsTrue}} \vdash (\text{IsTrue True}) \rightarrow (\text{IsTrue False}) \rightarrow^\pi (\Pi_{x^*} p \rightarrow p) \rightarrow (\Pi_{x^*} p) \]

As we have \( \Delta_{\text{IsTrue}}, a: (\text{True} =_{\text{Bool}} \text{False}) \vdash^\pi a \text{IsTrue} : (\text{IsTrue True}) \rightarrow (\text{IsTrue False}), \) and \( \Delta_{\text{IsTrue}}, a: (\text{True} =_{\text{Bool}} \text{False}) \vdash^\pi (\Pi_{x^*} p \rightarrow p) \rightarrow (\Pi_{x^*} p) : \ast, \) we obtain by \((\beta, \pi\text{-conversion})\):

\[ \Delta_{\text{IsTrue}}, a: (\text{True} =_{\text{Bool}} \text{False}) \vdash^\pi a \text{IsTrue} : (\Pi_{x^*} p \rightarrow p) \rightarrow (\Pi_{x^*} p) \]

By (application) and a proof of the true proposition we obtain a proof of the false proposition:

\[ \Delta_{\text{IsTrue}}, a: (\text{True} =_{\text{Bool}} \text{False}) \vdash^\pi a \text{IsTrue} (\lambda_{x^*}. \lambda x.p.x) : (\Pi_{x^*} p) \]

Thus we have by (abstraction):

\[ \Delta_{\text{IsTrue}} \vdash^\pi \lambda a:(\text{True} =_{\text{Bool}} \text{False}). a \text{IsTrue} (\lambda_{x^*}. \lambda x.p.x) : \neg (\text{True} =_{\text{Bool}} \text{False}) \]

Using the embedding of the booleans in the propositions we can use boolean decision procedures in formal proofs.

**Example 6.1.38** Recall the definition of \( \text{Leq} \) of Example 6.1.10. The value of this boolean relation for canonical arguments is available in proofs via the \((\beta, \pi\text{-conversion})\) rule. Notice that the proof of the true proposition is a proof for all valid statements \( \text{Leq} \) \( m \) \( n \). For instance, we can derive \( \Delta_{\text{Leq}}, \Delta_{\text{IsTrue}} \vdash^\pi \lambda_{x^*}. \lambda x.p.x : \text{IsTrue}(\text{Leq}(S O)(S(S O))) \), because we have:

\[ \Delta_{\text{Leq}}, \Delta_{\text{IsTrue}} \vdash^\pi \lambda a: (\text{IsTrue}(\text{Leq}(S O)(S(S O)))) : \ast, \]

\[ \Delta_{\text{Leq}}, \Delta_{\text{IsTrue}} \vdash^\pi \lambda_{x^*}. \lambda x.p.x : (\Pi_{x^*} p \rightarrow p), \]

\[ \Delta_{\text{Leq}}, \Delta_{\text{IsTrue}} \vdash^\pi (\text{IsTrue}(\text{Leq}(S O)(S(S O)))) \rightarrow^\ast (\Pi_{x^*} p \rightarrow p) \]

We can use the identity function to prove \( \neg (\text{Leq} \) \( m \) \( n \) \) for non valid statements \( \text{Leq} \) \( m \) \( n \). For instance, we can derive: \( \Delta_{\text{Leq}}, \Delta_{\text{IsTrue}} \vdash^\pi \lambda a: (\neg (\text{IsTrue}(\text{Leq}(S O)(O)))) \), because we have:

\[ \Delta_{\text{Leq}}, \Delta_{\text{IsTrue}} \vdash^\pi (\text{IsTrue}(\text{Leq}(S O)(O))) \rightarrow (\Pi_{x^*} p) : \ast, \]

\[ \Delta_{\text{Leq}}, \Delta_{\text{IsTrue}} \vdash^\pi \lambda a: (\text{IsTrue}(\text{Leq}(S O)(O))), a : (\neg (\text{IsTrue}(\text{Leq}(S O)(O)))) \rightarrow (\text{IsTrue}(\text{Leq}(S O)(O))) \]

\[ \Delta_{\text{Leq}}, \Delta_{\text{IsTrue}} \vdash^\pi (\text{IsTrue}(\text{Leq}(S O)(O))) \rightarrow^\ast \Pi_{x^*} p \]

**Problem 6.1.39** In order to use the system for proof checking we must have decidable type checking. Unfortunately, the \((\beta, \pi\text{-conversion})\) rule threatens the decidability of type checking, because it is possible to encode an unsolved mathematical problem as a type checking problem.

Let \( U \) be a decidable property on natural numbers, such that \( \exists n \ (Un) \) is an unsolved mathematical problem.

1. Define a function \( F_u : \text{Nat} \rightarrow \text{Bool} \) which encodes \( U \).
2. Define a function \( \text{Some}_u \) which computes \( \exists n \ (Un) \) as:

\[ \text{Some}_u : \text{Nat} \rightarrow \text{Bool} = \text{Some}_u \ n \rightarrow \text{Or}(F_u \ n) \ (\text{Some}_u \ (S \ n)) \]
3. Let $\Delta$ be a legal pseudo context containing the definitions of $\text{IsTrue}$, $\text{Or}$ $F_u$, and $\text{Some}_u$. We can solve the type checking problem $\Delta \vdash \lambda x: p. \lambda x: p. x: \text{IsTrue}(\text{Some}_u 0)$ if and only if we can solve the problem $\exists n \ (Un)$.

The problem is caused by the possibility to define functions with infinite computational behaviour. Therefore we should not allow such functions in order to keep type checking decidable. Since it would not be practical to demand a termination proof for each function definition, we prefer to restrict ourselves to an easily recognizable class of functions which have a finite computational behaviour. Therefore we introduce a relation $<_s$ on pseudo terms, that is similar to the notion of ‘structurally smaller’ for Constructor Systems, presented in Definition 5.2.13.

**Definition 6.1.40**
1. A binary relation $<_s$ on pseudo terms is defined as follows:
   
   (a) $t_i <_s Ct_1 \ldots t_n$.
   
   (b) $Ct_1 \ldots t_n <_s Ct_1 \ldots t_{i-1} u t_{i+1} \ldots t_n$, if $t_i <_s u$.
   
   (c) $t_1 <_s t_3$, if $t_1 <_s t_2$ and $t_2 <_s t_3$.

   If $t <_s u$, we say that $t$ is **structurally smaller** than $u$.

2. Let $<_s^{\text{lex}}$ be the **lexicographical** extension of $<_s$ (similarly defined as in Definition 5.2.12).
   
   A rule $F p_1 \ldots p_n \Rightarrow t$ is **argument decreasing**, if each occurrence of $F$ in $t$ is in a pseudo-term $Ft_1 \ldots t_n$, such that $t_1, \ldots, t_n <_s^{\text{lex}} p_1, \ldots, p_n$.
   
   A function definition $F: u = F$ is **argument decreasing**, if all $r_i$ are argument decreasing.

**Example 6.1.41** We have $l <_s \text{Cons} \ (S \ n) \ l$, but not $O <_s S \ n$. The function definition for $\text{Leq}$ of Example 6.1.10 is argument decreasing. The function definition for $\text{Some}_u$ is not.

Now we can present the final typing rules for function definitions that replace the rules of Definition 6.1.21.

**Definition 6.1.42** Typing rules for function definitions:
Definition 6.1.43 The ternary relation $\vdash_\pi$ is defined by the typing rules of Definition 6.1.17, 6.1.36 and 6.1.42.

Example 6.1.44 The function definitions for \textit{Leq} of Example 6.1.10, and \textit{NatInd} of Example 6.1.23 are typable using the rules of Definition 6.1.42. The function definitions for \textit{Pred}, and \textit{AllZero} of Problem 6.1.24 are \textit{not} typable with the typing rules above, because of the demand for exhaustiveness. The function definition for \textit{Some}_\_\_ of Problem 6.1.39 is \textit{not} typable as it is not argument decreasing.

Now that all typing rules have been defined we can formally describe our extension of $\lambda$HOL.

Definition 6.1.45 \textit{Higher Order Logic with Pattern Matching} is the tuple $(\mathcal{T}_\pi, \mathcal{X}_\pi, \{\rightarrow_\theta, \rightarrow_\sigma\}, \vdash_\pi)$.

We will use the abbreviation $\lambda$HOL$_\pi$ for the formal language Higher Order Logic with Pattern Matching.

Remark 6.1.46 The function definitions of $\lambda$HOL$_\pi$ impose less restrictions on the recursive calls of the rules than the guarded definitions of Coq (see Section 1.3):

1. In a guarded definition one argument is selected for iteration, whereas all arguments in a function definition may be used.

2. The rules in a function definition must be argument decreasing. This restriction is based on a lexicographical extension of the relation ‘structurally smaller’, which is less restrictive than the relation ‘subterm’. If we consider guarded definitions that have an inhabitant of an algebraic data type on the ‘iterating position’ then the recursive calls must have a subterm of this term on that position.
6.2 Properties of the Extended System

In the previous section we defined an extension of the type system λHOL with algebraic data types and function definitions by pattern matching. We will show that this type system has several nice properties. Recall that we call a pseudo context legal if some statement can be derived from it. First we prove that the reductions induced by a function definition in a legal context are confluent. This means that each pseudo term has at most one normal form in a legal context. Next we show that each legal term has a unique type modulo rewriting. Together with confluence this property imposes restrictions on the possible types of a term. This is important for type checking, that is verifying whether a term has a certain type. Finally we prove that the system has the property of subject reduction; this means that rewriting a legal term yields a term with the same type. The type of a term provides information about this term. Therefore it is important that rewriting a term does not change its type.

First we show that the system handles constants well. A constant is either the defined constant of a function definition, or an element of Universes, and occurs at most once as defined constant of a function definition in a legal context.

Lemma 6.2.1 1. If Γ is a legal context then Universes ∩ consts(Γ) = ∅.

2. If Γ₁, F : t = r, Γ₂ is a legal context then F ∉ consts(Γ₁) ∪ consts(Γ₂).

3. If Γ₁, x : t, Γ₂ is a legal context then x ∉ FV(Γ₁) ∪ FV(Γ₂).

Proof

1. As Γ is legal we have Γ ⊢ x t : u for some u, t ∈ T₁. The result follows with induction on the derivation of Γ ⊢ x t : u. We treat one case. The (fun-rule) is applied last: We have Γ, F : t = r, Γ₁ ⊢ x i : tᵢ and Γ₂, x : tᵢ ⊢ rhs(rᵢ)[F ← x] : tᵢ for all i ≤ |r|, and F ∉ consts(Γ) and F ∈ C. If G ∈ consts(Γ, F : t = r) then we have two cases.

   Either G ∈ consts(Γ). Then we obtain by the induction hypothesis G ∉ Universes.

   Or G = F. As F ∈ C we have F ∉ Universes.

2. As Γ₁, F : t = r, Γ₂ is a legal context we have Γ₁, F : t = r, Γ₂ ⊢ u₁ : u₂ for some u₁, u₂ ∈ T₁. The result follows with induction on the derivation of Γ₁, F : t = r, Γ₂ ⊢ u₁ : u₂.

3. As Γ₁, x : t, Γ₂ is a legal context we have Γ₁, x : t, Γ₂ ⊢ u₁ : u₂ for some u₁, u₂ ∈ T₁. The result follows with induction on the derivation of Γ₁, x : t, Γ₂ ⊢ u₁ : u₂.

First we prove a generalization of the (var-weak) rule, and that the predicate of a variable declaration in a legal context has a type universe as type, and that the statements induced by Axioms are derivable in any legal context.
Lemma 6.2.2 Let $\Gamma, \Delta$ be pseudo contexts. Let $s \in$ Universes. Let $x \in V_s$.

1. If $\Gamma, \Delta \vdash_{\pi} t : u$ and $\Gamma \vdash_{\pi} a : s$ and $x \not\in FV(\Gamma, \Delta)$ then $\Gamma, x : a, \Delta \vdash_{\pi} t : u$.
2. If $\Gamma, x : a, \Delta \vdash_{\pi} t : u$ then $\Gamma, x : a \vdash_{\pi} x : a$ and $\Gamma \vdash_{\pi} a : s$.
3. If $\Gamma \vdash_{\pi} t : u$ and $(s_1, s_2) \in$ Axioms then $\Gamma \vdash_{\pi} s_1 : s_2$.
4. If $\Gamma \vdash_{\pi} t : u$ then $\Gamma \vdash_{\pi} s' : \Box$ for $s' \in S$.

Proof

1. The case $\Delta = \epsilon$ is handled by the (var-weak) rule. We show with induction on the derivation of $\Delta' \vdash_{\pi} t : u$: if $\Delta' = \Gamma, \Delta$ and $\Delta \neq \epsilon$ and $\Gamma \vdash_{\pi} a : s$ and $x \not\in FV(\Delta')$ then $\Gamma, x : a, \Delta' \vdash_{\pi} t : u$.
2. By induction on the derivation of $\Gamma, x : a, \Delta \vdash_{\pi} t : u$.
3. By induction on the derivation of $\Gamma \vdash_{\pi} t : u$.
4. By induction on the derivation of $\Gamma \vdash_{\pi} t : u$.

Before we can express how substitution influences the typing relation, we need to define substitution for pseudo contexts.

Definition 6.2.3 Substitution of a pseudo term $u \in T_\pi$ for a variable $x$ in a pseudo context $\Gamma \in \mathcal{X}_\pi$, notation $\Gamma[x := u]$, is defined as follows:

1. $\epsilon[x := u] = \epsilon$.
2. \(\langle \Delta, y : t \rangle[x := u] = \langle \Delta[x := u], y : t[x := u] \rangle\).
3. \(\langle \Delta, F : t = \overline{F} \rangle[x := u] = \langle \Delta[x := u], F : t[x := u] = \overline{F} \rangle\).

The following lemma shows that replacing a variable with a term of the same type in a legal term yields a legal term.

Lemma 6.2.4 (substitution) Let $a, b, c, d \in T_\pi$ be pseudo terms, $x, y \in V$ variables, and $\Gamma, \Delta \in \mathcal{X}_\pi$ pseudo contexts.

1. $b[x := c][y := d] = b[y := d][x := c[y := d]]$, if $x \neq y$ and $x \not\in FV(d)$.
2. If $\Gamma, x : a, \Delta \vdash_{\pi} b : c$ and $\Gamma \vdash_{\pi} d : a$ then $\Gamma, \Delta[x := d] \vdash_{\pi} b[x := d] : c[x := d]$.

Proof

1. Structural induction on $b$.
2. Induction on the derivation of $\Gamma, x : a, \Delta \vdash_{\pi} b : c$ using part 1.
Lemma 6.2.5 Let \( \Gamma \) be a legal context. If \( F \vdash t = \bar{r} \in \text{fundefs}(\Gamma) \) then

1. \( \Gamma \vdash x \in F : t \).

2. \( \Gamma, \Gamma_i \vdash \text{lhs}(r_i) : t_i \) and \( \Gamma, \Gamma_i \vdash \text{rhs}(r_i) : t_i \), and \( \text{ctxthls}(\Gamma_i, \text{lhs}(r_i), t_i) \). for some pseudo context \( \Gamma_i \), and \( t_i \in T\pi \), all \( i \leq |\bar{r}| \).

3. \( F : t = \bar{r} \) is exhaustive and argument decreasing.

Proof
As \( \Gamma \) is legal we have \( \Gamma \vdash x : u \) for some \( t, u \in T\pi \). The results follow with induction on the derivation of \( \Gamma \vdash x : u \).

The following lemma analyses how a type assignment can be obtained.

Lemma 6.2.6 (generation) Let \( \Gamma \in \mathcal{X}\pi \) be a pseudo context and \( t \in T\pi \) a pseudo term.

1. If \( \Gamma \vdash s : t \) for \( s \in S \) then \( \Gamma \vdash t =_{\beta, \pi} s \).

2. If \( \Gamma \vdash C : t \) for \( C \in \Sigma \) with \( \tau(C) = s \) then \( \Gamma \vdash t =_{\beta, \pi} s \).

3. If \( \Gamma \vdash C : t \) for \( C \in C \), then either:
   
   \begin{enumerate}
   \item[(a)] \( C : s \in \text{Axioms and } \Gamma \vdash t =_{\beta, \pi} s \) for some \( s \in \text{Universes} \); or
   \item[(b)] \( C : u = \bar{r} \in \Gamma \) and \( \Gamma \vdash t =_{\beta, \pi} u \) for some \( u \in T\pi, \bar{r} \in \mathcal{R}^+ \).
   \end{enumerate}

4. If \( \Gamma \vdash x : v \) for \( v \in V\pi \) then \( \Gamma \vdash u : s \) and \( \Gamma \vdash u =_{\beta, \pi} t \) and \( v : u \in \Gamma \), for some \( u \in T\pi \).

5. If \( \Gamma \vdash (\Pi x : a. b) : t \) then \( \Gamma \vdash a : s_1, (\Gamma, x : a) \vdash b : s_2 \) and \( \Gamma \vdash t =_{\beta, \pi} s_3 \), for some \( (s_1, s_2, s_3) \in \text{Rules} \).

6. If \( \Gamma \vdash (\lambda x : a. b) : t \) then \( \Gamma \vdash (\Pi x : a. u) : s \), \( (\Gamma, x : a) \vdash b : u \) and \( \Gamma \vdash t =_{\beta, \pi} \Pi x : a. u \), for some \( u \in T\pi, s \in \text{Universes} \).

7. If \( \Gamma \vdash (fa) : t \) then either:
   
   \begin{enumerate}
   \item[(a)] \( f = Ca_1 \ldots a_{m-1}, \text{for } C \in \Sigma \) with \( \tau(C) = s_1 \times \ldots \times s_m \rightarrow s \), \( \Gamma \vdash t =_{\beta, \pi} s \), and
   \( \Gamma \vdash a_i : s_i \) (\( a = a_m \)), all \( i \leq m \); or
   \item[(b)] \( \Gamma \vdash f : (\Pi x : u_1, u_2), \Gamma \vdash a : u_1, \text{and } \Gamma \vdash t =_{\beta, \pi} u_2[x := a] \), for some \( u_1, u_2 \in T\pi \).
   \end{enumerate}

Proof
Induction on the derivation. Notice that except for the weakening rules (\( \text{var-weak} \)), (\( \text{fun-weak} \)), and (\( \beta, \pi \)-conversion), the structure of the term determines which typing rules may have been applied in the derivation.

A legal type either has a type universe as type, or is a type universe. The domain and range of a legal function type have a universe as type.
Corollary 6.2.7 1. If $\Gamma \vdash t : u$ then $u = \Delta$ or $\Gamma \vdash t : s$, for some $s \in \text{Universes}$.

2. If $\Gamma \vdash f : (\Pi x : t . u)$ then $\Gamma \vdash t : s_1$ and $\Gamma , x : t \vdash u : s_2$ and $(s_1 , s_2 , s_3) \in \text{Rules}$ for some $s_1 , s_2 , s_3 \in \text{Universes}$.

Proof

1. By induction on the derivation of $\Gamma \vdash t : u$ using Lemma 6.2.2, 6.2.4 and 6.2.6.

2. By the previous and 5. of Lemma 6.2.6. \hfill \Box

Confluence

In this subsection we will consider a legal context $\Gamma$. We will show that the reduction relation $\Gamma \vdash t_1 \rightarrow \rightarrow \rightarrow t_2$ is confluent. The main idea behind the proof is that parallel reduction of redexes of the same reduction rule is a subcommutative relation.

Definition 6.2.8 Let $F : t = \bar{r} \in \Gamma$. Let $i \leq |\bar{r}|$. The relation $\rightarrow_{F,i}$ is defined as follows:

1. $t_1 \rightarrow_{F,i} t_2$, if subst(lhs($r_i$), $\bar{\sigma}$) = $t_1$, subst(rhs($r_i$), $\bar{\sigma}$) = $t_2$, and $t_1 \#_s \text{lhs}(r_j)$, all $j < i$.

2. $t_1 \rightarrow_{F,i} t_1$.

3. If $t_1 \rightarrow_{F,i} t_3$ and $t_2 \rightarrow_{F,i} t_4$ then:
   
   (a) $t_1 t_2 \rightarrow_{F,i} t_3 t_4$.
   
   (b) $\lambda x : t_1 . t_2 \rightarrow_{F,i} \lambda x : t_3 . t_4$.
   
   (c) $\Pi x : t_1 . t_2 \rightarrow_{F,i} \Pi x : t_3 . t_4$.

We will write $t_1 \rightarrow_{F,i} t_2$ if $t_1 \rightarrow_{F,i} t_2$, for some $F , i$.

Remark 6.2.9 Notice that the following properties hold:

1. If $\Gamma \vdash t_1 \rightarrow t_2$ then $t_1 \rightarrow t_2$.

2. If $t_1 \rightarrow t_2$ then $\Gamma \vdash t_1 \rightarrow t_2$.

First we show that a term can only be an enabled redex for one rule.

Lemma 6.2.10 Let $F : t = \bar{r} , G : u = \bar{q} \in \text{fundefs}(\Gamma)$. If $a \Rightarrow b_1$ is an enabled $r_k$-reduction step for $F : t = \bar{r}$ and $a \Rightarrow b_2$ is an enabled $q_l$-reduction step for $G : u = \bar{q}$, then $b_1 = b_2$.

Proof

As $a \Rightarrow b_1$ is an enabled $r_k$-reduction step for $F : t = \bar{r}$, we have $a = \text{subst}(\text{lhs}(r_k), \bar{\sigma})$, $b_1 = \text{subst}(\text{rhs}(r_k), \bar{\sigma})$, and $\text{lhs}(r_j) \#_s a$, all $j < k$. As $a \Rightarrow b_2$ is an enabled $q_l$-reduction step for $G : u = \bar{q}$, we have $a = \text{subst}(\text{lhs}(q_l), \bar{\tau})$, $b_2 = \text{subst}(\text{rhs}(q_l), \bar{\tau})$, and $\text{lhs}(q_j) \#_s a$, all $j < l$. Thus we have $F = G$ and also $\bar{r} = \bar{q}$, because a function can only be defined once
in a legal context. We must have $k = l$, because we cannot have $h \not \equiv^a \alpha$ and $h = a$. Thus $b_1 = b_2$.

We can extend a relation on pseudo terms to substitution sequences, by applying it on all the pseudo terms in a substitution sequence.

**Definition 6.2.11** Let $\bar{\sigma} = (x_1, t_1), \ldots, (x_n, t_n)$ be a substitution sequence. Let $\rightarrow_X$ be a relation on pseudo terms. We write $\bar{\sigma} \rightarrow_X \bar{\tau}$, if $\bar{\tau} = (x_1, u_1), \ldots, (x_n, u_n)$ and $t_i \rightarrow_X u_i$, for all $i \leq n$.

We show that internal reduction in a redex yields a redex of the same rule.

**Lemma 6.2.12** Let $\bar{\sigma}$ be a standard substitution sequence.

1. Let $p \in P$ be a linear pattern. If $\text{sub}(p, \bar{\sigma}) \rightarrow_{F;i} u$ then $u = \text{sub}(p, \bar{\tau})$ and $\bar{\sigma} \rightarrow_{F;i} \bar{\tau}$, for some standard substitution sequence $\bar{\tau}$.

2. Assume $G; t = \bar{\tau} \in \Gamma$, $\text{lhs}(r_j) = Gp_1 \ldots p_n$, and $k < n$. If $\text{sub}(Gp_1 \ldots p_k, \bar{\sigma}) \rightarrow_{F;i} u$ then $u = \text{sub}(Gp_1 \ldots p_k, \bar{\tau})$ and $\bar{\sigma} \rightarrow_{F;i} \bar{\tau}$, for some standard substitution sequence $\bar{\tau}$.

**Proof**


2. Induction on $k$.

**Lemma 6.2.13** Let $\bar{\sigma}$ be a standard substitution sequence. If $\bar{\sigma} \rightarrow_{F;i} \bar{\tau}$ then $\text{sub}(t, \bar{\sigma}) \rightarrow_{F;i} \text{sub}(t, \bar{\tau})$.

**Proof**

Structural induction on $t$.

We show that parallel reduction is subcommutative.

**Proposition 6.2.14**

1. Assume $G; t = \bar{\tau} \in \Gamma$. If $t_1 \Rightarrow t_2$ is an enabled $r_j$-reduction step for $G; t = \bar{\tau}$ and $t_1 \rightarrow_{F;i} t_3$ then $t_2 \rightarrow_{F;i} t_4$ and $t_3 \rightarrow_{G;j} t_4$, for some $t_4$.

2. If $t_1 \rightarrow_{F;i} t_3$ and $t_1 \rightarrow_{G;j} t_2$ then $t_2 \rightarrow_{F;i} t_4$ and $t_3 \rightarrow_{G;j} t_4$, for some $t_4$.

**Proof**

1. Induction on the derivation of $t_1 \rightarrow_{F;i} t_3$. Use Lemmas 6.2.10, 6.2.12, and 6.2.13.

2. Induction on the derivation of $t_1 \rightarrow_{G;j} t_2$. Use first part of proposition.

**Corollary 6.2.15** (confluence) $\pi$-reduction is confluent.

**Proof**

Using Proposition 6.2.14 we obtain subcommutativity of $\rightarrow^\parallel_\pi$ (see Figure 6.1). By Lemma 2.1.5 we obtain confluence for $\rightarrow_\pi$. 

\[\square\]
Commuting reductions

In this subsection we will consider a legal context $\Gamma$. We will show that the reduction relation $\Gamma \vdash t_1 \rightarrow_{\pi} t_2$ commutes with $\rightarrow_{\beta}$. Unfortunately, we cannot use Lemma 2.1.7 for $\rightarrow_{\beta}$ and $\rightarrow_{\pi}$, because both relations can duplicate redexes. It is also not possible to use this lemma to prove that $\rightarrow_{\beta}$ and $\rightarrow_{\pi}$ commute, because of the following counter example.

Example 6.2.16 Let $\Delta$ be a legal context containing the definitions of the functions \texttt{Leq} (see Example 6.1.10) and \texttt{Last} (see Problem 6.1.27). Consider the term $(\lambda m:\texttt{Nat}.\texttt{Leq} \ (S \ O) \ (S \ m)) \ (\texttt{Last} \ (\texttt{Cons} \ n \ \texttt{Nil})).$ This term can be reduced as follows:

1. $(\lambda m:\texttt{Nat}.\texttt{Leq} \ (S \ O) \ (S \ m)) \ (\texttt{Last} \ (\texttt{Cons} \ n \ \texttt{Nil})) \rightarrow_{\beta} \texttt{Leq} \ (S \ O) \ (S \ (\texttt{Last} \ (\texttt{Cons} \ n \ \texttt{Nil})))$

2. $\Delta \vdash (\lambda m:\texttt{Nat}.\texttt{Leq} \ (S \ O) \ (S \ m)) \ (\texttt{Last} \ (\texttt{Cons} \ n \ \texttt{Nil})) \rightarrow_{\pi} (\lambda m:\texttt{Nat}.\texttt{Leq} \ O \ m) \ n$

For the last reduct we have $(\lambda m:\texttt{Nat}.\texttt{Leq} \ O \ m) \ n \rightarrow_{\beta} \texttt{Leq} \ O \ n$. But we cannot reduce the first reduct $\texttt{Leq} \ (S \ O) \ (S \ (\texttt{Last} \ (\texttt{Cons} \ n \ \texttt{Nil})))$ in one $\rightarrow_{\pi}$-step to $\texttt{Leq} \ O \ n$, because the two parallel redex occurrences $\texttt{Leq} \ (S \ O) \ (S \ m)$ and $\texttt{Last} \ (\texttt{Cons} \ n \ \texttt{Nil})$ of the original term are transformed into nested redex occurrences by the $\rightarrow_{\beta}$-step.

In order to overcome this problem, we will define a reduction relation $\rightarrow_{\pi}^p$ that can contract nested $\rightarrow_{\pi}$-redex occurrences, from the inside to the outside, and show that it commutes with $\rightarrow_{\beta}$. Recall Definition 6.2.11 that formalizes the extension of a relation on pseudo terms to substitution sequences.

Definition 6.2.17 The relation $\rightarrow_{\pi}^p$ is defined as follows:

1. $t_1 \rightarrow_{\pi}^p t_1$.

2. $t_1 \rightarrow_{\pi}^p t_2$, if $Ft=\bar{r} \in \Gamma$, $\text{subst}(\text{lhs}(r_i), \bar{\sigma}) = t_1$, $\text{subst}(\text{rhs}(r_i), \bar{\sigma}) = t_2$, $\bar{\sigma} \rightarrow_{\pi}^p \bar{r}$, and $t_1 \not\rightarrow_{\pi}^{\text{Arg}} \text{lhs}(r_j)$, all $j < i$.

3. If $t_1 \rightarrow_{\pi}^p t_3$ and $t_2 \rightarrow_{\pi}^p t_4$ then:
   
   (a) $t_1t_2 \rightarrow_{\pi}^p t_3t_4$. 

(b) $\lambda x.t_1.t_2 \rightarrow^\iota_\pi \lambda x.t_3.t_4$.
(c) $\Pi x.t_1.t_2 \rightarrow^\iota_\pi \Pi x.t_3.t_4$.

Remark 6.2.18 Notice that the following properties hold:

1. If $\Gamma \vdash t_1 \rightarrow^\pi t_2$ then $t_1 \rightarrow^\iota_\pi t_2$.
2. If $t_1 \rightarrow^\iota_\pi t_2$ then $\Gamma \vdash t_1 \rightarrow^\pi t_2$.

We show that $\rightarrow^\iota_\pi$ is preserved under substitution.

Lemma 6.2.19

1. If $u_1 \rightarrow^\iota_\pi u_2$ then $t[x := u_1] \rightarrow^\iota_\pi t[x := u_2]$.

2. If $t_1 \rightarrow^\iota_\pi t_2$ and $u_1 \rightarrow^\iota_\pi u_2$ then $t_1[x := u_1] \rightarrow^\iota_\pi t_2[x := u_2]$.

Proof

1. Structural induction on $t$.

2. Induction on the derivation of $t_1 \rightarrow^\iota_\pi t_2$.

We will treat one case: $t_1 = \text{sub}(\text{lhs}(r_i), \bar{s})$, $t_2 = \text{sub}(\text{rhs}(r_i), \bar{r})$, $\bar{s} \rightarrow^\iota_\pi \bar{r}$, and $t_1 \#_{\pi} \text{lhs}(r_j)$, all $j < i$. Let $\bar{s} = (x_1, s_1), \ldots, (x_n, s_n)$, and $\bar{r} = (x_1, w_1), \ldots, (x_n, w_n)$. We may assume that $\bar{s}$ is standard and that $\{x_1, \ldots, x_n\} \cap (FV(u_1) \cup \{x\}) = \emptyset$. Let $\bar{s}_2 = (x_1, s_1[x := u_1], \ldots, (x_n, s_n[x := u_1])$, and $\bar{r}_2 = (x_1, w_1[x := u_2]), \ldots, (x_n, w_n[x := u_2])$. By the induction hypothesis we have $s_j[x := u_1] \rightarrow^\iota_\pi w_j[x := u_2]$, for all $j$. Thus $\bar{s}_2 \rightarrow^\iota_\pi \bar{r}_2$. As $t_1[x := u_1] = \text{sub}(\text{lhs}(r_i), \bar{s}_2)$, $t_2[x := u_2] = \text{sub}(\text{rhs}(r_i), \bar{r}_2)$, and $t_1[x := u_1] \#_{\pi} \text{lhs}(r_j)$ all $j < i$, we have $t_1[x := u_1] \rightarrow^\iota_\pi t_2[x := u_2]$. \hfill $\Box$

We show that internal $\beta$-reduction in a redex yields a redex of the same rule.

Lemma 6.2.20 Let $\bar{s}$ be a standard substitution sequence.

1. Let $p \in \mathcal{P}$ be a linear pattern. If $\text{sub}(p, \bar{s}) \rightarrow_\beta u$ then $u = \text{sub}(p, \bar{r})$ and $\bar{s} \rightarrow^\pi \bar{r}$, for some standard substitution sequence $\bar{r}$.

2. Assume $Gt = \bar{r} \in \Gamma$. If $\text{sub}(\text{lhs}(r_j), \bar{s}) \rightarrow_\beta u$ then $u = \text{sub}(\text{lhs}(r_j), \bar{r})$ and $\bar{s} \rightarrow^\pi \bar{r}$, for some standard substitution sequence $\bar{r}$.

Proof


2. Induction on the number of arguments of $\text{lhs}(r_j)$.

Lemma 6.2.21 Let $\bar{s}$ be a standard substitution sequence. If $\bar{s} \rightarrow_\beta \bar{r}$ then $\text{sub}(t, \bar{s}) \rightarrow_\beta \text{sub}(t, \bar{r})$. \hfill $\Box$
Proof
Structural induction on $t$. \qed

We show that $\rightarrow^{io}_\pi$ is preserved under $\beta$-reduction.

Proposition 6.2.22 If $t_1 \rightarrow_\beta t_2$ and $t_1 \rightarrow^{io}_\pi t_3$ then $t_2 \rightarrow^{io}_\pi t_4$ and $t_3 \rightarrow_\beta t_4$ for some $t_4$.

Proof
Induction on the derivation of $t_1 \rightarrow^{io}_\pi t_3$.
First case $t_1 \rightarrow^{io}_\pi t_1$ is trivial.
Second case $t_1 = \text{subst}(\text{lhs}(r_i), \bar{\sigma})$ $t_3 = \text{subst}(\text{rhs}(r_i), \bar{\tau})$, $\bar{\sigma} \rightarrow^{io}_\pi \bar{\tau}$, and $t_1 \not\Rightarrow^{\text{Arg}}_\pi \text{lhs}(r_j)$, all $j < i$.
By Lemma 6.2.20 we have $t_2 = \text{subst}(\text{lhs}(r_i), \bar{\tau}_1)$, and $\bar{\sigma} \rightarrow^{io}_\pi \bar{\tau}_1$ for some substitution sequence $\bar{\tau}_1$. Using the induction hypothesis we can obtain a substitution sequence $\bar{\tau}_2$ such that $\bar{\tau} \rightarrow_\beta \bar{\tau}_2$ and $\bar{\tau}_1 \rightarrow^{io}_\pi \bar{\tau}_2$. By Lemma 6.2.21 we have $\text{subst}(\text{rhs}(r_i), \bar{\tau}) \rightarrow_\beta \text{subst}(\text{rhs}(r_i), \bar{\tau}_2)$. We also have $\text{subst}(\text{lhs}(r_i), \bar{\tau}_1) \rightarrow^{io}_\pi \text{subst}(\text{rhs}(r_i), \bar{\tau}_2)$, as $t_2 \not\Rightarrow^{\text{Arg}}_\pi \text{lhs}(r_j)$ for all $j < i$. Thus $t_2$ and $t_3$ have a common reduct.
For last case the only interesting subcase is: $t_1 = (\lambda x : u_1, u_2)u_3$, and $t_2 = u_3[x := u_3]$ and $t_3 = (\lambda x : u'_1, u'_2)u'_3$, with $u_i \rightarrow^{io}_\pi u'_i$, for $i \in \{1, 2, 3\}$. By Lemma 6.2.19 we obtain $u_2[x := u_3] \rightarrow^{io}_\pi u'_2[x := u'_3]$. By definition $t_3 \rightarrow_\beta u'_2[x := u'_3]$. The other subcases follow easily from the induction hypothesis. \qed

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (t1) {$t_1$};
  \node (t2) [right of=t1] {$t_2$};
  \node (t3) [below of=t1] {$t_3$};
  \node (t4) [right of=t3] {$t_4$};

  \draw[->] (t1) -- (t2) node[above, midway] {\scriptsize $\beta$};
  \draw[->] (t1) -- (t3) node[right, midway] {\scriptsize $\rightarrow^{io}_\pi$};
  \draw[->] (t3) -- (t4) node[above, midway] {\scriptsize $\beta$};
\end{tikzpicture}
\caption{$\rightarrow_\beta$ and $\rightarrow^{io}_\pi$ commute}
\end{figure}

Corollary 6.2.23 (commuting reductions) $\rightarrow_\beta$ and $\rightarrow^{io}_\pi$ commute.

Proof
By Lemma 2.1.7 and Proposition 6.2.22 $\rightarrow_\beta$ and $\rightarrow^{io}_\pi$ commute. The result follows from the observation that $\rightarrow^{io}_\pi = \rightarrow^{io}_\pi \pi$. \qed

Corollary 6.2.24 (confluence) The relation $\rightarrow_{\beta, \pi}$ is confluent.

Proof
As $\rightarrow_\beta$ and $\rightarrow_\pi$ commute and are confluent, we have using Lemma 2.1.6 that $\rightarrow_{\beta, \pi}$ is confluent. \qed
Properties of the Extended System

Uniqueness of types

We will prove in this subsection that a term has a unique type modulo $=_{\beta,\pi}$.

Lemma 6.2.25 Let $\Gamma$ be a legal context.

1. If $t_1 \rightarrow_{\beta} t_2$ then $t_1[x := a] \rightarrow_{\beta} t_2[x := a]$.

2. If $t_1 \rightarrow_{\beta} t_2$ then $t_1[x := a] \rightarrow_{\beta} t_2[x := a]$.

3. If $\Gamma \vdash t_1 \rightarrow_{\beta,\pi} t_2$ then $\Gamma \vdash t_1[x := a] \rightarrow_{\beta,\pi} t_2[x := a]$.

4. If $\Gamma \vdash t_1 =_{\beta,\pi} t_2$ then $\Gamma \vdash t_1[x := a] =_{\beta,\pi} t_2[x := a]$.

Proof

Using our conventions about renaming variables in pseudo terms and rules, we may assume that $FV(a) \cap (BV(t_1) \cup BV(t_2)) = \emptyset$.

1. We have $t_1 = C[(\lambda y.b)c]$ and $t_2 = C[b[y := c]]$, for some $y \neq x$. By part 1. of Lemma 6.2.4 we obtain $((\lambda y.b)c)[x := a] \rightarrow_{\beta} b[y := c][x := a]$. Thus $t_1[x := a] \rightarrow_{\beta} t_2[x := a]$.

2. Induction on the derivation of $t_1 \rightarrow_{\beta} t_2$.

3. Induction on the derivation of $\Gamma \vdash t_1 \rightarrow_{\beta,\pi} t_2$.

4. Follows from the previous, as $\rightarrow_{\beta,\pi}$ is confluent.

Before we can prove that each legal term has a unique type, we must show that the two cases of Lemma 6.2.6 for typing terms of form $(fa)$ exclude each other.

Lemma 6.2.26 Let $C \in \Sigma$ with $\tau(C) = s_1 \times \cdots \times s_m \rightarrow s$, where $1 \leq m$. If $\Gamma \vdash a_1 \cdots a_k : t$ then $k = m$ and $\Gamma \vdash t =_{\beta,\pi} s$, and $\Gamma \vdash a_i : s_i$, all $i \leq m$.

Proof

Induction on the derivation of $\Gamma \vdash a_1 \cdots a_k : t$. Assume the (application) rule has been applied last. Then we have $\Gamma \vdash a_1 \cdots a_{k-1} : \Pi x : t_1.t_2$ and $\Gamma \vdash a_k : t_1$. By the induction hypothesis we obtain $\Gamma \vdash \Pi x : t_1.t_2 =_{\beta,\pi} s$. As $s \in S$ is a normal form we have $\Gamma \vdash \Pi x.t_1.t_2 \rightarrow_{\beta,\pi} s$ by confluence of $\rightarrow_{\beta,\pi}$. As any reduc of $\Pi x.t_1.t_2$ is of form $\Pi x.t_1'.t_2'$, this is not possible. Thus the (application) rule cannot have been applied last. The only applicable rules are (data-constr) and ($\beta,\pi$-conversion) and the weakening rules.

Proposition 6.2.27 (uniqueness of types) If $\Gamma \vdash a : t_1$ and $\Gamma \vdash a : t_2$ then $\Gamma \vdash t_1 =_{\beta,\pi} t_2$.

Proof

Induction on the structure of $a$. We will only treat some of the cases.

Case $a = (u_1u_2)$.

Assume $\Gamma \vdash u_1u_2 : t_1$ and $\Gamma \vdash u_1u_2 : t_2$. Using Lemma 6.2.6 and 6.2.26 we have two excluding cases:
1. \( u_1 = C a_1 \ldots a_{l-1} \) for some \( C \in \Sigma \) with \( \tau(C) = s_1 \times \ldots \times s_1 \rightarrow s \) and \( \Gamma \vdash t_1 =_{\beta, \pi} s \) and \( \Gamma \vdash t_2 =_{\beta, \pi} s \). Or we have:

2. \( u_1 \neq C a_1 \ldots a_k \), all \( C \in \Sigma, \ 0 \leq k, a_1, \ldots, a_k \in T_\pi \). Then we have \( \Gamma \vdash u_1 : (\Pi x:w_1, w_2) \) and \( \Gamma \vdash u_2 : w_1, \) for some \( w_1, w_2 \) and \( \Gamma \vdash t_1 =_{\beta, \pi} w_2[x := u_2] \), and \( \Gamma \vdash u_1 : (\Pi x:w_3, w_4) \) and \( \Gamma \vdash u_2 : w_3, \) for some \( w_3, w_4 \) and \( \Gamma \vdash t_1 =_{\beta, \pi} w_4[x := u_2] \). By the induction hypothesis we have \( \Gamma \vdash \Pi x:w_1, w_2 =_{\beta, \pi} \Pi x:w_3, w_4 \). Thus we have \( \Gamma \vdash w_2 =_{\beta, \pi} w_4 \). By Lemma 6.2.25 we have \( \Gamma \vdash w_2[x := u_2] =_{\beta, \pi} w_4[x := u_2] \). Thus we have \( \Gamma \vdash t_1 =_{\beta, \pi} t_2 \).

Case \( a = \Pi x:u_1, u_2 \).

By the generation lemma we have \( \Gamma \vdash u_1 : s_1 \), and \( (\Gamma, v : u_1) \vdash u_2 : s_2 \), and \( (s_1, s_2, s_3) \in \) Rules, and \( \Gamma \vdash t_1 =_{\beta, \pi} s_3 \). Also we have \( \Gamma \vdash u_1 : s_4 \), and \( (\Gamma, v : u_1) \vdash u_2 : s_5 \), and \( (s_4, s_5, s_6) \in \) Rules, and \( \Gamma \vdash t_2 =_{\beta, \pi} s_6 \). By the induction hypothesis we have \( \Gamma \vdash s_1 =_{\beta, \pi} s_4 \) and \( \Gamma, v : u_1 \vdash s_2 =_{\beta, \pi} s_5 \). Thus we have \( s_1 = s_4 \) and \( s_2 = s_5 \). Thus we have \( s_3 = s_6 \). Therefore we have \( \Gamma \vdash t_1 =_{\beta, \pi} t_2 \).

**Subject reduction**

In this subsection we will prove that the type of a term is preserved under reduction. Thus if \( \Gamma \vdash t : u \) and \( \Gamma \vdash t \rightarrow_{\beta, \pi} t' \) then \( \Gamma \vdash t' : u \).

The main thing we have to show is that if \( \Gamma \vdash t : u \) and \( t \Rightarrow t' \) is a reduction step for a rule in \( \Gamma \), then \( \Gamma \vdash t' : u \). We will prove that the types of the terms substituted for the variables in the left-hand side of a rule are fully determined by the types of these variables. Using this property we will show that both sides of an instance of a rule have the same type.

First we introduce a relation that specifies which context is needed to derive a type for an instance of the left-hand side of a rule in some context.

**Definition 6.2.28** Let \( \Gamma \) be a pseudo context. A pseudo context is sufficient for a pseudo term in \( \Gamma \) is defined as follows:

1. \( \epsilon \) is sufficient for \( t \in T_\pi \) in \( \Gamma \), if \( FV(t) \subseteq FV(\Gamma) \).

2. \( x : A \) is sufficient for \( x \in V \) in \( \Gamma \), if \( x \not\in FV(\Gamma), A \in T_\pi \).

3. \( \Delta_1, \Delta_2 \) is sufficient for \( (t_1, t_2) \) in \( \Gamma \), if \( \Delta_1 \) is sufficient for \( t_1 \) in \( \Gamma \) and \( \Delta_2 \) is sufficient for \( t_2 \) in \( \Gamma \).

**Lemma 6.2.29** If \( \Gamma, \Delta \vdash x : l \) and \( \text{ctxthls}(\Delta, l, t') \) then \( \Delta \) is sufficient for \( l \) in \( \Gamma \) and \( FV(l) = FV(\Delta) \).

**Proof**

Assume \( \Gamma, \Delta \vdash x : l \) and \( \text{ctxthls}(\Delta, l \Rightarrow r, t') \). First we prove by induction on the derivation of \( \text{ctxtpat}(\Delta_1, p, t') \): if \( \Gamma, \Delta \vdash x : p : t \) and \( \Delta = \Delta_0, \Delta_1, \Delta_2 \) and \( \text{ctxtpat}(\Delta_1, p, t') \) then \( \Delta_1 \) is sufficient for \( p \) in \( \Gamma \) and \( FV(p) = FV(\Delta_1) \). The result follows with induction on the number of arguments of the rule. \( \square \)
Lemma 6.2.30 Let $\Gamma, \Delta$ be pseudo contexts. Let $t$ be a pseudo term.

1. If $\Delta$ is sufficient for $t$ in $\Gamma$ then $FV(t) \subseteq FV(\Gamma) \cup FV(\Delta)$.

2. If $\Gamma \vdash_{\pi} a : w$ and $\Gamma, x : w, \Delta \vdash_{\pi} t : u$ and $x : w, \Delta$ is sufficient for $t$ in $\Gamma$ then $\Delta[x := a]$ is sufficient for $t[x := a]$ in $\Gamma$.

Proof

1. Induction on the derivation of $\Delta$ is sufficient for $t$ in $\Gamma$.

2. Induction on the derivation of $x : w, \Delta$ is sufficient for $t$ in $\Gamma$.

The variables in an instance of a left-hand side of a rule, for which a term may be substituted, are always arguments of a function.

Definition 6.2.31 A pseudo term $a$ is an argument in a pseudo term $t$, if $t = ht_1 \ldots t_n$ for $h \in \Sigma \cup C, 1 \leq n$, and either

1. $a = t_i$, or

2. $a$ is an argument in $t_i$, for some $1 \leq i \leq n$.

Remark 6.2.32 For any rule $l \Rightarrow r$ each $x \in FV(l)$ is an argument in $l$.

A variable that is an argument of a pseudo term $t$ is also an argument of the pseudo term obtained from substituting some pseudo term for another variable in $t$.

Lemma 6.2.33 If $x \in V$ is an argument in $t$ and $y \neq x$ then $x$ is an argument in $t[y := u]$.

Proof

Induction on the proof that $x$ is an argument in $t$.

The type of a term substituted for the first variable in the left-hand side of a rule is fully determined by the type of this variable.

Lemma 6.2.34 Let $\sigma = (x_1, a_1), \ldots, (x_k, a_k)$ be a standard substitution sequence, and let $\Delta = x_1 : w_1, \ldots, x_k : w_k$ be a pseudo context $(1 \leq k)$. If $x_1$ is an argument in $t$, $\Gamma, \Delta \vdash_{\pi} t : u$ and $\Gamma \vdash_{\pi} \text{subst}(t, \sigma) : u'$ and $\Delta$ is sufficient for $t$ in $\Gamma$ then $\Gamma \vdash_{\pi} a_1 : w_1$.

Proof We write $\Delta_1 \leq \Delta$ if $\Delta = \Delta_1, \Delta_2$, for some $\Delta_2$. We show by induction on the proof of $x_1$ is an argument in $t$: if $\Gamma, \Delta \vdash_{\pi} t : u$ and $\Gamma \vdash_{\pi} \text{subst}(t, \sigma) : u'$ and $\Delta' \leq \Delta$ is sufficient for $t$ in $\Gamma$ then $\Gamma \vdash_{\pi} a_1 : w_1$.

As $\Gamma, \Delta \vdash_{\pi} t : u$ we have $\Gamma, \Delta \vdash_{\pi} x_1 : w_1$ and $\Gamma \vdash_{\pi} w_1 : s$ by Lemma 6.2.2.

We have $t = ht_1 \ldots t_n (1 \leq n)$, and $h \in \Sigma \cup C$, as $x_1$ is an argument in $t$. By Lemma 6.2.6 for $\Gamma, \Delta \vdash_{\pi} t_i : u$ we either have:

1. $h = C \in \Sigma$ and $\tau(C) = d_1 \times \ldots \times d_n \rightarrow d$, and $\Gamma, \Delta \vdash_{\pi} t_i : d_i$, or
2. \( h \in C, \Gamma, \Delta \vdash_{\pi} h t_1 \ldots t_{i-1} : \Pi y : u_i, u_i', \) and \( \Gamma, \Delta \vdash_{\pi} t_i : u_i, \) all \( i \leq n. \)

**Base case:** \( t_j = x_1 \) for some \( 1 \leq j \leq n. \) As \( \Delta' \leq \Delta \) is sufficient for \( t_i \) in \( \Gamma \) we must have \( \epsilon \) is sufficient for \( h t_1 \ldots t_{j-1} \) and \( x_1 \) is sufficient for \( t_j \) in \( \Gamma. \) Thus \( \text{subst}(h t_1 \ldots t_j, \tilde{\sigma}) = h t_1 \ldots t_{j-1} a_1. \) By Lemma 6.2.6 for \( \Gamma \vdash_{\pi} \text{subst}(t, \tilde{\sigma}) : u' \) we have in case 1. \( \Gamma \vdash_{\pi} a_1 : d_j \) and in case 2. \( \Gamma \vdash_{\pi} a_1 : u_i'' \) and \( \Gamma \vdash u_j = \beta, \pi u_i'' \) by uniqueness of types for \( h t_1 \ldots t_{j-1}. \) In both cases by uniqueness of types (for \( x_1 \)) and by \( (\beta, \pi\text{-conversion}) \) we have \( \Gamma \vdash_{\pi} a_1 : w_1. \)

**Induction case:** We have \( x_1 \) is argument in \( t_j, \) for some \( j \leq n. \) As \( \Delta' \leq \Delta \) is sufficient for \( t \) in \( \Gamma \) we must have \( \epsilon \) is sufficient for \( h t_1 \ldots t_{j-1} \) and \( \Delta'' \leq \Delta' \) is sufficient for \( t_j \) in \( \Gamma. \) By the induction hypothesis we obtain \( \Gamma \vdash_{\pi} a_1 : w_1. \)

\[ \square \]

The types of the terms substituted for the variables in the left-hand side of a rule are fully determined by the types of these variables.

**Lemma 6.2.35** Let \( \tilde{\sigma} = (x_1, a_1), \ldots, (x_k, a_k) \) be a standard substitution sequence, and let \( \Delta = x_1 : w_1, \ldots, x_k : w_k \) be a pseudo context \( (1 \leq k). \) Assume \( x_1, \ldots, x_k \) are arguments in \( t \) and \( \Gamma, \Delta \vdash_{\pi} t : u \) and \( \Gamma \vdash_{\pi} \text{subst}(t, \tilde{\sigma}) : u' \) and \( \Delta \) is sufficient for \( t \) in \( \Gamma. \) Then \( \Gamma \vdash_{\pi} a_1 : w_1[x_1 := a_1] \ldots [x_{i-1} := a_{i-1}], \) all \( 1 \leq i \leq k. \)

**Proof** Induction on \( k. \)

**Case** \( k = 1. \) Follows by the previous lemma.

**Induction case:** \( k = j + 1. \) Let \( \Delta = x_1 : w_1, \ldots, w_{j+1} : w_{j+1} \). Assume \( x_1, \ldots, x_{j+1} \) are arguments in \( t \) and \( \Gamma, \Delta \vdash_{\pi} t : u \) and \( \Gamma \vdash_{\pi} t[x_1 := a_1] \ldots [x_{j+1} := a_{j+1}] : u' \) and \( \Delta \) is sufficient for \( t \) in \( \Gamma. \) By the previous lemma we have \( \Gamma \vdash_{\pi} a_1 : w_1. \) Let \( \Delta' = x_2 : w_2[x_1 := a_1], \ldots, w_{j+1} : w_{j+1}[x_1 := a_1]. \) By the substitution lemma 6.2.4 we obtain \( \Gamma, \Delta' \vdash_{\pi} t[x_1 := a_1] : u[x_1 := a_1]. \) As \( \Delta' \) is sufficient for \( t[x_1 := a_1] \) by Lemma 6.2.30 and \( x_2, \ldots, x_{j+1} \) are arguments in \( t[x_1 := a_1] \) by Lemma 6.2.33, we have by the induction hypothesis \( \Gamma \vdash_{\pi} a_1 : (w_1[x_1 := a_1])[x_2 := a_2] \ldots [x_{i-1} := a_{i-1}], \) all \( 2 \leq i \leq j + 1. \)

\[ \square \]

The legal instance of the left-hand side of a rule is not an inhabitant of \( \Delta. \)

**Lemma 6.2.36**

1. \( \Gamma \not\vdash_{\pi} \Delta : t. \)

2. If \( \Gamma \not\vdash_{\pi} (fa) : \Delta \)

**Proof**

1. By the generation lemma \( \Gamma \vdash_{\pi} \Delta : t \) implies \( (\Delta, s) \in \text{Axioms for some } s \in \text{Universes}. \)

2. Show by induction on the derivation that \( \Gamma \vdash_{\pi} (fa) : t \) implies \( t \neq \Delta. \)

\[ \square \]

The instance of the right-hand side of a rule has the same type as the legal instance of its left-hand side.
Lemma 6.2.37 Let $\bar{\sigma} = (x_1, a_1), \ldots, (x_k, a_k)$ be a standard substitution sequence and let $\Delta = x_1: w_1, \ldots, x_k: w_k$ be a pseudo context ($1 \leq k$). If $x_1, \ldots, x_k$ are arguments in $l$, and $\Gamma, \Delta \vdash \pi l : u$ and $\Gamma, \Delta \vdash \pi r : u$ and $\Gamma \vdash \pi \text{subst}(l, \bar{\sigma}) : u'$, and $\Delta$ is sufficient for $l$ in $\Gamma$ then $\Gamma \vdash \pi \text{subst}(r, \bar{\sigma}) : u'$.

Proof

By the previous lemma we obtain $\Gamma \vdash \pi a_i : w_i[x_1 := a_1] \ldots [x_{i-1} := a_{i-1}]$, all $i \leq k$. Using the substitution lemma 6.2.4 we obtain $\Gamma \vdash \pi \text{subst}(r, \bar{\sigma}) : \text{subst}(u, \bar{\sigma})$ and also $\Gamma \vdash \pi \text{subst}(l, \bar{\sigma}) : \text{subst}(u, \bar{\sigma})$. As $\Gamma \vdash \pi \text{subst}(l, \bar{\sigma}) : u'$, we have by uniqueness of types that $\Gamma \vdash u' =_{\beta, \pi} \text{subst}(u, \bar{\sigma})$. By Corollary 6.2.7 and Lemma 6.2.36 we obtain $\Gamma \vdash \pi u':s$, for some $s \in \text{Universes}$. Using the ($\beta, \pi$-conversion) rule we obtain $\Gamma \vdash \pi \text{subst}(r, \bar{\sigma}) : u'$.

In the next proposition we prove that the type of a legal term is preserved under reduction. This proof follows the structure of the proof for subject reduction for $\rightarrow_{\beta}$ for Pure Type Systems in [4].

Proposition 6.2.38 (subject reduction) If $\Gamma \vdash \pi u_1 : t$ and $\Gamma \vdash u_1 \rightarrow_{\beta, \pi} u_2$ then $\Gamma \vdash \pi u_2 : t$.

Proof

We write $\Gamma \rightarrow_{\beta, \pi} \Gamma'$, if $\Gamma = \Gamma_1, x: A, \Gamma_2$ and $\Gamma' = \Gamma_1, x: A', \Gamma_2$ and $\Gamma_1 \vdash A \rightarrow_{\beta, \pi} A'$.

Consider the following statements:

1. If $\Gamma \vdash \pi u_1 : t$ and $\Gamma \vdash u_1 \rightarrow_{\beta, \pi} u_2$ then $\Gamma \vdash \pi u_2 : t$.

2. If $\Gamma \vdash \pi u_1 : t$ and $\Gamma \rightarrow_{\beta, \pi} \Gamma'$ then $\Gamma' \vdash \pi u_1 : t$.

These will be proved simultaneously by induction on the derivation of $\Gamma \vdash \pi u_1 : t$. We will treat two cases.

First case. The last applied rule is the (application) rule. We have $\Gamma \vdash \pi (f a) : B[x := a]$, because $\Gamma \vdash \pi f : \Pi x : A. B$ and $\Gamma \vdash \pi a : A$.

1. Assume $\Gamma \vdash (f a) \rightarrow_{\beta, \pi} u_2$. We have four possibilities.

   $\Gamma \vdash f \rightarrow_{\beta, \pi} f'$ and $u_2 = f'a$. By the induction hypothesis we have $\Gamma \vdash \pi f' : \Pi x : A. B$.

   Then we obtain $\Gamma \vdash \pi (f'a) : B[x := a]$.

   $\Gamma \vdash a \rightarrow_{\beta, \pi} a'$ and $u_2 = fa'$. By the induction hypothesis we have $\Gamma \vdash \pi a' : A$.

   By the (application) rule we get $\Gamma \vdash \pi (fa') : B[x := a']$. Using the generation lemma 6.2.6 we obtain $\Gamma, x : a \vdash \pi B : s$. By the substitution lemma 6.2.4 we have $\Gamma \vdash \pi B[x := a] : s$. Using the ($\beta, \pi$-conversion) rule we obtain $\Gamma \vdash \pi (fa') : B[x := a]$.

F: $t' = r \in \text{fundefs}(\Gamma)$, and $(fa) \Rightarrow u_2$ is an enabled $r_1$-reduction step. As $\Gamma \vdash \pi (fa) : B[x := a]$, we have $\Gamma, \Delta \vdash \pi \text{lhs}(r_i) : u'$ and $\Gamma, \Delta \vdash \pi \text{rhs}(r_i) : u'$, and $\text{ctxtlhs}(\Delta, \text{lhs}(r_i), t')$ by Lemma 6.2.5. Let $\Delta = x_1 : A_1, \ldots, x_n : A_n$. As $FV(\Delta) = FV(\text{lhs}(r_i))$ by Lemma 6.2.29 and $FV(fa) \subseteq FV(\Gamma)$, we
Priority Rewriting in a Type System

\begin{align*}
\text{subst}(\text{lhs}(r_i), \sigma) \text{ and } u_2 = \text{subst}(\text{rhs}(r_i), \sigma), \text{ for some standard substitution sequence } \sigma = (x_1, t_1), \ldots, (x_n, t_n).
\end{align*}

As \( \Delta \) is sufficient according to Lemma 6.2.29 we have by Lemma 6.2.37
\( \Gamma \vdash \sigma \text{subst}(\text{rhs}(r_i), \sigma) : B[x := a] \). Thus \( \Gamma \vdash u_2 : B[x := a] \).

\( f = \lambda x : A'. b \) and \( u_2 = b[x := a] \). Using Lemma 6.2.6 for \( \Gamma \vdash A : A' \) we get \( \Gamma, x : A' \vdash b : B' \), and \( \Gamma \vdash \Pi x : A'. B' : s, \) and \( \Gamma \vdash \Pi x : A. B = \beta, \pi \Pi x : A'. B', \) for some \( B' \in \mathcal{T}_s, \) \( s \in \text{Universes} \). By Corollary 6.2.7 we get \( \Gamma \vdash A' : s', \) for some \( s' \in \text{Universes} \). Using the \((\beta, \pi\text{-conversion})\) rule we obtain \( \Gamma \vdash a : A' \), as \( \Gamma \vdash A = \beta, \pi A' \). Thus by the substitution lemma we obtain \( \Gamma \vdash b[x := a] : B'[x := a] \). By Corollary 6.2.7 and Lemma 6.2.36 we obtain \( \Gamma \vdash B[x := a] : s'' \), for some \( s'' \in \text{Universes} \), as \( \Gamma \vdash f a : B[x := a] \).

Again by the conversion rule we obtain \( \Gamma \vdash b[x := a] : B[x := a] \), as \( \Gamma \vdash B'[x := a] = \beta, \pi B[x := a] \). Thus \( \Gamma \vdash u_2 : B[x := a] \).

2. If \( \Gamma \rightarrow_{\beta, \pi} \Gamma' \) then we have by the induction hypothesis \( \Gamma' \vdash f : \Pi x : A. B \) and \( \Gamma' \vdash a : A \). Thus we have \( \Gamma \vdash (f a) : B[x := a] \).

Second case. The last applied rule is the (fun) rule. We have \( \Gamma, F : t \rightarrow_{\pi} F : t \), because \( \Gamma, x : t, \Gamma \vdash \pi (\text{lhs}(r_i)[F \leftarrow x]) : t_i \) and \( \Gamma, x : t, \Gamma \vdash \pi (\text{rhs}(r_i)[F \leftarrow x]) : t_i \), and \( \text{ctxth}(\Gamma_i, \text{lhs}(r_i), t), \) for all \( i \leq |\tau|, \) and exhaustive \( F : t \rightarrow \tau \), and argument decreasing \( (F t = \tau), \) and \( F \in C_\tau, F \notin \text{consts(}\Gamma) \), for some \( s \in \text{Universes} \).

1. We don’t have \( \Gamma, F : t \rightarrow_{\pi} F \rightarrow_{\beta, \pi} u \), as any \( F \)-redex has at least 1 argument and \( F \) is not a \( \rightarrow_{\beta, \pi} \)-redex.

2. Assume \( \Gamma, F : t \rightarrow_{\pi} \beta, \pi \Gamma' \). Then we must have \( \Gamma' = \Gamma'' \), \( F : t \rightarrow_{\tau} \), and \( \Gamma \rightarrow_{\beta, \pi} \Gamma'' \).

By the induction hypothesis we obtain \( \Gamma'' : x : t, \Gamma \vdash \pi (\text{lhs}(r_i)[F \leftarrow x]) : t_i \) and \( \Gamma'' : x : t, \Gamma \vdash \pi (\text{rhs}(r_i)[F \leftarrow x]) : t_i \). We have \( F \notin \text{consts(}\Gamma'') \), as \( F \notin \text{consts(}\Gamma) \).

Using the (fun) rule we obtain \( \Gamma' : t \rightarrow_{\pi} F : t \).

Because of subject reduction, it makes sense to consider the ARS consisting of the set of terms that have a type in \( \Gamma \), with the reduction relation \( \rightarrow_{\beta, \pi} \) determined by a context \( \Gamma \).

6.3 Encoding of Pattern Matching by Inductive Types

We want to analyse the relation between the system \text{\textit{\texttt{\Lambda HOL}}} with pattern matching and the system \text{\textit{\texttt{\Lambda HOL}}} with inductive types. It is clear that inductive types ([30]) are more expressive than algebraic data types, because inductive types may be parameterized and their constructors may have dependent types, whereas algebraic data types are not parameterized and their constructors only have non dependent types. For instance, it is possible to define an parameterized inductive type for lists of a certain length. We can represent any algebraic data type signature by inductive types (even in our simplified version of inductive types presented in Section 4.2). On the other hand, we can represent the induction principle of any algebraic type by a function definition, as structural induction is
argument decreasing. Thus if we would only allow inductive types that represent algebraic data types, we could represent all terms in this restricted version of λHOL₄ as terms of λHOL₅. In this section we will describe how we can encode terms of λHOL₅ as terms of λHOL₄, and we will analyse how the encoding affects the typing relation.

The encoding of an $S$-sorted algebraic data type signature $\Sigma$ is easy. Each sort with its constructors can be encoded by an inductive type. If we already have an encoding for all algebraic data types on which a sort $s \in S$ depends, then the sort $s$ with constructors $C_1, \ldots, C_n$ can be represented by an inductive type $\text{Ind}(x: \square\{T_1(x), \ldots, T_n(x)\})$, where $x \in V_\Delta$ and $T_i(x) = t_1 \to \ldots \to t_j \to x$ is the function type obtained from $\tau(C_i) = s_1 \times \ldots \times s_j \to s$ by replacing $s$ with $x$ and each other sort by its encoding. As an algebraic data type signature does not contain mutually dependent sorts, we can start the encoding with a non dependent sort.

**Definition 6.3.1** Let $s \in S$. Then $N_s$ denotes the number of algebraic data type constructors of sort $s$, $C_{s,i}$ denotes the $i$th algebraic data type constructor of sort $s$, and $A_{s,i}$ denotes the arity of $C_{s,i}$ for $1 \leq i \leq N_s$. Then $B_s$ denotes a term built from non-recursive algebraic data type constructors only, such that $\epsilon \vdash_{\pi} B_s : s$. The inductive type that is equivalent with $s$ is denoted by $s$. We denote the inductive constructor $\text{Constr}(i, s)$ by $C_{s,i}$.

**Example 6.3.2** We have $\text{List} = \text{Ind}(l: \square\{l, \text{Nat} \to l \to l\})$. We have $\text{Nil} = \text{Constr}(1, \text{List})$, and $\text{Cons} = \text{Constr}(2, \text{List})$.

We can define an encoding $[\_]_\epsilon$ for legal terms in a context without function definitions, which is indicated by the subscript $\epsilon$, as follows.

**Definition 6.3.3** We define a map $[\_]_\epsilon : \mathcal{T}_\pi \to \mathcal{T}_\iota$ as follows:

1. $[s]_\epsilon = s$, for $s \in S$.
2. $[C]_\epsilon = C$, for $C \in \Sigma$.
3. $[t_1 t_2]_\epsilon = [t_1]_\epsilon [t_2]_\epsilon$.
4. $[\lambda x.t_1.t_2]_\epsilon = \lambda x : [t_1]_\epsilon [t_2]_\epsilon$.
5. $[[\Pi x.t_1.t_2]]_\epsilon = \Pi x : [t_1]_\epsilon [t_2]_\epsilon$.
6. $[v]_\epsilon = v$, for $v \in V$.
7. $[F]_\epsilon = F$, for $F \in C$.

**Example 6.3.4** We have $[\lambda x: \text{Nat}.\text{Cons} \ x \ \text{Nil}]_\epsilon = \lambda x: \text{Nat}.\text{Cons} \ x \ \text{Nil}$.

**Remark 6.3.5** We have $\epsilon \vdash_{\pi} t : u \iff \epsilon \vdash_{\iota} [t]_\epsilon [u]_\epsilon$, for all $t, u \in \mathcal{T}_\pi$. 
Recall from Corollary 6.2.7 that we have two options if $\Gamma \vdash_{\pi} t : u$. Namely $u = \Delta$ or $\Gamma \vdash_{\pi} u : s$, for some $s \in \text{Universes}$. As Universes has three elements and each term has a unique type, we can split the set of legal terms into four disjoint subsets. We will analyse which terms are part of which subset, in order to determine the best way to define an encoding for function definitions.

The type universe $\Delta$ has only $\Box$ as inhabitant, and $\Box$ has no other types.

**Lemma 6.3.6** Let $t, u \in T_{\pi}$ be pseudo terms.

1. If $\Gamma \vdash_{\pi} t : \Delta$ then $t = \Box$.

2. If $\Gamma \vdash_{\pi} \Box : u$ then $u = \Delta$.

**Proof**

1. Induction on the derivation of $\Gamma \vdash_{\pi} t : \Delta$. Use part 1. of Lemma 6.2.36, for (var-start), ($\beta, \pi$-conversion), and part 2. for (application).

2. By Lemma 6.2.6 we obtain $\Gamma \vdash u =_{\beta, \pi} \Delta$. By Corollary 6.2.24 (confluence for $\rightarrow_{\beta, \pi}$) we have $\Gamma \vdash u \rightarrow_{\beta, \pi} \Delta$. By Proposition 6.2.38 (subject reduction), and Lemma 6.2.36 $u$ has no type. Thus by Corollary 6.2.7 we must have $u = \Delta$. $\square$

We will now specify the syntax of terms of type $\Box$.

**Definition 6.3.7** We define a set of kinds, notation $\mathcal{K}$, as the smallest set satisfying:

1. $* \in \mathcal{K}$.

2. $s \in \mathcal{K}$, for $s \in S$.

3. $v \in \mathcal{K}$, if $v \in V_{\Delta}$.

4. $k_{1} \rightarrow k_{2} \in \mathcal{K}$, if $k_{1}, k_{2} \in \mathcal{K}$.

A term of type $\Box$ is a kind.

**Lemma 6.3.8** Let $t, u \in T_{\pi}$ be pseudo terms.

1. If $\Gamma \vdash_{\pi} t : \Box$ then $t \in \mathcal{K}$.

2. If $k \in \mathcal{K}$ and $\Gamma \vdash_{\pi} k : u$ then $u = \Box$.

**Proof**

1. Induction on the derivation of $\Gamma \vdash_{\pi} t : \Box$ using Lemma 6.3.6. We treat the interesting cases.

   Case (application). Use part 2. of Corollary 6.2.7.
Case ($\beta, \pi$-conversion). We have: $\Gamma \vdash_\pi t : \Box$, because $\Gamma \vdash_\pi t : u$ and $\Gamma \vdash_\pi \Box : \Delta$ and $u =_{\beta, \pi} \Box$. By confluence we have $u \rightarrow_{\beta, \pi} \Box$. By part 1. of Corollary 6.2.7 we must have $\Gamma \vdash_\pi u : s$, for some $s \in \text{Universes}$. By Proposition 6.2.38 we have $\Gamma \vdash_\pi \Box : s$ and thus we obtain $s = \Delta$ by part 2. of Lemma 6.3.6. By part 1. of the same lemma we must have $u = \Box$. By the induction hypothesis we obtain $t \in \mathcal{K}$.

2. Structural induction on $k \in \mathcal{K}$.

Case $k = *$ or $k = \Box$. By Lemma 6.2.6 we obtain $\Gamma \vdash u =_{\beta, \pi} \Box$. By Corollary 6.2.24 (confluence for $\rightarrow_{\beta, \pi}$) we have $\Gamma \vdash u \rightarrow_{\beta, \pi} \Box$. By Corollary 6.2.7, Proposition 6.2.38 (subject reduction), and part 2. of Lemma 6.3.6 we have $\Gamma \vdash_\pi u : \Delta$. By part 1. of Lemma 6.3.6 we have $u = \Box$.

Case $k = v \in V_\Delta$. By Lemma 6.2.6 we obtain $\Gamma \vdash u =_{\beta, \pi} u'$ and $\Gamma \vdash_\pi u' : \Delta$ By part 1. of Lemma 6.3.6 we have $u' = \Box$. Just like in the previous case we obtain $u = \Box$.

Case $k = k_1 \rightarrow k_2$. By Lemma 6.2.6 we obtain $\Gamma \vdash_\pi k_1 : s_1$ and $\Gamma \vdash_\pi k_2 : s_2$ and $u \vdash s_3 =_{\beta, \pi}$ for some $(s_1, s_2, s_3) \in \text{Rules}$. By the induction hypothesis we obtain $s_1 = \Box$ and $s_2 = \Box$. By inspecting Rules we obtain $s_3 = \Box$. Just like before we obtain $t = \Box$.

We will now analyse which terms have a kind as type.

**Definition 6.3.9** We define a set of pseudo terms $\mathcal{T}_\mathcal{K}$ as follows:

1. $C \in \mathcal{T}_\mathcal{K}$, for $C \in \Sigma$.
2. $F \in \mathcal{T}_\mathcal{K}$, for $F \in C_0$.
3. $v \in \mathcal{T}_\mathcal{K}$, for $v \in V_0$.
4. $\lambda v.k.t \in \mathcal{T}_\mathcal{K}$, for $v \in V_0, k \in \mathcal{K}, t \in \mathcal{T}_\mathcal{K}$.
5. $t_1 t_2 \in \mathcal{T}_\mathcal{K}$, for $t_1, t_2 \in \mathcal{T}_\mathcal{K}$.
6. $\Pi v.k.t$, for $v \in V_0, k \in \mathcal{K}, t \in \mathcal{T}_\mathcal{K}$.
7. $t_1 \rightarrow t_2$, for $t_1, t_2 \in \mathcal{T}_\mathcal{K}$.

A term that has a kind as type is a member of $\mathcal{T}_\mathcal{K}$.

**Lemma 6.3.10** Let $t, u \in \mathcal{T}_\pi, k \in \mathcal{K}$ be pseudo terms.

1. If $\Gamma \vdash_\pi t : k$ then $t \in \mathcal{T}_\mathcal{K}$.
2. If $t \in \mathcal{T}_\mathcal{K}$ and $\Gamma \vdash_\pi t : u$ then $u \in \mathcal{K}$. 
Proof

1. Induction on the derivation of $\Gamma \vdash x \colon t \rightarrow k$ using Lemma 6.3.8. For (application) we have: $\Gamma \vdash x \colon (fa) \colon u[\bar{x} \leftarrow a]$, because $\Gamma \vdash f \colon P \rightarrow t \colon u$ and $\Gamma \vdash a \colon t \colon u$. By Corollary 6.2.7 we obtain $\Gamma \vdash x \colon t \colon s_1$, and $\Gamma, x \vdash u \colon s_2$ and $(s_1, s_2, s_3) \in \text{Rules}$, for some $s_1, s_2 \in \text{Universes}$. By Lemma 6.2.4 we obtain $\Gamma \vdash u \colon x \leftarrow a \colon s_2$. We must have $s_2 = \Box$, as $u[\bar{x} \leftarrow a] \in K$. Thus by inspecting Rules we have $s_1 = \Box$. Thus by Lemma 6.3.8 we have $t \vdash u \in K$. By (product) we obtain $\Gamma \vdash \Pi x \colon t \vdash u \in K$. By the induction hypothesis we obtain $f, a \in T_K$.

2. Structural induction on $t \in T_K$. $\square$

The right-hand sides of the rules of a legal function definition $F \vdash k \equiv \tau$ must be members of $T_K$ and therefore they do not contain defined constants in $C_r$. Thus for translating $F$ we do not need to be able to translate constants in $C_r$. For translating a function $G \vdash t \equiv \tau$ with $t \in T_K$ we might need to translate defined constants in $C_r$. Therefore we will give a detailed description of the encoding of function definitions $F \vdash k \equiv \tau$ with $F \in C_r$ first. Then we will establish properties of this encoding with respect to typing and reduction. Finally we will discuss how function definitions with defined constants in $C_r$, that represent proofs, could be encoded. We will not describe the syntax of the remaining terms, that have a member of $T_K$ as type.

Encoding of non-recursive functions

We will start with the encoding of non-recursive functions. Using the Elim constructor of inductive types we can inspect the head constructor of a term. We will define a transformation of function definitions, that models the effect of inspecting the head constructor of an argument.

Definition 6.3.11 Let $C \in \Sigma$ with $\tau(C) = s_1 \times \ldots \times s_m \rightarrow s$ ($0 \leq m$).

1. We define a function ‘RemoveCons’ on a sequence of rules as follows:

   (a) $\text{RemoveCons}_{C, i}(\epsilon) = \epsilon$,

   (b) $\text{RemoveCons}_{C, i}((Fp_1 \ldots p_{i-1} (Cq_1 \ldots q_m) p_{i+1} \ldots p_n \Rightarrow u, \tau)) = \\
\langle Fp_1 \ldots p_{i-1} q_1 \ldots q_m p_{i+1} \ldots p_n \Rightarrow u, \text{RemoveCons}_{C, i}(\tau) \rangle,$

   (c) $\text{RemoveCons}_{C, i}((Fp_1 \ldots p_{i-1} vp_{i+1} \ldots p_n \Rightarrow u, \tau)) = \\
\langle Fp_1 \ldots p_{i-1} z_1 \ldots z_m p_{i+1} \ldots p_n \Rightarrow u[v \leftarrow Cz_1 \ldots z_m], \text{RemoveCons}_{C, i}(\tau) \rangle,$

   for $v \in V$, where $z_1, \ldots, z_m \in V \setminus FV(Fp_1 \ldots p_n),$

   (d) $\text{RemoveCons}_{C, i}((l_1 \rightarrow u, \tau)) = \text{RemoveCons}_{C, i}(\tau)$, otherwise.

2. We define a function ‘RemoveCons’ on a type ($\in T_K$) as follows:

   (a) $\text{RemoveCons}_{C, i}((\Pi x : t \vdash u) = \\
\left\{ \begin{array}{ll}
\Pi z_1 : s_1 \ldots \Pi z_m : s_m. u[\bar{x} \leftarrow Cz_1 \ldots z_m] & \text{if } i = 1 \\
\Pi x : t. \text{RemoveCons}_{C, i-1}(u) & \text{otherwise}
\end{array} \right.,$

   where $z_1, \ldots, z_m \in V \setminus FV(\Pi x : t \vdash u),$
(b) RemoveCons(\textit{C}_i; t) = t, otherwise.

3. We define a function ‘RemoveCons’ on a function definition as follows:
   \text{RemoveCons}_{\textit{C}_i}(\textit{F}:\textit{T} = \textit{r}) = \textit{F}: \text{RemoveCons}_{\textit{C}_i}(\textit{T}) = \text{RemoveCons}_{\textit{C}_i}(\textit{r}).

**Example 6.3.12** Removing the successor from the rules for \texttt{Pred}:

\begin{align*}
\text{RemoveCons}_{\textit{S}_{\textit{cons}}} & (\textit{Pred}: \texttt{Nat} \rightarrow \texttt{Nat} \Rightarrow \text{Pred}(S \ x) \Rightarrow x, \text{Pred} O \Rightarrow O) = \\
\text{Pred}: \texttt{Nat} \rightarrow \texttt{Nat} & \Rightarrow \text{Pred} x \Rightarrow x
\end{align*}

Removing the constructor from the rules for \texttt{Head}:

\begin{align*}
\text{RemoveCons}_{\textit{S}_{\textit{cons}}}(\textit{Head}: \texttt{List} \rightarrow \texttt{Nat} \Rightarrow \text{Head}(\text{Cons} \ n \ l) \Rightarrow n, \text{Head} \ \text{Nil} \Rightarrow O) = \\
\text{Head}: \texttt{Nat} \rightarrow \texttt{Nat} & \Rightarrow \text{Head} \ n \ l \Rightarrow n
\end{align*}

**Remark 6.3.13** Assume \(F: t = r\) is an exhaustive function definition with \(n\) arguments and with a matching position \(i\) of sort \(s\). Assume \(|A_{s,l}| = k \ (l \leq N_s)\). We have

\[F_{t_1 \ldots t_{i-1}(c_{s,l}u_1 \ldots u_k)t_{i+1} \ldots t_n} \Rightarrow u\]

is an (enabled) reduction step for \(F: t = r\), if and only if \(F_{t_1 \ldots t_{i-1}(u_1 \ldots u_k)t_{i+1} \ldots t_n} \Rightarrow u\) is an (enabled) reduction step for \(\text{RemoveCons}_{\textit{C}_i}(F: t = r)\). Thus \(\text{RemoveCons}_{\textit{C}_i}(F: t = r)\) is also exhaustively defined.

Let \(t' = \text{RemoveCons}_{\textit{C}_i}(t)\), and \(r' = \text{RemoveCons}_{\textit{C}_i}(r)\). If \(\Gamma, F: t = r \vdash r F: t\) and \(F: t = r\) is non-recursive then

\[\begin{align*}
\Gamma, \text{RemoveCons}_{\textit{C}_i}(F: t = r) & \vdash r F: t' \\
\Gamma & \vdash r F: t' \text{ all } j \leq |r'|
\end{align*}

(otherwise)

Before we can specify the encoding for non-recursive functions we have to define same auxiliary functions.

**Definition 6.3.14**

1. \(\text{Abstract}_{i+1}((\Pi x: t_1. t_2, u) = \lambda x: t_1. \text{Abstract}_i(t_2, u))\).

2. \(\text{Abstract}_i(t_1, u) = u\), otherwise.

**Remark 6.3.15** Let \(\Delta = x_1: t_1, \ldots, x_n: t_n\). Let \(t = \Pi x_1: t_1, \ldots, x_n: t_n.t_{n+1}\). If \(\Gamma \vdash t: s\), for \(s \in \text{Universes}\), and \(\Delta, \Gamma \vdash u: t_{n+1}\) then \(\Gamma \vdash \text{Abstract}_n(t, u): t\).

**Definition 6.3.16** We define a function ‘\(\text{RemoveP}_{i}\)’ that removes the \(i^{th}\) \(\Pi\) in a type \((\in T_i)\).

1. \(\text{RemoveP}_{i}(\Pi x: t. u) = \begin{cases} 
  u & \text{ (if } i = 1) \\
  \Pi x: t. \text{RemoveP}_{i-1}(u) & \text{ (otherwise) }
\end{cases}\)

2. \(\text{RemoveP}_{i}(t) = t\), otherwise.

**Remark 6.3.17** Let \(t = t_1 \rightarrow \ldots \rightarrow t_n \rightarrow t_{n+1}\). If \(\Gamma \vdash t: s\) then \(\Gamma \vdash \text{RemoveP}_{j}(t): s\), for \(j \leq n, s \in \{*, \Box\}\).

We define an auxiliary function ‘\(\text{CallCons}_s\)’ on a sequence of sorts \((\in S^*)\) and two pseudo terms \((\in T_i)\), that we need in the definition of a function by structural induction on a term of type \(s\).
Definition 6.3.18 Let \( s \in S \), and \( t, f \in T_i \).

1. \( \text{CallCons}_s(\varepsilon, t, f) = f \).

2. \( \text{CallCons}_s((s_1, \bar{u}), t, f) = \begin{cases} \lambda x : s_1. \lambda \cdot t. \text{CallCons}_s(\bar{u}, t, f) & (s_1 = s) \\ \lambda x : s_1. \text{CallCons}_s(\bar{u}, t, f) & \text{(otherwise)} \end{cases}, \) where \( z \in V \setminus \text{FV}(f) \).

Remark 6.3.19 Let \( t = t_1 \rightarrow \ldots \rightarrow s \rightarrow t_{i+1} \), and let \( M_j = \text{CallCons}_s(A_{s,j}, \text{RemovePi}_i([t_i], f_j)) \) (\( j \leq N_s \)).

1. If \( \Gamma \vdash f_j : [\text{RemoveCons}_{c_{s,j}, i}(t)] \), then
   \( \Gamma, v : s \vdash \text{Elim}(s, \text{RemovePi}_i([t_i]), v)\{\bar{M}\} : \text{RemovePi}_i([t_i]), \) for \( v \in V \setminus \text{FV}(\Gamma) \).

2. \( \text{Elim}(s, t', C_{s,j} q_1 \ldots q_{|A_{s,j}|})\{\bar{M}\} \rightarrow_{\rho, f_j q_1 \ldots q_{|A_{s,j}|}} \).

For restoring the position of the arguments in front of a removed constructor, we need auxiliary functions \( \text{FirstPI}_{i,j} \) and \( \text{First}_{i,j} \).

Definition 6.3.20

1. We define a function \( '\text{FirstPI}_{i,j}' \) that moves the \( i \)-th until the \( i + j - 1 \)-th \( \Pi \) in a type to the front.

   (a) \( \text{FirstPI}_{i,j}(\Pi x_1 : t_1 \ldots \Pi x_n : t_n.t_{n+1}) = \Pi x_i : t_i \ldots \Pi x_{i+j-1} : t_{i+j-1}. \Pi x_1 : t_1 \ldots \Pi x_{i-1}. t_{i-1}. \Pi x_{i+j}. t_{i+j} \ldots \Pi x_n.t_{n+1} \), if \( i > 1, j > 0 \)

   (b) \( \text{FirstPI}_{i,j}(t) = t \), otherwise

2. We define a function \( '\text{First}_{i,j}' \) that moves the \( i \)-th until the \( i + j - 1 \)-th argument of a function to the front.

   (a) \( \text{First}_{i,j}(f, \Pi x_1 : t_1 \ldots \Pi x_n : t_n.t_{n+1}) = \lambda x_i : t_i \ldots \lambda x_{i+j-1} : t_{i+j-1}. \lambda x_1 : t_1 \ldots \lambda x_{i-1}. t_{i-1}. \lambda x_{i+j}. t_{i+j} \ldots \lambda x_n.t_{n+1}. f x_1 \ldots x_n \), if \( i > 1, j > 0 \)

   (b) \( \text{First}_{i,j}(f, t) = f \), otherwise

Remark 6.3.21 Let \( t = \Pi x_1.d_1. \ldots. \Pi x_n.t_{n+1} \).

1. If \( \Gamma \vdash f : t \) \( \) and \( x_1, \ldots, x_{i-1} \notin \text{FV}(t_k) \), for \( i \leq k < i + j \), then
   \( \Gamma \vdash \text{First}_{i,j}(f, t) : \text{First}_{i,j}(t) \).

2. \( \text{First}_{i,j}(f, t) a_1 \ldots a_{i+j-1} a_1 \ldots a_{i-1}. a_{i+j} \ldots a_n \rightarrow_{\rho} f a_1 \ldots a_n \).

Definition 6.3.22 We write \( \text{Match1}(F : k = \tilde{r}) = (i, s) \), if \( i \) is the first matching position of sort \( s \) of \( r_1 \). For a function definition whose first rule does not have a matching position we define \( \text{AbstractRule}(F : k_1 \rightarrow \ldots \rightarrow k_n \rightarrow k_{n+1} = (Fv_1 \ldots v_n \Rightarrow u, \tilde{r})) = \lambda u_1 : k_1 \ldots \lambda v_n : k_n. u \).
We define a function ‘Match’ that encodes a non-recursive, exhaustive function definition. If the first rule of this function definition does not have a matching position it can be encoded using AbstractRule. Otherwise we can inspect the head constructor of the argument on the first matching position and then use the encoding of the function definition in which this constructor is removed.

**Definition 6.3.23** Let \( \Gamma \) be a sequence of function definitions such that \([-\] \( \Gamma \) is defined. Let \( F:k = \bar{r} \) be a non-recursive, exhaustive function definition. \( \text{Match}_\Gamma(F:k = \bar{r}) = \)

\[
\begin{align*}
&\text{First}_{2,i-1}(\lambda x:s.\text{Elim}(s,k',x)\{M\},s \rightarrow k'), \\
&\text{Where } k' = \text{RemovePi}_i([k]_\Gamma), M_j = \text{CallCons}_s(A_{s,j},k'), \\
&\text{First}_{1,i}(\text{Match}_\Gamma(\text{RemoveCons}_{s,j,i}(F:k = \bar{r}))), \\
&\text{[RemoveCons}_{s,j,i}(k)]_\Gamma, \text{ for } 1 \leq j \leq N_s, \\
&\text{AbstractRule}(F:k = \bar{r})_\Gamma, \\
&\text{if } \text{Match}(F:k = \bar{r}) = (i,s) \\
&\text{otherwise}
\end{align*}
\]

**Example 6.3.24** We can encode the predecessor function as follows:

\[
\text{Match}_\lambda(\text{Pred:Nat} \rightarrow \text{Nat} = \text{Pred} (S \ x) \rightarrow x, \text{Pred} \ O \rightarrow O) = \\
\lambda z_1: \text{Nat}.\text{Elim}(\text{Nat}, \text{Nat}, \text{z}_1)\{O, \lambda z_2: \text{Nat}.\lambda z: \text{Nat}.(\lambda x: \text{Nat}.x) \ z_2\}
\]

This encoded function, below denoted as \( \text{Pred} \), has the following computational behaviour:

\[
\text{Pred}(S \ x) \rightarrow_{\beta,\alpha} x, \\
\text{Pred} \ O \rightarrow_{\beta,\alpha} O
\]

Using Match we can simulate any enabled reduction step of an application of a non-recursive function to canonical arguments.

**Lemma 6.3.25** Let \( F : k = \bar{r} \) be an exhaustive, left-linear function definition for \( k \in \mathcal{K} \). Assume \( F \) has \( n \) arguments and \( m \) matching positions \( i_1 \) of sort \( s_1, \ldots, i_m \) of sort \( s_m \). Assume \( k = k_1 \rightarrow \ldots \rightarrow k_n \rightarrow k_{n+1} \). Let \( p_1, \ldots, p_n \in \mathcal{P} \) be patterns such that \( \epsilon \vdash p_{i_j} : s_j \), for all \( j \leq m \). If \( Fp_1 \ldots p_n \Rightarrow u \) is an enabled reduction step for \( F:k = \bar{r} \), then \( \text{Match}_\lambda(F:k = \bar{r})[p_1]_c \ldots [p_n]_c \rightarrow_{\beta,\alpha} [u]_c \).

**Proof** Induction on the definition of \( \text{Match}_\lambda(F:k = \bar{r}) \).

**Base case:** \( r_1 = Fp_1 \ldots p_n \Rightarrow w \). We have \( (\lambda v_1:[k_1]_c) \ldots (\lambda v_n:[k_n]_c \cdot [w]_c)[p_1]_c \ldots [p_n]_c \rightarrow_{\beta} [w]_c[v_1 := [p_1]_c] \ldots [v_n := [p_n]_c] = [w[v_1 := p_1] \ldots [v_n := p_n]]_c \).

**Case** \( \text{Match}(F:k = \bar{r}) = (i,s) \). Let \( p_1, \ldots, p_n \) be patterns such that \( \epsilon \vdash p_{i_j} : s_j \), all \( j \leq m \).

Assume \( Fp_1 \ldots p_n \Rightarrow u \) is an enabled reduction step for \( F:k = \bar{r} \). We must have \( p_i = C_{s_i,j}q_1 \ldots q_{A_{s_i,i}} \) for some \( j \leq N_n \). Thus \( Fp_1 \ldots p_{i-1} q_1 \ldots q_{A_{s_i,j}}[p_{i+1}]_c \ldots p_n \) is an enabled reduction step for \( \text{RemoveCons}_{C_{s_i,j},i}(F:k = \bar{r}) \). By the induction hypothesis

\[
\text{Match}_\epsilon(\text{RemoveCons}_{C_{s_i,j},i}(F:k = \bar{r}))[p_1]_c \ldots [p_{i-1}]_c[q_1]_c \ldots [q_{A_{s_i,j}}]_c[p_{i+1}]_c \ldots [p_n]_c \\
\rightarrow_{\beta,\alpha} [u]_c.
\]

Considering the definitions of First, CallCons and Elim we obtain

\[
\text{Match}_\lambda(F:k = \bar{r})[p_1]_c \ldots [p_n]_c \rightarrow_{\beta,\alpha} [u]_c.
\]

We can extend the encoding of pseudo terms to pseudo contexts as follows.
Definition 6.3.26 Let \( \Gamma \) be a pseudo context such that \([-\_]_\Gamma : \mathcal{T}_\pi \to \mathcal{T}_i \) is defined. We define a function \([-\_]_\Gamma \) on pseudo contexts as follows:

1. \([\epsilon]_\Gamma = \epsilon\).
2. \([\Delta, x : t]_\Gamma = \begin{cases} [\Delta]_\Gamma, x : [t]_\Gamma & \text{(if } t \in \mathcal{K} \cup \{\square\}) \\ [\Delta]_\Gamma & \text{(otherwise)} \end{cases}\).
3. \([\Delta, F : t = \bar{r}]_\Gamma = [\Delta]_\Gamma\).

The typing relation for kinds and their inhabitants is preserved by the encoding.

Lemma 6.3.27 Let \( \Gamma \) be a sequence of variable declarations.

1. If \( \Gamma \vdash x : \square \) then \( [\Gamma]_\epsilon \vdash [k]_\epsilon : \square \).
2. If \( \Gamma \vdash x : a : k \) then \( [\Gamma]_\epsilon \vdash [a]_\epsilon : [k]_\epsilon \), for \( k \in \mathcal{K} \).

Proof

1. By induction on the derivation of \( \Gamma \vdash x : \square \).
2. By induction on the derivation of \( \Gamma \vdash x : a : k \).

\( \square \)

Lemma 6.3.28 Let \( \Gamma \) be a sequence of variable declarations. Let \( F : k = \bar{r} \) be a non-recursive function definition for \( k \in \mathcal{K} \). If \( \Gamma, F : k = \bar{r} \vdash F : k \) then \( [\Gamma]_\epsilon \vdash \text{Match}_\epsilon (F : k = \bar{r}) : [k]_\epsilon \).

Proof

Induction on the definition of \( \text{Match}_\epsilon (F : k = \bar{r}) \). Use Lemma 6.3.27.

\( \square \)

According to Lemma 6.3.25 the encoded version of a function definition \( F : k = \bar{r} \) can simulate any reduction step of an application of \( F \) to canonical values. But this is not true for an application of \( F \) to arbitrary (type correct) arguments. We will illustrate this in the next example.

Example 6.3.29 Consider the definition of a function \texttt{IsZeroNil} that tests whether its argument is a list containing only a \texttt{O}.

Let \( \Delta = \)

\texttt{IsZeroNil:List \to Bool =}

\texttt{IsZeroNil(Cons O Nil) \Rightarrow True,}

\texttt{IsZeroNil l \Rightarrow False}

It has encoding \( \text{Match}_\epsilon (\Delta) = \)

\( \lambda z_1 : \text{List.Elim(List, Bool, z_1)} \{ \text{False, } \lambda z_2 : \text{Nat.} \lambda z_3 : \text{List.} \lambda : \text{Bool.} \)

\( (\lambda z_4 : \text{Nat.Elim(Nat, List \to Bool, z_4)} \{ \lambda z_5 : \text{List.Elim(List, Bool, z_5)} \{ \text{True, } \lambda z_6 : \text{Nat.} \lambda z_7 : \text{List.} \lambda : \text{Bool.} (\lambda v_1 : \text{Nat.} \lambda v_2 : \text{List.False} : z_6 z_7),

\lambda z_8 : \text{Nat.} \lambda : \text{List \to Bool.} (\lambda v_1 : \text{Nat.} \lambda v_2 : \text{List.False} z_8) ) z_2 z_3). \)
We have $\Delta \vdash \text{isZeroNil}(\text{Cons } m \ (\text{Cons } n \ \text{Nil})) \rightarrow_{\pi} \text{False}$, but we do not have $\text{Match}_{\pi}(\Delta)(\text{Cons } m \ (\text{Cons } n \ \text{Nil})) \rightarrow_{\beta, \delta} \text{False}$, because the encoded function tries to inspect the variable $m$ before $\text{Cons } n \ \text{Nil}$.

It is not always possible to simulate every $\rightarrow_{\pi}$ reduction step on a legal term by a sequence of $\rightarrow_{\beta, \delta}$ reduction steps on an 'equivalent' term, because a function defined by pattern matching can inspect several arguments arbitrarily deep at the same time, whereas a function defined by induction can only inspect the head constructor of one argument. Thus if the left-hand side of a rule of a function definition has a subterm with more than one non-variable arguments then any encoding of this function using inductive types has to choose which argument is inspected encoding first. For such function definitions it is impossible to obtain a reduction preserving encoding.

**Sequential version of $\rightarrow_{\pi}$**

We will now give a precise description of a sequential version of $\rightarrow_{\pi}$ that can be simulated by the encoding.

**Definition 6.3.30**  
1. We define a relation $t$ is *incompatible* with $u$ at position $\overline{p}$, notation $t \#_{\pi} u$, as follows:

   (a) $C_1 t_1 \ldots t_m \#_{\pi} C_2 u_1 \ldots u_n$, if $C_1 \neq C_2$, $C_1, C_2 \in \Sigma$.
   
   (b) $C t_1 \ldots t_n \#_{i \pi} C u_1 \ldots u_n$, if $t_i \#_{\pi} u_i$, and subst$(t_j, \overline{\sigma}) = u_j$, all $j < i$, some $\overline{\sigma}, C \in \Sigma$.

2. Two pseudo terms $t, u$ have an *incompatible argument* at position $i \overline{p}$, notation $t \#_{i \pi} u$, if $t = F t_1 \ldots t_n$ and $u = F u_1 \ldots u_n$, for some $F \in C$, and $t_i \#_{\pi} u_i$ for some $i$. We write $F t_1 \ldots t_n \#_{i \pi} F u_1 \ldots u_n$, if $F t_1 \ldots t_n \#_{i \pi} F u_1 \ldots u_n$ and subst$(t_j, \overline{\sigma}) = u_j$, all $j < i$, some $\overline{\sigma}$.

**Remark 6.3.31** Notice that we have:

1. $t \#_{\pi} u \Rightarrow t \#_{s} u$.

2. $t \#_{s} u \Rightarrow t \#_{s} u$.

**Definition 6.3.32** Let $F : t = \overline{r}$ be a function definition. An $r_i$-reduction step $u_1 \Rightarrow u_2$ is *sequentially enabled* for $F : t = \overline{r}$, if $\text{lhs}(r_j) \#_{s} u_1$, or $\text{lhs}(r_j) \#_{s} u_1$ and $\text{lhs}(r_k) \#_{s} u_1$ for some $k < j$, $\overline{p}, \overline{q}$, for all $j < i$.

**Example 6.3.33** Consider the function of Example 6.3.29. We have

\[
\text{isZeroNil} \ (\text{Cons } (S \ n) \ \text{Nil}) \Rightarrow \text{False} \text{ is sequentially enabled for } \Delta, \text{ but }
\text{isZeroNil} \ (\text{Cons } m \ (\text{Cons } n \ \text{Nil})) \Rightarrow \text{False} \text{ is not sequentially enabled for } \Delta.
\]

**Definition 6.3.34** Let $\Gamma$ be a pseudo context. We define a relation $\rightarrow_{s}$ as follows:
\[ \Gamma \vdash C[t] \rightarrow_s C[u], \text{ if } t \Rightarrow u \text{ is a sequentially enabled } r_i\text{-reduction step} \]

for \( F:t'=t \in \text{ fundefs}(\Gamma), C[ ] \in \mathcal{H}_\pi \).

**Definition 6.3.35** We define a ternary relation \( \vdash_s \) by the typing rules for \( \vdash_\pi \) of Definition 6.1.17 and 6.1.42 (thus without the \((\beta, \pi\text{-conversion})\) rule) and with the following rule for \( \beta, s\text{-conversion} \):

\[
\frac{\Gamma \vdash_s a: \ast \quad \Gamma \vdash_s t:b \quad \Gamma \vdash a=_{\beta,s} b}{\Gamma \vdash_s t:a}
\]

**Remark 6.3.36** The relations defined above are restricted versions of the relations of \( \lambda\text{HOL}_\pi \).

1. If \( \Gamma \vdash t \rightarrow_s u \) then \( \Gamma \vdash t \rightarrow_\pi u \).

2. If \( \Gamma \vdash_s t:u \) then \( \Gamma \vdash_\pi t:u \).

Now we can define the sequential version of Higher Order Logic with Pattern Matching.

**Definition 6.3.37** The system \( \lambda\text{HOL}_s \) is the tuple \( (\tau_\pi, \chi_\pi, \{ \rightarrow_\beta, \rightarrow_s \}, \vdash_s) \).

Several fundamental properties that hold for \( \lambda\text{HOL}_\pi \) are also valid for \( \lambda\text{HOL}_s \).

**Proposition 6.3.38**

1. If \( \Gamma \vdash_s t:u \) then \( u = \Delta \) or \( \Gamma \vdash_\pi u:s \), for some \( s \in \text{Universes} \).

2. \( \rightarrow_s \) and \( \rightarrow_\beta \) are confluent.

3. If \( \Gamma \vdash_s t_1:u_1 \) and \( \Gamma \vdash_s t_2:u_2 \) then \( \Gamma \vdash u_1 =_{\beta,s} u_2 \).

4. If \( \Gamma \vdash_s t_1:u \) and \( \Gamma \vdash t_1 \rightarrow_s t_2 \) then \( \Gamma \vdash_t t_2:u \).

**Proof**

The proofs of Corollary 6.2.7, and confluence for \( \rightarrow_\beta, \pi \) and uniqueness of types and subject reduction for \( \lambda\text{HOL}_\pi \) can be transformed easily into proofs for the statements above. \( \square \)

**Encoding of recursive functions**

Until now we have seen how *non-recursive* functions can be encoded using inductive types. We will show how exhaustive, argument decreasing functions can be encoded. The complexity of the reduction behaviour of such functions is determined by the number of decreasing positions. Each recursive call of such a function has a structurally smaller argument on a decreasing position, but the succeeding arguments may be more complex.

**Definition 6.3.39**

1. We write \( t_1, \ldots, t_n(>_s)_{k\leq j} u_1, \ldots, u_n \) if \( t_i >_s u_i \) and \( t_j = u_j \) for all \( 1 \leq j < i \).
2. We say that \( i \) is a decreasing position for \( F: t \Rightarrow \bar{r} \) if for some \( j \) there is an occurrence \( F u_1 \ldots u_n \) of \( F \) in \( \text{rhs}(r_j) \) such that \( \text{lhs}(r_j) = F t_1 \ldots t_n \) and \( t_1, \ldots, t_n(>s) \Rightarrow u_1, \ldots, u_n \).

**Example 6.3.40** The function definition for \( \text{Leq} \) of Example 6.1.10 has decreasing position

1. The Ackermann function can be defined as:

\[
\begin{align*}
\text{Ack}: \text{Nat} & \to \text{Nat} \to \text{Nat} = \\
\text{Ack} O n & \Rightarrow S n, \\
\text{Ack} (S m) O & \Rightarrow \text{Ack} m (S O), \\
\text{Ack} (S m) (S n) & \Rightarrow \text{Ack} m (\text{Ack} (S m) n)
\end{align*}
\]

This function definition has decreasing positions 1 and 2.

We can simplify function definitions by replacing all recursive calls on the first argument position by an application of a fresh variable, that is an additional argument on the left-hand sides.

**Definition 6.3.41** Let \( F: t \Rightarrow \bar{r} \) be an argument decreasing function definition with a first argument decreasing position \( i \) of sort \( s \), notation \( \text{Decr}_1(F: t \Rightarrow \bar{r}) = (i, s) \). Let \( G \in \mathcal{C} \) be a function symbol, and let \( f \in \forall \mathcal{V} \forall \mathcal{F} V(F: t \Rightarrow \bar{r}) \).

1. We define \( \text{ReplDecr}_n^{G}(F: p_1 \ldots p_n \Rightarrow u) = G f p_1 \ldots p_n \Rightarrow (u'[F \leftarrow G f]) \), where \( u' \) is the pseudo term obtained from \( u \) by replacing each occurrence \( F u_1 \ldots u_n \), such that \( p_1, \ldots, p_n(>s) \Rightarrow u_1, \ldots, u_n \), by \( f u_1 \ldots u_n \).

2. \( \text{RmDecr}_1^{G}(F: t \Rightarrow \bar{r}) = G t \Rightarrow t = \bar{R} \), where \( R_j = \text{ReplDecr}_n^{G}(r_j) \), for \( j \leq |\bar{r}| \).

**Example 6.3.42** Let \( \Delta_{\text{Leq}} \) be the function definition for \( \leq \) on \( \text{Nat} \) of Example 6.1.10. Its non-recursive version \( \text{RmDecr}_1^{\text{Leq}_{\text{nr}}}(\Delta_{\text{Leq}}) = \)

\[
\begin{align*}
\text{Leq}_{\text{nr}}(\text{Nat} \Rightarrow \text{Nat} \Rightarrow \text{Bool}) & \Rightarrow (\text{Nat} \Rightarrow \text{Nat} \Rightarrow \text{Bool}) = \\
\text{Leq}_{\text{nr}} f O y & \Rightarrow \text{True} \\
\text{Leq}_{\text{nr}} f (S x) O & \Rightarrow \text{False} \\
\text{Leq}_{\text{nr}} f (S x) (S y) & \Rightarrow f x y
\end{align*}
\]

When we apply this function to \( \text{Leq} \) it has the same reduction behaviour as \( \text{Leq} \):

\[
\begin{align*}
\text{RmDecr}_1^{\text{Leq}_{\text{nr}}}(\Delta_{\text{Leq}}) & \vdash \text{Leq}_{\text{nr}} \text{Leq} O y \Rightarrow_s \text{True}, \\
\text{RmDecr}_1^{\text{Leq}_{\text{nr}}}(\Delta_{\text{Leq}}) & \vdash \text{Leq}_{\text{nr}} \text{Leq} (S x) O \Rightarrow_s \text{False}, \\
\text{RmDecr}_1^{\text{Leq}_{\text{nr}}}(\Delta_{\text{Leq}}) & \vdash \text{Leq}_{\text{nr}} \text{Leq} (S x) (S y) \Rightarrow_s \text{Leq} x y.
\end{align*}
\]

Note that we have for \( b \in \{\text{True}, \text{False}\} \) and any \( l \):

\[
\begin{align*}
\text{RmDecr}_1^{\text{Leq}_{\text{nr}}}(\Delta_{\text{Leq}}) & \vdash \text{Leq}_{\text{nr}}^{n+1} l (S^n O)(S^m O) \Rightarrow_s b, \\
& \text{if and only if } \Delta_{\text{Leq}} \vdash \text{Leq} (S^n O) (S^m O) \Rightarrow_s b.
\end{align*}
\]
Lemma 6.3.43 Let $F:k = \tau \mapsto F:k$ be a recursive, argument decreasing function definition with $k \in \mathcal{K}$. Let $G \in \mathcal{C}_\Gamma$ with $G \notin \text{consts}(\Gamma)$.

If $\Gamma, F:k = \tau \mapsto F:k$ then $\Gamma, \text{RmDecr1}^G(\Gamma)(F:k = \tau) \vdash_\Delta G:k \rightarrow k$.

Proof
Trivial. \hfill \Box

The example illustrates that we can compute the result of applying an argument decreasing, recursive function $F$ to canonical arguments by iterating $\text{RmDecr1}(F)$. We will define a function ‘Repeat’ that applies a function $f$ (typically the encoding of $\text{RmDecr1}(F)$) a given number of times to an argument $d$ of type $t$.

Definition 6.3.44 $\text{Repeat}(t, d, f) = \lambda n: \text{Nat}. \text{Repeat}'(t, d, f, n)$.

Where $\text{Repeat}'(t, d, f, n) = \text{Elim}(\text{Nat}, t, n)\{d, \lambda n: \text{Nat}. f\}$.

Remark 6.3.45 $\text{Repeat}'(t, d, f, S(n)) \rightarrow_{\beta, \Delta} f(\text{Repeat}'(t, d, f, n))$.

The type of $\text{Repeat}$ is determined by its first argument.

Lemma 6.3.46 If $\Gamma \vdash t : \Box, \Gamma \vdash d : t$ and $\Gamma \vdash f : t \rightarrow t$ then $\Gamma \vdash \text{Repeat}(t, d, f) : \text{Nat} \rightarrow t$.

Proof
Trivial. \hfill \Box

We just need to know how many times we must apply the encoding of $\text{RmDecr1}(F)$. Note that the number of iterations is limited by the number of symbols of the argument at the first decreasing position of $F$, as $F$ is argument decreasing. Thus we need a function for counting the number of function symbols of a (canonical) term of type $s$ for each $s \in S$.

Definition 6.3.47 Recall the definition of the inductive type $\text{Nat}$ in Example 6.1.23 and the encoding of the addition on this type by the function $\text{Plus}$ of Example 4.2.12. For each $s \in S$ we define a function ‘$\text{NrConstr}_s$’ that counts the number of constructor symbols of a canonical inhabitant of $s$, such that $\epsilon \vdash \text{NrConstr}_s : s \rightarrow \text{Nat}$. These functions $\text{NrConstr}_s$ have the following reduction behaviour:

\[
\begin{align*}
\text{NrConstr}_s(C_s, t) & \rightarrow_{\beta, \Delta} S(O), \text{ if } \tau(C_s, t) = s \\
\text{NrConstr}_s(C_s, a) & \rightarrow_{\beta, \Delta} S(\text{NrConstr}_s(a)), \text{ if } \tau(C_s, a) = s' \rightarrow s \\
\text{NrConstr}_s(C_s, \overline{a_1 \ldots a_n}) & \rightarrow_{\beta, \Delta} S(\text{Plus}(\text{NrConstr}_s(a_1), \ldots, \text{Plus}(\text{NrConstr}_s(a_{n-1}), \text{NrConstr}_s(a_n)))), \text{ if } \tau(C_s, \overline{a_1 \ldots a_n}) = s_1 \times \ldots \times s_n \rightarrow s, 2 \leq n.
\end{align*}
\]

Example 6.3.48 The function $\text{NrConstr}_s$ that counts the number of constructor symbols of a canonical list is defined by:

\[
\begin{align*}
\text{NrConstr}_s(l) = \\
\text{Elim}(\text{List}, \text{Nat}, l)\{S(O), \lambda a: \text{Nat}. \lambda m: \text{List}. \lambda n_m: \text{Nat}. S(\text{Plus}(\text{NrConstr}_s(a), n_m))\}
\end{align*}
\]
The function \( \text{NrConstrList} \) has the following reduction behaviour:

\[
\begin{align*}
\text{NrConstrList}(\text{Nil}) & \rightarrow_\beta, \alpha \text{ S(O)} \\
\text{NrConstrList}(\text{Cons a l}) & \rightarrow_\beta, \alpha \text{ S(Plus(NrConstrNat(a), NrConstrList(l)))}
\end{align*}
\]

We define a function \( \text{ArgCount}_{(i,a)} \) that for a given type \( T \) and function \( g : \text{Nat} \rightarrow T \) (typically the application of \( \text{Repeat} \) to an encoding of \( \text{RmDecr1}(F) \)) yields a function of type \( T \) that applies \( g \) to ‘the number of constructor symbols’ of the \( i \)th argument.

**Definition 6.3.49** Let \( T = \Pi x_1 : t_1, \ldots, \Pi x_i : t_i.t_{i+1} \). Let \( s \in S \). We define a function \( \text{ArgCount}_{(i,a)} \) as follows:

\( \text{ArgCount}_{(i,a)}(T, g) = \lambda x_1 : t_1, \ldots, \lambda x_i : t_i.g(\text{S(NrConstr}(x_i))) \) \( x_1 \ldots x_i \).

**Remark 6.3.50** Let \( T = \Pi x_1 : t_1, \ldots, \Pi x_i : t_i.t_{i+1} \). If \( \Gamma \vdash_\beta g : \text{Nat} \rightarrow T \), and \( t_i = s \in S \), then \( \Gamma \vdash_\beta \text{ArgCount}_{(i,a)}(T, g) : T \).

If we want to use \( \text{Repeat} \) to encode a recursive function of type \( k \in K \) we must provide a default value (of type \( k \)) for the result of applying \( \text{Repeat} \) to \( \text{O} \). For this purpose we define a function ‘\( \text{Fundefault} \)’.

**Definition 6.3.51** Let \( F : k = \tilde{r} \) be an argument decreasing, exhaustive function definition with \( n \) arguments and \( m \) matching positions \( i_1 \) of sort \( s_1 \), dots, \( i_m \) of sort \( s_m \). Let \( k = k_1 \rightarrow \ldots \rightarrow k_n \rightarrow k_{n+1} \in K \). Let \( v_1, \ldots, v_n \) be a sequence of distinct variables. Let \( p_1, \ldots, p_n \) be the sequence of patterns, such that \( p_i = B_{s_j} \) for \( j \leq m \) and \( p_i = v_i \) for non-matching positions \( l \leq n \). Let \( u \) be the unique term such that \( Fp_1 \ldots p_n \Rightarrow u \) is a sequentially enabled reduction step for \( F : k = \tilde{r} \). We define \( \text{Fundefault}(F : k = \tilde{r}) = \lambda v_1 : k_1, \ldots, \lambda v_n : k_n.u \).

**Example 6.3.52** Let \( \Delta_{\text{Leq}} \) be the function definition of Example 6.1.10. We have \( \text{Fundefault}(\Delta_{\text{Leq}}) = \lambda v_1 : \text{Nat}. \lambda v_2 : \text{Nat}. \text{True} \).

Fundefault has the type of its function definition.

**Lemma 6.3.53** If \( \Gamma, F : k = \tilde{r} \vdash_\beta F : k \) then \( \Gamma \vdash_\beta \text{Fundefault}(F : k = \tilde{r}) : k \).

**Proof**

As \( \Gamma, F : k = \tilde{r} \vdash_\beta F : k \) the function definition \( F : k = \tilde{r} \) is argument decreasing and exhaustive. Assume \( F : k = \tilde{r} \) has \( n \) arguments and \( m \) matching positions \( i_1 \) of sort \( s_1 \), dots, \( i_m \) of sort \( s_m \). Let \( k = k_1 \rightarrow \ldots \rightarrow k_n \rightarrow k_{n+1} \in K \). By the definition of Fundefault there are variables \( v_1, \ldots, v_n \), patterns \( p_1, \ldots, p_n \), and a term \( u \) such that \( p_i = B_{s_j} \) for \( j \leq m \) and \( p_i = v_i \) for non-matching positions \( l \leq n \) and \( Fp_1 \ldots p_n \Rightarrow u \) is a sequentially enabled reduction step for \( F : k = \tilde{r} \) and \( \text{Fundefault}(F : k = \tilde{r}) = \lambda v_1 : k_1, \ldots, \lambda v_n : k_n.u \). We may assume that \( v_1, \ldots, v_n \notin \text{FV}(\Gamma) \). We have \( e \vdash_\beta B_{s_j} : s_j \) and \( k_{i_j} = s_j \) for all \( j \leq m \). Thus we have \((\Gamma, F : k = \tilde{r}, v_1 : k_1, \ldots, v_n : k_n) \vdash_\beta Fp_1 \ldots p_n : k_{n+1} \). By subject reduction we have \((\Gamma, F : k = \tilde{r}, v_1 : k_1, \ldots, v_n : k_n) \vdash_\beta u : k_{n+1} \). Thus \( \Gamma, F : k = \tilde{r} \vdash_\beta \lambda v_1 : k_1, \ldots, \lambda v_n : k_n.u : k \). As \( F \) is argument decreasing, \( F \) does not occur in \( u \). Thus \( \Gamma \vdash_\beta \lambda v_1 : k_1, \ldots, \lambda v_n : k_n.u : k \).

Now we can specify the encoding of a function definition given an encoding for the function definitions on which it is based.
**Definition 6.3.54** Let \( F: k = \vec{r} \) be an argument decreasing, exhaustive function definition with \( k \in \mathcal{K} \). Let \( \Gamma \) be a sequence of function definitions such that \([-]_\Gamma \) is defined.

\[\text{Transform}_\Gamma(F: k = \vec{r}) = \begin{cases} \text{ArgCount}_{(i, s)}([k]_\Gamma, \text{Repeat}([k]_\Gamma, [\text{Fundef}(F: k = \vec{r})]_\Gamma), & \text{if } \text{Decr}_1(F: k = \vec{r}) = (i, s) \\ \text{Transform}_\Gamma(\text{RmDecr}_1(F: k = \vec{r})) & \text{(otherwise)} \end{cases} \]

We can extend the encoding for terms with defined constants in \( \Gamma \) to an encoding for terms with defined constants in \( \Gamma', F: k = \vec{r} \) using the encoding for \( F: k = \vec{r} \) in \( \Gamma \) as follows.

**Definition 6.3.55** Let \( \Gamma \) be a sequence of function definitions such that \([-]_\Gamma \) is defined. Let \( F: k = \vec{r} \) be an exhaustive, argument decreasing function definition such that \( F \notin \text{consts}(\Gamma), k \in \mathcal{K} \). We define a function \([-]_{\Gamma, F: k = \vec{r}} \) as follows:

1. \([G]_{\Gamma, F: k = \vec{r}} = \begin{cases} \text{Transform}_\Gamma(F : k = \vec{r}) & \text{(if } G = F \text{)} \\ [G]_\Gamma & \text{(otherwise)} \end{cases} \), for \( G \in \mathcal{C} \).
2. \([t_1 t_2]_{\Gamma, F: k = \vec{r}} = [t_1]_{\Gamma, F: k = \vec{r}} [t_2]_{\Gamma, F: k = \vec{r}} \).
3. \([\lambda x. t_1 t_2]_{\Gamma, F: k = \vec{r}} = \lambda x. [t_1]_{\Gamma, F: k = \vec{r}} [t_2]_{\Gamma, F: k = \vec{r}} \).
4. \([\Pi x. t_1 t_2]_{\Gamma, F: k = \vec{r}} = \Pi x. [t_1]_{\Gamma, F: k = \vec{r}} [t_2]_{\Gamma, F: k = \vec{r}} \).
5. \([u]_{\Gamma, F: k = \vec{r}} = [u]_\Gamma \), for \( u \in \mathcal{V} \cup \mathcal{S} \cup \mathcal{\Sigma} \).

The encoding of kinds does not change by extending the encoding with function definitions.

**Lemma 6.3.56** Assume \([-]_\Gamma \) is defined. \([k]_\Gamma = [k]_\vec{r} \) for all \( k \in \mathcal{K} \).

**Proof**
Structural induction on \( k \).

The encoding is preserved under substitution.

**Lemma 6.3.57** Let \( \Gamma \) be a sequence of function definitions such that \([-]_\Gamma \) is defined. We have \([t[x := u]]_\Gamma = [t]_\Gamma[x := [u]_\Gamma] \).

**Proof**
Structural induction on \( t \).

We want to prove that the encoding of a legal term (in \( \Gamma \)) is a legal term. For this purpose we define a function 'fundef\(_K\)' that extracts the function definitions of \( \Gamma \) that we need in the encoding.

**Definition 6.3.58** Let \( \Gamma \) be a pseudo context. We define fundef\(_K\)(\( \Gamma \)) as follows:

1. fundef\(_K\)(\( \epsilon \)) = \( \epsilon \).
2. \( \text{fundefs}_\mathcal{K}((F:t = \bar{r}, \Gamma)) = \begin{cases} \{ (F:t = \bar{r}, \text{fundefs}_\mathcal{K}(\Gamma)) \} & \text{if } t \in \mathcal{K} \\ \text{fundefs}_\mathcal{K}(\Gamma) & \text{otherwise} \end{cases} \).

3. \( \text{fundefs}_\mathcal{K}((u:t, \Gamma)) = \text{fundefs}_\mathcal{K}(\Gamma). \)

**Remark 6.3.59** If \( \Gamma \) is legal and \( \Delta = \text{fundefs}_\mathcal{K}(\Gamma) \) then \([-]_\Delta \) is defined.

The encoding of a term \( t \) does not change by adding function definitions whose defined constants do not occur in \( t \).

**Lemma 6.3.60** Assume \( \Gamma \vdash_s t : u \). Let \( \Gamma' = \text{fundefs}_\mathcal{K}(\Gamma) \). If \([-]\) is defined then \([t]_{\Gamma', \Delta} = [t]_{\Gamma'} \).

**Proof**

Structural induction on \( t \).

**Lemma 6.3.61** Let \( \Delta \) be sequence of function definitions such that \([-]_\Delta \) is defined. Assume we have \( \Gamma \vdash_s t : u \) and \( \text{fundefs}_\mathcal{K}(\Gamma) = \Delta \) implies \( [\Gamma]_\Delta \vdash_{s\Delta} [t]_\Delta = [u]_\Delta \), for all \( \Gamma \in \mathcal{X}, t \in \mathcal{T}, u \in \mathcal{K} \cup \{ \square \} \). If \( \Gamma, F: k = \bar{r}; F : k \) and \( \text{fundefs}_\mathcal{K}(\Gamma) = \Delta \) then

1. \( [\Gamma]_\Delta \vdash_1 \text{Match}_\Delta(F: k = \bar{r}): [k]_\Delta \), if \( F: k = \bar{r} \) is non-recursive.

2. \( [\Gamma]_\Delta \vdash_1 \text{Transform}_\Delta(F: k = \bar{r}): [k]_\Delta \).

**Proof**

1. Let \( n \) be the number of arguments of \( F : k = \bar{r} \) and \( k = k_1 \to \ldots \to k_n \to k_{n+1} \).

Induction on the definition of \( \text{Match}_\Delta(F: k = \bar{r}) \).

**Base case.** \( r_1 = F v_1 \ldots v_n \Rightarrow u \), for \( v_1, \ldots, v_n \in V \). We have \( \Gamma, f : k, v_1 : k_1, \ldots, v_n : k_n \vdash_s u[F \leftarrow f] : k_{n+1} \). Using abstraction we obtain \( \Gamma, f : k \vdash_s \lambda v_1 : k_1 \ldots \lambda v_n : k_n. u[F \leftarrow f] : k \). As \( f \notin \mathcal{FV}(u[F \leftarrow f]) \), we have \( \Gamma \vdash_s \lambda v_1 : k_1 \ldots \lambda v_n : k_n. u : [k]_\Delta \). Thus \( [\Gamma]_\Delta \vdash_1 [\lambda v_1 : k_1 \ldots \lambda v_n : k_n. u] : [k]_\Delta \).

**Case Match** \( 1(F: k = \bar{r}) = (i, s) \). Let \( j \leq N_s \). Let \( \vec{R}_j = \text{RemoveCons}_{\mathcal{C}_{\mathcal{I}}}(\bar{r}) \). Let \( k_j = \text{RemoveCons}_{\mathcal{C}_{\mathcal{I}}}(k) \). Let \( M_j = \text{CallCons}_{\mathcal{C}_{\mathcal{I}}}((A_{s,j}, \text{RemoveP}_{\mathcal{I}}([k]_\Delta), \text{First}_{i,A_{s,j}}(\text{Match}_\Delta(F' : k_j = \vec{R}_j), [k]_\Delta))). \)

**Case** \( |A_{s,j}| + n > 1 \). We have \( \Gamma, F : k_j = \vec{R}_j \vdash_1 F : k_j \). By the induction hypothesis we obtain \( [\Gamma]_\Delta \vdash_1 \text{Match}_\Delta(F' : k_j = \vec{R}_j) : [k]_\Delta \).

**Otherwise** \( |A_{s,j}| = 0 \) and \( n = 1 \). We have \( \text{Match}_\Delta(F : k_j = \vec{R}_j) = \text{rhs}((R_j)_1) \Delta \) and \( \Gamma \vdash_1 \text{rhs}((R_j)_1) : k_j \). Thus we obtain \([\Gamma]_\Delta \vdash_1 \text{Match}_\Delta(F' : k_j = \vec{R}_j) : [k]_\Delta \).

Combining all these functions we get

\[ [\Gamma]_\Delta, z : s \vdash \text{Elim}(s, \text{RemoveP}_{\mathcal{I}}([k]_\Delta), z, \vec{M}) : \text{RemoveP}_{\mathcal{I}}([k]_\Delta) \]

Thus we obtain \( [\Gamma]_\Delta \vdash_1 \text{Match}_\Delta(F : k = \bar{r}) : [k]_\Delta \).

2. Induction on the definition of \( \text{Transform}_\Delta(F : k = \bar{r}) \) using part 1. and Lemma 6.3.46 and 6.3.53. \( \square \)
The typing relation is preserved by the encoding.

**Lemma 6.3.62** Let $\Delta = \text{fundefs}_K(\Gamma)$. Assume $[-]_\Delta$ is defined.

1. If $\Gamma \vdash_s k : \Box$ then $[\Gamma]_\Delta \vdash_1 [k]_\Delta : \Box$.

2. If $\Gamma \vdash_s a : k$ then $[\Gamma]_\Delta \vdash_1 [a]_\Delta : [k]_\Delta$, for $k \in K$.

**Proof**

1. Induction on the derivation of $\Gamma \vdash_s k : \Box$.

2. Induction on the number of function definitions in $\Gamma$. Case $\Delta = \epsilon$ by Lemma 6.3.27. Case $|\Delta| = n + 1$ by induction on the derivation of $\Gamma \vdash_s a : k$. For (fun) rule use part 2. of Lemma 6.3.61. \hfill $\square$

## 6.4 Reduction Behaviour of the Encoding

In this section we will analyse the reduction behaviour of encoded legal terms. We will show that we can simulate reduction on legal terms in $T_K$ by their encodings. We will use this result to prove that the reduction relation $\rightarrow_{\beta,s}$ is terminating for legal terms in $T_K$.

The encoded version of a sequentially enabled reducible term $u$ reduces to its encoding in one step.

**Lemma 6.4.1** Let $F : k = \bar{r}$ be a non-recursive function definition with $n$ arguments such that $\Gamma, F : k = \bar{r}$ is a legal context and $k \in K$. Let $k = k_1 \rightarrow \ldots \rightarrow k_n \rightarrow k_{n+1}$, and let $\Gamma' = \text{fundefs}_K(\Gamma)$. Let $\Delta$ be a pseudo context and let $\Delta' = \text{fundefs}_K(\Delta)$, Assume $[-]_{\Gamma', \Delta}$ is defined. If $Fu_1 \ldots u_n \Rightarrow w$ is a sequentially enabled $\alpha$-reduction step for $F : k = \bar{r}$ and $\Gamma, \Delta \vdash_s u_j : k_j$ for all $j \leq n$ then $\text{Match}_\alpha(F : k = \bar{r})[u_1]_{\Gamma', \Delta} \ldots [u_n]_{\Gamma', \Delta} \rightarrow_{\beta,s} [w]_{\Gamma', \Delta}$.

**Proof** Induction on the definition of $\text{Match}_\alpha(F : k = \bar{r})$. We have $\text{sub}(\text{lhs}(r_i), \bar{s}) = Fu_1 \ldots u_n$ and $\text{sub}(\text{rhs}(r_i), \bar{s}) = w$.

**Base case** $r_1 = Fu_1 \ldots u_n \Rightarrow u$, for $u_1, \ldots, u_n \in V$. Use Lemma 6.3.57 and 6.3.60.

**Case** $\text{Match}_\alpha(F : k = \bar{r}) = (j, s)$. Considering the definition of sequentially enabled reduction step, we must have $u_j = Cq_1 \ldots q_m$ and $\tau(C) = s_1 \times \ldots \times s_m \Rightarrow s$, for some $C \in \Sigma$, as $\Gamma, \Delta \vdash_s u_j : s$. Thus we have $Fu_1 \ldots u_{j-1}q_1 \ldots q_m u_{j+1} \ldots u_n \Rightarrow w$ is a sequentially enabled reduction step for $\text{RemoveCons}_{\Sigma,j}(F : k = \bar{r})$.

**Case** $m + n > 1$. We have $\Gamma, \text{RemoveCons}_{\Sigma,j}(F : k = \bar{r})$ is a legal context and $\Gamma, \Delta \vdash_s q_j : s_j$, for all $j' \leq m$. By the induction hypothesis we obtain $\text{Match}_\alpha(\text{RemoveCons}_{\Sigma,j}(F : k = \bar{r})[u_1]_{\Gamma', \Delta} \ldots [u_{j-1}]_{\Gamma', \Delta} [q_1]_{\Gamma', \Delta} \ldots [q_m]_{\Gamma', \Delta} [u_{j+1}]_{\Gamma', \Delta} \ldots [u_n]_{\Gamma', \Delta} \rightarrow_{\beta,s} [w]_{\Gamma', \Delta}$. Using the properties of First, CallCons and Elim we obtain the result.
Otherwise \( m = 0 \) and \( n = 1 \). We have \( r_i = F \cdot C \Rightarrow w \) and \( \text{Match}_r(F : k = \vec{r}) = [w]r \)
and \( \Gamma \vdash_s w : k' \). By Lemma 6.3.60 we obtain \( \text{Match}_r(F : k = \vec{r}) \rightarrow_{\beta, s} [w]r, \Delta' \). \( \square \)

Now that we know the reduction behaviour of the translation of non-recursive functions, we will analyse recursive functions. By reducing the encoding of a sequentially enabled redex of a recursive function we must unfold the definition. Therefore the result is not always an encoding of the reduct (if it contains a recursive call). We will illustrate this in an example.

**Example 6.4.2** Let \( \Delta \) be the function definition for \( \text{Leq} \) of Example 6.1.10. Its non-recursive version can be found in Example 6.3.42 and is used in the encoding for \( \text{Leq} \) (see Definition 6.3.55). This encoding has the following reduction behaviour.

Let \( M = \text{Match}_\lambda(\text{RmDecr}^{\text{Leq}}(\Delta)) \), \( T = \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool} \), \( D = [\text{Fundefault}(\Delta)]_\epsilon \).

\[
[\text{Leq} (S m) (S n)]_\Delta \rightarrow_{\beta, \Delta} M(\text{Repeat}(T, D, M, \text{NrConstr}_\text{Nat}([S m]_\Delta))(S[m]_\Delta)(S[n]_\Delta)
\rightarrow_{\beta, \Delta} \text{Repeat}(T, D, M, S(\text{NrConstr}_\text{Nat}([m]_\Delta)))([m]_\Delta[n]_\Delta)
\rightarrow_{\beta, \Delta} M(\text{Repeat}(T, D, M, \text{NrConstr}_\text{Nat}([m]_\Delta)))([m]_\Delta[n]_\Delta)
\]

Note that the reduct is not equal to \([\text{Leq} m n]_\Delta\), but can be obtained from it by reduction. In general the situation is a bit more complicated, as a recursive call can reduce the number of constructor symbols of an argument on a decreasing position with more than one.

We introduce a relation \( \leq_S \) that represents \( \leq \) on \( \text{Nat} \).

**Definition 6.4.3** We define a relation \( \leq_S \) as follows:

1. \( n \leq_S n \).
2. If \( m \leq_S n \) then \( m \leq_S S n \).

**Remark 6.4.4** If \( m \leq_S n \) and \( m \rightarrow_{\beta, \Delta} m' \) then \( n \rightarrow_{\beta, \Delta} n' \) and \( m' \leq_S n' \), for some \( n' \).

Using the relation \( \leq_S \) we can specify the shape of a reduct of the encoding of a function definition.

**Definition 6.4.5** Let \( \Gamma \) be a sequence of function definitions. We define a relation \( \text{unfolds} \) on an argument decreasing, exhaustive function definition, a sequence of pseudo terms \((\in \mathcal{T}_I)\), and a pseudo term as follows:

1. \( \text{unfolds}_\Gamma(F : k = \vec{r}, \vec{a}, \text{Match}_\Gamma(F : k = \vec{r})) \), if \( F : k = \vec{r} \) is non-recursive and has \( |\vec{a}| \) arguments.
2. \( \text{unfolds}_\Gamma(F : k = \vec{r}, \vec{a}, f(\text{Repeat}_\Gamma([k]_\Gamma, [\text{Fundefault}(F : k = \vec{r}])_\Gamma, \text{Transform}_\Gamma(\text{RmDecr}^F(F : k = \vec{r}), l)))) \), if \( \text{Decr}(F : k = \vec{r}) = (i, s) \), and \( \text{unfolds}_\Gamma(\text{RmDecr}^F(F : k = \vec{r}), (g, \vec{a}), f) \), and \( \text{NrConstr}_\lambda(a_i) \rightarrow_{\beta, \Delta} l' \leq_S l \).
We say $u_1$ unfolds to $u_2$ for $w$ in $\Gamma$, if $w = F a_1 \ldots a_n$, $u_1 = g b_1 \ldots b_n$, $\Gamma = (\Gamma_1, F:k = \bar{r}, \Gamma_2)$, $F:k = \bar{r}$ has $n$ arguments, unfolds$_\Gamma(F:k = \bar{r}, (b_1, \ldots, b_n), f)$, and $u_2 = f b_1 \ldots b_n$.

The relation unfolds is stable under $\rightarrow_{\beta,s}$.

**Lemma 6.4.6** If unfolds$_\Gamma(F:k = \bar{r}, \bar{a}, f)$ and $a_i \rightarrow_{\beta,s} b$ for some $i$ then unfolds$_\Gamma(F:k = \bar{r}, (a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n), f')$ and $f \rightarrow_{\beta,s} f'$ for some $f'$.

**Proof** Induction on the number of argument decreasing positions of $F:k = \bar{r}$.

$F:k = \bar{r}$ is non-recursive. We have $f = \text{Match}_\Gamma(F:k = \bar{r})$. Take $f' = f$ and we obtain unfolds$_\Gamma(F:k = \bar{r}, (a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n), f')$ and $f \rightarrow_{\beta,s} f'$.

$\text{Decr}_I(F:k = \bar{r}) = (j, s)$. We have unfolds$_\Gamma(\text{RmDecr}_I^n(F:k = \bar{r}), (g, \bar{a}), f_1)$, and $f = f_1(\text{Repeat'}([k]_\Gamma, [\text{Fundefault}(F:k = \bar{r})]_\Gamma, \text{Transform}_\Gamma(\text{RmDecr}_I^n(F:k = \bar{r}), l)))$ and $\text{NtConstr}_a(a_i) \rightarrow_{\beta,s} l \leq S l$. By the induction hypothesis we obtain $f'_1$ such that unfolds$_\Gamma(\text{RmDecr}_I^n(F:k = \bar{r}), (g, a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n), f'_1)$ and $f_1 \rightarrow_{\beta,s} f'_1$.

Case $i \neq j$. We have unfolds$_\Gamma(F:k = \bar{r}, (a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n), f'_1(\text{Repeat'}([k]_\Gamma, [\text{Fundefault}(F:k = \bar{r})]_\Gamma, \text{Transform}_\Gamma(\text{RmDecr}_I^n(F:k = \bar{r}), l)))$.

Case $i = j$. We have by confluence of $\rightarrow_{\beta,s}$ that $\text{NtConstr}_a(b) \rightarrow_{\beta,s} l'_1$ and $l' \rightarrow_{\beta,s} l_1$ for some $l'_1$. Thus $l \rightarrow_{\beta,s} l_1$ and $l'_1 \leq S l_1$ for some $l_1$. Thus we have unfolds$_\Gamma(F:k = \bar{r}, (a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n), f'_1(\text{Repeat'}([k]_\Gamma, [\text{Fundefault}(F:k = \bar{r})]_\Gamma, \text{Transform}_\Gamma(\text{RmDecr}_I^n(F:k = \bar{r}), l_1))))$.

Using the relation unfolds we can define a relation $\sim$ between a term in $T_\pi$, and a term in $T_i$ indicating that the second term can be obtained from the encoding of the first term by unfolding encodings of function definitions.

**Definition 6.4.7** Let $\Gamma$ be a sequence of function definitions such that $[-]_\Gamma$ is defined. We define a relation $\sim$ on a pseudo term $t \in T_\pi$ and a pseudo term $t' \in T_i$ as follows:

1. $t \sim [t]_\Gamma$, for $t \in C \cup V \cup S \cup \Sigma$.

2. If $a \sim a'$ and $b \sim b'$ then
   
   (a) $\lambda x.a.b \sim \lambda x.a'.b'$.
   (b) $\Pi x.a.b \sim \Pi x.a'.b'$.
   (c) $ab \sim \begin{cases} c & (a'b' unfolds to c for ab in \Gamma) \\ a'b' & \text{(otherwise)} \end{cases}$

If we want to emphasize $\Gamma$ we write $\Gamma \vdash t \sim t'$. We say $t'$ is an unfolded encoding of $t$ in $\Gamma$, if $\Gamma \vdash t \sim t'$.

In the next lemmas we will show that any sequentially enabled reduction step of a closed legal term can be simulated by a reduction on an unfolded encoding of it.
Lemma 6.4.8 If \( t \sim t' \) and \( u \sim u' \) then \( t'u' \longrightarrow_{\alpha} w \) with \( tu \sim w \).

**Proof** The only interesting case is \( t = F_{a_1} \ldots a_{n-1} \) and \( \Gamma = \Gamma_1, F:k = \vec{r}, \Gamma_2 \) and \( F:k = \vec{r} \) has \( n \) arguments. By definition we must have \( t' = [F]_{\Gamma_1}, a'_1 \ldots a'_{n-1} \) with \( a_j \sim a'_j \), all \( j < n \). Let \( a'_n = u' \). We must show \( t'u' \longrightarrow_{\alpha} f a'_1 \ldots a'_n \) with unfold\( s_{\vec{r}}(F:k = \vec{r}, \langle a'_1 \ldots a'_n \rangle, f) \).

We will show by induction on the definition of Transform\( _{\vec{r}}(F:k = \vec{r}) \):

\( \text{Transform}_{\vec{r}}(F:k = \vec{r}) b_1 \ldots b_n \longrightarrow_{\beta, \alpha} f b_1 \ldots b_n \) with unfold\( s_{\vec{r}}(F:k = \vec{r}, \vec{b}, f) \), where \( n \) is the number of arguments of \( F:k = \vec{r} \).

**Case** \( F:k = \vec{r} \) *non-recursive*. We have Transform\( _{\vec{r}}(F:k = \vec{r}) b_1 \ldots b_n \longrightarrow_{\beta, \alpha} \) Match\( _{\vec{r}}(F:k = \vec{r}) b_1 \ldots b_n \) and unfold\( s_{\vec{r}}(F:k = \vec{r}, \vec{b}, \text{Match}_{\vec{r}}(F:k = \vec{r})) \).

**Case** Dec\( 1(F:k = \vec{r}) = (i, s) \). Let \( T = \text{Transform}_{\vec{r}}(\text{RmDecr}_1^F(F:k = \vec{r})) \) and \( D = \text{Fundefault}_1[F:k = \vec{r}] \). Now we have:

\( \text{Transform}_{\vec{r}}(F:k = \vec{r}) b_1 \ldots b_n \longrightarrow_{\beta, \alpha} \text{Repeat}'([k]_{\Gamma_1}, D, T, \text{S(NrConstr}_s(b_i))) b_1 \ldots b_n \)

By the induction hypothesis we have \( T([k]_{\Gamma_1}, D, T, \text{NrConstr}_s(b_i))) b_1 \ldots b_n \)

\( \longrightarrow_{\beta, \alpha} f(\text{Repeat}'([k]_{\Gamma_1}, D, T, \text{NrConstr}_s(b_i))) b_1 \ldots b_n \)

Thus unfold\( s_{\vec{r}}(F:k = \vec{r}, \langle b_1, \ldots, b_n \rangle, f(\text{Repeat}'([k]_{\Gamma_1}, D, T, \text{NrConstr}_s(b_i)))) \) holds.

The encoding of a pseudo term reduces to a pseudo term that is an unfolded encoding of it.

**Lemma 6.4.9** Assume \( [-]_\Gamma \) is defined. \([t]_\Gamma \longrightarrow_{\beta, \alpha} t' \) with \( \Gamma \vdash t \sim t' \).

**Proof** Structural induction on \( t \). The only interesting case is handled by Lemma 6.4.8. □

Each legal term \( \in T_K \) has a legal unfolded encoding.

**Corollary 6.4.10** Let \( \Gamma \) be a pseudo context, let \( t \in T_K \), and let \( \Delta = \text{fundef}_K(\Gamma) \).

If \( \Gamma \vdash t:k \) then \( [\Gamma]_\Delta \vdash_t t':[k]_\Delta \) and \( \Delta \vdash t \sim t' \) for some \( t' \in T \).

**Proof** By Lemma 6.3.62 we have \( [\Gamma]_\Delta \vdash_t [k]_\Delta \). By Lemma 6.4.9 we obtain \( t' \) such that \( [t]_\Delta \longrightarrow_{\beta, \alpha} t' \) and \( \Delta \vdash t \sim t' \). By subject reduction we obtain \( [\Gamma]_\Delta \vdash_t t':[k]_\Delta \). □

We can obtain an unfolded encoding of a substitution instance by reduction

**Lemma 6.4.11** If \( t \sim t' \) and \( u \sim u' \) then \( t'[x := u'] \longrightarrow_{\beta, \alpha} w \) with \( t[x := u] \sim w \).
Proof
Structural induction on \( t \). Use Lemma 6.4.8 for \( t = (t_1 \ t_2) \).

A sequentially enabled reduction step of a non-recursive function can be simulated by its unfolded encoding.

**Lemma 6.4.12** Let \( F:k=\bar{r} \) be a non-recursive function definition with \( n \) arguments such that \( \Gamma, F:k=\bar{r} \) is a legal context and \( k \in \mathcal{K} \). Let \( k = k_1 \to \ldots \to k_n \to k_{n+1} \), and let \( \Gamma' = \text{unfndefs}_k(\Gamma) \). Let \( \Delta \) be a pseudo context and let \( \Delta' = \text{unfndefs}_k(\Delta) \). Assume \([-\] \( \Gamma', \Delta' \) is defined. Assume \( F u_1 \ldots u_n \Rightarrow w \) is a sequentially enabled \( \rho \)-reduction step for \( F:k=\bar{r} \) and \( \Gamma, \Delta \vdash \bar{r}_j : k_j \) and \( \Delta', \Delta' \vdash u_j \sim u'_j \) for all \( j \leq n \).

Then \( \text{Match}_r(F:k=\bar{r})u'_1 \ldots u'_n \rightarrow^{\rho \beta \sigma} w' \) and \( \Gamma', \Delta' \vdash w \sim w' \).

**Proof**
Similar to the proof of Lemma 6.4.1.

The encoding of a recursive function counts the number of constructor symbols of the arguments on decreasing positions. This is only possible if these arguments have a canonical normal form (consisting of constructors only). Therefore we will demand that these arguments do not contain free variables.

**Lemma 6.4.13** Let \( s \in S \). If \( \Gamma \vdash_1 t : s \), \( \text{FV}(t) = \emptyset \), and \( t \) is in \( \rightarrow_{\beta, \sigma} \) normal form then \( t = \text{Constr}(i, s)t_1 \ldots t_n \), \( \tau(C_{s,i}) = s_1 \times \ldots \times s_n \rightarrow s \) and \( \Gamma \vdash_1 t_j : s_j \), for all \( j \leq n \), some \( i, n, t_1, \ldots t_n \).

**Proof**
By structural induction on \( t \in \mathcal{T}_i \) we prove if \( \text{FV}(t) = \emptyset \) and \( t \) is in \( \rightarrow_{\beta, \sigma} \) normal form and \( \Gamma \vdash_1 t : \prod x_1.t_1 \ldots \prod x_n.t_n.s \) then \( t = \lambda x_1.t_1 \ldots \lambda x_n.u_n.u \) (if \( 1 \leq n \)) or \( t = \text{Constr}(i, s)t_1 \ldots t_m \), \( \tau(C_{s,i}) = s_1 \times \ldots \times s_{n+m} \rightarrow s \) and \( \Gamma \vdash_1 t_j : s_j \), for all \( j \leq m \).

The number of constructor symbols of a canonical inhabitant of the encoding of a sort \( s \) can be computed by \( \text{NrConstr}_s \).

**Corollary 6.4.14** If \( \Gamma \vdash_1 t : s \) and \( \text{FV}(t) = \emptyset \) then \( \text{NrConstr}_s(t) \rightarrow_{\beta, \sigma} S^{n+1}O \), for some \( n \).

**Proof**
As \( \rightarrow_{\beta, \sigma} \) is strongly normalizing for legal terms, \( \text{NrConstr}_s(t) \) has some normal form \( u \), since \( \Gamma \vdash_1 \text{NrConstr}_s(t) : \text{Nat} \). By subject reduction and Lemma 6.4.13 we must have \( u = S^nO \) for some \( n \). As \( \text{NrConstr}_s(t) \) does not reduce to \( O \) we must have \( 1 \leq n \).

The closed instance of a pattern has more constructor symbols than the instance of any structurally smaller pattern. Thus a sequentially enabled reduction step reduces the number of constructor symbols on a argument decreasing position.

**Lemma 6.4.15** Let \( p, q \in \mathcal{P} \). Assume \( q <_s p \), \( \Delta \vdash \text{subst}(p, \bar{r}) \sim p' \), \( \Delta \vdash \text{subst}(q, \bar{r}) \sim q' \), \( \text{FV}(p') = \emptyset \), and \( \Gamma \vdash_1 q' : s_1, \Gamma \vdash_1 q' : s_2 \), and \( \text{NrConstr}_{s_1}(p') \rightarrow_{\beta, \sigma} s_1 \leq s_2 \).

Then \( j_2 \rightarrow_{\beta, \sigma} S j_3 \) and \( \text{NrConstr}_{s_2}(q') \rightarrow_{\beta, \sigma} s_4 \leq s j_3 \), for some \( j_3, j_4 \).
Proof
Induction on the derivation of \( q \lessdot p \). Use Corollary 6.4.14.

The next two lemmas are needed to show that an unfolded encoding of a sequentially enabled reduct of a recursive function \( F : k = \vec{r} \) can be reduced to an unfolded encoding of the reduct, if we know that this holds for \( \text{RmDecr}^{\vec{r}} \). The first lemma treats the recursive calls for the first argument decreasing position.

Lemma 6.4.16 Let \( \Delta \) be a sequence of function definitions such that \( [-]_\Delta \) is defined. Assume \( F : k = \vec{r} \in \Delta \) is a function definition with \( n \) arguments and \( \text{Decr} F : k = \vec{r} = (i, s) \) and \( k \in \mathcal{K} \). Assume \( p_1, \ldots, p_n, (\vec{a}_0 \vec{a}_1 \ldots \vec{a}_n, \vec{r}) \rightarrow f g a_0 \ldots a_n \), and \( \Delta \vdash \text{subst} (F q_1 \ldots q_n, \vec{a} \vec{b}_1 \ldots \vec{b}_n, \vec{r}) \rightarrow u \). Then \( g b_0 \ldots b_n \rightarrow_{\beta, u} \) with \( \Delta \vdash \text{subst} (F q_1 \ldots q_n, \vec{a} \vec{b}_1 \ldots \vec{b}_n) \rightarrow u \).

Proof
Let \( \Delta = \Gamma_1, F : k = \vec{r} \in \Gamma_2 \). Let \( k' = [k]_{\Gamma_1}, D = [\text{Fundefault} (F : k = \vec{r})]_{\Gamma_1} \), and \( M = \text{Transform}_{\Gamma_1} (\text{RmDecr}^{\vec{r}} (F : k = \vec{r})) \). By definition of \( \rightarrow \) we have \( g = \text{Repeat} (k', D, M, j) \), such that \( \text{NrConstr} (\vec{a}_j) \rightarrow_{\beta, j} j_2 \leq j \). By Lemma 6.4.15 we have \( j \rightarrow_{\beta, j} F S_{j_3} \) with \( \text{NrConstr} (b_j) \rightarrow_{\beta, j} j_4 \leq S_{j_3} \). Then

\[
\text{Repeat} (k', D, M, j) b_1 \ldots b_n \rightarrow_{\beta, j} \text{Repeat} (k', D, M, j_3) b_1 \ldots b_n \rightarrow_{\beta, j} M (\text{Repeat} (k', D, M, j_3)) b_1 \ldots b_n
\]

Similar to the proof of 6.4.8 we obtain \( h \) such that \( M (\text{Repeat} (k', D, M, j_3)) b_1 \ldots b_n \rightarrow_{\beta, j} h b_1 \ldots b_n \) with \( \text{unfolds}_{\Gamma_1} (F : k = \vec{r}, \vec{b}_1 \ldots \vec{b}_n, \vec{a} \vec{b}_1 \ldots \vec{b}_n) \). So \( [F]_{\Gamma_1} b_1 \ldots b_n \) unfolds to \( h b_1 \ldots b_n \) for \( \text{subst} (F q_1 \ldots q_n, \vec{a} \vec{b}_1 \ldots \vec{b}_n) \) in \( \Gamma_1 \). Thus \( \Delta \vdash \text{subst} (F q_1 \ldots q_n, \vec{a} \vec{b}_1 \ldots \vec{b}_n) \rightarrow h b_1 \ldots b_n \).

Lemma 6.4.17 Let \( F : k = \vec{r} \) be a function definition with \( n \) arguments and \( \text{Decr} F : k = \vec{r} = (i, s) \) such that \( \Gamma \), \( F : k = \vec{r} \) is a legal context and \( k \in \mathcal{K} \). Let \( k = k_1 \rightarrow \ldots \rightarrow k_n \rightarrow k_{n+1} \), and let \( \Gamma' = \text{fundef}_{\mathcal{K}} (\Gamma) \). Assume \( F u_1 \ldots u_n \rightarrow w_1 \) is a sequentially enabled reduction step for \( F : k = \vec{r} \) and \( F V (a_i) = \emptyset \) and \( \Gamma' \vdash u_j \rightarrow u'_j \) and \( \Gamma \vdash u_j : k_j \) and \( [\Gamma]_{\vec{r}, u_j} : [k_j]_{\vec{r}} \) for all \( j \leq n \) and \( \text{unfolds}_{\Gamma} (F : k = \vec{r}, \vec{u} \vec{M} f) \). Let \( v \in V \setminus F V (\Gamma) \). Assume \( F u_1 \ldots u_n \rightarrow w_2 \) is a sequentially enabled reduction step for \( \text{RmDecr}^{\vec{r}} (F : k = \vec{r}) \) and \( \Gamma', \text{RmDecr}^{\vec{r}} (F : k = \vec{r}) \vdash w_2 \rightarrow w'_2 \) and \( \Gamma, v : k]_{\vec{r}, \text{RmDecr}^{\vec{r}} (F : k = \vec{r})} : w'_2 : [k]_{\vec{r}} \).

Then \( w'_2 [v := f] \rightarrow_{\beta, w'_1} \) such that \( \Gamma', F : k = \vec{r} \vdash w_1 \rightarrow w'_1 \).

Proof
Let \( F : k \rightarrow k' = \vec{r} = \text{RmDecr}^{\vec{r}} (F : k = \vec{r}) \) with \( r'_j = \text{ReplDecr}^{\vec{r}} (i, v') (r_j) \) for all \( j \leq |\vec{r}| \), for some \( v' \in V \). Let \( \vec{a} \vec{b}_1 \ldots \vec{b}_n \) be a standard substitution sequence such that \( \text{subst} (\text{lhs} (r'_j), \vec{a} \vec{b}_1 \ldots \vec{b}_n, \vec{r}) = F u_1 \ldots u_n \) and \( \text{subst} (\vec{r} (r'_j), \vec{a} \vec{b}_1 \ldots \vec{b}_n) = w_2 \) and \( \text{subst} (x, \vec{a} \vec{b}_1 \ldots \vec{b}_n) = x \) for all \( x \in V \setminus F V (\text{lhs} (r'_j)) \). Let \( u = \text{rhs} (r'_j) \). Notice that we have \( \text{rhs} (r'_j) [F v' \leftarrow F] [v' := F] = \text{rhs} (r_j) \), and thus \( \text{subst} (w [F v' \leftarrow F] [v' := F], \vec{a} \vec{b}_1 \ldots \vec{b}_n) = w_1 \). By structural induction on \( w \) we prove:

1. \( \Gamma', (F : k \rightarrow k' = \vec{r}) \vdash \text{subst} (w, \vec{a} \vec{b}_1 \ldots \vec{b}_n) \rightarrow w' \), and
2. \( \Gamma, F : k \rightarrow k = \vec{r}, v : k, \Delta \vdash, \text{sub}(w, \delta) : k' \) and \([\Gamma, v : k, \Delta]_{[F,k\leftarrow k = \vec{r}]} \vdash w' : [k']_\epsilon\), for some \( k' \in \mathcal{K} \) and some sequence of variable declarations \( \Delta \), and

3. each occurrence of \( F \) in \( w \) is in a subterm \( F v' q_1 \ldots q_n \) such that \( p_1, \ldots, p_n(s) \) \( \leq \) \( q_1, \ldots, q_n \) with \( i < j \), and

4. each occurrence of \( v' \) in \( w \) is either in a subterm \( F v' q_1 \ldots q_n \) or in a subterm \( v' q_1 \ldots q_n \) such that \( p_1, \ldots, p_n(s) \) \( \leq \) \( q_1, \ldots, q_n \),

then either

1. \( w'[v := f] \rightarrow_{\beta_\delta} w'' \) such that \( \Gamma', F : k = \vec{r} \vdash \text{sub}(w[Fv' \leftarrow F][v' := F], \delta) \rightarrow w'' \), or

2. \( w' = H h'_1 \ldots h'_m \) with \( m < n \) and \((w = v'h_1 \ldots h_m \text{ or } w = Fv'h_1 \ldots h_m) \) and \( h'_j[v := f] \rightarrow_{\beta_\delta} h''_j \) such that \( \Gamma', F : k = \vec{r} \vdash \text{sub}(h_j[Fv' \leftarrow F][v' := F], \delta) \rightarrow h''_j \), all \( j \leq m \).

Case \( w \in V \): We have \( v \notin FV(w') \), as \( v \notin FV(\text{sub}(w, \delta)) \). For case \( w = v' q_1 \ldots q_n \) we need Lemma 6.4.16. \( \square \)

Any sequentially enabled reduction step of a closed legal term can be simulated by a reduction on any unfolded encoding.

**Lemma 6.4.18** Let \( \Delta' \) be a pseudo context, and let \( \Delta = \text{fundefs}_K(\Delta') \). Let \( F : k = \vec{r} \in \Delta \). Assume \( \Delta' \vdash_s F a_1 \ldots a_n : k' \), \( \Delta \vdash F a_1 \ldots a_n \rightarrow f \), \([\Delta']_{\Delta} \vdash f : [k']_\epsilon \) and \( F a_1 \ldots a_n \Rightarrow u \) is a sequentially enabled reduction step for \( F : k = \vec{r} \), and \( FV(a_j) = \emptyset \) for all argument decreasing positions \( j \) of \( F : k = \vec{r} \). Then \( f \rightarrow_{\beta_\delta}^* u' \) and \( \Delta \vdash u \rightarrow u' \) for some \( u' \in \mathcal{T}_i \).

**Proof** Let \( F' \in C \setminus (\text{consts}(\Delta') \cup \text{Universes}) \). Let \( F' : k = \vec{r} \) be the function definition obtained from \( F : k = \vec{r} \) by replacing each \( F \) with \( F' \). Notice that we have \( \Delta \vdash t[F' \leftarrow F] \rightarrow t' \) if \( \Delta, F' : k = \vec{r} \vdash t \rightarrow t' \) and \( \Delta', F' : k = \vec{r} t \vdash t' : k' \). We will prove by induction on the number of argument decreasing positions of \( F' : k = \vec{r} : k' \).

Let \( n \) be the number of arguments of \( F' : k = \vec{r} \). Let \( k = k_1 \rightarrow \ldots \rightarrow k_n \rightarrow k_{n+1} \).

Assume \( \Delta \vdash a_j \rightarrow a'_j \) and \( \Delta', \Gamma \vdash a_j : k_j \) and \([\Delta', \Gamma]_{\Delta} \vdash a'_j : [k_j]_\epsilon \) for all \( j \leq n \) and some sequence of variable declarations \( \Gamma \), and \( \text{unfolds}_A(F' : k = \vec{r}, a'_j, f'') \), and \([\Delta']_{\Delta} \vdash f' : [k]_\epsilon \).

If \( F a_1 \ldots a_n \Rightarrow u \) is a sequentially enabled reduction step for \( F' : k = \vec{r} \) and \( FV(a_j) = \emptyset \) for all argument decreasing positions \( j \) of \( F' : k = \vec{r} \) then \( f a'_1 \ldots a'_n \rightarrow_{\beta_\delta}^* u' \) such that \( \Delta, F' : k = \vec{r} \vdash u \rightarrow u' \).

**Case** \( F' \) is not recursive. We are done by Lemma 6.4.12.

**Case** \( \text{DecrI}(F' : k = \vec{r}) = (i, s) \). Assume \( F' a_1 \ldots a_n \Rightarrow u \) is a sequentially enabled \( r_j \)-reduction step for \( F' : k = \vec{r} \), and \( \Delta \vdash a_j \rightarrow a'_j \) and \([\Delta', \Gamma]_{\Delta} \vdash a'_j : [k_j]_\epsilon \), all \( j \leq n \), and \( \text{unfolds}_A(F' : k = \vec{r}, a'_j, f'') \), and \([\Delta']_{\Delta} \vdash f' : [k]_\epsilon \). Let \( v \in V \setminus FV(\Delta', \Gamma) \). We have \( F' a_1 \ldots a_n \Rightarrow w \) is a sequentially enabled reduction step for \( \text{ReplDecrI}_{i,s}(F') : k = \vec{r} \).\]
We have $f' = Mf''$ with unfolds$_\Delta$(RmDecr$^F_r$(F': $k = r'$), (v, a'), M) and 
$[\Delta']_{\Delta} |_{a_1} M : [k \rightarrow k]_{\Delta}$ and $[\Delta']_{\Delta} |_{a_1} f'' : [k]_{\Delta}$, and $[\Delta', \Gamma, v : k]_{\Delta} |_{a_1} v : [k]_{\Delta}$. Thus we have
$\Delta, \text{RmDecr}^F_r(F': k = r') \vdash F' \nu a_1 \ldots a_n \sim M \nu a_1 \ldots a_n$. By the induction hypothesis
$M \nu a'_1 \ldots a'_n \rightarrow_{\beta, u} w'$ with $\Delta, \text{RmDecr}^F_r(F': k = r') \vdash w \rightarrow w'$. Thus we have
$Mf'' a'_1 \ldots a'_n \rightarrow_{\beta, u} w'[v := f'']$. By Lemma 6.4.17 we obtain $w'[v := f''] \rightarrow_{\beta, u} u'$ with $\Delta, F': k = r' \vdash u \rightarrow u'$. $\square$

Now that we have shown that a sequentially enabled reduction step can be simulated by a reduction on their unfolded encodings, we want to obtain a similar result for $\rightarrow_s$. If we want to use the previous result we must guarantee that the contracted redex occurrence has closed terms on all argument decreasing positions. Terms on argument decreasing positions have a type $s \in S$. We will define a subset $K^S$ of $K$ that contains $S$ and all function types with a sort $s \in S$ as range. All terms of sort $s$ contain at least one subterm that has a type in $K^S$.

**Definition 6.4.19** We define a set of pseudo terms $K^S$ as the smallest set satisfying:

1. $s \in K^S$, for $s \in S$.
2. $k \rightarrow f \in K^S$, if $k \in K$, $f \in K^S$.

A pseudo context $\Gamma$ is $K^S$-free if for all $x : t \in \Gamma$ we have $t \notin K^S$. The set $K^S_i \subset T_i$ is the counter part for $K^S$. Thus $f \in K^S \iff [f]_e \in K^S_i$.

A term on an argument decreasing position that is typable in a $K^S$-free context contains no free variables.

**Remark 6.4.20** If $\Gamma \vdash t : f$ and $\Gamma$ is $K^S_i$-free then FV(t) = $\emptyset$, for $f \in K^S$. Assume $[-]_\Delta$ is defined. We have $\Gamma$ is $K^S$-free $\iff [\Gamma]_\Delta$ is $K^S_i$-free.

If a term is typable in a $K^S$-free pseudo context $\Gamma$, then it may contain subterms that are only typable in a not $K^S$-free extension of $\Gamma$. To indicate this, we introduce the notion of 'bound' redex occurrence.

**Definition 6.4.21** A redex occurrence $r$ in a pseudo term $t$ is **bound** if there are $k \in K_S$, $x \in V$, $u \in T_r$, such that:

1. $r$ is a subterm of a term $u$, and
2. $\lambda x : k . u$ or $\Pi x : k . u$ is a subterm of $t$.

We introduce 'bound' and 'free' reduction in order to indicate whether the contracted redex occurrence is bound or not.

**Definition 6.4.22** Assume $\Gamma \vdash t \rightarrow_s t'$. We write $\Gamma \vdash t \rightarrow^b_s t'$ if the contracted redex occurrence is bound in $t$, and $\Gamma \vdash t \rightarrow^f_s t'$ otherwise (free redex occurrence). We will use a similar notation for $\rightarrow_\beta$ and $\rightarrow_s$. 


Remark 6.4.23 Let $\Gamma$ be a $\mathcal{K}^S$-free pseudo context such that $\Gamma \vdash_s t : u$. If $r$ is a free redex occurrence in $t$ then $\Gamma, \Delta \vdash_s r : u'$ for some term $u'$, and some $\mathcal{K}^S$-free pseudo context $\Delta$.

A free reduction step starting with a legal term in a $\mathcal{K}^S$-free context can be simulated by a reduction starting with any legal unfolded encoding of this term.

Proposition 6.4.24 Let $\Delta'$ be a $\mathcal{K}^S$-free pseudo context, and let $\Delta = \text{fundefs}_K(\Delta')$. Assume $\Delta' \vdash_s t_1 : k$, and $\Delta \vdash t_1 \leadsto t'_1$, and $[\Delta']_{\Delta} \vdash t'_1 : [k]_{\Delta}$, for $k \in \mathcal{K}$.
If $\Delta' \vdash t_1 \leadsto^f_{\beta,s} t_2$ then $t'_1 \leadsto^+_{\beta,s} t'_2$ and $\Delta \vdash t_2 \leadsto t'_2$.

Proof
Induction on the derivation of $\Delta' \vdash_s t_1 : k$.

Case (abstraction)-rule was last applied. We have $t_1 = \lambda x : k_1, b, b_0$ and $\Delta' \vdash_s k_1 \rightarrow k_2 : \Box$, for $k_1 \in \mathcal{K}$. We have $t'_1 = \lambda x : [k_1]_{\Delta}, b' \rightarrow b'_2$ with $[\Delta']_{\Delta} \vdash_s b' : [k_2]_{\Delta}$, $b \leadsto b'$, and $[\Delta']_{\Delta} \vdash k_1 \leadsto k_2 : \Box$. We must have $k_1 \notin \mathcal{K}^S$, as $\Delta' \vdash t_1 \leadsto^f_{\beta,s} t_2$. Thus $\Delta', x : k_1$ is $\mathcal{K}^S$-free. By the induction hypothesis we obtain $t'_1 \leadsto^+_{\beta,s} t'_2$ and $t_2 \leadsto t'_2$.

Case (application)-rule was last applied. We have $t_1 = b_1 b_2$, and $\Delta' \vdash_s b_1 : k_1 \rightarrow k_2$ and $\Delta' \vdash_s b_2 : k_1$. We have four subcases for $\Delta' \vdash t_1 \leadsto^f_{\beta,s} t_2$:

1. $b_1 = \lambda x : k_1, b_3$ and $t_2 = b_3[x := b_2]$. Then $t'_1 = (\lambda x : [k_1]_{\Delta}, b'_3) b'_2$ with $b'_3 \leadsto b'_2$ and $b_2 \leadsto b'_2$. By definition $t'_1 \leadsto^+_{\beta,s} b'_2[x := b'_2]$. Using Lemma 6.4.11 we obtain $b'_2[x := b'_2] \rightarrow^+_{\beta,s} w$ with $b_3[x := b_2] \leadsto w$.

2. $t_1 \leadsto t_2$ is a sequentially enabled $r_i$-reduction step for $F : k' = \vec{r}$ in $\Delta$. We have $t_1 = Fa_1 \ldots a_n$. As $\Delta'$ is $\mathcal{K}^S$-free we have $\text{FV}(a_j) = \emptyset$ for all argument decreasing positions $j$ of $F : k' = \vec{r}$. Thus by Lemma 6.4.18 we obtain $t'_1 \leadsto^+_{\beta,s} w$ with $t_2 \leadsto w$.

3. $\Delta' \vdash b_2 \rightarrow^f_{\beta,s} b_2'$ and $t_2 = b_1 b_2'$. As $t_1 \leadsto t'_1$ we have $t'_1 = c_1 c_2$.

Either have $c_1 c_2$ unfolds to $c_1 c_2$ for $b_1 b_2$ in $\Delta$ and $b_1 \leadsto c_1'$ and $b_2 \leadsto c_2$, for some $c_1'$. By the induction hypothesis we obtain $c_2 \rightarrow^+_{\beta,s} w$ with $b'_2 \leadsto w$.

By Lemma 6.4.6 we obtain a pseudo term $c_1'$ such that $c_1 \rightarrow^+_{\beta,s} c_1'$ and $c_1 w$ unfolds to $c_1 w$ for $b_1 b'_2$ in $\Delta$. Thus $t'_1 \rightarrow^+_{\beta,s} c_1' w$ and $t_2 \leadsto c_1' w$.

Or we have $b_1 \leadsto c_1$ and $b_2 \leadsto c_2$. By the induction hypothesis we obtain $c_2 \rightarrow^+_{\beta,s} w$ with $b_2' \leadsto w$. Thus $t'_1 \rightarrow^+_{\beta,s} c_1 w$ and $t_2 \leadsto c_1 w$.

4. $\Delta' \vdash b_1 \rightarrow^f_{\beta,s} b_1'$ and $t_2 = b_1' b_2$. Similar as previous case.

Corollary 6.4.25 $\rightarrow^f_{\beta,s}$ is strongly normalizing for legal $T_K$-terms in a $\mathcal{K}^S$-free pseudo context.
Proof

Let $\Delta'$ be a $\mathcal{K}^S$-free pseudo context, and let $\Delta = \text{undefs}_K(\Delta')$. Assume $\Delta' \vdash_t t_1 : k$, for $t_1 \in T_{\mathcal{K}}, k \in \mathcal{K}$. Assume there exists an infinite reduction sequence $t_1 \rightarrow_{\beta,s} t_2 \rightarrow_{\beta,s} t_3 \rightarrow_{\beta,s} \ldots$.

By Corollary 6.4.10 we have $[\Delta]'_\Delta \vdash_t t_1' : [k]_\Delta$, and $\Delta \vdash t_1 \sim t_1'$, for some $t_1' \in T_I$. By Proposition 6.4.24 we obtain $t_{j+1}'$ with $t_j' \rightarrow_{\beta,s} t_{j+1}'$ and $\Delta \vdash t_{j+1} \sim t_{j+1}'$ (and $[\Delta]'_\Delta \vdash t_{j+1}' : [k]_\Delta$ by subject reduction), for any $1 \leq j$ (see Figure 6.3). This would contradict the fact that $\rightarrow_{\beta,s}$ is strongly normalizing for legal terms (see Theorem 4.2.27).

\[ \square \]

Proof of strong normalization for $\rightarrow_{\beta,s}$

We will now show that $\rightarrow_{\beta,s}$ is strongly normalizing for legal $T_{\mathcal{K}}$-terms by providing a simulation of $\rightarrow_{\beta,s}$ by $\rightarrow_{\beta,s}^f$. For this purpose we add for each bound redex occurrence an equivalent free redex occurrence. We will define a relation $t \preceq_f u$ between terms indicating that $u$ is obtained from $t$ by adding a free redex occurrence for each bound redex occurrence.

We define a function ‘dummy$_S$’ that gives as a term of the type of the argument $\in \mathcal{K}^S$). Recall that $B_s$ denotes a canonical value of type $s$, for $s \in S$.

Definition 6.4.26 We define a function dummy$_S : \mathcal{K}^S \rightarrow \mathcal{T}_\pi$ as follows:

1. dummy$_S(s) = B_s$ for $s \in S$.

2. dummy$_S(k_1 \rightarrow k_2) = \lambda \cdot k_1.\text{dummy}_S(k_2) \in \mathcal{K}$, for $k_1 \in \mathcal{K}, k_2 \in \mathcal{K}^S$.

Remark 6.4.27 If $\Gamma \vdash_s k : \Box$ then $\Gamma \vdash_s \text{dummy}_S(k) : k$ for $k \in \mathcal{K}^S$.

We define a binary relation $\preceq_f$ on pseudo terms, that indicates that the second pseudo term is obtained from the first one by adding free redex occurrences.

Definition 6.4.28 We define a relation $\preceq_f$ on pseudo terms ($\in \mathcal{T}_\pi$) as follows:

1. $t \preceq_f t$, for $t \in C, V, S, \Sigma$.

2. if $A \preceq_f A'$ and $B \preceq_f B'$ then
(a) \( AB \preceq_f A'B' \).

(b) \( \lambda x:A.B \preceq_f \begin{cases} \lambda x:C.\lambda x:A.B' & (\text{if } A \in K^S) \\ \lambda x:A'.B' & (\text{otherwise}) \end{cases} \).

(c) \( \Pi x:A.B \preceq_f \begin{cases} \Pi x:C.\Pi x:A.B' & (\text{if } A \in K^S, B \in T_K) \\ \Pi x:A'.B' & (\text{otherwise}) \end{cases} \).

The added free redex occurrences can be removed by \( \beta \)-reduction.

Lemma 6.4.29 If \( t \preceq_f t' \) then \( t' \rightarrow_\beta t \).

Proof

By induction on the derivation of \( t \preceq_f t' \).

\( \square \)

The extension of a legal term with free redex occurrences can be done in such a way that its type is preserved.

Lemma 6.4.30 If \( \Gamma \vdash_s t : u \) then \( t \preceq_f t' \) and \( \Gamma \vdash_s t' : u \) for some \( t' \).

Proof

If \( \Gamma \vdash_s t : u \) then by Proposition 6.3.38 we have \( u = \Delta \) or \( \Gamma \vdash_s u : s \) for some \( s \in \text{Universes} \).

By Lemma 6.3.6 and 6.3.8 we have \( t \preceq_f t \) if \( u = \Delta \) or \( \Gamma \vdash_s u : \Delta \). The cases \( \Gamma \vdash_s u : \square \) and \( \Gamma \vdash_s u : * \) and follow by induction on the derivation of \( \Gamma \vdash_s t : u \).

\( \square \)

We extend the relation \( \preceq_f \) to pseudo contexts. Roughly speaking, we have \( \Gamma \preceq_f \Delta \), if \( \Delta \) is obtained from \( \Gamma \) by replacing the right-hand sides of all rules by their extensions with free redex occurrences.

Definition 6.4.31 We define a relation \( \preceq_f \) on pseudo contexts as follows:

1. \( \varepsilon \preceq_f \varepsilon \).

2. If \( \Delta_1 \preceq_f \Delta_2 \) then \( \Delta_1, x : A \preceq_f \Delta_2, x : A \).

3. If \( \Delta_1 \preceq_f \Delta_2 \), and \( r_i \preceq_f q_i \) for all \( i \leq n \), then
   \( \Delta_1, F : t = (l_1 \Rightarrow r_1, \ldots, l_n \Rightarrow r_n) \preceq_f \Delta_2, F : t = (l_1 \Rightarrow q_1, \ldots, l_n \Rightarrow q_n) \).

We can simulate the reduction relation \( \rightarrow_s \), induced by a context, by reductions induced by extensions of this context.

Lemma 6.4.32 Assume \( \Delta \) is a legal context such that \( \Gamma \preceq_f \Delta \).

If \( \Gamma \vdash t_1 \rightarrow_s t_2 \) then \( \Delta \vdash t_1 \rightarrow_s t_3 \) and \( t_3 \rightarrow_\beta t_2 \) for some \( t_3 \).

Proof

By definition of \( \rightarrow_s \) we have \( \Gamma = (\Gamma_1, F : t = \tilde{r}, \Gamma_2) \), and \( t_1 = C[\text{subst}(\text{lhs}(r_i), \tilde{r})] \), and \( t_2 = C[\text{subst}(\text{rhs}(r_i), \tilde{r})] \), for some \( F \in C, C[ \ ] \in \mathcal{T}_r \), and \( \Gamma_1, \Gamma_2 \in \mathcal{X}_r \). As \( \Gamma \preceq_f \Delta \) we have \( \Delta = \Delta_1, F : t = \tilde{q} \), \( \Delta_2 \) with \( \Gamma_1 \preceq_f \Delta_1 \), and \( \Gamma_2 \preceq_f \Delta_2 \), and \( \text{lhs}(r_i) = \text{lhs}(q_i) \) and
rhys(r_i) \leq_f rhys(q_i). By definition we have \( \Delta \vdash t_1 \rightarrow_s C[\text{subst}(rhys(q_i), \vec{\sigma})] \). By Lemma 6.4.29 we have \( rhys(q_i) \rightarrow_{\beta} rhys(r_i) \), as \( r_i \leq_f q_i \). Thus by Lemma 6.2.25 we have \( \text{subst}(q_i, \vec{\sigma}) \rightarrow_{\beta} \text{subst}(r_i, \vec{\sigma}) \).

\[ \square \]

A legal context that is obtained from extending a context \( \Gamma \) with free redex occurrences has the same typing relation as \( \Gamma \).

**Lemma 6.4.33** If \( \Delta \) is a legal context and \( \Gamma \leq_f \Delta \) and \( \Gamma \vdash_s t : u \) then \( \Delta \vdash_s t : u \).

**Proof**
Induction on the derivation of \( \Gamma \vdash_s t : u \).
For \((\beta, s\text{-conversion})\) use previous lemma.

\[ \square \]

The extension of a legal context can be done in such a way that a legal context is obtained.

**Lemma 6.4.34** For any legal context \( \Gamma \) there exists a legal context \( \Gamma' \) such that \( \Gamma \leq_f \Gamma' \).

**Proof**
As \( \Gamma \) is legal we have \( \Gamma \vdash_s t : u \) for some \( t, u \). The result follows with induction on the derivation of \( \Gamma \vdash_s t : u \). The case where the \((\text{fun})\)-rule is applied, follows from Lemma 6.4.30.

\[ \square \]

We define a relation \( \rightarrow_{\beta'} \) that can contract an added free and its bound \( \beta\text{-redex} \) occurrence in one step.

**Definition 6.4.35** We define a compatible relation \( \rightarrow_{\beta'} \) initiated by the following rules:

1. \((\lambda x : \mathcal{A}. \lambda y : \mathcal{B}. f) bc \rightarrow_{\beta'} g[x := b][y := c] \) if \( f \in \mathcal{K}^S \).

2. \((\lambda x : \mathcal{A}. b) c \rightarrow_{\beta'} b[x := c] \) if \( b \neq \lambda y : \mathcal{B}. g \), for some \( f \in \mathcal{K}^S, g \in \mathcal{T}_s \).

**Remark 6.4.36** If \( t \rightarrow_{\beta'} u \) then \( t \rightarrow_{\beta}^+ u \).

Let \( X, Y \in \{\beta, \beta', \pi, s\} \). In this lemma we will show the possible forms of a contracted bound redex.

**Lemma 6.4.37**
1. If \( t_1 t_2 \rightarrow_{\beta}^X t_3 \) then \( t_3 = t_1' t_2' \) and either
   
   \( (a) \) \( t_1 \rightarrow_{\beta}^X t_1' \), or
   
   \( (b) \) \( t_2 \rightarrow_{\beta}^X t_2' \).

2. If \( \lambda y : t_1.t_2 \rightarrow_{\beta}^X t_3 \) then \( t_3 = \lambda y : t_1' t_2' \) and either
   
   \( (a) \) \( t_1 \rightarrow_{\beta}^X t_1' \), or
   
   \( (b) \) \( t_2 \rightarrow_{\beta}^X t_2' \), or
   
   \( (c) \) \( t_2 \rightarrow_{\beta}^X t_2' \) and \( t_1 \in \mathcal{K}^S \).
3. If $\Pi y:t_1.t_2 \rightarrow^b_X t_3$ then $t_3 = \Pi y:t'_1.t'_2$ and either
   
   (a) $t_1 \rightarrow^b_X t'_1$, or
   (b) $t_2 \rightarrow^b_X t'_2$, or
   (c) $t_2 \rightarrow^f_X t'_2$ and $t_1 \in \mathcal{K}^S$.

Proof
By definition of $\rightarrow^b_X$. \hfill \Box

Lemma 6.4.38 If $t_1 \rightarrow^b_X t_2$ then

1. $t_3[x := t_1] \rightarrow^b_X t_3[x := t_2]$.
2. $t_1[x := t_3] \rightarrow^b_X t_2[x := t_3]$.
3. If $u_1 \rightarrow^f_X u_2$ then $u_1[x := t_3] \rightarrow^f_X u_2[x := t_3]$.

Proof

1. By structural induction on $t_3$.
2. We have $t_1[x := t_3] \rightarrow_X t_2[x := t_3]$.
3. We have $u_1[x := t_3] \rightarrow_X u_2[x := t_3]$. \hfill \Box

Using the next lemma we can see that if we have a number of bound reduction steps followed by a free reduction step, we can do the free reduction step first. This is illustrated in Figure 6.4.

![Figure 6.4: Postponement of bound reduction](image)

Lemma 6.4.39 If $t_1 \rightarrow^f_Y t_2$ and $t_2 \rightarrow^f_X u_2$ then $t_1 \rightarrow^f_X u'_1 \rightarrow^f_Y u_1 \rightarrow^b_Y u_2$, for some $u'_1, u_1 \in T_\pi$.

Proof Let $T'_\pi = T_\pi \backslash \mathcal{K}^S$. We define the set $\mathcal{H}'_\pi$ of free terms with a hole as follows:
\[ \mathcal{H}_s^l = \mathcal{H}_s^l \uparrow T_s \ni T_s \uparrow \mathcal{H}_s^l \ni \lambda V.\mathcal{H}_s^l, T_s \ni \lambda V:T_s, \mathcal{H}_s^l \ni \Pi V: \mathcal{H}_s^l, T_s \ni \Pi V: T_s, \mathcal{H}_s^l. \]

We write \( C[l] \rightarrow Y C'[l] \) if \( C[l], C'[l] \in \mathcal{H}_s^l \) and \( C[t] \rightarrow Y C'[t] \) for all \( t \in T_s \).

If \( t_2 \rightarrow X^l u_2 \) then \( t_2 = C[l] \) and \( u_2 = C[r] \) and \( l \rightarrow X r \) for some \( l, r \in T_s \). \( C[l] \in \mathcal{H}_s^l \). If \( t_1 \rightarrow Y C[l] \) and \( C[l] \in \mathcal{H}_s^l \) then by Lemma 6.4.37 we have two cases.

Either \( t_1 = C'[l] \) and \( C'[l] \in \mathcal{H}_s^l \) and \( C'[l] \rightarrow Y C'[r] \) and \( C'[r] \rightarrow Y C[r] \).

Or \( t_1 = C[l'] \) and \( l' \rightarrow Y l \). Follows by structural induction on \( l \) using the definition of \( \rightarrow X \) for all \( X \in \{ \beta, \beta', \pi, s \} \) and Lemma 6.4.37 and 6.4.38. The only interesting case is \( X = \beta \) and \( l = (\lambda x:k.w_2)w_2 \) and \( r = w_3[x := w_2] \) and \( l' = (\lambda x:k.w_1)w_2 \) and \( w_1 \rightarrow Y^1 \) \( w_3 \) and \( k \in \mathcal{K}^\beta \). We have \( C[l'] \rightarrow Y^1 C[w_1[x := w_2]] \) and by Lemma 6.4.38 we obtain \( C[w_1[x := w_2]] \rightarrow Y^1 C[w_3[x := w_2]] \).

\( \square \)

The extension with free redex occurrences is preserved under substitution.

**Lemma 6.4.40** Let \( x \in V_0, t_3 \in T_K \). If \( t_2 \leq_f t_2' \) and \( t_3 \leq_f t_3' \) then \( t_2[x := t_3] \leq_f t_2'[x := t_3'] \).

**Proof**

Induction on the derivation of \( t_2 \leq_f t_2' \). We will treat one case.

Case \( t_2 = \Pi y:A.B \) and \( t_2' = (\lambda x:A.B')(B'[y := \text{dummy}_Y(A))] \), with \( A \in \mathcal{K}^\pi \), \( B \in T_K \) such that \( B \leq_f B' \). By the variable convention we have \( x \neq y \). We have \( A[x := t_3] = A = A[x := t_3'] \) and \( \text{dummy}_Y(A)[x := t_3'] = \text{dummy}_Y(A) \), as \( x \notin FV(A), FV(\text{dummy}_Y(A)) \).

And we have \( B[x := t_3] \in T_K \), as \( B, t_3 \in T_K \) and \( x \in V_0 \). Thus we have \( t_2[x := t_3] = \Pi y:A.B[x := t_3] \) and \( t_2'[x := t_3'] = (\lambda x:A.B'[x := t_3']) \).

By Lemma 6.2.4. By the induction hypothesis we have \( B[x := t_3] \leq_f B'[x := t_3'] \). Thus \( t_2[x := t_3] \leq_f t_2'[x := t_3'] \).

\( \square \)

A reduction step starting with a term can be simulated by a number of bound reduction steps followed by a free reduction step starting with any legal extension with free redex occurrences of this term.

**Proposition 6.4.41** Assume \( t_1 \leq_f t_1' \), and \( \Gamma \vdash t_1 : k \), for some \( k \in \mathcal{K} \). If \( \Gamma \vdash t_2 \rightarrow_{\beta,s} t_2 \) and \( \Gamma \leq_f \Delta \) then \( \Delta \vdash t_2' \rightarrow_{\beta,s} t_2'' \) and \( \Delta \vdash t_2' \rightarrow_{\beta,s} t_2'' \) and \( t_2 \leq_f t_2' \), for some \( t_2', t_2'' \).

**Proof**

Induction on the proof of \( t_1 \leq_f t_1' \). We will treat the interesting cases.

1. \( t_1 = f_1 a_1 \) and \( t_1' = f_1' a_1' \) and \( f_1 \leq_f f_1' \) and \( a_1 \leq_f a_1' \).

As \( \Gamma \vdash t_1 : k \), we must have \( \Gamma \vdash f_1 : k' \rightarrow k \) and \( \Gamma \vdash a_1' : k' \), for some \( k' \).

We have four subcases for \( \Gamma \vdash t_1 \rightarrow_{\beta,s} t_2 \).

**Case** \( t_2 = f_2 a_1 \), with \( \Gamma \vdash f_1 \rightarrow_{\beta,s} f_2 \). Follows easily from induction hypothesis.

**Case** \( t_2 = f_1 a_2 \), with \( \Gamma \vdash a_1 \rightarrow_{\beta,s} a_2 \). Follows easily from the induction hypothesis.
Case $f_1 = \lambda x:k'.c$, and $t_2 = c[x := a_1]$. We have $t'_1 \rightarrow^f_{\beta,s} c'[x := c'_1]$, and $c \leq_f c'$ for some $c'$, for both cases for $f'_1$. By Lemma 6.4.40 we have $c[x := a_1] \leq_f c'[x := c'_1]$.

Case $t_1 = \text{subst}(\text{lhs}(r_1), \sigma)$ and $t_2 = \text{subst}(\text{rhs}(r_1), \sigma)$, and $\Gamma = \Gamma_1, F:k' = \tau, \Gamma_2$. We have $\Delta = \Delta_1, \Gamma: k' = \tau, \Gamma_2$, and $\Delta \leq_f \Delta_1$, and $\Gamma_2 \leq_f \Delta_2$ and $\text{lhs}(r_j) = \text{lhs}(q_j)$, and $\text{rhs}(r_j) \leq_f \text{rhs}(q_j)$, for all $j \leq |\tau|$. We must have $t'_1 = \text{subst}(\text{lhs}(r_1), \tau) = \text{subst}(\text{lhs}(q_1), \tau)$ with $\sigma \leq_f \tau$ (see Definition 6.2.11), for some substitution sequence $\tau$. By definition we have $\Delta \vdash \text{subst}(\text{lhs}(q_1), \tau) \rightarrow^f_{\beta,s} \text{subst}(\text{rhs}(q_1), \tau)$. By Lemma 6.4.40 we have $\text{subst}(\text{rhs}(r_1), \sigma) \leq_f \text{subst}(\text{rhs}(q_1), \tau)$.

2. $t_1 = \lambda x:k_1.c_1$. We must have $t_2 = \lambda x:k_1.c_2$ with $\Gamma \vdash c_1 \rightarrow_{\beta,s} c_2$. We have two subcases.

Case $k_1 \in \mathcal{K}^S$. We have $t'_1 = (\lambda:k_2.\lambda x:k_1.c_1)(c'_1[x := \text{dummy}_S(k_1)])$ and $c_1 \leq_f c'_1$, for some $c'_1, k_2$. As $\Gamma \vdash c'_1 : k_2$, we must have $\Gamma \vdash c'_1[x := \text{dummy}_S(k_1)] : k_2$ and $\Gamma, x : k_1 \vdash c'_2 : k_2$ and $k_2 \in \mathcal{K}$. By the induction hypothesis we obtain $c'_2, c'_2'$ such that $\Delta, x : k_1 \vdash c'_1 \rightarrow_{\beta,s} c'_2$ and $\Delta, x : k_1 \vdash c'_2 \rightarrow_{\beta,s} c'_2'$ and $c_2 \leq_f c'_2$. Thus we have $\Delta \vdash \lambda : k_2, \lambda x : k_1.c'_1 \rightarrow_{\beta,s} \lambda : k_2, \lambda x : k_1.c'_2$. We also have $\Delta \vdash c'_1[x := \text{dummy}_S(k_1)] \rightarrow_{\beta,s} c'_2[x := \text{dummy}_S(k_1)]$ and $\Delta \vdash c'_2[x := \text{dummy}_S(k_1)] \rightarrow_{\beta,s} c'_2[x := \text{dummy}_S(k_1)]$. Let $t'_2 = (\lambda : k_2, \lambda x : k_1.c'_2)(c'_2[x := \text{dummy}_S(k_1)])$, and $t'_2 = (\lambda : k_2, \lambda x : k_1.c'_2)(c'_2[x := \text{dummy}_S(k_1)])$. Now we have $\Delta \vdash t'_1 \rightarrow_{\beta,s} t'_2$ and $\Delta \vdash t'_2 \rightarrow_{\beta,s} t'_2$. By definition we have $t_2 \leq_f t'_2$.

Case $k_1 \notin \mathcal{K}^S$. Follows easily from the induction hypothesis. \hfill \Box

A consequence of this proposition is that any $\rightarrow_{\beta,s}$-reduction sequence in $\Gamma$ starting with a legal term can be simulated by a $\rightarrow_{\beta,s}$-reduction sequence of at least the same length and starting with a legal term. We will prove this in the next corollary.

![Figure 6.5: Simulation of reduction by free reduction](image)

**Corollary 6.4.42** $\rightarrow_{\beta,s}$ is strongly normalizing for legal terms in $\mathcal{T}_\mathcal{K}$.
Proof
Assume $\Gamma_0 \vdash_s t_0 : k$. By applying the substitution lemma we can obtain a $K^S$-free context $\Gamma_1$ and a term $t_1$ (an instance of $t_0$) such that $\Gamma_1 \vdash_s t_1 : k$. As $\rightarrow_{\beta,s}$ is preserved under substitution it is sufficient to show that there is not an infinite reduction sequence starting with $t_1$. Using Lemma 6.4.30 we obtain a term $u_1$ such that $t_1 \leq_f u_1$ and $\Gamma_1 \vdash_s u_1 : k$. Using Lemmas 6.4.34, 6.4.33 we obtain a context $\Delta$ such that $\Gamma_1 \leq_f \Delta$ and $\Delta \vdash_s u_1 : k$.
Using Proposition 6.4.41 and Lemma 6.4.39 we can simulate any $\rightarrow_{\beta,s}$-reduction sequence starting with $t_1$ by a $\rightarrow_{\beta',s}$ reduction sequence of at least the same length starting with $u_1$ (see Figure 6.5). As $\rightarrow_{\beta,s}$ is strongly normalizing for legal terms in $\mathcal{T}_K$ by Corollary 6.4.25, we have that any reduction sequence starting with $t_1$ is finite. $\Box$

Encoding of proofs
If we want to use the method for encoding terms in $\mathcal{T}_K$ to encode formal proofs we encounter several problems:

1. For deriving the type of a formal proof the conversion rule may be needed (see the formal proof in Example 6.1.37). Moreover a proposition may be part of a formal proof, and therefore its reduction behaviour can be needed in the derivation of its type. As we have seen before, the encoding does not always preserve the reduction behaviour of terms in $\mathcal{T}_K$ that contain variables.

2. The encoding of recursive proofs of the form $F : t = \tau$ would require a default value that has the encoding of $t$ as type (see Definition 6.3.51). Thus an encoding of this proof of $t$ would require a proof of the encoding of $t$.

The first problem can be solved, by using an encoding for propositions that are part of formal proofs, that does not count the number of constructors of its arguments. Instead, the encoding of these propositions should be instantiated with sufficiently large numbers, such that all reduction steps needed for type correctness can be performed by the encoding. In this approach we would replace all recursive calls in a recursive function by an extra variable at once (instead of replacing only the recursive calls for the first decreasing position by a variable as in Definition 6.3.41). For determining the maximum number of iterations that are needed for the encoding, we need a derivation tree for the type of the term that will be encoded.

The second problem can only be solved by adding an assumption of the false proposition to the context of the encoding. Of course, this is not a desirable solution.

If we consider this problem from a mathematical perspective, it is not important how the proof of a proposition is encoded, as long as some proof of the encoding of the proposition is given. Thus only the type, and not the structure or reduction behaviour of the proof are interesting. A proof defined by a recursive function definition can be interpreted as a proof by well-founded induction on the tuples containing the number of constructor symbols of the arguments on the decreasing positions combined with reasoning by cases. Clearly, we can represent these reasoning principles in $\lambda$HOL with inductive types. Using these
representations we could encode proofs without the need for an assumption of the false proposition. But still we would need the encoding, that we sketched above, for propositions that occur in proofs. As a precise description of an encoding for proofs would be quite complicated, we will not give such a formal definition.

6.5 Typability

We will show that typability and type checking are decidable for $\vdash_s$, that is the restricted version of the typing relation $\vdash_\pi$ of $\lambda$HOL with pattern matching, presented in Definition 6.3.35. For this purpose we will define a syntax directed type inference system $\vdash_{sd}$. Its definition is inspired by the rules for a syntax directed version of ECC ([26]), the type system for LEGO, given in [31]. Following the ideas of Pollack we will show that $\vdash_{sd}$ is sound and complete for $\vdash_s$. Finally we will prove that typability and type checking are decidable for $\vdash_{sd}$.

We call the system $\vdash_{sd}$, that is presented in Figure 6.6, syntax directed, because for a given pseudo context $\Gamma$ and pseudo term $a$ at most one rule can be applicable to derive a type $t$ such that $\Gamma \vdash_{sd} a : t$. For each rule the pseudo contexts and subjects of its premises are fully determined by the pseudo context and subject of the conclusion (except for the choice of the fresh variable replacing the function symbol in the sdfun-rules). The main restrictions of $\vdash_{sd}$ with respect to $\vdash_s$ are that weakening is only allowed for basic terms and that conversion is only allowed when it is needed.

**Lemma 6.5.1** $\vdash_{sd}$ is sound for $\vdash_s$.

**Proof** We have to prove $\Gamma \vdash_{sd} a : t \Rightarrow \Gamma \vdash_s a : t$. Induction on the derivation of $\Gamma \vdash_{sd} a : t$. We discuss the interesting cases.

*Case (sdfun), (sdfun-weak), or (sapplication) is applied last.* By part 1. of Proposition 6.3.38 and Lemma 6.2.36 we obtain that the $(\beta, s$-conversion) rule may be applied.

The (sabstraction) rule is applied last. We have $\Gamma \vdash_{sd} \lambda x : t.a : \Pi x : t.u$, because $\Gamma \vdash_{sd} t : s$ and $\Gamma, x : t \vdash_{sd} a : u$ and $(s = * \text{ and } u \in \mathcal{T}_K)$ or $(s = \Box \text{ and } u \in \mathcal{K} \cup \mathcal{T}_K))$. By the induction hypothesis we have $\Gamma \vdash_s t : s$ and $\Gamma, x : t \vdash_s a : u$. By Lemma 6.3.38 we obtain $\Gamma, x : t \vdash_s u : s'$ for some $s' \in \text{Universes}$. If $u \in \mathcal{K}$ then $s' = \Box$ by Lemma 6.3.8. If $u \in \mathcal{T}_K$ then $s' = *$ by Lemma 6.3.10. By the (product) rule we obtain $\Gamma \vdash_s \Pi x : t.u : s'$.

Thus by (abstraction) we have $\Gamma \vdash_s \lambda x : t.a : \Pi x : t.u$. \qed

**Lemma 6.5.2** $\vdash_{sd}$ is complete for $\vdash_s$.

**Proof** We have to prove $\Gamma \vdash_s a : t$ implies $\Gamma \vdash_{sd} a : u$ and $\Gamma \vdash t =_{\beta,s} u$, for some $u$. Notice that we can always postpone the use of the weakening rules until the subject has become a basic term in a derivation in $\vdash_s$. Thus we may assume that the weakening rules are only used with a basic term as subject. Induction on the derivation of $\Gamma \vdash_s a : t$. There are two interesting cases.
(sdaxiom) \[ e \vdash_{sd} s_1 : s_2 \] if \((s_1, s_2) \in \text{Axioms}\)

(advar-start) \[ \frac{\Gamma \vdash_{sd} t:s}{\Gamma, x : t \vdash_{sd} x : t} \] if \(x \in V_\Gamma \setminus FV(\Gamma)\)

(advar-weak) \[ \frac{\Gamma \vdash_{sd} b : t \quad \Gamma \vdash_{sd} u:s}{\Gamma, x : u \vdash_{sd} b : t} \] if \(x \in V_\Gamma \setminus FV(\Gamma), b \neq x\)

(sddata-type) \[ e \vdash_{sd} d : \square \] for \(d \in S\)

(sddata-const) \[ e \vdash_{sd} C : d \] if \(\tau(C) = d\)

(sddata-constr) \[ \frac{\Gamma \vdash_{sd} t_1 : d_1 \ldots \Gamma \vdash_{sd} t_n : d_n}{\Gamma \vdash_{sd} C t_1 \ldots t_n : d} \] if \(\tau(C) = d_1 \times \ldots \times d_n \rightarrow d, 1 \leq n\)

(sdfun) \[ \frac{\Gamma, x : t, \Gamma \vdash_{sd} (\text{lhs}(r_i)[F \leftarrow x]): u_i \quad \Gamma, x : t, \Gamma \vdash_{sd} (\text{rhs}(r_i)[F \leftarrow x]): u_i |^F_i}{\Gamma, \delta : \tau(t) \vdash_{sd} F : t} \] if \(F \notin \text{consts}(\Gamma)\)
- \text{ctxtlhs}(\Gamma_1, \text{lhs}(r_i), t)
- \text{exhaustive}(F_\delta = \tau)
- \text{argument decreasing}(F_\delta = \tau)
- \(\Gamma \vdash t_i =_{\beta, \alpha} u_i\)
- \(F \in C_\alpha, x \in V_\Gamma \setminus FV(\Gamma, \Gamma_1)\)

(sdfun-weak) \[ \frac{\Gamma \vdash_{sd} b : u}{\Gamma, x : t, \Gamma \vdash_{sd} \text{lhs}(r_i)[F \leftarrow x] : u_i \quad \Gamma, x : t, \Gamma \vdash_{sd} \text{rhs}(r_i)[F \leftarrow x] : u_i |^F_i}{\Gamma, \delta : \tau(t) \vdash_{sd} b \leftarrow F : u} \] if \(F \notin \text{consts}(\Gamma)\)
- \text{ctxtlhs}(\Gamma_1, \text{lhs}(r_i), t)
- \text{exhaustive}(F_\delta = \tau)
- \text{argument decreasing}(F_\delta = \tau)
- \(\Gamma \vdash t_i =_{\beta, \alpha} u_i\)
- \(F \in C_\alpha, x \in V_\Gamma \setminus FV(\Gamma, \Gamma_1), b \neq F\)

(sdproduct) \[ \frac{\Gamma \vdash_{sd} t : s_1 \quad \Gamma \vdash_{sd} u : s_2}{\Gamma \vdash_{sd} (\Pi x : t . u) : s_3} \] if \((s_1, s_2, s_3) \in \text{Rules}\)

(sdabstraction) \[ \frac{\Gamma \vdash_{sd} s \quad \Gamma \vdash_{sd} (\lambda x : t . a) : (\Pi x : t . u)}{\Gamma \vdash_{sd} (\lambda x : t . a) : (\Pi x : t . u)} \] if \(s = * \land u \in T_\mathcal{K}, \text{or} \ s = \square \land u \in \mathcal{K} \cup T_\mathcal{K}\)

(sdapplication) \[ \frac{\Gamma \vdash_{sd} f : t \quad \Gamma \vdash_{sd} a : u}{\Gamma \vdash_{sd} f \leftarrow a : t_2[x := a]} \] if \(\Gamma \vdash t \rightarrow_{\beta, \alpha} t_1 \quad \Pi x : t_1 . t_2, \Gamma \vdash u =_{\beta, \alpha} t_1 \quad f \neq C a_1 \ldots a_n, \text{all} \ C \in \Sigma, a_1, \ldots, a_n, 0 \leq n\)

Figure 6.6: Syntax directed typing rules
In the rules of Figure 6.6 we use the following notation:

\begin{align*}
  s, s_1, s_2, s_3 & \in \text{Universes;} \\
  d, d_1, \ldots, d_n & \in S; \\
  b & \in V \cup S \cup C \cup \Sigma; \\
  a, a_1, \ldots, a_n, f, t, t', t_i, u, u_i & \in \mathcal{T}_\pi; \\
  \bar{r} & \in \mathcal{R}^+. \\
\end{align*}

The sets \( \mathcal{K} \) and \( \mathcal{T}_\mathcal{K} \) are defined in Definitions 6.3.7, 6.3.9.

The last applied rule was the (application) rule: \( \Gamma \vdash \lambda x e : B[x := e] \), because

\( \Gamma \vdash f : \Pi x : A.B, \) and \( \Gamma \vdash e : A \). By the induction hypothesis we have \( \Gamma \vdash_{\text{sd}} f : t_1 \), \( \Gamma \vdash \Pi x : A.B \rightarrow_{\beta, s} t_1 \), and \( \Gamma \vdash e : t_2 \), \( \Gamma \vdash A =_{\beta, s} t_2 \). As \( \rightarrow_{\beta, s} \) is confluent we have \( \Gamma \vdash t_1 \rightarrow_{\beta, s} \Pi x : A'.B' \) and \( \Gamma \vdash \Pi x : A.B \rightarrow_{\beta, s} \Pi x : A'.B' \). We also have \( \Gamma \vdash t_2 =_{\beta, s} A' \).

Using (sadapplication) we obtain \( \Gamma \vdash_{\text{sd}} f e : B'[x := e] \). As \( \Gamma \vdash B =_{\beta, s} B' \) we also have \( \Gamma \vdash B[x := e] =_{\beta, s} B'[x := e] \).

The last applied rule was the (abstraction) rule: \( \Gamma \vdash \lambda x : A.b : \Pi x : A.B \), because \( \Gamma, x : A \vdash_{\text{sd}} b : B \) and \( \Gamma \vdash \Pi x : A.B : s \). By the induction hypothesis we have \( \Gamma, x : A \vdash_{\text{sd}} b : t_1 \), \( \langle \Gamma, x : A \rangle \vdash t_1 =_{\beta, s} B \) and \( \Gamma \vdash_{\text{sd}} \Pi x : A.B : t_2 \), and \( \Gamma \vdash t_2 =_{\beta, s} s \). To obtain the last result the (sdproduct) rule must have been applied. Thus we have \( \Gamma \vdash_{\text{sd}} A : s_1 \), \( \langle \Gamma, x : A \rangle \vdash_{\text{sd}} B : s_2 \), and \( t_2 = s_3 \) and \( (s_1, s_2, s_3) \in \text{Rules} \). By soundness we have \( \Gamma, x : A \vdash_{\text{sd}} b : t_1 \), and \( \Gamma, x : A \vdash_{\text{sd}} B : s_2 \). Thus we have \( \Gamma \vdash_{\text{sd}} t_1 : s_2 \) as \( \Gamma \vdash t_1 =_{\beta, s} B \).

Considering Rules we either have \( s_1 = * \) and \( s_2 = * \), or \( s_1 = \square \) and \( s_2 \in \{ *, \square \} \).

Case \( s_2 = * \). Then \( t_1 \in \mathcal{T}_\mathcal{K} \) by Lemma 6.3.10.

Case \( s_2 = \square \). Then \( t_1 \in \mathcal{K} \) by Lemma 6.3.8.

Thus we have \( \Gamma \vdash_{\text{sd}} \lambda x : A.b : \Pi x : A.t_1 \) by the (sadabstraction) rule. Moreover we have \( \Gamma \vdash \Pi x : A.B =_{\beta, s} \Pi x : A.t_1 \).

\( \square \)

**Type synthesis algorithm**

The rules for \( \vdash_{\text{sd}} \) suggest a type synthesis algorithm for a given pseudo context \( \Gamma \) and term \( t \).

**Construction of a quasi derivation tree:**

First we construct a quasi derivation tree, that is a derivation tree in which the predicates are replaced by question marks, as follows:

1. Start with the quasi statement \( \Gamma \vdash_{\text{sd}} t ? \).
2. If there is a quasi statement \( \Gamma' \vdash_{\text{sd}} t' ? \) on top of the tree such that \( \Gamma' \) is nonempty or \( t' \notin \Sigma \cup S \cup \text{Universes} \), find all rules that might be applicable. Otherwise stop: the tree is ready.
3. Verify the type-independent syntactic constraints for the found rules.

4. If the constraints are met for one rule, compute the pseudo contexts and subjects of its premises, add them on top of the old top (replacing the types with ?), and continue with step 2. Otherwise fail: the term is not typable in the given context.

If this first step is successful we have obtained a tree of which all quasi statements on the top consist of an empty pseudo context and a basic term.

*Fill in types in leaves:*

The second step consists of replacing all question marks on top by types such that valid statements are obtained. If this is not possible, then the term is not typable in the given context.

*Complete the derivation tree:*

If the second step is successful, then we can try to complete the derivation tree as follows:

1. If there is a quasi rule application with typed premises and a ? as predicate of the conclusion, then verify the constraints for the rule that depend on the predicates of the premises and not on the predicate of the conclusion. Otherwise stop: the derivation is finished.

2. If these constraints are met, compute the type for the conclusion and replace the ? with this type. Otherwise fail.

3. Verify the constraints that depend on the predicate of the conclusion.

4. If these constraints are met, continue with step 1. Otherwise fail.

**Example 6.5.3** We will demonstrate the use of the algorithm for deriving a type for \((\lambda x:\text{Nat}.s \ x)\) 0.

1. The first step is to construct a quasi derivation tree. We start with the quasi statement:

\[ \epsilon_{\text{sd}}(\lambda x:\text{Nat}.s \ x) \text{ O?} \]

The only applicable rule is the \((\text{sdapplication})\)-rule. We obtain:

\[
\frac{\epsilon_{\text{sd}} \lambda x:\text{Nat}.s \ x \text{ ?} \quad \epsilon_{\text{sd}} \text{ 0 ?}}{\epsilon_{\text{sd}} (\lambda x:\text{Nat}.s \ x) \text{ O?}}
\]

As the subject of the quasi statement on the left is an abstraction, the algorithm continues with this quasi statement. Finally we obtain the following quasi derivation tree:
\[\begin{align*}
\epsilon &\vdash_{sd} \text{ Nat} \, ? \\
x &\vdash_{sd} \text{ Nat} \, x \\
\epsilon &\vdash_{sd} \text{ Nat} \, ? \\
x &\vdash_{sd} S \, x \\
\epsilon &\vdash_{sd} \lambda x. \text{ Nat} \, S \, x \\
\epsilon &\vdash_{sd} (\lambda x. \text{ Nat} \, S \, x) \, O \, ? \\
\end{align*}\]

2. Notice that all quasi statements on top must of the form $\epsilon \vdash_{sd} ?$, with $b \in \Sigma \cup S \cup \text{Universes}$. Thus for determining the types of the leaves we only need to consider the rules (sddata-const), (sddata-type), and (sdaxiom). As $\text{Nat} \in S$ we fill in $\square$ for $?$ in $\epsilon \vdash_{sd} \text{ Nat} \, ?$. After filling in types in the leaves we obtain:

\[\begin{align*}
\epsilon &\vdash_{sd} \text{ Nat} : \square \\
x &\vdash_{sd} \text{ Nat} \, x \\
\epsilon &\vdash_{sd} \text{ Nat} : \square \\
x &\vdash_{sd} S \, x \\
\epsilon &\vdash_{sd} \lambda x. \text{ Nat} \, S \, x \\
\epsilon &\vdash_{sd} (\lambda x. \text{ Nat} \, S \, x) \, O \, ? \\
\end{align*}\]

3. Now we will complete the derivation tree. The only quasi rule application with typed premises and a $?$ as type of the conclusion is:

\[\begin{align*}
\epsilon &\vdash_{sd} \text{ Nat} : \square \\
x &\vdash_{sd} \text{ Nat} \, x \\
\end{align*}\]

The only applicable rule is (sddvar-start). Therefore we replace the $?$ with $\text{Nat}$. Now the algorithm continues with the quasi rule application below, as its premise has just been given a type. Finally we obtain the following derivation:

\[\begin{align*}
\epsilon &\vdash_{sd} \text{ Nat} : \square \\
x &\vdash_{sd} \text{ Nat} \, x \\
\epsilon &\vdash_{sd} \text{ Nat} : \square \\
x &\vdash_{sd} S \, x \\
\epsilon &\vdash_{sd} \lambda x. \text{ Nat} \, S \, x \\
\epsilon &\vdash_{sd} (\lambda x. \text{ Nat} \, S \, x) \, O : \text{Nat} \\
\end{align*}\]

We will prove that the type synthesis algorithm terminates and is correct. First we show that we can always compute the normal form of the type of a term.

**Lemma 6.5.4** If $\Gamma \vdash_{s} a : t$ then $t$ is strongly normalizing.

**Proof**

By part 1. of Proposition 6.3.38 we have $t = \Delta$ or $\Gamma \vdash_{s} t : s$, for $s \in \text{Universes}$. Thus we have $t \in \{\square, \Delta\}$, or $t \in K$ or $t \in T_{K}$. The last case is treated by Corollary 6.4.42. In the other cases $t$ is in normal form.
Proposition 6.5.5  Typability is decidable for $\vdash_s$.

Proof
Using completeness and soundness it is sufficient to prove that typability in $\vdash_{sd}$ is decidable. We will show that the type synthesis algorithm terminates and that it decides typability. We define the length of a pseudo term as follows: $\text{length}(\Pi x : A. B) = \text{length}(AB) = \text{length}(\lambda x : A. B) = 1 + \text{length}(A) + \text{length}(B)$, $\text{length}(b) = 1$, for $b \in V \cup S \cup C \cup \Sigma$; and we define the length of a pseudo context as the sum of the lengths of the types in its variable declarations. We define a measure of a (quasi) statement as follows: $\text{measure}(\Gamma \vdash_{sd} a : t) = (|\text{fundefs}(\Gamma)|, \text{length}(\Gamma) + \text{length}(a))$.

For each rule in $\vdash_{sd}$ we have that the measure of each premise is smaller than the measure of the conclusion ($(m_1, m_2) < (n_1, n_2)$ if $m_1 < n_1$ or $m_1 = n_1$ and $m_2 < n_2$). Thus the first step of the algorithm terminates, as the syntactic constraints are decidable. As the rules are syntax directed at most one rule can be applicable after verifying the syntactic constraints. Thus failure of the first step implies that the pseudo term is not typable in the pseudo context.

The second step of the algorithm terminates as the side conditions for (sdaxiom), (sddatatype) and (sddata-const) are decidable. If this step fails, then a premise needed for obtaining a valid derivation does not hold; whence the pseudo term is not typable in the pseudo context.

The conditions in the third step are decidable, as for legal types $\equiv_{\beta, s}$ is decidable due to Lemma 6.5.4 and confluence for $\to_{\beta, s}$. If the third step fails, then a constraint for a rule needed for a valid derivation cannot be met; therefore the pseudo term is not typable in the pseudo context. If the algorithm succeeds, a valid derivation tree has been constructed.

Remark 6.5.6 We can decide type checking for $\vdash_s$ as follows: Assume we want to know whether $\Gamma \vdash_s a : t$.

1. Use the type synthesis algorithm on $\Gamma \vdash_{sd} a : ?$. If it fails then $\Gamma \not\vdash_s a : t$.
2. Otherwise we obtain a type $u_1$ such that $\Gamma \vdash_s a : u_1$. If $t = u_1$ then we have $\Gamma \vdash_s a : t$.
3. Otherwise use the type synthesis algorithm on $\Gamma \vdash_{sd} t : ?$. If it fails then $\Gamma \not\vdash_s a : t$.
4. Otherwise we get a type $u_2$ such that $\Gamma \vdash_s t : u_2$. If $u_2 \notin \text{Universes}$ then $\Gamma \not\vdash_s a : t$, because the ($\beta, s$-conversion) rule cannot be applied.
5. Now we have $\Gamma \vdash_s a : t$ if and only if $\Gamma \vdash t =_{\beta, s} u_1$. By Lemma 6.5.4 we can compute the normal form of $u_1$. As $u_2 \in \text{Universes}$ we can compute the normal form of $t$ according to the proof of Lemma 6.5.4. Thus by confluence of $\to_{\beta, s}$ we have $\Gamma \vdash t =_{\beta, s} u_1$ if and only if $t$ and $u_1$ have the same normal form.
Chapter 7

Proof Development using Priority Rewriting

In the previous chapter we have described an extension of the type system Higher Order Logic with priority rewriting, and we have showed that the obtained language is suited for automated verification. In this chapter we will describe how we have extended the proof development system LEGO with priority rewriting. First we briefly describe the proof assistant LEGO. Then we discuss how algebraic data types and function definitions are represented in our prototype, and how the typing rules for these new constructs are verified in cooperation with the proof checking facilities offered by LEGO. We describe the commands for entering algebraic data types and function definitions, and a tactic for interactively constructing a proof by reasoning by cases.

In order to test our implementation we have done a case study. We present the formalization of a method for proving primality by computations in this prototype. This method is based on witnesses that can be easily computed by a Computer Algebra System. Using a top-down approach we have refined the statement of the correctness of this method to a number of basic properties of standard functions. Thus we have constructed a formal proof of the correctness of the method for proving primality based on the assumptions that these basic properties hold. Finally we discuss the experience we obtained from this test case.

7.1 LEGO

In this section we will shortly describe the proof development system LEGO [27] that implements an extension of Higher Order Logic with inductive types (see Section 4.2). The main extra constructs of the language implemented by LEGO are definitions, strong sum types ($\Sigma$-types), and an infinite hierarchy of type universes.

Besides facilities for type checking and type synthesis (computing the type of a term) LEGO also provides support for interactive proof development by tactics. Formalizing a mathematical theory in LEGO means constructing a mathematical context consisting
of variable declarations and definitions. Depending on its predicate a variable declaration should be interpreted as an assumption or an introduction of a primitive notion. A definition just gives a name to a typable expression.

A tactic is an algorithm that makes constructing formal proofs easier in several ways. First of all, it reverses the order in which the rules of the formal system must be applied. Thus starting with a conclusion a tactic computes the statements that must be proved for the application of a formal rule. Second, the use of tactics increases the abstraction level of the reasoning steps, because a tactic can invoke several rules of the formal system. Third, tactics can automatically fill in details in a formal proof. Moreover, we do not have to pay attention to the proof-object that is constructed by the tactics.

**Example 7.1.1** The most frequently used tactic in LEGO is the Refine tactic that tries to prove a certain proposition by unifying it with a specialization of a given proof-object. For instance, we can prove a proposition $P(0)$ by invoking the Refine tactic with a proof of the proposition $\forall n \ P(n)$, that automatically specializes it for 0.

If the given proof-object proves a conditional proposition the Refine tactic computes the instances of the conditions that still have to be proved. For instance, applying the Refine tactic for a proposition $Q(b)$ with a proof of the conditional proposition $\forall m, n \ C(m, n) \rightarrow Q(n)$ results in a request for a term $?_m$ and a proof of $C(?_m, b)$.

We will not give a full description of the syntax for LEGO expressions and commands, because we will mainly use the standard notation of typed lambda-calculus. We hope that the following short characterization provides sufficient information for understanding the rare examples in LEGO syntax.

**Notation 7.1.2** The LEGO syntax for pseudo terms is a bit different from the usual notation. A pseudo term $\lambda x : t . b$ is denoted in LEGO as $[x : t] b$, and a pseudo term $\Pi x : t . u$ is denoted as $\{x : t\} u$. The type universe for propositions ($\ast$) is denoted as $\text{Prop}$, and the type universe for sets ($\Omega$) is denoted as $\text{Type}(\Omega)$. The expression $\text{Type}(n)$ is a type universe in LEGO, for each natural number $n$.

A user command begins after the LEGO prompt (Lego> ) and ends with a semi colon (;). An interactive proof is started by the command Goal $prop$, where $prop$ is the proposition one wants to prove. If the proposition has been proved, then the system responds with the message *** QED ***.

The main motivation to choose LEGO is that it allows us to experiment with rewrite rules. Thus one can give a constant a computational meaning by specifying rewrite rules for it. The only restrictions that are imposed on rewrite rules are that both sides of a rule should have the same type. Thus it is possible to represent non-terminating and non-confluent Term Rewriting Systems in LEGO. The rewriting facilities that are offered by the proof assistant Coq impose severe restrictions on the rules (see Section 1.3), and therefore Coq does not provide enough freedom to implement function definitions. Defining rewrite rules in LEGO changes the equality relation on terms that is used in the conversion rule: the computational meaning of constants defined by rewrite rules can be used in formal proofs.
In the next section we will use this feature to implement function definitions by pattern matching.

**Strong sum types**

A strong sum type is a generalization of the cartesian product, because it provides a type for pairs of which the type of the second component may depend on the first component. We will only give a general description of this type construct, and will not discuss matters that are concerned with its implementation in LEGO.

A strong sum type has syntax $\Sigma x : A . B$, where $x$ is a variable that may occur in $B$. If $x$ does not occur in $B$, $\Sigma x : A . B$ is the ordinary product type $A \times B$. An example of a type that cannot be represented by $\times$ is the type for positive natural numbers, that can be defined as $\Sigma n : \text{Nat}. 0 < n$. A typical inhabitant of this type is a pair consisting of a natural number and a proof of its positivity.

The pair constructor for $\Sigma$-types does not only need the two components, but also its $\Sigma$-type. Thus a typical dependent pair has form $(a, b)_{\Sigma x : A . B}$. This last argument is needed to guarantee the uniqueness of the type of the dependent pair and to keep type checking feasible. Furthermore, we have two operators for extracting the first and the second component of dependent pairs.

**7.2 LEGO with Pattern Matching**

In this section we describe our extension of the proof development system LEGO with algebraic data types and function definitions as defined in Section 6.1. The first task of the extension is *type checking*: we must be able to verify typing judgements using the typing rules for this type system. The second task is assistance in *proof development*: the user of the tool must be supported in formalizing mathematics in the type system.

**Type checking**

For fulfilling the first task our tool consists of the following parts:

1. Representations for an algebraic data type signature and function definitions.

2. A data base controlling the construction of an algebraic data type signature and a legal context of function definitions.

3. An interface with LEGO.

The reliability of our tool depends on these parts and (the type checking facilities of) LEGO.

First we need to describe how algebraic data types and function definitions are represented. An algebraic data type consists of a sort and a number of function symbols (constructors) with their arity. We represent it by an ML type `datadef`. A function
definition is represented by an ML type `fundef` that for each rule contains a local context (assigning a type to the variables in the left-hand side) and a type (the type of the right-hand side).

Second an algebraic data type signature is represented by the ML type `datadef list` and a context is represented by the ML type `redef list`. The heart of the implementation is formed by a data base that contains the representations of an algebraic data type signature and a context of function definitions. The data base can be freely inspected, but adding a new algebraic data type or a function definition to the data base is only possible via special functions. These partial functions only succeed if the new algebraic data type or function definition is correct with respect to the current data base. The initial data base is empty. We will now describe how to add elements to the data base. Assume we have constructed a data base representing an algebraic data type signature $\Sigma$ and a context of function definitions $\Gamma$.

We can **add an algebraic data type definition** with sort $d$ to the data base, if:

1. The names in each arity are sorts of $\Sigma$ or equal to $d$.
2. At least one constructor has an arity in which $d$ does *not* occur.
3. The names of the constructors and $d$ are all distinct and are not in the data base.

Thus instead of starting with a fixed algebraic data type signature (as in the formal description) we allow the construction of an algebraic data type signature in a sequential way.

We can **add a function definition** to the data base if:

1. All its rules have the same number of correct $\Sigma$-patterns in the left-hand sides.
2. Its rules are argument decreasing and weakly head-orthogonal.
3. Its rules are exhaustive in $\Sigma$.
4. The local context of each rule is sufficient for its left-hand side.
5. The name of the function is not in the data base.

The demand for weak head-orthogonality, that guarantees confluence, is needed because we interpret the rules of a function definition as (ordinary) rewrite rules (and not as priority rewrite rules as in our formal description). We will motivate this choice when we discuss the translation to LEGO.

We need a **translation** of algebraic data types and function definitions to LEGO in order to be able to verify that the types of both sides of each rule are the same and to use the reduction relation induced by the function definitions in the data base. LEGO has an undocumented feature for adding rewrite rules to the context. A rule is accepted if both sides have the same type in the provided context. The computational meaning for these rules is similar to the rewrite relation in TRSs: a left-hand side of an instance of a rule is
reduced to the right-hand side. If more rules are applicable it is unknown which rule will be applied. Note that the rewrite rules determine the equivalence relation on types used in the conversion rule.

If we want to translate a function definition in our formal system to a number of LEGO rules we must transform it from a priority rewrite system into a rewrite system. As this transformation is not entirely trivial (see Section 5.3), we want to keep it out of the type checking part (that determines the reliability) of our tool. Therefore we translate formal proofs in LEGO with Pattern Matching in two stages to LEGO code (see Figure 7.1). We will define type checking for a language, that we call LEGO with Rewriting, that formalizes function definitions based on ordinary rewriting. In our formal system we use priority rewriting as a method to guarantee confluence. In order to preserve this property for the reduction relation induced by a function definition we impose the extra condition of weak head-orthogonality (see Definition 5.2.6), that is easy to verify.

An algebraic data type definition is translated to LEGO code as a number of constant declarations. A function definition is translated to LEGO code as a constant declaration together with a number of rewrite rules. Note that if the translated version is accepted by LEGO, for each rule the type of the left-hand side is equal (modulo rewriting) to the type of the right-hand side.

Thus type checking an algebraic data type or a function definition is a combination of adding the new definition to the data base and entering its translated version in LEGO. If LEGO or the data base detect an error the data base and the LEGO context are restored to their original values.

**Interactive proof development**

When doing formal mathematics one does not want to work directly at the level of type checking. Therefore we made a user interface that transforms the textual input of a user into type checking commands. In this way the user does not need to know which ML types represent algebraic data types and function definitions or which algorithms are used for type checking. Thus the user can concentrate on giving correct formal definitions. If the user makes a mistake the tool responds with an error message.

Via the user interface one can try to enter an algebraic data type or a function definition, or give a LEGO command. In the first two cases the new definition is verified using the type checking functions described above.
Definition 7.2.1 We have one command for defining an algebraic data type. First one specifies the sort of the algebraic data type and then its constructors with their types. It has the following grammar:

\[
\text{Data } \ id=id\_type \ | \ \ldots \ | \ id\_type \quad \text{Where } type=id \mid id \to type.
\]

In the grammar above \( id \) stands for identifier.

Example 7.2.2 The algebraic data type for natural numbers can be defined by the following command:

\[
\text{Data } \text{Nat}=0\_\text{Nat} \ | \ \text{S} \_\text{Nat} \to \text{Nat}
\]

Entering a function definition can be done via a command in which a function definition is given in textual form or via a tactic in which the right hand sides of the rules are constructed by LEGO tactics. For entering function definitions with small right-hand sides the first command will be sufficient. The tactic is useful for constructing a proof by reasoning by cases.

Definition 7.2.3 The command for entering a function definition directly has the following grammar:

\[
\text{Rec } \ id\_\text{term}=\text{rule } \to \ \ldots \to \ \text{rule} \quad \text{With } \text{rule}=id\_\text{pat}^+ \to \text{term}.
\]

Where \( \text{term} \) is a LEGO term and \( \text{pat} \) is a pattern.

The rules are separated by \( \to \) to indicate that they are regarded as priority rewrite rules. The obtained function definition is transformed by an algorithm, that is based on the function ‘transform’ described in Section 5.3, into a weakly head-orthogonal function definition whose rules are regarded as ordinary rewrite rules. For each rule a local context and a type for the right-hand side are computed. Finally the correctness of the function definition is verified.

Example 7.2.4 We can define the boolean relation \( \leq \) on natural numbers by:

\[
\text{Rec } \text{Leq} \_\text{Nat} \to \text{Nat} \to \text{Bool}=
\]

\[
\text{Leq } \text{0 } y \to \text{True }>
\]

\[
\text{Leq } (\text{S} \_x) (\text{S} \_y) \to \text{Leq } x \_y >
\]

\[
\text{Leq } x \_y \to \text{False}
\]

Notice that the transformation algorithm transforms the last rule into \( \text{Leq } (\text{S} \_x) \_0 \to \text{False} \).

Definition 7.2.5 The tactic for entering function definitions is started by specifying the name and type of the function with the command: \text{Rec } \ id\_\text{term};

Then the following procedure is repeated until no cases are left:

1. The user enters the patterns of the left-hand side.
2. The local context for this rule and the type of the right hand side of the rule are computed.

3. The user is asked to solve a goal of this type (to construct the right-hand side).

4. If the goal is solved, the cases that still have to be treated to obtain an exhaustive definition are displayed.

The obtained function definition is transformed into a weakly head-orthogonal function definition whose rules are regarded as ordinary rewrite rules. Finally the correctness of the transformed function is verified.

**Example 7.2.6** We will demonstrate the use of this tactic in a proof of the reflexivity of \(\text{Leq}\) (see Example 7.2.4). The user input is displayed in **bold**. We have defined **trivial** as the true proposition with inhabitant **trivialprf** (see Example 3.2.2). A short introduction to LEGO syntax is given in Notation 7.1.2.

The user enters the name and type of the function:

```lego
Rec Leq_refl:\{x:Nat\}IsTrue(Leq x x);
```

The tactic responds by asking for a left-hand side:

```lego
Lhs ? Leq_refl
```

The user enters: \(O\); The tactic shows the goal for the right-hand side:

```lego
Goal
?0 : IsTrue (Leq 0 0)
```

The user enters **Refine trivialprf**; The result of this command is displayed:

```lego
Refine by trivialprf
*** QED ***
```

The goal is solved as \(\text{IsTrue}(\text{Leq} \ 0 \ 0)\) is equal to \(\text{trivial}\) modulo rewriting. The left cases are displayed and the user is asked to enter a left-hand side:

```lego
left cases for Leq_refl:\{x:Nat\}IsTrue (Leq x x)
```

```lego
Leq_refl (S #)
```

The tactic provides the right local context and displays the goal that must be solved:

```lego
decl y : Nat
Goal
?0 : IsTrue (Leq (S y) (S y))
```

The user enters **Refine Leq_refl y**; which solves the goal as \(\text{Leq} \ (S \ y) \ (S \ y)\) reduces to \(\text{Leq} \ y \ y\). The tactic responds with:

```lego
Refine by Leq_refl y
*** QED ***
```

As all cases are treated the obtained function definition is transformed, verified and entered in LEGO:

```lego
decl Leq_refl : \{x:Nat\}IsTrue(Leq x x)
```
LEGOf With Pattern Matching

[[y: Nat]
 Leq_ref1 0 ==> trivialprf
\| Leq_ref1 (S y) ==> Leq_ref1 y]
added reductions

Polymorphism

In previous sections we described how our prototype implements the formal language of Section 6.1. In fact, we have implemented a more general language with polymorphic algebraic data types and function definitions.

Example 7.2.7 We can define the algebraic data type for polymorphic lists as:

\[ \text{Data List a} = \text{Nil:List} \mid \text{Cons:a->List->List} \]

In this definition \(a\) is a generic type variable for which one may fill in any legal set type. For instance, List Nat is an instance of this algebraic data type and has constructors Nil Nat:List Nat, and Cons Nat: Nat->(List Nat)->(List Nat).

Example 7.2.8 We can define a polymorphic function that maps a function over a list as:

\[
\text{Rec Map:\{a, b: Type(0)\}(a->b)->(List a)->(List b)=}
\text{Map a b f (Nil a) ==> Nil b >}
\text{Map a b f (Cons a x l) ==> Cons b (f x) (Map a b f l)}
\]

Recall that Type(0) is the LEGO notation for the type of sets (\(\square\)).

Notice that the syntax for polymorphic function definitions is rather redundant, as the generic type variables are repeated after each occurrence of the function name in the rules, and the types of the polymorphic constructors in the patterns are explicitly mentioned, whereas they can be derived from the type of the function definition. Therefore a more convenient notation would be:

\[
\text{Rec Map a b:(a->b)->(List a)->(List b)=}
\text{Map f Nil ==> Nil b >}
\text{Map f (Cons x l) ==> Cons b (f x) (Map f l)}
\]

This latter syntax is quite close to the syntax used in functional programming languages.

The main advantage of polymorphic algebraic data types and functions is that a whole class of operators can be specified by one definition. For instance, the polymorphic function Map can be used to transform lists of any type. Polymorphism is also an abstraction mechanism, since a property that holds for a polymorphic function also holds for all its instances.
7.3 Proving Primality by Computations

In this section we will discuss how we can verify the primality of a number using computations. As we must be able to do serious computations we will use a binary representation for the natural numbers. First we will formalize binary numbers in our type system.

**Definition 7.3.1** Algebraic data types for binary representation of natural numbers:

<table>
<thead>
<tr>
<th>sort</th>
<th>Bit, Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td>0:</td>
</tr>
<tr>
<td></td>
<td>1:</td>
</tr>
<tr>
<td></td>
<td>Nul:</td>
</tr>
<tr>
<td></td>
<td>Bnc: Bin \times Bit \rightarrow Bin</td>
</tr>
</tbody>
</table>

As a short cut we will write 1011 in stead of Bnc (Bnc (Bnc (Nul 1) 0) 1) 1. Thus we will also denote Bnc Nul 1 by 1; from the context in which 1 appears it should be clear whether the bit or the binary number is meant.

**Definition 7.3.2** We can define addition on Bin by the following function definitions:

\[
\begin{align*}
\text{Inc:} & \text{Bin} \rightarrow \text{Bin} = \\
\text{Inc Nul} & \Rightarrow 1, \\
\text{Inc (Bnc n o)} & \Rightarrow \text{Bnc n 1}, \\
\text{Inc (Bnc n 1)} & \Rightarrow \text{Bnc (Inc n) 0} \\

\Delta_{\text{Add}} = & \\
\text{Add:} & \text{Bin} \rightarrow \text{Bit} \rightarrow \text{Bin} = \\
\text{Addbit n 1} & \Rightarrow \text{Inc n}, \\
\text{Addbit n 0} & \Rightarrow n \\
\text{Add:} & \text{Bin} \rightarrow \text{Bin} \rightarrow \text{Bin} = \\
\text{Add n Nul} & \Rightarrow n, \\
\text{Add Nul n} & \Rightarrow n, \\
\text{Add (Bnc m a) (Bnc n b)} & \Rightarrow \text{Addbit (Bnc (Add m n) a) b} \\
\end{align*}
\]

Recall from the previous chapter that function definitions give a computational meaning to defined symbols.

**Example 7.3.3** For instance we can compute the value of 5 + 3 as follows:

\[
\Delta_{\text{Add}} \vdash \text{Add 101 11} \rightarrow \text{1000}.
\]

**Remark 7.3.4** The algebraic data type Bin does not provide a unique representation of the natural numbers. For instance, both 10 and 010 represent the number 2. A binary number is canonical if it is Nul or its first bit is 1. We can define a function Canonical : Bin \rightarrow Bool, that tests this property. Note that the functions in the previous definition map canonical binary numbers to canonical binary numbers. In the rest of the section we will leave out all canonicity conditions to improve the readability of the statements.
Recall that a number \( p \) is prime if its only divisors are 1 and \( p \), and if \( 1 < p \). We have defined some more standard functions and relations on Bin, but we will not show these definitions. Based on these function definitions we give an algorithmic definition of primality in our system. In the definitions below we ‘abuse’ our function definition mechanism for simulating ordinary definitions.

**Definition 7.3.5** Operational variants of divisibility and primality:

\[
\begin{align*}
\text{Divides:} & \quad \text{Bin} \rightarrow \text{Bin} \rightarrow \text{Bool} = \\
& \quad \text{Divides } l \; m \Rightarrow \text{Or } (\text{Eq } m \; \text{Nul}) \; (\text{And } (\text{Less } l \; \text{Nul}) \; (\text{Eq } (\text{Mod } m \; l) \; \text{Nul})) \\
\text{PrimeCondition:} & \quad \text{Bin} \rightarrow \text{Bin} \rightarrow \text{Bool} = \\
& \quad \text{PrimeCondition } p \; q \Rightarrow \text{Not}(\text{Divides } q \; p) \\
\text{Prime:} & \quad \text{Bin} \rightarrow \text{Bool} = \\
& \quad \text{Prime } p \Rightarrow \text{And } (\text{Less } l \; p) \; (\text{All } (\text{PrimeCondition } p) \; (\text{FirstSqrt } p))
\end{align*}
\]

In this definition the expression FirstSqrt \( p \) produces the binary representation of the list \([2, 3, \ldots, \lfloor \sqrt{p} \rfloor]\), and All tests whether a predicate holds for all members of a list.

**Example 7.3.6** As we have defined primality using rewrite rules we can compute it for any binary number. For instance, Prime 111 reduces to True, in accordance with \( 7 \) is prime.

Unfortunately, it is not very efficient to use the definition to decide the primality for large numbers in this way. If enough prime divisors of \( n - 1 \) can be found, one can verify the primality of \( n \) by the following condition.

**Lemma 7.3.7 (Critère de Pocklington)** Let \( n \in \mathbb{N} \), \( n > 1 \), with \( n - 1 = q \cdot m \) such that \( q = q_1 \cdots q_t \) for certain primes \( q_1, \ldots, q_t \). Suppose \( a \in \mathbb{Z} \) satisfies \( a^{n-1} = 1 \in \mathbb{Z}_n \) and \( \gcd(a^{\frac{n-1}{q_i}} - 1, n) = 1 \) for all \( i = 1, \ldots, t \). If \( q \geq \sqrt{n} \), then \( n \) is a prime.

**Proof**
Let \( p \) be any prime divisor of \( n \). Put \( b = a^m \). Then \( b^t = a^{n-1} = 1 \in \mathbb{Z}_n \), so \( b \) has order a divisor of \( q \) in the cyclic group \( \mathbb{Z}_p^\times \). On the other hand, if \( i \in \{1, \ldots, t\} \), then, by hypothesis, there are \( \alpha, \beta \in \mathbb{Z} \) with \( \alpha(a^{\frac{n-1}{q_i}} - 1) + \beta n = 1 \), so \( b^\frac{\alpha}{q_i} = 1 \in \mathbb{Z}_p \), which is absurd. Hence, \( b \) has order precisely \( q \) in \( \mathbb{Z}_p^\times \). Consequently, \( \sqrt{n} \leq q \leq p - 1 \). Finding that \( n \) has no prime divisors less than or equal to \( \sqrt{n} \), we conclude that it is prime.

**Example 7.3.8** We can check that 1223 is prime as follows. Take \( q = 47 \) (note that 1222 = 47 * 26) and \( a = 5 \) (this 5 is a generator of \( \mathbb{Z}_q^\times \)) and verify that \( 5^{1222} = 1 \mod 1223 \), and \( \gcd(5^{\frac{1222}{47}} - 1, 1223) = 1 \), and \( 47 \geq \sqrt{1223} \).
For using this criterion to prove the primality of a number \( p \) the assistance of a Computer Algebra System for computing the prime factors of \( p - 1 \) and a generator for \( \mathbb{Z}_p^* \) is very helpful. The verification of the primality of \( p \) using the obtained numbers can be done in our tool. We will formalize this method in our type system. We will not give all definitions of the functions we need to compute the Critère de Pocklington. We will only discuss the differences in our definition with respect to the original one.

**Definition 7.3.9** We need an algebraic data type for lists of binary numbers:

<table>
<thead>
<tr>
<th>sort</th>
<th>BinList</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td>BNil: BinList</td>
</tr>
<tr>
<td></td>
<td>BCons: Bin \times BinList \to BinList</td>
</tr>
</tbody>
</table>

We define a function \texttt{MullList} that multiplies the members of a list as:

\[
\text{MullList}: \text{BinList} \to \text{Bin} = \\
\text{MullList} \ B\text{Nil} \quad \Rightarrow \quad 1 \\
\text{MullList} \ (\text{BCons} \ m \ l) \quad \Rightarrow \quad \text{Mul} \ m \ (\text{MullList} \ l)
\]

Now we can test the condition \( \gcd(a^n - 1, n) = 1 \) for a given list of primes \( q_1, \ldots, q_l \) as follows:

\[
\text{Pockgcdtest}: \text{BinList} \to \text{Bin} \to \text{Bin} \to \text{Bin} \to \text{Bool} = \\
\text{Pockgcdtest} \ B\text{Nil} \ a \ m \quad \Rightarrow \quad \text{True} \\
\text{Pockgcdtest} \ (\text{BCons} \ q \ l) \ n \ a \ m \quad \Rightarrow \quad \text{And} \ (\text{Pockgcdtest} \ l \ n \ a \ (\text{Mul} \ m \ q)) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ Quad
We will try to prove this statement in our system using a top-down approach. Thus we must construct an inhabitant of this type in a legal context containing the definitions for Pockprimetest and not containing axioms (variable declarations). We will give an impression of the construction of a formal proof for this statement without paying attention to the constructed object (just like in most proof development systems). Recall that we leave out all canonicity conditions to improve the readability of the statements.

**Definition 7.3.11** We define the notion order as follows:

\[
\begin{align*}
\text{UnitWitness}: & \text{Bin} \rightarrow \text{Bin} \rightarrow \text{Bool} = \\
\text{UnitWitness } a p m & \Rightarrow \text{And} \left( \text{Eq} \left( \text{Mod} \left( \text{Exp} \ a \ m \right) \ p \right) \ 1 \right) \left( \text{Less} \ Nul \ m \right) \\
\text{IsOrder}: & \text{Bin} \rightarrow \text{Bin} \rightarrow \text{Bin} \rightarrow + = \\
\text{IsOrder } a p o & \Rightarrow (\text{IsTrue}(\text{UnitWitness } a \ p \ o)) \land \\
& \forall w \in \text{Bin}. (\text{IsTrue}(\text{UnitWitness } a \ p \ w)) \rightarrow \text{IsTrue}(\text{Divides } a \ w)
\end{align*}
\]

We will present simpler statements, such that if we prove these refining statements, we can easily obtain a proof of the original refined statement. We call the procedure of replacing a statement by simpler statements refinement.

**Refinement 7.3.12** \( \forall n, a, m \in \text{Bin}. \forall l \in \text{BinList}. (\text{IsTrue}(\text{All Prime } l)) \rightarrow (\text{IsTrue}(\text{Pockprimetest } n \ a \ m \ l)) \rightarrow \text{IsTrue}(\text{Prime } n) \) by

1. \( \forall n \in \text{Bin}. (\text{IsTrue}(\text{Less } 1 \ n)) \rightarrow \neg (\text{IsTrue}(\text{Prime } n)) \rightarrow \\
\exists p \in \text{Bin}. \text{IsTrue}(\text{And} \left( \text{And} \left( \text{Divides } p \ n \right) \left( \text{Less } 1 \ p \right) \right) \left( \text{Leq} \left( \text{Mul} \ p \ p \ n \right) \right))
\]

2. \( \forall n, a, m, p \in \text{Bin}. \forall l \in \text{BinList}. (\text{IsTrue}(\text{Pockprimetest } n \ a \ m \ l)) \rightarrow \text{IsTrue}(\text{All Prime } l)) \rightarrow (\text{IsTrue}(\text{And} \left( \text{Divides } p \ n \right) \left( \text{Less } 1 \ p \right)) \rightarrow \text{IsOrder} \left( \text{Exp} \ a \ m \right) \ p \left( \text{MulList } l \right)
\]

3. \( \forall x, p, o \in \text{Bin}. (\text{IsTrue}(\text{Less } 1 \ p)) \rightarrow (\text{IsOrder } x \ p \ o) \rightarrow \text{IsTrue}(\text{Less } o \ p)
\]

4. \( \forall n, p, o \in \text{Bin}. (\text{IsTrue}(\text{Leq } n \left( \text{Mul} \ o \ o \right))) \rightarrow \text{IsTrue}(\text{Less } o \ p) \rightarrow \text{IsTrue}(\text{Less } n \left( \text{Mul} \ p \ p \right))
\]

5. \( \forall x, y \in \text{Bin}. (\text{IsTrue}(\text{Leq } x \ y)) \rightarrow \neg (\text{IsTrue}(\text{Less } y \ x))
\]

In each refinement step we construct a formal proof of the refined statement based on the assumption of the refining statements. If we have proof-objects for all these statements, we can easily obtain a proof-object for the refined statement.

We can repeat the refinement procedure for the refining statements. For instance, we can refine the second part of Refinement 7.3.12 as follows:

**Refinement 7.3.13** \( \forall n, a, m, p \in \text{Bin}. \forall l \in \text{BinList}. (\text{IsTrue}(\text{Pockprimetest } n \ a \ m \ l)) \rightarrow (\text{IsTrue}(\text{All Prime } l)) \rightarrow (\text{IsTrue}(\text{And} \left( \text{Divides } p \ n \right) \left( \text{Less } 1 \ p \right))) \rightarrow \\
\text{IsOrder} \left( \text{Exp} \ a \ m \right) \ p \left( \text{MulList } l \right) \) by

1. \( \forall n, a, m, p \in \text{Bin}. \forall l \in \text{BinList}. (\text{IsTrue}(\text{Pockprimetest } n \ a \ m \ l)) \rightarrow \\
(\text{IsTrue}(\text{And} \left( \text{Divides } p \ n \right) \left( \text{Less } 1 \ p \right))) \rightarrow \text{IsTrue}(\text{UnitWitness} \left( \text{Exp} \ a \ m \right) \ p \left( \text{MulList } l \right))
\]
2. \( \forall a, p, w \in \text{Bin}.(\text{IsTrue}(\text{Less } 1 \, p)) \rightarrow (\text{IsTrue}(\text{UnitWitness } a \, p \, w)) \rightarrow \exists o \in \text{Bin}.\text{IsOrder } a \, p \, o \)

3. \( \forall n, a, m, p \in \text{Bin}.\forall l \in \text{BinList}.(\text{IsTrue}(\text{Pockprimetest } n \, a \, m \, l)) \rightarrow (\text{IsTrue}(\text{AllPrime } l) \rightarrow (\text{IsTrue}(\text{And } (\text{Divides } p \, n) \, (\text{Less } 1 \, p))) \rightarrow \forall o \in \text{Bin}.(\text{IsTrue}(\text{UnitWitness } (\text{Exp } a \, m) \, p \, o)) \rightarrow (\text{IsTrue}(\text{Divides } o \, (\text{MulList } l))) \rightarrow \text{IsTrue}(\text{Eq } o \, (\text{MulList } l)) \)

We can refine the first part of Refinement 7.3.13 as follows:

Refinement 7.3.14 \( \forall n, a, m, p \in \text{Bin}.\forall l \in \text{BinList}.(\text{IsTrue}(\text{Pockprimetest } n \, a \, m \, l)) \rightarrow (\text{IsTrue}(\text{And } (\text{Divides } p \, n) \, (\text{Less } 1 \, p))) \rightarrow \text{IsTrue}(\text{UnitWitness } (\text{Exp } a \, m) \, p \, (\text{MulList } l)) \) by

1. \( \forall a, b, c, r \in \text{Bin}.(\text{IsTrue}(\text{Divides } b \, a)) \rightarrow (\text{IsTrue}(\text{Less } \text{Nul } a)) \rightarrow (\text{IsTrue}(\text{Eq } (\text{Mod } c \, a) \, r)) \rightarrow (\text{IsTrue}(\text{Less } r \, b)) \rightarrow \text{IsTrue}(\text{Eq } (\text{Mod } c \, b) \, r) \)

2. \( \forall a, b, c \in \text{Bin}.\text{IsTrue}(\text{Eq } (\text{Exp } (\text{Exp } a \, b) \, c) \, (\text{Exp } a \, (\text{Mul } b \, c))) \)

3. \( \forall l, m \in \text{Bin}.(\text{IsTrue}(\text{Less } \text{Nul } (\text{Mul } l \, m))) \rightarrow (\text{IsTrue}(\text{Less } \text{Nul } m)) \)

4. \( \forall a, b, c \in \text{Bin}.(\text{IsTrue}(\text{Less } (\text{Plus } a \, c) \, b)) \rightarrow \text{IsTrue}(\text{Less } a \, (\text{Sub } b \, c)) \)

The statements of the last Refinement are basic properties of standard functions. We have refined all statements of Refinement 7.3.12 until we reached statements of this basic level.

Remark 7.3.15 We have constructed a context \( \Delta \) and a proof-object \( P \) such that:

\[ \Delta |-_P : \forall n, a, m \in \text{Bin}.\forall l \in \text{BinList}.(\text{IsTrue}(\text{AllPrime } l)) \rightarrow (\text{IsTrue}(\text{Pockprimetest } n \, a \, m \, l)) \rightarrow \text{IsTrue}(\text{Prime } n) \]

Using this \( P \) we can obtain a formal proof of \( \text{Prime } n \) for any prime number \( n \). Such a proof is based on witnesses that must be computed by a Computer Algebra System. Notice that the obtained formal proof is verified completely in LEGO and does not require extra axioms or external computations. Thus we use the CAS as a guide (see Section 4.1). Besides definitions, the context \( \Delta \) contains axioms, that state basic properties of standard functions.

Example 7.3.16 Using the proof-object \( P \), that formally proves the Critère de Pocklington, we obtain a formal proof of the primality of 1223 (see Example 7.3.8) as follows:
Step 1: Computation of the witnesses that are needed for the construction of the formal proof in a Maple session.

```
> ifactor(1223-1);
(2) (13) (47)
> primroot(1223);
5
```

Step 2: Verification of the LEGO proof script, that is based on these witnesses, that generates a formal proof of the primality of 1223.

```
Goal IsTrue(Prime(100111000111));
Refine P;
Refine 101;
Refine 11010;
Refine BCons 101111 BNil;
Refine trivialprf;
Refine trivialprf;
Save prime1223;
```

The Maple function \( \text{ifactor}(n) \) factorizes the integer \( n \), and the command \( \text{primroot}(p) \) computes a generator of the group \( \mathbb{Z}_p^* \).

After four \texttt{Refine} steps of the LEGO proof script have been verified, two goals still must be proved:

1. \( \text{IsTrue(All Prime (BCons 101111 BNil))} \);
2. \( \text{IsTrue(PockPrimetest 100111000111 101 11010 (BCons 101111 BNil))} \).

Both expressions reduce to \( \text{IsTrue(True)} \), which is proved by the proof-object \texttt{trivialprf} (see Example 7.2.6).

The process of refinement requires human interaction, since finding a number of useful, provable, simpler statements that lead to the proof of a given statement is not a decidable problem.

**Experience**

The formalization of this example was a test case for the use of priority rewriting in interactive proof development. Most operators on the binary numbers could be easily represented as function definitions. The restriction that function definitions should be argument decreasing only influenced the definition of \( \text{Gcd} \) (greatest common divisor) and functions that are of primitive recursive nature such as testing a property for all numbers smaller than some number. As the natural definitions for these functions are not argument decreasing, we were forced to use auxiliary functions in order to be able to define them.

**Example 7.3.17** For computing \( \text{Gcd} \ a \ b \) (for positive \( a \)) we want to have a recursive call \( \text{Gcd} \ (\text{Mod} \ b \ a) \ a \). Unfortunately, this call is not argument decreasing. Therefore we defined an auxiliary function \( \text{Gcdfun} \) with a counter as extra argument in order to obtain a argument decreasing definition.
\[
\begin{align*}
\text{Gcdfun:} & \text{Bin } \rightarrow \text{ Bin } \rightarrow \text{ Bin } = \\
\text{Gcdfun } t & \text{ Nul } b \quad \Rightarrow \quad b \\
\text{Gcdfun } (\text{Bnc } t \ u) & \text{ a } b \quad \Rightarrow \quad \text{Gcdfun } t (\text{Mod } b \ a) \ a \\
\text{Gcdfun } \text{Nul } a & \ b \quad \Rightarrow \quad \text{Nul} \\
\text{Gcd:} & \text{Bin } \rightarrow \text{ Bin } \rightarrow \text{ Bin } = \\
\text{Gcd } a & \ b \quad \Rightarrow \quad \text{Gcdfun } (\text{Bnc } (\text{Mul } a \ b) \ 1) \ a \ b
\end{align*}
\]

The value of the counter initiated by a call of Gcd is large enough to prevent an application of the last rule of Gcdfun, that is only needed to obtain an exhaustive definition.

In order to allow the definition of functions by primitive recursion we have defined a function \textit{Natrec}$_A$ of type \( A \rightarrow (\text{Bin } \rightarrow A \rightarrow A) \rightarrow (\text{Bin } \rightarrow A) \) for a 'set' \( A \). Similarly we defined a function for doing natural induction.

In order to be able to hide the canonicity conditions for inhabitants of Bin in the proofs we used \( \Sigma \)-types (see Section 7.1). Thus we used pairs consisting of a binary number \( b \) and a proof that \( b \) is canonical. Furthermore we proved that the operators on Bin preserve canonicity via reasoning by cases and used these proofs to construct operators that map canonical binary numbers to canonical binary numbers. The tactic for entering function definitions made the construction of these proofs relatively easy, as it provides the goal for a selected case and displays the untreated cases after it has been proved. It really makes a difference to be able to hide the canonicity conditions, since they are a substantial part of the formal proof.

First the operators should be defined in such a way that their fundamental properties can be proved easily. Once the correctness proof has been established one can improve the speed of the computations by defining faster equivalent operators. For instance, in the definition of \textit{Pockprimetest} we have to compute \( \text{Mod } (\text{Exp } a (\text{Sub } n \ 1)) \ n \). We can define a more efficient operator \textit{Expmod} that combines the operators \textit{Exp} and \textit{Mod}, that works with smaller intermediate results.

Another way to improve the speed of the computation of \textit{Pockprimetest} is to use better computable criteria. For instance, we can safely replace the test

\[ \text{Eq } (\text{Gcd } (\text{Sub } (\text{Exp } a (\text{Mul } m (\text{MulList } l)))) \ 1) \ n \ 1 \]

by

\[ \text{Eq } (\text{Gcd } (\text{Sub } (\text{ExpMod } a (\text{Mul } m (\text{MulList } l))) \ n \ 1) \ n \ 1 \]

in \textit{Pockgcdtest}, because we have \( \text{gcd}(x - 1, n) = \text{gcd}((x, n) - 1, n) \), if \( \text{mod}(x, n) \neq 0 \).

This condition is satisfied if the other tests of \textit{Pockprimetest} succeed.

Using tactics one can construct a proof using the available knowledge in the context. In order to avoid that a user of a proof development system gets lost during the construction of a formal proof, the number of steps in such a proof should be kept low (say below hundred). Thus it is not feasible to formalize large proofs by solving a goal using tactics. The
construction of a large formal proof requires a top-down approach in which the complexity of the statements that still have to be proved can be decreased step by step. Unfortunately, this method for constructing large proofs is not really supported by LEGO (and many other proof development systems). We would like to be able to

1. introduce a statement \( t \), that one wants to prove in a context \( \Gamma \), as a conjecture;

2. replace a conjecture \( u \) by several weaker conjectures \( u_1, \ldots, u_n \) (\( 0 \leq n \)), if a proof-object \( p \) such that that \( \Gamma, x_1:u_1, \ldots, x_n:u_n \vdash p : u \) is provided;

3. automatically obtain a proof-object \( P \) such that \( \Gamma \vdash P : t \), if no conjectures are left.

At first sight one might think that conjectures can be represented as assumptions. The introduction of a conjecture \( t \) in context \( \Gamma \) would then be represented as

\[
\frac{\Gamma \vdash t : \ast}{\Gamma ; x : t \vdash t : t} \quad \text{if} \quad x \notin FV(\Gamma)
\]

A refinement step in which a conjecture \( x : u \) is refined by conjectures \( \Delta = x_1 : u_1, \ldots, x_n : u_n \) can be represented as

\[
\frac{\Gamma_1 \vdash u : \ast \quad \Gamma_1 ; x : u, \Gamma_2 \vdash q : t \quad (\Gamma_1 \vdash u_i : \ast)^{i=1}_{p} \quad \Gamma_1, \Delta \vdash r : u \quad \Gamma_1, \Delta, \Gamma_2 \vdash q[x := r] : t}{\Gamma_1, \Delta \vdash r : u \quad \text{if} \quad FV(\Delta) \cap FV(x : u, \Gamma_2) = \emptyset}
\]

Thus a refinement step would consist of the replacement of an assumption by a number of assumptions. Unfortunately, an efficient procedure that takes care of this is not available in LEGO (only the last context item can easily be replaced by a number of context items). In principle one could write a tactic that performs this task, but executing such a tactic would be time consuming because all context items after the replaced item would have to be removed and entered after the new conjectures are added. Without such a tactic one could only refine the last conjecture which is quite unnatural.

A more practical solution is to represent the conjectures and (the proof-objects provided in) the refinement steps only in the proof development system and not in the context of the proof checker. The introduction of a statement \( t \) as a conjecture can be implemented by the verification of \( \Gamma \vdash t : \ast \) by the proof checker and the storage of the conjecture \( t \) in the proof development system. A refinement step, that is certified by proof-object \( r \), where the conjecture \( x : u \) is replaced by conjectures \( \Delta = x_1 : u_1, \ldots, x_n : u_n \) can be implemented by the verification of \( \Gamma \vdash u_i : \ast \) (all \( i \leq n \)) and \( \Gamma, \Delta \vdash r : u \) by the proof checker and the storage of the refinement step in the proof development system. If the conjectures \( \Delta \) are proved then we obtain a proof of the conjecture as follows:

\[
\frac{\Gamma, \Delta \vdash r : u \quad (\Gamma \vdash p_i : u_i)^{i=1}_{p}}{\Gamma \vdash r[x_1 := p_1 \ldots x_n := p_n] : u}
\]

Thus the construction of proofs in the proof checker is postponed until a refinement step does not introduce new conjectures. If all conjectures are proved using the refinement procedure we obtain a proof of the original conjecture in this way.
Notice that the number of conjectures (and the size of the constructed proof-object) can
grow very fast. In order to keep the size of the constructed proof-object small one should
add definitions, that give names to proofs of successfully refined conjectures, to the context.
Using this approach we finally obtain a sequence of definitions $\Gamma'$ and a proof-object $p$ such
that $\Gamma, \Gamma' + \pi \vdash p : t$. 
Bibliography


Index

β-reduction, 22
β, π-conversion, 85
λ-reduction, 39
π-reduction, 85

Abstract Reduction Systems, 11
abstraction mechanisms, 4
algebraic data type, 14
almost orthogonal, 57
argument decreasing, 59
  - function definition, 87
ARS, 11

bound variable, 21

commutes, 12
Computer Algebra Systems, 29
confluence, 18
  - for →β,π, 96
  - for →β, 27
  - for →π, 93
confluent, 12, 19, 44
  - enabled reduction, 75
Constructor Systems, 56
collection rule, 42
Critère de Pocklington, 149
critical pair, 18

enabled reduction, 62, 85
exhaustive, 56
  - function definition, 83

formal mathematics, 2
free variable, 21
function definition, 79
functional programming languages, 14, 54

Higher Order Logic, 24
inductive pseudo terms, 36
inductive types, 36
inhabitant, 3, 24
habitation, 28
instantiation of pseudo term, 80
legal, 25
LEGO, 140
lexicographical path order, 17
most general unifier, 18

normal form, 12
oracle type, 50

pattern, 79
pattern matching, 54
polymorphism, 147
predicate, 23
priority, 55
Priority Constructor System, 61
priority rewrite system, 61
proof checking, 4
proof development system, 5
proof-object, 3
propositions-as-types, 2, 25
pseudo context, 23, 80
pseudo term, 22, 78
  - with a hole, 22, 78
PTS, 27
Pure Type System, 27

redex, 16
redex occurrence, 16
reduct, 11
Index

reduction, 11
  - sequence, 12
  - step, 11, 16, 80
refinement, 151
reliable, 4
rewrite rule, 15
rule, 79

statement, 23
strong normalization, 61
strong sum type, 142
strongly incompatible, 62, 84
strongly normalizing, 12, 44
structurally smaller, 59, 87
subcommutative, 12
subject, 23
subject reduction, 27, 101
substitution, 16, 22
substitution sequence, 80
subterm, 16, 22

tactic, 30, 141
term, 15
Term Rewriting System, 13
term with a hole, 16
top-down approach, 155
TRS, 15
typability, 28, 139
type checking, 28, 139
type theory, 2
type universe, 23
typing rules, 20, 23
  - for λHOL (→), 23
  - for λHOL_τ (→_τ), 88
  - for algebraic data types, 81
  - for function definitions, 87
  - for inductive types, 43
unifiable, 18
uniqueness of types, 97

variable declaration, 23
weakly head-orthogonal, 57
weakly orthogonal, 19
Notation

\[ \rightarrow, 11 \]
\[ \rightarrow_i, 11 \]
\[ \rightarrow_i^*, 11 \]
\[ \rightarrow_i^+, 11 \]
\[ \rightarrow_n, 12 \]
\[ \equiv, 12 \]
\[ \rightarrow_{i,j}, 12 \]
\[ \mathcal{A}^*, 13 \]
\[ \epsilon, 13 \]
\[ \delta, 13 \]
\[ |\bar{d}|, 13 \]
\[ \mathcal{A}^+, 13 \]
\[ \Sigma, 13 \]
\[ \tau(G), 13 \]
\[ s_1 \times \ldots \times s_n \rightarrow s, 13 \]
\[ V, 15 \]
\[ V_i, 15 \]
\[ \mathcal{T}(\Sigma, V), 15 \]
\[ \text{var}(t), 15 \]
\[ \mathcal{R}(\Sigma, V), 15 \]
\[ t^\sigma, 16 \]
\[ C(\Sigma, V), 16 \]
\[ C[t], 16, 22 \]
\[ >_{\text{ipo}}, 17 \]
\[ \text{mgu}(t, u), 18 \]
\[ V, 20 \]
\[ C, 20 \]
\[ \mathbb{T}, 20 \]
\[ \lambda x.t.b, 21 \]
\[ \Pi x.u, 21 \]
\[ BV(t), 21, 36, 78 \]
\[ FV(t), 21, 36, 78 \]
\[ u[x := t], 22, 37, 78 \]
\[ \mathcal{H}, 22 \]
\[ \rightarrow_{\beta}, 22 \]
\[ \mathcal{X}, 23 \]
Universes, 23
Axioms, 23
Rules, 23
\[ \Gamma \vdash a : t, 23 \]
\[ \lambda \text{HOL}, 25 \]
\[ \lambda \text{CC}, 28 \]
\[ \mathcal{T}, 36 \]
\[ \text{Constr}(n, t), 36 \]
\[ \text{Elim}(F, Q, z)(\bar{t}), 36 \]
\[ \text{Ind}(v : \square)(\bar{t}), 36 \]
\[ \mathcal{H}, 37 \]
\[ \rightarrow_t, 39 \]
\[ \mathcal{X}_i, 40 \]
\[ \Gamma \vdash a : t, 43 \]
\[ \lambda \text{HOL}_4, 44 \]
\[ \mathcal{T}(C, V), 56 \]
\[ \text{CS}(D, C, R), 56 \]
\[ >_{\text{lex}}, 59 \]
\[ <_x, 59, 87 \]
\[ \text{PCS}(D, C, R, >), 61 \]
\[ \#_s, 62, 84 \]
\[ \#_{\text{Arg}}, 62, 84 \]
\[ \rightarrow_{c}, 62 \]
\[ t_0, 65 \]
\[ \geq, 65 \]
todo^{\mathcal{F}}(\bar{I}), 68
new(\bar{I}, m), 69
\[ \text{subs}(\bar{I}, m), 69 \]
\[ \text{transform}(D, C, R, >), 73 \]
\[ S, 78 \]
\[ \mathcal{T}, 78 \]
\[ C_x, 78 \]
\[ \mathcal{H}_x, 78 \]
\[ \mathcal{P}, 79 \]

162
\( \mathcal{R} \), 79
\( F; t = \bar{r} \), 79
\( X_r \), 80
fundefs(\( \Gamma \)), 80
consts(\( \Gamma \)), 80
subst(\( t, \sigma \)), 80
\( \Gamma \vdash t \rightarrow_{\pi} u \), 80
\( \Gamma \vdash \pi a : t \), 81, 85, 87
cxtpat(\( \Gamma, p, t \)), 81
cxtlhs(\( \Gamma, t, t \)), 81
\( \Gamma \vdash t \rightarrow_{\pi} u \), 85
\( \lambda \text{HOL}_{\pi} \), 88
\( C_{s,i} \), 103
\( A_{s,i} \), 103
\( B_s \), 103
\( s \), 103
\( \Phi_{s,i} \), 103
\( [t]_t \), 103
\( \mathcal{K} \), 104
\( \mathcal{I} \), 105
Match\( _{\Gamma}(F; k = \bar{r}) \), 109
\( [\Delta]_\Gamma \), 110
\( \Gamma \vdash t \rightarrow_s u \), 111
\( \Gamma \vdash \pi a : t \), 112
\( \lambda \text{HOL}_{\pi} \), 112
\( (>\_s)_{\text{max}} \), 112
Decr\( _{\Gamma}(F; t = \bar{r}) \), 113
RmDecr\( _{\Gamma}(F; t = \bar{r}) \), 113
Repeat(\( t, d, f \)), 114
Repeat\( 't, d, f, n \), 114
NrConstr\( _s(t) \), 114
Transform\( _{\Gamma}(F; k = \bar{r}) \), 116
\( [t]_\Gamma, F; k = \bar{r} \), 116
fundefs\( _{\mathcal{K}}(\Gamma) \), 116
\( \leq_s \), 119
\( \Gamma \vdash t \leadsto u \), 120
\( \kappa^S \), 125
t \( \leq_f u \), 127
\( \Gamma \leq_f \Delta \), 127
\( \rightarrow_{\sigma'} \), 129
\( \Gamma \vdash \pi d a : t \), 134
Samenvatting

Wiskunde kan beoefend worden op verschillende niveaus van abstractie en precisie. Aan de ene kant van het spectrum bevindt zich de informele wiskunde waarin op een hoog abstract niveau wordt geredeneerd. Aan de andere kant bevindt zich de formele wiskunde waarin alles tot in het kleinste detail wordt uitgewerkt. In de formele wiskunde werkt men met een vaste taal, waarvan de uitleg en logische regels precies zijn omschreven. Het is daarom mogelijk om met behulp van een computerprogramma de correctheid van uitleg en logische regels precies zijn omschreven. Een door een computer gecontroleerd bewijs heeft een grote betrouwbaarheid, omdat een computer geen details over het hoofd kan zien. Om het construeren van formele bewijzen te vergemakkelijken, kan men bewijsontwikkelingsystemen gebruiken waarmee men interactief kan werken.

De introductie van de computer heeft de wiskunde sterk beïnvloed. Bepaalde methodes voor het oplossen van wiskundige problemen, die grote berekeningen vereisen, kan men nu uitvoeren met behulp van computerprogramma's. Het kunnen uitvoeren van dit soort berekeningen is een middel om het abstractieniveau te verhogen, omdat een kleine expressie een grote berekening kan presenteren. In functionele programmeertalen kan men functies specifiek maken door middel van een aantal vergelijkingen. Deze vergelijkingen worden opgevat als herschrijvingen door ze van links naar rechts te orienteren. Een rekenstap bestaat uit het vervangen van de linkerpartij van een herschrijving door de rechterpartij.

In dit proefschrift wordt beschreven hoe de kloof tussen informele en formele wiskunde kan worden verkleind met behulp van berekeningen. We beginnen met een formele taal waarin de structuur van de expressies is vastgelegd. Omdat niet alle expressies in deze taal zinvol zijn, hebben we typingsregels die bepalen welke expressies worden opgevat als wiskundige objecten. Deze formele taal is geschikt voor het axiomatiseren van wiskundige theorieën, maar kan geen berekeningen presenteren. We breiden deze taal uit met functiedefinities gebaseerd op herschrijvingen. De berekeningen die door functiedefinities gespecificeerd worden, kunnen binnen de taal uitgevoerd worden en zijn ook toegestaan als redeneerstap. Dus met behulp van functiedefinities kan het abstractieniveau van formele bewijzen verhoogd worden. Behalve berekeningen kan men ook redeneringen met gevalsonderscheiding presenteren als functiedefinities. Hierdoor wordt het mogelijk om bepaalde eigenschappen van functies te bewijzen in de taal zelf.

De eisen waaraan een functiedefinitie moet voldoen garanderen dat alleen functies gedefinieerd kunnen worden waarvan het resultaat voor alle mogelijke argumenten in een eindig aantal stappen berekend kan worden. De berekeningswijze garandeert dat het resultaat van het toepassen van een functie op een argument uniek is. Door deze eigenschappen is het mogelijk om de correctheid van uitleg en logische regels precies zijn omschreven. In de formele wiskunde werkt men met een computerprogramma waarmee functiedefinities interactief kunnen worden geconstrueerd en gecontroleerd.
Curriculum Vitae

Naam: Hugo Johannes Elbers.


Middelbare school: VWO, Canisius College-Mater Dei, Nijmegen, 1982-1988
   Vakken: Duits, Economie 1, Engels, Natuurkunde, Nederlands, Scheikunde,
   Wiskunde A, Wiskunde B.

Universitaire opleiding:


      Klop.
Stellingen

behorende bij het proefschrift

Connecting Informal and Formal Mathematics

van

Hugo Elbers
1. Het gebruik van prioriteitsherschrijfregels voor het definiëren van functies is een goed middel om het abstractie-niveau van formele bewijzen te verhogen. (dit proefschrift)

2. Het formeel specificeren van de taken die door software uitgevoerd moeten worden in een bewijsontwikkelingssysteem, leidt tot een goede documentatie van de software en voorkomt veel fouten.

3. Het formaliseren van wiskundige bewijzen met bewijsontwikkelingssystemen leidt tot bewustwording en grotere kennis van de (meta-theoretische) redeneermethodes die door wiskundigen gehanteerd worden.

4. Het formalisme van Pure Type Systems is een goede basis voor OpenMath, de in ontwikkeling zijnde standaardtaal voor het uitwisselen van wiskundige objecten tussen computerprogramma’s (zie [1]).


6. Zolang het praktisch niet haalbaar is om correcte software te produceren, dient men af te zien van het automatiseren van essentiële processen waarbij storingen grote negatieve gevolgen kunnen hebben (zoals het in gevaar brengen van mensenlevens of het lamleggen van een deel van de economie).

7. Het door Maurice de Hond naar analogie van het woord ‘analfabeet’ ingevoerde begrip ‘digibeet’ (zie [2]), voor mensen die niet met computers kunnen werken, kan beter vervangen worden door het woord ‘adigitalist’.
8. Het invoeren van het stemmen per computer is een typisch voorbeeld van onnodige en ongewenste automatisering, aangezien het voordeel (snel tellen van de stemmen) niet opwiegt tegen de nadelen (de betrouwbaarheid van de uitslag is niet door iedereen te controleren en (stem)computers zijn niet gegarandeerd storingvrij).

9. In het algemeen wordt een roman door één persoon vertaald, om op de eenvoudigste wijze een goed product te krijgen. Dit ondersteunt de stelling dat concurrentie op het spoor beter niet kan worden ingevoerd.

10. Gezien het grote aantal herhalingen van programma’s en sportuitzendingen met verslagen van onbelangrijke wedstrijden, heeft de publieke omroep aan twee netten meer dan genoeg.

Referenties
