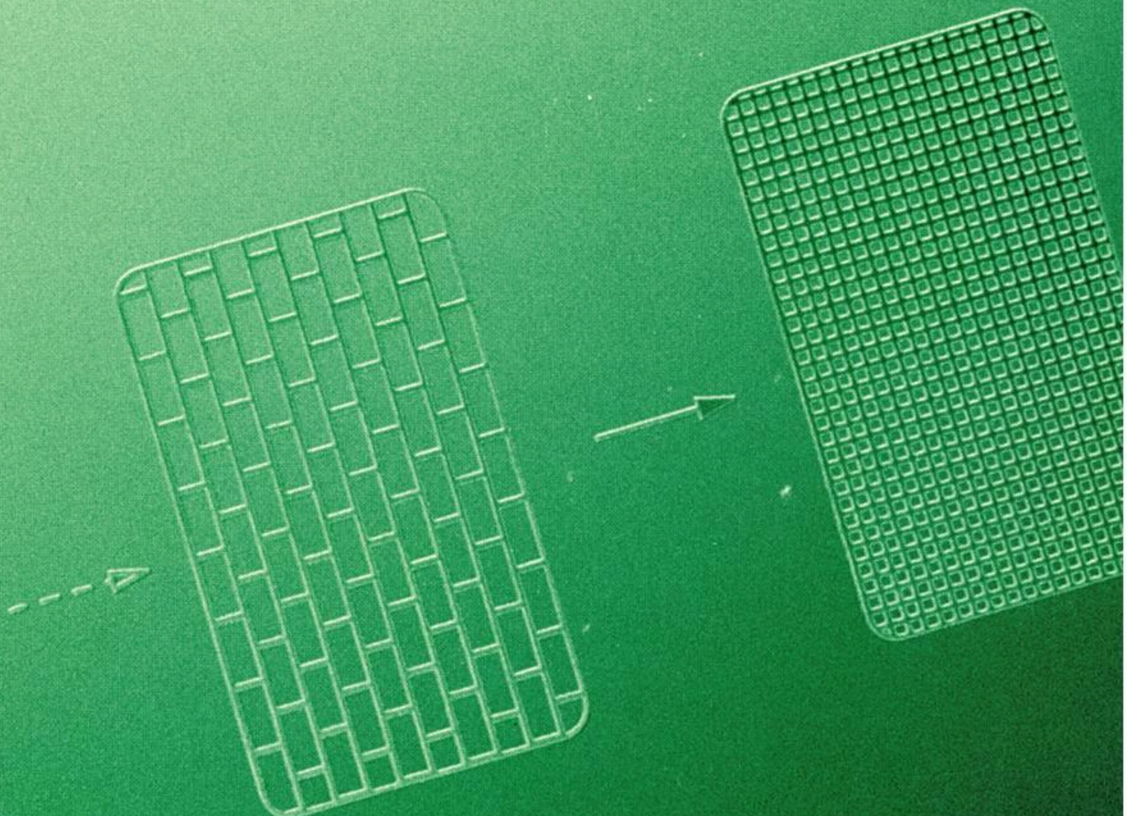
Connecting Informal and Formal Mathematics



Hugo Johannes Elbers

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]).

5. Net zoals het internet het gebruik van computers interessant heeft gemaakt voor de 'gewone man', zal een door iedereen uitbreidbare standaardbibliotheek met door computers geverifieerde wiskunde dit voor wiskundigen kunnen doen.

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 opweegt 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

[1] OpenMath Homepage. URL `http://www.openmath.org`.

[2] Maurice de Hond. Dankzij de snelheid van het licht. Het Spectrum, 1995.

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

Connecting informal and formal mathematics / by Hugo Johannes Elbers.
- Eindhoven : Eindhoven University of Technology, 1998.
Proefschrift. - ISBN 90-386-0611-7
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

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

$$\mathsf{Type} = P \mid \mathsf{Type} \rightarrow \mathsf{Type}, \text{ where } P \text{ is an infinite set of proposition variables.}$$

We represent proofs by terms; they have syntax

$$\mathsf{Term} = V \mid \mathsf{Term}\ \mathsf{Term} \mid \lambda V{:}P.\mathsf{Term}, \text{ where } V \text{ is an infinite set of term variables.}$$

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 $\frac{A_1 \ldots A_n}{C}$ should be interpreted as 'if A_1 and

... and A_n then C'. The following typing rules formalize the notion $\Gamma \vdash_{\rightarrow} p{:}t$, where Γ is a sequence of *assumptions*, and $p \in \mathsf{Term}$, and $t \in \mathsf{Type}$.

(assumption)	$\dfrac{x{:}P\in\Gamma}{\Gamma\vdash_{\rightarrow} x{:}P}$
($\rightarrow$-elimination)	$\dfrac{\Gamma\vdash_{\rightarrow} M{:}P\rightarrow Q \quad \Gamma\vdash_{\rightarrow} N{:}P}{\Gamma\vdash_{\rightarrow} (MN){:}Q}$
($\rightarrow$-introduction)	$\dfrac{\Gamma,x{:}P\vdash_{\rightarrow} M{:}Q}{\Gamma\vdash_{\rightarrow} (\lambda x{:}P.M){:}P\rightarrow Q}$

If $\Gamma \vdash_{\rightarrow} t:p$ we say that t has type p (or t is an *inhabitant* of p) in Γ. 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.

$$\dfrac{\dfrac{x_2 : P_1 \rightarrow P_2 \in \Gamma}{\Gamma \vdash_{\rightarrow} x_2 : P_1 \rightarrow P_2} \quad \dfrac{x_1 : P_1 \in \Gamma}{\Gamma \vdash_{\rightarrow} x_1 : P_1}}{\Gamma \vdash_{\rightarrow} x_2\ x_1 : P_2}$$

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 Γ). 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.

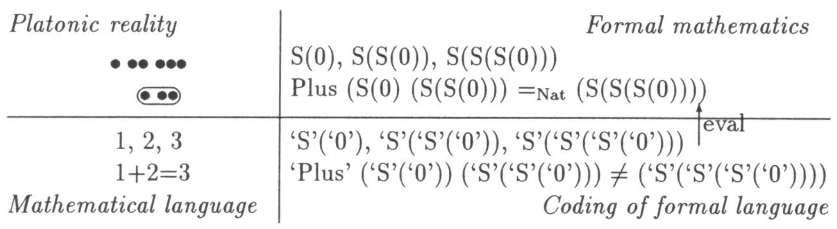

Platonic reality
Formal mathematics
● ●● ●●●
S(0), S(S(0)), S(S(S(0)))
Plus (S(0) (S(S(0))) $=_{\text{Nat}}$ (S(S(S(0))))
eval
1, 2, 3
'S'('0'), 'S'('S'('0')), 'S'('S'('S'('0')))
1+2=3
'Plus' ('S'('0')) ('S'('S'('0'))) $\neq$ ('S'('S'('S'('0'))))
Mathematical language
Coding of formal language

Figure 1.1: Representation of natural numbers in several semantic levels

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

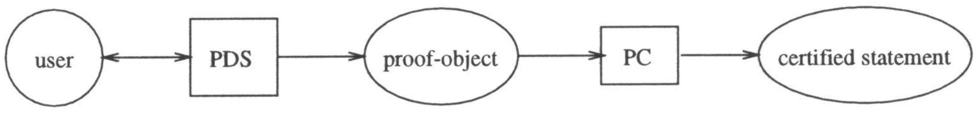


Figure 1.2: Interactive proof development using a Proof Development System

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.

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 `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 **`Fix`** we can define recursive functions. The recursive calls of such a definition should be *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 kth 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 kth 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 **`Half`** defined by the equations `Half(0)=0`, `Half(S(0))=0`, `Half(S(S(n)))=S(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 **`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]).

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 `Leq` with two constructors of type `Leq 0 x` and (`Leq x y`) $\rightarrow$ (`Leq` (`S(x)`) (`S(y)`)) respectively. Now, we can prove $\neg$(`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 $P \rightarrow \perp$, where $\perp$ denotes the false proposition. Thus this function f takes an argument of type `Leq (S(x)) 0` and yields a proof of the false proposition ($\perp$). The definition of f contains no rules, because all possible cases for this argument are treated, as no constructor of `Leq` can yield a term of type `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 `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 λHOL_π, 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 λ-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

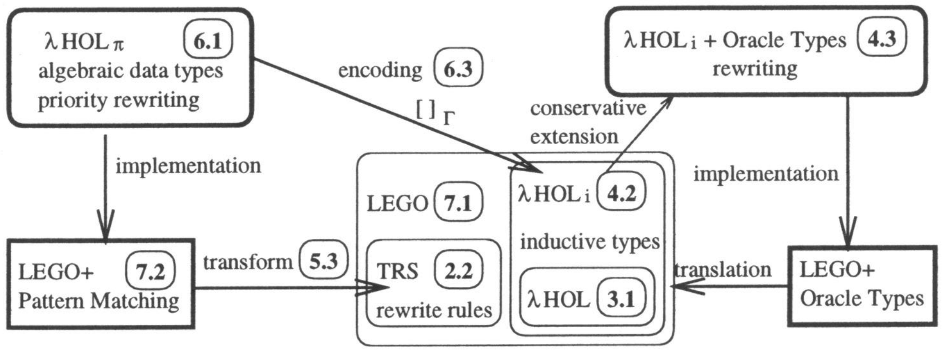


Figure 1.3: Overview of the relations between the systems

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 Reductions 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 $\rightarrow_i^=$. Thus $a \rightarrow_i^= a$, 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 $\twoheadrightarrow_i$. The *equivalence* relation generated by $\rightarrow_i$ is denoted by $=_i$. 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 $a \twoheadrightarrow_i b$, $a \twoheadrightarrow_i c$, there exists an element $d \in A$ such that $b \twoheadrightarrow_i d$ and $c \twoheadrightarrow_i d$.
2. $\rightarrow_i$ *commutes* with $\rightarrow_j$, if for any two reductions $a \twoheadrightarrow_i b$, $a \twoheadrightarrow_j c$, there exists an element $d \in A$ such that $b \twoheadrightarrow_j d$ and $c \twoheadrightarrow_i d$.
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 $\twoheadrightarrow_i \; = \; \twoheadrightarrow_j$ 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 $b \twoheadrightarrow_j d$ 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 $b \twoheadrightarrow a$ 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 ϵ. 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 ith 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 $\langle a_1, \ldots, a_n \rangle$ 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* Σ is a set of function symbols $\mathcal{F}$ together with a typing function $\tau : \mathcal{F} \rightarrow S^+$. We call S the *sorts* of Σ. For ease of notation, we just write $G \in \Sigma$ instead of $G \in \mathcal{F}$.

The typing function τ of a signature Σ 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.

sort	nat,list		
func	`0:`		nat
	`S:`	nat $\rightarrow$	nat
	`Nil:`		list
	`Cons:`	nat $\times$ list $\rightarrow$	list
	`Length:`	list $\rightarrow$	nat

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 {nat, list}, the function symbols are {0, S, Nil, Cons, Length} and we have τ(0)=nat, τ(S)=nat $\rightarrow$ 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 Σ be an S-sorted signature.

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

   We say that Σ is strict if all sorts are strict in Σ.

2. A sort s *depends* on $t \in S$ in Σ, if $s \neq t$, $\tau(F) = s_1 \times \ldots \times s_n \to s$ for some $F \in \Sigma$, and
   (a) $t = s_i$, or
   (b) s_i depends on t in Σ, for some $i \leq n$.

3. We call Σ an *algebraic data type*-signature if Σ 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:

sort	nat, tree, forest		
func	`0`:		nat
	`S`:	nat $\to$	nat
	`Node`:	nat $\times$ forest $\to$	tree
	`Nil`:		forest
	`Cons`:	tree $\times$ forest $\to$	forest

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-indexed) *mapping of I-indexed sets $f : A \to B$* is an I-indexed set of functions $\langle f_i : A_i \to B_i \mid i \in I \rangle$.

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 ${}^{s}x$ 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 $\mathcal{T}(\Sigma, V)$ of *terms* over an S-sorted signature Σ is the smallest set satisfying the following clauses:

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

For $t \in \mathcal{T}(\Sigma, V)$, 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 $t \in \mathcal{T}(\Sigma)$.

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(S(0))`, `Length(`${}^{\text{list}}y$`)` and `Cons(S(`${}^{\text{nat}}x$`),Nil)`.
The expressions `Nil(0)`, `S(Nil)` and `Cons(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 $\mathcal{R}(\Sigma, V)$ of *rewrite rules* over an S-sorted signature Σ is the smallest set such that: if $t_1, t_2 \in \mathcal{T}(\Sigma, V)_s$, and $t_1 \notin V$ and $\text{var}(t_2) \subseteq \text{var}(t_1)$ then $(t_1, t_2) \in \mathcal{R}(\Sigma, V)_s$. We will denote a rewrite rule $(t_1, t_2) \in \mathcal{R}(\Sigma, V)$ 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:

```
Length(Nil) → 0
Length(Cons(x,y)) → S(Length(y))
```

Notice that `S(Length(y))` $\rightarrow$ `Length(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 Σ and an S-indexed set of rewrite rules over Σ.

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 Σ be an S-sorted signature.

1. A *substitution* σ is a mapping of S-indexed sets $\sigma : V \to \mathcal{T}(\Sigma, V)$
2. Instantiating a term t with substitution σ, notation t^σ, 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 $t_i \in \mathcal{T}(\Sigma, V)$.
3. The *composition* of two substitutions σ and τ, notation $\tau \circ \sigma$, is defined as $(\tau \circ \sigma)(x) = (\sigma(x))^\tau$, for $x \in V$.

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

`Cons(x,y)`$^\sigma$=`Cons(0,Nil)`
`S(Length(y))`$^\sigma$=`S(Length(Nil))`

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

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

1. ${}^s[\ ] \in \mathcal{C}(\Sigma, V)_{s,s}$, for all $s \in S$.
2. $F(t_1, \ldots, t_{i-1}, C[\ ], t_{i+1}, \ldots, t_n) \in \mathcal{C}(\Sigma, V)_{s_0,s_{n+1}}$, for all $C[\ ] \in \mathcal{C}(\Sigma, V)_{s_0,s_i}$, all $F \in \Sigma$ with $\tau(F) = s_1 \times \ldots \times s_n \to s_{n+1}$, and all $t_j \in \mathcal{T}(\Sigma, V)_{s_j}$ $(1 \le i, j \le n, i \ne j)$.

A term with a hole $C[\ ] \in \mathcal{C}(\Sigma, V)_{s_1,s_2}$ is said to have *arity* s_1 and *sort* s_2. The result of replacing the hole in $C[\ ]$ with a term $t \in \mathcal{T}(\Sigma, V)_{s_1}$, denotation $C[t]$, is a term in $\mathcal{T}(\Sigma, V)_{s_2}$. We say that t is a *subterm* of $C[t]$. If $C[\ ] \not\equiv [\ ]$, 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: ${}^{\text{nat}}[\ ]$, `S(S(`${}^{\text{nat}}[\ ]$`))`, and `Length(Cons(0,`${}^{\text{list}}[\ ]$`))`.
We have ${}^{\text{nat}}[$`0`$]$ = `0`, and `Cons(x,`${}^{\text{list}}[$`Cons(S(0),Nil)`$]$`)`= `Cons(x,Cons(S(0),Nil))`.
The expressions `0`, `Cons(`${}^{\text{nat}}[\ ]$,${}^{\text{list}}[\ ]$`)`, and `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 $r : t_1 \to t_2 \in \mathcal{R}(\Sigma, V)_{s_1}$ be a rewrite rule. Let $\sigma : V \to \mathcal{T}(\Sigma, V)$ be a substitution, and $C[\ ] \in \mathcal{C}(\Sigma, V)_{s_1,s_2}$ a context. Then $C[t_1^\sigma] \to_r C[t_2^\sigma]$ is a *reduction step* of *sort* s_2. We say $C[t_1^\sigma]$ is a *redex*, and t_1^σ is a *redex occurrence in* $C[t_1^\sigma]$. We call $\to_r$ the *one-step reduction relation* generated by r.

Example 2.2.19 Using the rewrite rules for `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 $\langle \mathcal{T}(\Sigma, V), (\rightarrow_r)_{r \in R} \rangle$. 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 $\rhd$ be a strict partial order on Σ (thus $G \not\rhd G$ for $G \in \Sigma$). The relation $>_{\text{lpo}}$ on terms is defined as follows:

1. $F(t_1, \ldots, t_n) >_{\text{lpo}} t_i$, for $i \leq n$.
2. $F(t_1, \ldots, t_n) >_{\text{lpo}} G(u_1, \ldots, u_m)$, if $F \rhd G \wedge \forall i \leq m \; F(t_1, \ldots, t_n) >_{\text{lpo}} u_i$.
3. $F(t_1, \ldots, t_n) >_{\text{lpo}} F(t_1, \ldots, t_{i-1}, u_i, \ldots, u_n)$, if $t_i >_{\text{lpo}} u_i \wedge \forall j > i \; F(t_1, \ldots, t_n) >_{\text{lpo}} u_j$.
4. $t_1 >_{\text{lpo}} t_3$, if $t_1 >_{\text{lpo}} t_2 \wedge t_2 >_{\text{lpo}} t_3$.

Example 2.2.22 Let Σ be the signature of Example 2.2.4. We can define a strict partial order $\rhd$ on Σ by: `Length` $\rhd$ `S` and `Length` $\rhd$ `0`. Now we have `S(0)` $>_{\text{lpo}}$ `0`, and `Length(Nil)` $>_{\text{lpo}}$ `0`, and `Length(Cons(x,y))` $>_{\text{lpo}}$ `S(Length(y))`.
We do not have `S(x)` $>_{\text{lpo}}$ `0`, because `S` and `0` are not ordered.

Theorem 2.2.23 *Let (Σ, R) be an S-sorted Term Rewriting System. Assume $\rhd$ is a well-founded strict partial order on Σ. If $l >_{\text{lpo}} r$, for every rule $l \rightarrow r \in R$, 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 × 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 Σ 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 σ is a *unifier* for t and u. A substitution σ is a *most general unifier* for t and u, notation $\sigma = \text{mgu}(t, u)$, if σ is a unifier for t and u, and for all unifiers τ for t and u there exists a substitution σ' with $\sigma' \circ \sigma = \tau$.

If two terms t and u are unifiable, then there exists a σ such that σ=mgu(t, u); notice that σ 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 σ, such that $l_2 = C[t]$ and $\sigma = \text{mgu}(t, l_1)$. The term l_2^σ can be reduced in two possible ways: $C[t]^\sigma \rightarrow C[r_1]^\sigma$ and $l_2^\sigma \rightarrow r_2^\sigma$. Now the pair of reducts $\langle C[r_1]^\sigma, r_2^\sigma \rangle$ 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 $\langle t_1, t_2 \rangle$ 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 $\langle$`False`, `True`$\rangle$. 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:

```
sort  nat
func  0:                   nat
      S:            nat →  nat
      Plus:   nat × nat →  nat
rule  Plus(0,x)          → x
      Plus(y,0)          → y
      Plus(S(x),S(y))    → S(S(Plus(x,y)))
```

This TRS has one critical pair $\langle \texttt{0}, \texttt{0} \rangle$.

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

Definition 2.2.30 Let (Σ, R) be an S-sorted Term Rewriting System. (Σ, R) is *weakly orthogonal* if (Σ, 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 (λHOL), 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 $x \mapsto f_x$ to an argument a yielding f_a. Mathematical theories can be represented in λHOL 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 $\mathcal{C}$ an infinite set of constants. The sets V and $\mathcal{C}$ are disjoint. The set of *pseudo terms* $\mathcal{T}$ is defined as follows:

$$\mathcal{T} = V \mid \mathcal{C} \mid (\mathcal{T}\mathcal{T}) \mid \lambda V{:}\mathcal{T}.\mathcal{T} \mid \Pi V{:}\mathcal{T}.\mathcal{T}$$

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 $t \to u$ instead of $\Pi x : t.u$. We will use the convention that application associates to the left and $\to$ associates to the right; thus we may write *Plus x Zero* instead of ((*Plus* x) *Zero*), and $Nat \to Nat \to Bool$ instead of $Nat \to (Nat \to Bool)$. 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 β-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 $C \in \mathcal{C}$.
(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 $C \in \mathcal{C}$.
(c) $FV(fa) = FV(f) \cup FV(a)$.
(d) $FV(\lambda x{:}t.b) = FV(t) \cup (FV(b) \backslash \{x\})$.
(e) $FV(\Pi x{:}t.b) = FV(t) \cup (FV(b) \backslash \{x\})$.

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. $y[x := t] = \begin{cases} t & \text{(if } x = y) \\ y & \text{(otherwise)} \end{cases}$ for $y \in V$.
2. $C[x := t] = C$, for $C \in \mathcal{C}$.
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 $(\lambda y{:}Nat.Plus\ x\ y)[x := O] = \lambda y{:}Nat.Plus\ O\ y$, and $(\lambda y{:}Nat.Plus\ x\ y)[y := O] = \lambda y{:}Nat.Plus\ x\ y$.

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:

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

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 $y \notin FV(t_2)$.

Definition 3.1.10 The notion of β-reduction is defined as follows:

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

Notice that because of the variable convention we have $\mathrm{FV}((\lambda v{:}t.b)a) \cap \mathrm{BV}((\lambda v{:}t.b)a) = \emptyset$. As $\mathrm{FV}(a) \subseteq \mathrm{FV}((\lambda v{:}t.b)a)$, and $\mathrm{BV}(b) \subseteq \mathrm{BV}((\lambda v{:}t.b)a)$, 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 $S((\lambda x{:}Nat.Plus\ x\ x)\ (S\ O)) \rightarrow_\beta S(Plus\ (S\ O)\ (S\ O))$.

Not all pseudo terms represent mathematical objects. For instance, *Eq* $(S\ O)$ *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 $a, t \in \mathcal{T}$. 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 $\mathcal{C}$ that contains the constants that will be used to represent type universes. Usually the set Universes is called Sorts (or $\mathcal{S}$), 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 $\Box$, and the constant $\triangle$ is the type of $\Box$. 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 $(\Box, \triangle) \in$ Axioms indicates that $\Box$ has type $\triangle$. A triple $(s_1, s_2, s_3) \in$ 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:

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

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

(axiom)	$\epsilon\vdash C:s$	if $(C,s)\in$ Axioms
(var-start)	$\dfrac{\Gamma\vdash t{:}s}{\Gamma,x{:}t\vdash x{:}t}$	if $x\in V_s, x\notin FV(\Gamma)$
(var-weak)	$\dfrac{\Gamma\vdash a{:}t \quad \Gamma\vdash u{:}s}{\Gamma,x{:}u\vdash a{:}t}$	if $x\in V_s, x\notin FV(\Gamma)$
(product)	$\dfrac{\Gamma\vdash t{:}s_1 \quad \Gamma,x{:}t\vdash u{:}s_2}{\Gamma\vdash(\Pi x{:}t.u){:}s_3}$	if $(s_1,s_2,s_3)\in$ Rules
(abstraction)	$\dfrac{\Gamma,x{:}t\vdash b{:}u \quad \Gamma\vdash(\Pi x{:}t.u){:}s}{\Gamma\vdash(\lambda x{:}t.b){:}(\Pi x{:}t.u)}$	
(application)	$\dfrac{\Gamma\vdash f{:}(\Pi x{:}t.u) \quad \Gamma\vdash a{:}t}{\Gamma\vdash(fa){:}u[x{:=}a]}$	
(β-conversion)	$\dfrac{\Gamma\vdash a{:}t \quad \Gamma\vdash t'{:}s \quad t{=}_\beta t'}{\Gamma\vdash a{:}t'}$	

In the rules above we use the following notation:
$C\in\mathcal{C}$ $\quad s,s_1,s_2,s_3\in$ Universes $\quad x\in V$ $\quad \Gamma\in\mathcal{X}$ $\quad a,b,f,t,t',u\in\mathcal{T}$
Recall from Section 2.1 that $=_\beta$, used in the rule (β-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:

$$\cfrac{\epsilon\vdash *:\square \qquad \cfrac{\epsilon\vdash *:\square}{p:*\vdash p:*}}{\epsilon\vdash(\Pi p{:}*.p):*}$$

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 $\Gamma\vdash t:\triangle$ then $t=\square$.*
2. *We do not have '$\Gamma\vdash t:\square \Rightarrow t=*$'. For instance, we have $\epsilon\vdash *\rightarrow *:\square$.*
3. *Let $s\in$ Universes and $v\in V_s$. If $\Gamma\vdash v:t$ then $\Gamma\vdash t:s$.*

Now we can combine the notions of pseudo term, pseudo context, β-reduction, and type assignment to define the language of 'Higher Order Logic'.

Definition 3.1.18 *Higher Order Logic* is the tuple $(\mathcal{T},\mathcal{X},\rightarrow_\beta,\vdash)$.

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 $a \in \mathcal{T}$ is *legal* if we can derive $\Gamma \vdash a : t$ or $\Gamma \vdash t : a$, for some pseudo context Γ 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 λHOL 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 λHOL. 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 $p \to q$ 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 $\Pi p{:}*.p$ 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 λHOL by $\Pi p{:}*.p \to p$. This trivial proposition has a proof $\lambda p{:}*.\lambda x{:}p.x$, as we can derive $\epsilon \vdash (\lambda p{:}*.\lambda x{:}p.x) : (\Pi p{:}*.p \to p)$.

We can represent the logical connectives $\neg$, $\wedge$, and $\vee$ as:

$$\boxed{\begin{array}{ll} \neg p & = p \to (\Pi r{:}*.r) \\ p \wedge q & = \Pi r{:}*.(p \to q \to r) \to r \\ p \vee q & = \Pi r{:}*.(p \to r) \to (q \to r) \to r \end{array}}$$

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 $p \wedge q$. 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 $p \wedge q$ as predicate.

$$\dfrac{\dfrac{\dfrac{\dots}{\Gamma\vdash i:p\wedge q}\quad\dfrac{\dots}{\Gamma\vdash p:*}}{\Gamma\vdash i\ p:(p\to q\to p)\to p}\qquad\dfrac{\dfrac{\dots}{\Gamma,x:p\vdash(\lambda y{:}q.x):(q\to p)}\quad\dfrac{\dots}{\Gamma\vdash(p\to q\to p):*}}{\Gamma\vdash(\lambda x{:}p.\lambda y{:}q.x):(p\to q\to p)}}{\Gamma\vdash(i\ p\ (\lambda x{:}p.\lambda y{:}q.x)):p}$$

Figure 3.1: Derivation of p from the assumption $p\wedge q$ in context $\Gamma = p:*, q:*, i:p\wedge q$

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:

$$\boxed{\begin{array}{ll}\forall x\in T.p_x & = \Pi x{:}T.p_x\\ \exists x\in T.p_x & = \Pi r{:}*.(\Pi x{:}T.p_x\to r)\to r\\ x =_{\mathrm{T}} y & = \Pi p{:}(T\to *).(p\ x)\to(p\ y)\end{array}}$$

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

$$\begin{array}{l}T:\Box\vdash\lambda x{:}T.\lambda y{:}T.\lambda i{:}(x =_{\mathrm{T}} y).(i\ (\lambda z{:}T.(z =_{\mathrm{T}} x))\ (\lambda p{:}T\to *.\lambda j{:}(p\ x).j)):\\ \qquad(\forall x,y\in T.(x =_{\mathrm{T}} y)\to(y =_{\mathrm{T}} x))\end{array}$$

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

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:

$$\Delta_{\mathrm{Mon}} = \boxed{\begin{array}{ll}M: & \Box,\\ op: & M\to M\to M,\\ e: & M,\\ assoc: & \forall x,y,z\in M.op\ x\ (op\ y\ z) =_{\mathrm{M}} op\ (op\ x\ y)\ z,\\ neutL: & \forall x\in M.op\ e\ x =_{\mathrm{M}} x,\\ neutR: & \forall x\in M.op\ x\ e =_{\mathrm{M}} x\end{array}}$$

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 Δ_{Mon} by:

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

It is inhabited by the term:

$$\begin{aligned}&\lambda f{:}M.\lambda l{:}(\forall x \in M.op\ f\ x =_{\text{M}} x).\lambda r{:}(\forall x \in M.op\ x\ f =_{\text{M}} x).\\ &=_{\text{trans}} f\ (op\ f\ e)\ e\ (=_{\text{sym}} (op\ f\ e)\ f\ (neutR\ f))\ (l\ e)\end{aligned}$$

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

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 λHOL.

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 λHOL 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	$= \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box), (*, \Box, \Box)\}$

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]. $\Box$

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* $a \twoheadrightarrow_\beta a'$ *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 *λHOL is strongly normalizing.*

Proof
The Calculus of Constructions (λCC) is strongly normalizing (see [4]). We can embed λHOL in λCC via a reduction preserving transformation (see [16]). This transformation is done via a mapping E that is defined as follows. Let $t \in \mathcal{T}$. Then $E(t)$ denotes the pseudo term obtained from t by replacing each $\square$ with $*$, and each $\triangle$ with $\square$. Let $\Gamma \in \mathcal{X}$ be a pseudo context. Then $E(\Gamma)$ denotes the pseudo context obtained from Γ by replacing each predicate p by $E(p)$. If $\Gamma \vdash a : t$ in λHOL, then we have $E(\Gamma) \vdash E(a) : t'$ in λCC, where $t' = \begin{cases} \square & (\text{if } a = t_1 \rightarrow \ldots \rightarrow t_n \rightarrow * \ (0 \leq n)) \\ E(t) & (\text{otherwise}) \end{cases}$.
As λCC is strongly normalizing, the system λHOL 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 Γ, a, t. Does $\Gamma \vdash a : t$ hold?
2. *Typability* is the problem: Given Γ, a. Is there a t such that $\Gamma \vdash a : t$ holds?
3. *Inhabitation* is the problem: Given Γ, 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 λHOL.

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 λHOL. The provability of a property p (in t) corresponds to the inhabitation of $[p]$ (in $[t]$). As type checking is decidable in λHOL, 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 $F_{\mathrm{P}}(x)$ 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 λHOL (see Example 3.2.2):

$$\epsilon \vdash ? : (\Pi p{:}*.p \rightarrow p)$$

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:

$$\epsilon \vdash (\lambda p{:}*.\lambda h{:}p.h) : (\Pi p{:}*.p \rightarrow p)$$

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:

```
sort  M
func  e:                M
      ·:     M × M  →   M
rule  e·x           →  x
      x·e           →  x
      x·(y·z)       →  (x·y)·z
```

For instance, `(e·x)·(y·z)` and `(x·y)·(z·e)` have the same normal form `(x·y)·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:

$$\Delta_{\mathrm{N}} = \boxed{\begin{array}{ll} N: & \square, \\ 0: & N, \\ S: & N \to N, \\ \mathit{Snot0}: & \forall x \in N.\neg(S(x) =_{\mathrm{N}} 0), \\ \mathit{Sinjective}: & \forall x, y \in N.S(x) =_{\mathrm{N}} S(y) \to x =_{\mathrm{N}} y, \\ \mathit{induction}: & \forall P \in N \to *.(P0) \to (\forall n \in N.(Pn) \to (P(Sn))) \to \forall n \in N.Pn \end{array}}$$

Notice that $=_N$ 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:

$$\Delta_{\text{plus}} = \boxed{\begin{array}{ll} \mathit{plus}: & N \to N \to N, \\ \mathit{plus0}: & \forall y \in N.\mathit{plus}\ 0\ y =_N y, \\ \mathit{plusS}: & \forall x, y \in N.\mathit{plus}\ (Sx)\ y =_N S(\mathit{plus}\ x\ y) \end{array}}$$

The type N with binary operator *plus* and neutral element 0 is a monoid.

In the TRS of Example 4.1.2 we have the following reduction sequence:
$\mathtt{(e{\cdot}x){\cdot}(y{\cdot}z)} \to \mathtt{x{\cdot}(y{\cdot}z)} \to \mathtt{(x{\cdot}y){\cdot}z}$. This reduction sequence can be interpreted as the equation $\forall x, y, z \in N.\mathit{plus}\ (\mathit{plus}\ 0\ x)\ (\mathit{plus}\ y\ z) =_N \mathit{plus}\ (\mathit{plus}\ x\ y)\ z$. 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:

$$\begin{array}{rcll} \forall x \in N.\mathit{plus}\ 0\ x & =_N & x & \text{(plus0)} \\ \forall x \in N.\mathit{plus}\ x\ 0 & =_N & x & \text{(0identR)} \\ \forall x, y, z \in N.\mathit{plus}\ x\ (\mathit{plus}\ y\ z) & =_N & \mathit{plus}\ (\mathit{plus}\ x\ y)\ z & \text{(plus-assoc)} \end{array}$$

The names behind these equations refer to formal proofs of these equations. The first equation is proved by the axiom *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
$$\forall x_1, y_1, x_2, y_2 \in N.(x_1 =_N y_1) \to (x_2 =_N y_2) \to (\mathit{plus}\ x_1\ x_2 =_N \mathit{plus}\ y_1\ y_2) \qquad \text{(plus-resp)}$$

A *trace* of a reduction sequence $\mathtt{t}_1 \to \mathtt{t}_2 \to \ldots \to \mathtt{t}_n$ 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 $\mathtt{(e{\cdot}x){\cdot}(y{\cdot}z)} \to \mathtt{x{\cdot}(y{\cdot}z)}$ uses the first rule of `M`. Using (plus0) and (plus-resp) we obtain a proof of

$$plus\ (plus\ 0\ x)\ (plus\ y\ z) =_{\mathrm{N}} plus\ x\ (plus\ y\ z)$$

In a similar way the reduction step $\mathbf{x}\cdot(\mathbf{y}\cdot\mathbf{z}) \to (\mathbf{x}\cdot\mathbf{y})\cdot\mathbf{z}$ induces a proof of

$$plus\ x\ (plus\ y\ z) =_{\mathrm{N}} plus\ (plus\ x\ y)\ z$$

Using the transitivity of $=_{\mathrm{N}}$ we can link the obtained proofs to construct a proof of

$$plus\ (plus\ 0\ x)\ (plus\ y\ z) =_{\mathrm{N}} plus\ (plus\ x\ y)\ z$$

In a similar way we can construct a proof of

$$plus\ (plus\ x\ y)\ (plus\ z\ 0) =_{\mathrm{N}} plus\ (plus\ x\ y)\ z$$

We can combine the obtained proofs to construct the proof

$=_{\mathrm{N}}$ -trans (*plus* (*plus* 0 x) (*plus* y z)) (*plus* (*plus* x y) z) (*plus* (*plus* x y) (*plus* z 0))
($=_{\mathrm{N}}$ -trans (*plus* (*plus* 0 x) (*plus* y z)) (*plus* x (*plus* y z)) (*plus* (*plus* x y) z) (*plus-resp*
(*plus* 0 x) x (*plus* y z) (*plus* y z) (*plus0* x) ($=_{\mathrm{N}}$ -refl (*plus* y z))) (*plus-assoc* x y z))
($=_{\mathrm{N}}$ -sym (*plus* (*plus* x y) (*plus* z 0)) (*plus* (*plus* x y) z) (*plus-resp* (*plus* x y) (*plus* x y)
(*plus* z 0) z ($=_{\mathrm{N}}$ -refl (*plus* x y)) (*0identR* z)))

of the equality

$$plus\ (plus\ 0\ x)\ (plus\ y\ z) =_{\mathrm{N}} plus\ (plus\ x\ y)\ (plus\ z\ 0)$$

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 $\mathcal{O}(|t_1|)$ (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 $\mathcal{O}(|t_2|)$ (if the costs for substitution are linear in the number of symbols of the obtained term). Thus the computation of a reduction step $t_1 \to t_2$ can be done in time $\mathcal{O}(|t_1| + |t_2|)$. 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 $t_1 \to t_2$ are $\mathcal{O}(|t_1|^2 + |t_2|^2)$. 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
$f_1(\ldots(f_n(l^\sigma))) \to f_1(\ldots(f_n(r^\sigma)))$ requires a proof of $l^\sigma = r^\sigma$, which has length $\mathcal{O}(|l^\sigma|+|r^\sigma|)$,

and n compatibility proofs of the form 'if $f_{i+1}(\ldots(f_n(l^\sigma))) = f_{i+1}(\ldots((f_n(r^\sigma))))$ then $f_i(f_{i+1}(\ldots(f_n(l^\sigma)))) = f_i(f_{i+1}(\ldots((f_n(r^\sigma)))))$'. The ith compatibility proof adds length $1 + |l^\sigma| + (n-i) + |r^\sigma| + (n-i)$ to the proof. Thus the length of the compatibility part of the proof is $\Sigma_{i=1}^n |l^\sigma| + |r^\sigma| + 2 * (n-i) + 1 = n * (|l^\sigma| + |r^\sigma| + n))$. Therefore the total length of the proof has length $\mathcal{O}(|f_1(\ldots(f_n(l^\sigma)))|^2 + ||f_1(\ldots(f_n(r^\sigma)))|^2)$. 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 $t_1 \to t_2 \to \ldots \to t_n$ costs $\mathcal{O}(\Sigma_{i=1}^n |t_i|^2)$ (the verification of the applications of the transitivity rule costs $\mathcal{O}(\Sigma_{i=1}^n |t_i|))$. 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+y` $\to$ `y+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: $[x, [y, z]] + [y, [z, x]] + [z, [x, y]] = 0$. 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.

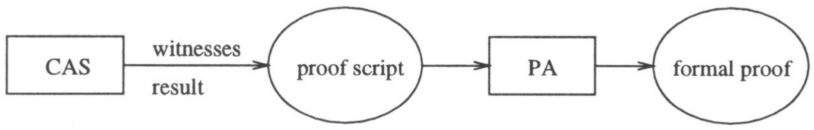


Figure 4.1: Using a Computer Algebra System as a guide for a Proof Assistant

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 $\text{gcd}(m, n) = g$. 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, $\text{gcd}(36, 56) = 4$ 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 Ind, Constr, 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* $\mathcal{T}_{\rm i}$ is defined as follows:

$$\mathcal{T}_{\rm i} = V \mid \mathcal{C} \mid (\mathcal{T}_{\rm i}\ \mathcal{T}_{\rm i}) \mid \lambda V{:}\mathcal{T}_{\rm i}.\mathcal{T}_{\rm i} \mid \Pi V{:}\mathcal{T}_{\rm i}.\mathcal{T}_{\rm i} \mid \mathsf{Ind}(V{:}\mathcal{T}_{\rm i})\{\mathcal{T}_{\rm i}^*\} \mid \mathsf{Constr}(\mathbb{N}, \mathcal{T}_{\rm i}) \mid \mathsf{Elim}(\mathcal{T}_{\rm i}, \mathcal{T}_{\rm i}, \mathcal{T}_{\rm i})\{\mathcal{T}_{\rm i}^*\}$$

Recall that $\mathcal{T}_{\rm i}^*$ denotes the set of sequences of elements in $\mathcal{T}_{\rm i}$.

A pseudo term of the form $\mathsf{Ind}(x{:}u)\{t_1, \ldots, t_n\}$ represents a type with n constructors; the type of the ith 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 $\Box$ 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 $\mathsf{Ind}(x{:}\Box)\{t_1, \ldots, t_n\}$.

The pseudo term $\mathsf{Constr}(i, I_0)$ represents the ith 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 $\mathsf{Constr}(i, I_0)\vec{a}$. A pseudo term of the form $\mathsf{Elim}(I_0, Q, b)\{f_1, \ldots, f_n\}$ 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 $\mathsf{Constr}(i, I_0)a_1 \ldots a_{m_i}$, 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 $\mathcal{T}$ for pseudo terms in $\mathcal{T}_{\rm i}$.

Definition 4.2.3 We define bound and free variables and substitution for pseudo terms ($\in \mathcal{T}_{\rm i}$) 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:
   (a) $BV(\mathsf{Ind}(x{:}u)\{t_1, \ldots, t_n\}) = \{x\} \cup BV(u) \cup BV(t_1) \cup \ldots \cup BV(t_n)$.
   (b) $BV(\mathsf{Constr}(i, t)) = BV(t)$.
   (c) $BV(\mathsf{Elim}(I, Q, a)\{t_1, \ldots, t_n\}) = BV(I) \cup BV(Q) \cup BV(a) \cup BV(t_1) \cup \ldots \cup BV(t_n)$.
2. The new rules for the set of free variables of a pseudo term are:
   (a) $FV(\mathsf{Ind}(x{:}u)\{t_1, \ldots, t_n\}) = FV(u) \cup ((FV(t_1) \cup \ldots \cup FV(t_n)) \setminus \{x\})$.
   (b) $FV(\mathsf{Constr}(i, t)) = FV(t)$.
   (c) $FV(\mathsf{Elim}(I, Q, a)\{t_1, \ldots, t_n\}) = FV(I) \cup FV(Q) \cup FV(a) \cup FV(t_1) \cup \ldots \cup (t_n)$.

3. The new rules for substitution are as follows:

   (a) $\mathsf{Ind}(y{:}u)\{t_1,\ldots,t_n\}[x:=b] =$
   $$\begin{cases} \mathsf{Ind}(y{:}u)\{t_1,\ldots,t_n\} & \text{(if } x=y\text{)} \\ \mathsf{Ind}(y{:}u[x:=b])\{t_1[x:=b],\ldots,t_n[x:=b]\} & \text{(otherwise)} \end{cases}$$

   (b) $\mathsf{Constr}(i,t)[x:=b] = \mathsf{Constr}(i,t[x:=b])$

   (c) $\mathsf{Elim}(I,Q,a)\{t_1,\ldots,t_n\}[x:=b] =$
   $\mathsf{Elim}(I[x:=b],Q[x:=b],a[x:=b])\{t_1[x:=b],\ldots,t_n[x:=b]\}$.

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 $\mathcal{H}_{\mathrm{i}}$ of *pseudo terms with a hole* as follows:

$$\mathcal{H}_{\mathrm{i}} = [\,] \mid \mathcal{H}_{\mathrm{i}}\ \mathcal{T} \mid \mathcal{T}\,\mathcal{H}_{\mathrm{i}} \mid \lambda V{:}\mathcal{H}_{\mathrm{i}}.\mathcal{T} \mid \lambda V{:}\mathcal{T}.\mathcal{H}_{\mathrm{i}} \mid \Pi V{:}\mathcal{H}_{\mathrm{i}}.\mathcal{T} \mid \Pi V{:}\mathcal{T}.\mathcal{H}_{\mathrm{i}} \mid \mathsf{Ind}(V{:}\mathcal{H}_{\mathrm{i}})\{\mathcal{T}_{\mathrm{i}}^*\} \mid \mathsf{Ind}(V{:}\mathcal{T}_{\mathrm{i}})\{\mathcal{T}_{\mathrm{i}}^*\mathcal{H}_{\mathrm{i}}\mathcal{T}_{\mathrm{i}}^*\} \mid \mathsf{Constr}(\mathbb{N},\mathcal{H}_{\mathrm{i}}) \mid \mathsf{Elim}(\mathcal{H}_{\mathrm{i}},\mathcal{T}_{\mathrm{i}},\mathcal{T}_{\mathrm{i}})\{\mathcal{T}_{\mathrm{i}}^*\} \mid \mathsf{Elim}(\mathcal{T}_{\mathrm{i}},\mathcal{H}_{\mathrm{i}},\mathcal{T}_{\mathrm{i}})\{\mathcal{T}_{\mathrm{i}}^*\} \mid \mathsf{Elim}(\mathcal{T}_{\mathrm{i}},\mathcal{T}_{\mathrm{i}},\mathcal{H}_{\mathrm{i}})\{\mathcal{T}_{\mathrm{i}}^*\} \mid \mathsf{Elim}(\mathcal{T}_{\mathrm{i}},\mathcal{T}_{\mathrm{i}},\mathcal{T}_{\mathrm{i}})\{\mathcal{T}_{\mathrm{i}}^*\mathcal{H}_{\mathrm{i}}\mathcal{T}_{\mathrm{i}}^*\}$$

Definition 4.2.5 The notion of β-reduction for pseudo terms ($\in \mathcal{T}_{\mathrm{i}}$) is defined as follows:

$$C[(\lambda v{:}t.b)a] \to_\beta C[b[v:=a]], \text{ for } C[\,] \in \mathcal{H}_{\mathrm{i}}.$$

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

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

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

1. $t = x$; or

2. $t = u \to c$, and $\mathrm{constr}_x(c)$, and either

   (a) $u = x$, or

(b) $\mathrm{FV}(u) = \emptyset$, $* \notin u$.

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 $x \to x$, and $\mathsf{Ind}(y{:}\Box)\{y, y\} \to x$ are types of constructor in x. If $y \in V_\triangle$ and $x \neq y$ then $y \to x$ is *not* a type of constructor in x.

A *(pseudo) inductive type* is a pseudo term of the form $\mathsf{Ind}(x:\Box)\{t_1, \ldots, t_n\}$ such that $\mathrm{constr}_x(t_i)$ 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 $\mathsf{Constr}(i, I_0)$ 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 **Nat**.

Example 4.2.10 We define an inductive type **Int** as follows:

$$\mathbf{Int} = \mathsf{Ind}(y{:}\Box)\{\mathbf{Nat} \to y, y, \mathbf{Nat} \to y\}$$

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

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

We will show how we can represent functions on inductive types using Elim.

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

We want a pseudo term f such that:

$$\begin{array}{rcl} f(\mathbf{Neg}\ n) & \twoheadrightarrow & \mathbf{Pos}\ n \\ f(\mathbf{Zero}) & \twoheadrightarrow & \mathbf{Zero} \\ f(\mathbf{Pos}\ n) & \twoheadrightarrow & \mathbf{Neg}\ n \end{array}$$

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

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

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

The desired reduction behaviour of *Negation* is:

$$\begin{array}{rclcl} \mathit{Negation}(\mathbf{Neg}\ n) & \twoheadrightarrow & f_1\ n & \to_\beta & \mathbf{Pos}\ n \\ \mathit{Negation}(\mathbf{Zero}) & \twoheadrightarrow & f_2 & = & \mathbf{Zero} \\ \mathit{Negation}(\mathbf{Pos}\ n) & \twoheadrightarrow & f_3\ n & \to_\beta & \mathbf{Neg}\ n \end{array}$$

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 **Nat** of Example 4.2.6. We want to represent the operator $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by a pseudo term g such that

$$\begin{array}{rcl} g\ \mathbf{O}\ y & \twoheadrightarrow & y \\ g\ (\mathbf{S}\ x)\ y & \twoheadrightarrow & \mathbf{S}\ (g\ x\ y) \end{array}$$

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

$$\mathit{Plus}(x, y) = \mathsf{Elim}(\mathbf{Nat}, \mathbf{Nat}, x)\{g_1, g_2\}$$

The desired reduction behaviour of the function *Plus* is as follows:

$$\begin{array}{rcccl} \mathit{Plus}(\mathbf{O}, y) & \twoheadrightarrow & g_1 & = & y \\ \mathit{Plus}(\mathbf{S}\ x, y) & \twoheadrightarrow & g_2\ x\ (\mathit{Plus}(x, y)) & \twoheadrightarrow_\beta & \mathbf{S}(\mathit{Plus}(x, y)) \end{array}$$

Before we can specify the computational behaviour of pseudo terms of the form $\mathsf{Elim}(I_0, Q, a)\{h_1, \ldots, h_n\}$, we need auxiliary functions $\mathrm{ElimFun}_x(t_i, h_i, F, I_0)$ 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 $\mathsf{Constr}(i, I_0)$ with type of constructor t_i. The third argument of $\mathrm{ElimFun}_x$, F, is a dummy function, that is supposed to compute the result for recursive calls. The last argument of $\mathrm{ElimFun}_x$, 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* $\mathrm{ElimFun}_x$ for a type of constructor in x and three terms is defined as follows:

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

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 $\to_\iota$ as follows:

$$C[\mathsf{Elim}(I, Q, \mathsf{Constr}(i, I')\vec{m})\{\vec{f}\}] \to_\iota C[\mathit{ElimFun}_x(T_i, f_i, \lambda z{:}I.\mathsf{Elim}(I, Q, z)\{\vec{f}\}, I)\ \vec{m}]$$

with $C[\,] \in \mathcal{H}_{\mathrm{i}}$, and $1 \leq i \leq |\vec{f}|$, and $I = \mathsf{Ind}(x{:}A)\{\vec{T}\}$, such that $|\vec{f}| = |\vec{T}|$ and $\mathrm{constr}_x(T_j)$ for all $j \leq |\vec{T}|$.

We will now demonstrate how this reduction relation $\rightarrow_\iota$ 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:

$$\begin{array}{rcccl} \mathit{Negation}(\mathbf{Neg}\ n) & \rightarrow_\iota & (\lambda p{:}\mathbf{Nat}.f_1\ p)\ n & \twoheadrightarrow_\beta & \mathbf{Pos}\ n \\ \mathit{Negation}(\mathbf{Zero}) & \rightarrow_\iota & f_2 & = & \mathbf{Zero} \\ \mathit{Negation}(\mathbf{Pos}\ n) & \rightarrow_\iota & (\lambda p{:}\mathbf{Nat}.f_3\ p)\ n & \twoheadrightarrow_\beta & \mathbf{Neg}\ n \end{array}$$

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

$$\begin{array}{rcccl} \mathit{Plus}(\mathbf{O}, y) & \rightarrow_\iota & g_1 & = & y \\ \mathit{Plus}(\mathbf{S}x, y) & \rightarrow_\iota & (\lambda p{:}\mathbf{Nat}.g_2\ p\ ((\lambda z{:}\mathbf{Nat}.\mathit{Plus}(z, y))\ p))\ x & \twoheadrightarrow_\beta & \mathbf{S}(\mathit{Plus}(x, y)) \end{array}$$

Notice that *Plus* is an *operational* version of the function *plus*, that is axiomatized in Example 4.1.4. We use the abbreviation $\mathit{Plus}(x, y)$ 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 $\mathcal{T}_{\mathrm{i}}$.

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

We will comment on the typing rules for the relation $\vdash_{\mathrm{i}}$ that are presented in Definition 4.2.25.

The first typing rule (ind) specifies that inductive types $\mathsf{Ind}(x : \Box)\{t_1, \ldots, t_n\}$ 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 $\epsilon \vdash_{\mathrm{i}} \mathbf{Nat} : \Box$ as follows:

$$\dfrac{\dfrac{\epsilon \vdash_{\mathrm{i}} \Box : \triangle}{x : \Box \vdash_{\mathrm{i}} x : \Box} \qquad \dfrac{\ldots}{x : \Box \vdash_{\mathrm{i}} x \rightarrow x : \Box}}{\epsilon \vdash_{\mathrm{i}} \mathsf{Ind}(x{:}\Box)\{x, x \rightarrow x\} : \Box}$$

The second typing rule (intro) specifies that the ith constructor $\mathsf{Constr}(i, I_0)$ 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 $\epsilon \vdash_{\mathrm{i}} \mathbf{S} : \mathbf{Nat} \rightarrow \mathbf{Nat}$.

Before we can specify typing rules for pseudo terms of the form $\mathsf{Elim}(I_0, Q, a)\{h_1, \ldots, h_n\}$, we need auxiliary functions $\mathrm{NodepElimType}_x(t_i, Q, I_0)$ that specify the types of the functions h_i, that determine the result for the ith constructor $\mathsf{Constr}(i, I_0)$. Recall that I_0 is the domain and Q is the range of the function specified by $\mathsf{Elim}(I_0, Q, a)\{h_1, \ldots, h_n\}$.

Remark 4.2.19 Recall the definition of *Negation*(j) that has specified domain **Int** and range **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 **Neg** n, **Zero**, and **Pos** n resp., can be typed as follows:

$$\begin{array}{l} j:\mathbf{Int}\vdash_{\mathsf{i}} f_1:\mathbf{Nat}\to\mathbf{Int} \\ j:\mathbf{Int}\vdash_{\mathsf{i}} f_2:\mathbf{Int} \\ j:\mathbf{Int}\vdash_{\mathsf{i}} f_3:\mathbf{Nat}\to\mathbf{Int} \end{array}$$

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

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

$$y:\mathbf{Nat}\vdash_{\mathsf{i}} g_2:\mathbf{Nat}\to\mathbf{Nat}\to\mathbf{Nat}$$

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

1. $\mathrm{NodepElimType}_x(x,Q,I) = Q$

2. $\mathrm{NodepElimType}_x(u\to c,Q,I) =$
$\begin{cases} I\to(Q\to\mathrm{NodepElimType}_x(c,Q,I)) & (\text{if } u = x) \\ u\to\mathrm{NodepElimType}_x(c,Q,I) & (\text{otherwise}) \end{cases}$

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

Example 4.2.21 Let Γ be the pseudo context $x:\mathbf{Nat}, y:\mathbf{Nat}$. Using the (nodep-elim) rule we can derive $\Gamma\vdash_{\mathsf{i}} \mathit{Plus}(x,y):\mathbf{Nat}$ (see Example 4.2.12) as follows:

$$\frac{\begin{array}{cccc} \cdots & \cdots & \cdots & \cdots \\ \Gamma\vdash_{\mathsf{i}}\mathbf{Nat}:\Box & \Gamma\vdash_{\mathsf{i}} g_1:\mathbf{Nat} & \Gamma\vdash_{\mathsf{i}} g_2:\mathbf{Nat}\to\mathbf{Nat}\to\mathbf{Nat} & \Gamma\vdash_{\mathsf{i}} x:\mathbf{Nat}\end{array}}{\Gamma\vdash_{\mathsf{i}}\mathsf{Elim}(\mathbf{Nat},\mathbf{Nat},x)\{g_1,g_2\}:\mathbf{Nat}}$$

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

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

$$\mathit{IntSplit} = \lambda j{:}\mathbf{Int}.(j =_{\mathbf{Int}} \mathbf{Zero} \vee (\exists n\in\mathbf{Nat}.j =_{\mathbf{Int}} \mathbf{Neg}\ n \vee j =_{\mathbf{Int}} \mathbf{Pos}\ n)).$$

A proof of $\forall j \in \mathbf{Int}.\mathit{IntSplit}\ j$ by reasoning by cases requires pseudo terms q_1, q_2, and q_3 such that:

$$\begin{aligned}
&\epsilon \vdash_{\mathrm{i}} q_1 : \forall n \in \mathbf{Nat}.\mathit{IntSplit}\ (\mathbf{Neg}\ n)\\
&\epsilon \vdash_{\mathrm{i}} q_2 : \mathit{IntSplit}\ \mathbf{Zero}\\
&\epsilon \vdash_{\mathrm{i}} q_3 : \forall n \in \mathbf{Nat}.\mathit{IntSplit}\ (\mathbf{Pos}\ n)
\end{aligned}$$

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

In the same way as $\mathrm{NodepElimType}_x(t_i, Q, I_0)$ specifies the type of functions h_i in pseudo terms of the form $\mathsf{Elim}(I_0, Q, a)\{h_1, \ldots, h_n\}$ such that Q is an inhabitant of $\Box$, $\mathrm{DepElimType}_x(t_i, P, \mathsf{Constr}(i, I_0), I_0)$ specifies the type of the proofs q_i in pseudo terms of the form $\mathsf{Elim}(I_0, P, b)\{q_1, \ldots, q_n\}$ such that P is an inhabitant of $I_0 \to *$.

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

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

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

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

The last typing rule of Definition 4.2.25 (conv) allows us to use the computational meaning of terms in proofs. This conversion rule replaces the (β-conversion) rule of λHOL. 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 *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 *Plus* to give a short proof of its associativity $(\mathit{Plus}(x, \mathit{Plus}(y, z)) =_{\mathbf{Nat}} \mathit{Plus}(\mathit{Plus}(x, y), z))$ by induction on x.

Let $\mathit{Assoc} = \lambda n{:}\mathbf{Nat}.\mathit{Plus}(n, \mathit{Plus}(y, z)) =_{\mathbf{Nat}} \mathit{Plus}(\mathit{Plus}(n, y), z)$;
let $\Gamma = \langle x : \mathbf{Nat}, y : \mathbf{Nat}, z : \mathbf{Nat}\rangle$. We want to solve the problem:

$$\Gamma \vdash_{\mathrm{i}} ? : \mathit{Assoc}\ x.$$

For doing induction on x we need to apply the (dep-elim)-rule. Thus we fill in $\mathsf{Elim}(\mathbf{Nat}, \mathit{Assoc}, x)\{?_1, ?_2\}$ for ?. Now we need to solve the problems:

$$\begin{aligned}
&\Gamma \vdash_{\mathrm{i}} ?_1 : \mathit{Assoc}\ \mathbf{O}\\
&\Gamma \vdash_{\mathrm{i}} ?_2 : (\forall x' \in \mathbf{Nat}.(\mathit{Assoc}\ x') \to (\mathit{Assoc}(\mathbf{S}\ x')))
\end{aligned}$$

The instantiation of the proof of the reflexivity of $=_{\mathbf{Nat}}$ ($=_{\mathbf{Nat}}$-*refl*) with *Plus*(y,z) has type *Plus*$(y,z) =_{\mathbf{Nat}}$ *Plus*(y,z), that is convertible (modulo $=_{\beta,\iota}$) with *Assoc* $\mathbf{O}$. Now we can use the (conv)-rule to derive that this term ($=_{\mathbf{Nat}}$-*refl* (*Plus*(y,z))) has type *Assoc* $\mathbf{O}$. Thus we can fill in this term for $?_1$. If we fill in $\lambda x'{:}\mathbf{Nat}.\lambda i{:}(\mathit{Assoc}\ x').?_3$ for $?_2$ we have to prove:

$$\Gamma, x'{:}\,\mathbf{Nat}, i{:}\,(\mathit{Assoc}\ x')\vdash_i ?_3 : (\mathit{Assoc}(\mathbf{S}\ x'))$$

For $?_3$ we can use a proof of the equivalent (modulo $=_{\beta,\iota}$) proposition $\mathbf{S}(\mathit{Plus}(x',\mathit{Plus}(y,z))) =_{\mathbf{Nat}} \mathbf{S}(\mathit{Plus}(\mathit{Plus}(x',y),z))$. Using the compatibility of $=_{\mathbf{Nat}}$ with respect to $\mathbf{S}$ we need to prove:

$$\Gamma, x'{:}\,\mathbf{Nat}, i{:}\,(\mathit{Assoc}\ x')\vdash_i ?_4 : (\mathit{Plus}(x',\mathit{Plus}(y,z))) =_{\mathbf{Nat}} (\mathit{Plus}(\mathit{Plus}(x',y),z))$$

We can fill in the assumption i (the induction hypothesis) for $?_4$, because its type (*Assoc* x') is convertible with the desired type $(\mathit{Plus}(x',\mathit{Plus}(y,z))) =_{\mathbf{Nat}} (\mathit{Plus}(\mathit{Plus}(x',y),z))$.

Here we present the typing rules for inductive types.

Definition 4.2.25 Typing rules for $\vdash_i$ are the rules for $\vdash$ of Definition 3.1.15 together with following rules for inductive types:

$$\text{(ind)}\quad \frac{(\Gamma,x{:}\Box\vdash_i C_j{:}\Box)_{j=1}^{|\vec{C}|}}{\Gamma\vdash_i \mathsf{Ind}(x{:}\Box)\{\vec{C}\}{:}\Box}\qquad \text{if } \mathrm{constr}_x(C_j),\ \text{all } j\le|\vec{C}|$$

$$\text{(intro)}\quad \frac{\Gamma\vdash_i \mathsf{Ind}(x{:}A)\{\vec{C}\}{:}T}{\Gamma\vdash_i \mathsf{Constr}(j,\mathsf{Ind}(x{:}A)\{\vec{C}\}){:}C_j[x{:=}\mathsf{Ind}(x{:}A)\{\vec{C}\}]}\qquad \text{if } j\le|\vec{C}|$$

$$\text{(nodep-elim)}\quad \frac{\Gamma\vdash_i Q{:}\Box\quad (\Gamma\vdash_i f_j{:}\mathrm{NodepElimType}_x(C_j,Q,I))_{j=1}^{n}\quad \Gamma\vdash_i t{:}I}{\Gamma\vdash_i \mathsf{Elim}(I,Q,t)\{f_1,\ldots,f_n\}{:}Q}$$
$$\text{with } I=\mathsf{Ind}(x{:}A)\{C_1,\ldots,C_n\}$$

$$\text{(dep-elim)}\quad \frac{\Gamma\vdash_i P{:}I{\to}{*}\quad (\Gamma\vdash_i f_j{:}\mathrm{DepElimType}_x(C_j,P,\mathsf{Constr}(j,I),I))_{j=1}^{|\vec{C}|}\quad \Gamma\vdash_i t{:}I}{\Gamma\vdash_i \mathsf{Elim}(I,P,t)\{f_1,\ldots,f_n\}{:}\ P\ t}$$
$$\text{with } I=\mathsf{Ind}(x{:}A)\{C_1,\ldots,C_n\}$$

$$\text{(conv)}\quad \frac{\Gamma\vdash_i a{:}T\quad \Gamma\vdash_i T{:}s\quad \Gamma\vdash_i U{:}s\quad T=_{\beta,\iota}U}{\Gamma\vdash_i a{:}U}\qquad s\in\text{Universes}$$

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 $(\mathcal{T}_i, \mathcal{X}_i, \{\to_\beta,\to_\iota\},\vdash_i)$.

We will use the abbreviation $\lambda\mathrm{HOL_i}$ for Higher Order Logic with inductive types.

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

Theorem 4.2.27 *The system λHOL_{i} has the following properties:*

1. *$\rightarrow_{\beta,\iota}$ is confluent.*
2. *If $\Gamma\vdash_{\mathrm{i}} t:u$ and $t \rightarrow_{\beta,\iota} t'$ then $\Gamma\vdash_{\mathrm{i}} t':u$.*
3. *$\rightarrow_{\beta,\iota}$ 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 $\rightarrow_{\beta,\iota}$ of $\lambda\mathrm{HOL_i}$ and λCC 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 λHOL in λCC (see proof of Theorem 3.3.5) by the mapping E, that replaces each $\Box$ with $*$ and each $\triangle$ with $\Box$, is extended to a map E_i on pseudo terms in $\mathcal{T}_{\mathrm{i}}$ by adding the following rules:

   (a) $E_i(\mathsf{Elim}(I,Q,t)\{f_1.\dots,f_n\}) =$
   $$\begin{cases} \mathsf{Elim}(E_i(I),\lambda\_{:}E_i(I).E_i(Q),E_i(t)) & (\text{if } Q = u_1 \rightarrow \dots \rightarrow u_{n+1}, \\ \{E_i(f_1).\dots,E_i(f_n)\} & \text{and } u_{n+1} \neq u' \rightarrow u'', \\ & \text{and } u_{n+1} \neq *) \\ \mathsf{Elim}(E_i(I),E_i(Q),E_i(t))\{E_i(f_1).\dots,E_i(f_n)\} & (\text{otherwise}) \end{cases}$$

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

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

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

   By $E_i(\Gamma)$ we denote the pseudo context obtained from Γ by replacing each predicate p by $E_i(p)$. If $\Gamma\vdash_{\mathrm{i}} a:t$ then we have $E_i(\Gamma)\vdash E_i(a):t'$ in λCC with inductive types, where $t' = \begin{cases} \Box & (\text{if } a = t_1 \rightarrow \dots \rightarrow t_n \rightarrow * \ (0 \leq n)) \\ E_i(t) & (\text{otherwise}) \end{cases}$. Our definition of the notion of 'simple type of constructor' in Definition 4.2.7 guarantees that any legal inductive type in $\lambda\mathrm{HOL_i}$ is mapped to a *small* inductive type in λCC with inductive types. The (nodep-elim) rule of λCC may only be used for small inductive types. Notice that only typable terms of the form $\mathsf{Elim}(I,u_1 \rightarrow \dots \rightarrow u_n \rightarrow *,t)\{\vec{f}\}$ 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 $t_1 \rightarrow \dots t_n \rightarrow t_{n+1}$ is small if all t_i have type $*$. $\Box$

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 λHOL is much larger than the proof for the associativity of *Plus*, that we described in Example 4.2.24. For instance, for proving the base case $(plus\ 0\ (plus\ y\ z) =_{\mathrm{N}} plus\ (plus\ 0\ y)\ z)$ we need *plus0* twice, and compatibility, symmetry, and transitivity of $=_{\mathrm{N}}$ once. As a comparison, the base case in the proof of associativity for the operator *Plus* requires only the reflexivity of $=_{\mathbf{Nat}}$, since the conversion rule allows us to use the computational behaviour of *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
$Plus(Plus(x,y), Plus(z,\mathbf{O}))$ does not reduce, because *Plus* only inspects the left argument. Thus this expression is not convertible with $Plus(Plus(x,y),z)$. But we know that we can prove $Plus(Plus(x,y), Plus(z,\mathbf{O})) =_{\mathbf{Nat}} Plus(Plus(x,y),z)$, as **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 Axiom with a typing rule of the following form:

$$\vdash \mathsf{Axiom}(\underline{R\vec{a}}) : \underline{R\vec{a}},\ \text{if}\ R(\vec{a}).$$

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 $\lambda\mathrm{HOL_i}$ of Section 4.2 with oracle types. Roughly speaking an oracle type consists of a complete TRS, and a representation

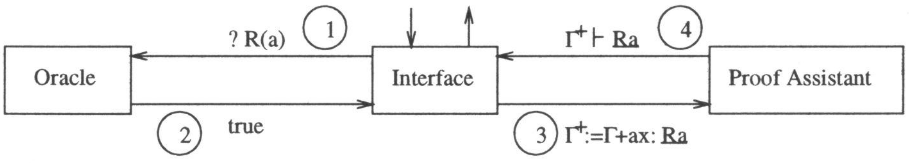


Figure 4.2: Extension of a Proof Assistant with an Oracle

(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 $\Sigma^{\mathtt{N}}$ 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 $(\Sigma^{\mathtt{N}}, R)$ 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 $(\Sigma^{\mathtt{N}}, R)$ of the TRS for `M` with the signature:

$$\begin{array}{lllll}
\textbf{sort} & \mathtt{N} & & & \\
\textbf{func} & \mathtt{0}: & & & \mathtt{N} \\
 & \mathtt{S}: & \mathtt{N} & \rightarrow & \mathtt{N} \\
 & \mathtt{var}_{\mathtt{M}}: & \mathtt{N} & \rightarrow & \mathtt{M}
\end{array}$$

2. An inductive type $\underline{\mathsf{M}} = \mathsf{Ind}(x : \square)\{x \rightarrow x \rightarrow x, x, \mathbf{Nat} \rightarrow x\}$ with constructors $\underline{\cdot} = \mathsf{Constr}(\underline{\mathsf{M}}, 1)$, $\underline{\mathsf{e}} = \mathsf{Constr}(\underline{\mathsf{M}}, 2)$, $\underline{\mathsf{var}_\mathsf{M}} = \mathsf{Constr}(\underline{\mathsf{M}}, 3)$, representing $\mathcal{T}(\Sigma, V)$. The constructors $\underline{\cdot}$ and $\underline{\mathsf{e}}$ represent the function symbols $\cdot$ and e of Σ. Applications of the constructor $\underline{\mathsf{var}_\mathsf{M}}$ to canonical pseudo terms of type **Nat**, such as $\underline{\mathsf{var}_\mathsf{M}}(\mathbf{O})$, represent variables.

3. A map $[-]_{\mathtt{M}} : \mathcal{T}(\Sigma^{\mathtt{M}})_{\mathtt{M}} \rightarrow \mathcal{T}_{\mathrm{i}}$ mapping closed terms, i.e. terms not containing variables, to their representations, such that

$$\begin{aligned} [\mathtt{x} \cdot \mathtt{y}]_{\mathtt{M}} &= \underline{\cdot}\,([\mathtt{x}]_{\mathtt{M}})\,([\mathtt{y}]_{\mathtt{M}}) \\ [\mathsf{e}]_{\mathtt{M}} &= \underline{\mathsf{e}} \\ [\mathtt{var_M(n)}]_{\mathtt{M}} &= \underline{\mathsf{var}_{\mathsf{M}}}([\mathtt{n}]_{\mathtt{N}}) \end{aligned}$$

   The map $[-]_{\mathtt{N}}$ maps closed terms of sort $\mathtt{N}$ to their representations in **Nat**.

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

$$\text{(rewrite)} \quad \epsilon \vdash \mathsf{Rewrite}([a]_{\mathtt{M}}, [b]_{\mathtt{M}}) : [a]_{\mathtt{M}} \sim_{\underline{\mathsf{M}}} [b]_{\mathtt{M}} \quad \text{if } a =_R b,\ a, b \in \mathcal{T}(\Sigma^{\mathtt{M}})_{\mathtt{M}}$$

   Where $\sim_{\underline{\mathsf{M}}}$ is the binary relation on $\underline{\mathsf{M}}$ defined as:
   $m_1 \sim_{\underline{\mathsf{M}}} m_2 = \forall r \in \underline{\mathsf{M}} \rightarrow \underline{\mathsf{M}} \rightarrow *.(\mathit{eqrel}(r) \wedge \mathit{monoidrel}(r)) \rightarrow r\ m_1\ m_2$.
   In this definition $\mathit{eqrel}(r)$ denotes equivalence of r, and we have

$$\mathit{monoidrel}(r) = \begin{array}{l} \forall x \in \underline{\mathsf{M}}.r\ (\underline{\cdot}\ x\ \underline{\mathsf{e}})\ x\ \wedge \\ \forall x \in \underline{\mathsf{M}}.r\ (\underline{\cdot}\ \underline{\mathsf{e}}\ x)\ x\ \wedge \\ \forall x, y, z \in \underline{\mathsf{M}}.r\ (\underline{\cdot}\ x\ (\underline{\cdot}\ y\ z))\ (\underline{\cdot}\ (\underline{\cdot}\ x\ y)\ z)\ \wedge \\ \forall x_1, y_1, x_2, y_2 \in \underline{\mathsf{M}}.(r\ x_1\ y_1) \rightarrow (r\ x_2\ y_2) \rightarrow r\ (\underline{\cdot}\ x_1\ x_2)\ (\underline{\cdot}\ y_1\ y_2) \end{array}$$

The typing rule for the new construct $\mathsf{Rewrite}$ 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.

	Representation of the TRS
	$\mathsf{e} \cdot (\mathtt{var_M}(0)) =_R \mathtt{var_M}(0)$
	$\downarrow [-]_{\mathtt{M}}$
$\mathsf{e} \cdot x_0 = x_0$	$\underline{\cdot}\ \underline{\mathsf{e}}\ (\underline{\mathsf{var}_{\mathsf{M}}}(\mathbf{O})) \sim_{\underline{\mathsf{M}}} (\underline{\mathsf{var}_{\mathsf{M}}}(\mathbf{O}))$
Mathematical language	*Coding of the equational theory*

Figure 4.3: Representation of a monoid by an oracle type

Example 4.3.2 Let $\mathtt{v}_0 = \mathtt{var_M}(0)$, $\mathtt{v}_1 = \mathtt{var_M}(\mathtt{S}(0))$, and $\mathtt{v}_2 = \mathtt{var_M}(\mathtt{S}(\mathtt{S}(0)))$ be closed terms $(\mathtt{v}_0, \mathtt{v}_1, \mathtt{v}_2 \in \mathcal{T}(\Sigma^{\mathtt{M}})_{\mathtt{M}})$. We have $(\mathsf{e} \cdot \mathtt{v}_0) \cdot (\mathtt{v}_1 \cdot \mathtt{v}_2) =_R (\mathtt{v}_0 \cdot \mathtt{v}_1) \cdot (\mathtt{v}_2 \cdot \mathsf{e})$.

Let $\underline{v}_0 = \underline{\mathsf{var}_{\mathsf{M}}}(\mathbf{O})$, $\underline{v}_1 = \underline{\mathsf{var}_{\mathsf{M}}}(\mathbf{S}(\mathbf{O}))$, and $\underline{v}_2 = \underline{\mathsf{var}_{\mathsf{M}}}(\mathbf{S}(\mathbf{S}(\mathbf{O})))$ be the inductive representations $(\underline{v}_0, \underline{v}_1, \underline{v}_2 \in \mathcal{T}_{\mathrm{i}})$ of the closed terms $\mathtt{v}_0, \mathtt{v}_1$ and $\mathtt{v}_2$. Thus $[\mathtt{v}_0]_{\mathtt{M}} = \underline{v}_0$, $[\mathtt{v}_1]_{\mathtt{M}} = \underline{v}_1$, and $[\mathtt{v}_2]_{\mathtt{M}} = \underline{v}_2$.

By (rewrite) we obtain a formal proof of $\underline{\cdot}\ (\underline{\cdot}\ \underline{\mathsf{e}}\ \underline{v}_0)\ (\underline{\cdot}\ \underline{v}_1\ \underline{v}_2) \sim_{\underline{\mathsf{M}}} \underline{\cdot}\ (\underline{\cdot}\ \underline{v}_0\ \underline{v}_1)\ (\underline{\cdot}\ \underline{v}_2\ \underline{\mathsf{e}})$ as follows:

$$\begin{array}{l} \epsilon \vdash \mathsf{Rewrite}(\underline{\cdot}\ (\underline{\cdot}\ \underline{\mathsf{e}}\ \underline{v}_0)\ (\underline{\cdot}\ \underline{v}_1\ \underline{v}_2), \underline{\cdot}\ (\underline{\cdot}\ \underline{v}_0\ \underline{v}_1)\ (\underline{\cdot}\ \underline{v}_2\ \underline{\mathsf{e}})) : \\ \underline{\cdot}\ (\underline{\cdot}\ \underline{\mathsf{e}}\ \underline{v}_0)\ (\underline{\cdot}\ \underline{v}_1\ \underline{v}_2) \sim_{\underline{\mathsf{M}}} \underline{\cdot}\ (\underline{\cdot}\ \underline{v}_0\ \underline{v}_1)\ (\underline{\cdot}\ \underline{v}_2\ \underline{\mathsf{e}}) \end{array}$$

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 $\cdot^{\mathsf{A}} : \mathsf{A} \to \mathsf{A} \to \mathsf{A}$ and constant $\mathbf{e}^{\mathsf{A}} : \mathsf{A}$ be the formal representation of a Σ-algebra. We define $int^{\mathrm{A}}_{\underline{\mathrm{M}}} : (\mathbf{Nat} \to \mathrm{A}) \to (\underline{\mathrm{M}} \to \mathrm{A})$ by structural induction on $\underline{\mathsf{M}}$, such that

$$\begin{array}{rcl}
int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ (\underline{\cdot}\ x\ y) & \twoheadrightarrow_{\beta,\iota} & \cdot^{\mathrm{A}}\ (int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ x)\ (int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ y) \\
int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ \underline{\mathrm{e}} & \twoheadrightarrow_{\beta,\iota} & \mathrm{e}^{\mathrm{A}} \\
int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ (\underline{\mathrm{var}}_{\underline{\mathrm{M}}}\ n) & \twoheadrightarrow_{\beta,\iota} & v\ n
\end{array}$$

By $model^{\mathrm{A}}_{\underline{\mathrm{M}}}$ we denote the property:

$$\forall \rho \in \mathbf{Nat} \to A.monoidrel(\lambda m_1, m_2{:}\underline{\mathrm{M}}.int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ m_1\ =_{\mathrm{A}} int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ m_2)$$

Notice that one of the consequences of $model^{\mathrm{A}}_{\underline{\mathrm{M}}}$ is:

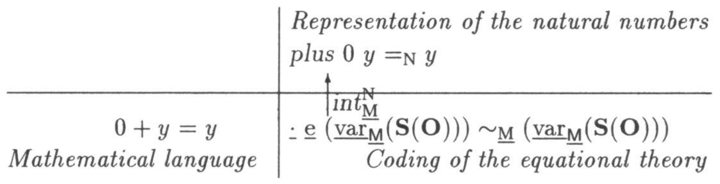
$$\forall \rho \in \mathbf{Nat} \to A.\forall x \in \underline{\mathrm{M}}.(\lambda m_1, m_2{:}\underline{\mathrm{M}}.int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ m_1\ =_{\mathrm{A}} int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ m_2)\ (\underline{\cdot}\ x\ \underline{\mathrm{e}})\ x$$

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

$$\forall \rho \in \mathbf{Nat} \to \mathrm{A}.\forall x \in \underline{\mathrm{M}}.\ \cdot^{\mathrm{A}}\ (int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ x)\ e^{\mathrm{A}} =_{\mathrm{A}} int^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ x$$

and it implies: $\forall a \in A.\ \cdot^{\mathrm{A}}\ \ a\ e^{\mathrm{A}} =_{\mathrm{A}} a$.

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.



Representation of the natural numbers
plus 0 y =N y
int^N_M
0 + y = y
Mathematical language
· e (var_M(S(O))) ~M (var_M(S(O)))
Coding of the equational theory

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 **Nat** with constructors **O** and **S** that is used in the representation of variables by $\underline{\mathrm{var}}_{\underline{\mathrm{M}}}$; third we have the sort `N` with function symbols `0` and `S` that is used in the representation of variables by $\mathtt{var}_{\mathtt{M}}$ in the TRS (Σ, R).

Example 4.3.4 Using the definition of $\sim_{\underline{\mathrm{M}}}$ we can prove that derivable equality is valid in each model:

$$\mathit{model}^{\mathrm{A}}_{\underline{\mathrm{M}}} \to (\forall m_1, m_2 \in \underline{\mathrm{M}}. m_1 \sim_{\underline{\mathrm{M}}} m_2 \to (\forall \rho \in \mathbf{Nat} \to \mathrm{A}. \mathit{int}^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ m_1 =_{\mathrm{A}} \mathit{int}^{\mathrm{A}}_{\underline{\mathrm{M}}}\ \rho\ m_2))$$

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

$$\forall \rho \in \mathbf{Nat} \to N. \mathit{int}^{\mathrm{N}}_{\underline{\mathrm{M}}}\ \rho\ (\underline{\cdot}\ (\underline{\cdot}\ \underline{\mathsf{e}}\ \underline{v}_0)\ (\underline{\cdot}\ \underline{v}_1\ \underline{v}_2)) \ =_{\mathrm{N}} \mathit{int}^{\mathrm{N}}_{\underline{\mathrm{M}}}\ \rho\ (\underline{\cdot}\ (\underline{\cdot}\ \underline{v}_0\ \underline{v}_1)\ (\underline{\cdot}\ \underline{v}_2\ \underline{\mathsf{e}}))$$

If we specialize this for a valuation function $\rho : \mathbf{Nat} \to N$ such that $\rho\ \mathbf{O} \twoheadrightarrow_{\beta,\iota} x$, $\rho\ (\mathbf{S}(\mathbf{O})) \twoheadrightarrow_{\beta,\iota} y$, and $\rho\ (\mathbf{S}(\mathbf{S}(\mathbf{O}))) \twoheadrightarrow_{\beta,\iota} z$ we obtain a proof of:

$$\mathit{plus}\ (\mathit{plus}\ 0\ x)\ (\mathit{plus}\ y\ z) =_{\mathrm{N}} \mathit{plus}\ (\mathit{plus}\ x\ y)\ (\mathit{plus}\ z\ 0)$$

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 Σ-terms and the valuation function for given members of a Σ-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 M, $\cdot$, e, and $\mathrm{var}_{\mathrm{M}}$. The following typing rules are added for typing these constants:

$$\epsilon \vdash \mathrm{M} : \Box \quad \epsilon \vdash \cdot : \mathrm{M} \to \mathrm{M} \to \mathrm{M} \quad \epsilon \vdash \mathrm{e} : \mathrm{M} \quad \epsilon \vdash \mathrm{var}_{\mathrm{M}} : \mathbf{Nat} \to \mathrm{M}$$

   and the following reduction rules:

$$\begin{array}{rcl} \cdot\ \mathrm{e}\ x & \to_\rho & x \\ \cdot\ x\ \mathrm{e} & \to_\rho & x \\ \cdot\ x\ (\cdot\ y\ z) & \to_\rho & \cdot\ (\cdot\ x\ y)\ z \end{array}$$

2. The representation of $\underline{\mathrm{M}}$ is the same.

3. The set of pseudo terms is extended with a construct $[-]$, that represents the 'inverse' of the map $[-]_{\mathsf{M}}$. It has the following typing rule:

$$\frac{\Gamma \vdash a : \underline{\mathrm{M}}}{\Gamma \vdash [a] : \mathrm{M}}$$

   and the following reduction rules:

$$\begin{array}{rcl} [\underline{\cdot}\ x\ y] & \to_\chi & \cdot\ [x]\ [y] \\ [\underline{\mathrm{e}}] & \to_\chi & \mathrm{e} \\ [\underline{\mathrm{var}}_{\mathrm{M}}(n)] & \to_\chi & \mathrm{var}_{\mathrm{M}}(n) \end{array}$$

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

$$\frac{\Gamma \vdash p : [a] =_{\mathrm{M}} [b]}{\Gamma \vdash \mathrm{noconf}\ p : a \sim_{\underline{M}} b}$$

5. The conversion rule is extended as follows: $\dfrac{\Gamma \vdash a : A \quad \Gamma \vdash B : s \quad A =_{\beta,\iota,\rho,\chi} B}{\Gamma \vdash a : B}$

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 $[-]_{\mathsf{M}}$. Therefore the executable map $[-]$ transforms arguments of type $\underline{\mathsf{M}}$ 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 $\mathit{model}^{\mathsf{A}}_{\underline{\mathsf{O}}}$.
The notions $\mathit{int}^{\mathsf{A}}_{\underline{\mathsf{O}}}$ and $\mathit{model}^{\mathsf{A}}_{\underline{\mathsf{O}}}$ are automatically generated by a tactic. The proof of $\mathit{model}^{\mathsf{A}}_{\underline{\mathsf{O}}}$ is used to generate a proof that each provable equality in $\underline{\mathsf{O}}$ is valid in A.

Prove an equality in a model A of an oracle type O.
First an equality $\mathsf{a} =_{\mathsf{A}} \mathsf{b}$ is transformed into an equivalent $\mathit{int}^{\mathsf{A}}_{\underline{\mathsf{O}}}\ \rho\ [\mathtt{t}]_{\mathsf{O}} =_{\mathsf{A}} \mathit{int}^{\mathsf{A}}_{\underline{\mathsf{O}}}\ \rho\ [\mathtt{u}]_{\mathsf{O}}$, by computing terms $\mathtt{t}$, $\mathtt{u}$ and a valuation function ρ, such that $\mathit{int}^{\mathsf{A}}_{\underline{\mathsf{O}}}\ \rho\ [\mathtt{t}]_{\mathsf{O}} \twoheadrightarrow_{\beta,\iota} \mathsf{a}$ and $\mathit{int}^{\mathsf{A}}_{\underline{\mathsf{O}}}\ \rho\ [\mathtt{u}]_{\mathsf{O}} \twoheadrightarrow_{\beta,\iota} \mathsf{b}$. If $\mathtt{t}$ and $\mathtt{u}$ have the same normal form, then the Rewrite axiom is used to produce a proof of $\mathsf{a} =_{\mathsf{A}} \mathsf{b}$ (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 *pattern matching* is a good candidate. Before we give a formal description of an extension of λHOL 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:

```
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:

$$\mathit{Fac}(n) = \mathsf{Elim}(\mathbf{Nat}, \mathbf{Nat}, n)\{\mathbf{S}\ \mathbf{O}, \lambda m{:}\mathbf{Nat}.\lambda f{:}\mathbf{Nat}.\mathit{Mul}(\mathbf{S}m, f)\}$$

This function has the following reduction behaviour:

$$\begin{array}{lll} \mathit{Fac}(\mathbf{S}\ \mathbf{O}) & \rightarrow_\iota & (\lambda p{:}\mathbf{Nat}.(\lambda m{:}\mathbf{Nat}.\lambda f{:}\mathbf{Nat}.\mathit{Mul}(\mathbf{S}m, f))\ p\ ((\lambda n'{:}\mathbf{Nat}.\mathit{Fac}(n'))\ p))\ \mathbf{O} \\ & \twoheadrightarrow_\beta & \mathit{Mul}(\mathbf{S}\ \mathbf{O}, \mathit{Fac}(\mathbf{O})) \end{array}$$

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 $\leq$ on natural numbers using recursors:

$$\begin{aligned}&\mathit{Leq}(m,n) = \mathsf{Elim}(\mathbf{Nat},\mathbf{Nat}\to\mathbf{Bool},m)\{\lambda n'{:}\mathbf{Nat}.\mathbf{True},\\ &\quad\lambda m_1{:}\mathbf{Nat}.\lambda l_1{:}\mathbf{Nat}\to\mathbf{Bool}.\lambda n_1{:}\mathbf{Nat}.\mathsf{Elim}(\mathbf{Nat},\mathbf{Bool},n_1)\{\mathbf{False},\\ &\quad\lambda n_2{:}\mathbf{Nat}.\lambda l_2{:}\mathbf{Bool}.l_1\ n_2\}\}\ n\end{aligned}$$

An equivalent definition using pattern matching:

```
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:

```
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(Σ, R) with the property that Σ can be divided into disjoint sets $\mathcal{D}$ and $\mathcal{C}$ such that every left-hand side $F(t_1, \ldots, t_n)$ of a rewrite rule of R satisfies $F \in \mathcal{D}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C},V)$.
A *pattern* is an element of $\mathcal{T}(\mathcal{C},V)$. 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$(\mathcal{D}, \mathcal{C}, R)$.

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
      Plus:   M × M →       M
rule  Plus(v,0)                → v
      Plus(x,Plus(y,z))        → Plus(Plus(x,y),z)
```

If we could divide the signature into defined symbols $\mathcal{D}$ and constructors $\mathcal{C}$, then certainly `Plus` should be in $\mathcal{D}$ 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$(\mathcal{D}, \mathcal{C}, R)$. A defined symbol $F \in \mathcal{D}$ *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 nth element of a list, to the signature of lists as follows:

```
def  Nth:  nat × list →   nat
rule  Nth(0,Cons(x,y))      → x
      Nth(S(n),Cons(x,y))   → Nth(n,y)
```

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

Remark 5.2.5 Notice that for *finite* Constructor Systems, that are specified using the method described above, *exhaustively definedness* is *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.

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

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
$\langle$`Plus(Plus(x,y),0)`, `Plus(x,y)`$\rangle$ (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 '*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.

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

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.*

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 σ, we have $F(p_1, \ldots, p_n)^\sigma = F(p_1^\sigma, \ldots, p_n^\sigma) \neq C(q_1^\sigma, \ldots, q_m^\sigma) = C(q_1, \ldots, q_m)^\sigma$, because $\mathcal{C}$ and $\mathcal{D}$ are disjoint. Thus $F(p_1, \ldots, p_n)$ is not unifiable with q. □

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 $l \to r \in R$. 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 $l_1 \to r_1 \in R$ on $l_2 \to r_2 \in R$. 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_2 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 CS($\mathcal{D}, \mathcal{C}, R$) 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 CS($\mathcal{D}, \mathcal{C}, R$) 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 CS($\mathcal{D}, \mathcal{C}, R$) 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 Σ-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 Σ 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 $u_1, \ldots, u_n (<_s)_{\text{lex}} t_1, \ldots, t_n$ for each occurrence $F(u_1, \ldots, u_n)$ of F in u. A rule $F(t_1, \ldots, t_n) \to r$ is *argument decreasing*, if r is argument decreasing for $F(t_1, \ldots, t_n)$. A TRS(Σ, R) is *argument decreasing* if all rules $r \in R$ are argument decreasing.
3. Let $\rhd$ be a strict partial order on Σ. A term t is *bounded by F*, if for each function symbol $G \neq F$ in t we have $F \rhd G$. A rule $F(t_1, \ldots, t_n) \to r$ is *bounded*, if r is bounded by F. A TRS(Σ, R) is *bounded* if all rules are bounded.
4. Let $\mathcal{D}, \mathcal{C}$ be disjoint S-sorted signatures. Let $\rhd$ be a binary relation on $\mathcal{D}$. We extend it to a binary relation $\rhd_{\mathcal{C}}$ on $\mathcal{D} \cup \mathcal{C}$ as follows:
   (a) $F \rhd_{\mathcal{C}} C$, for $F \in \mathcal{D}, C \in \mathcal{C}$.
   (b) $F \rhd_{\mathcal{C}} G$, if $F \rhd G$ for $F, G \in \mathcal{D}$.

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

Lemma 5.2.14 *Let Σ be an S-sorted signature. Let $\rhd$ be strict partial order on Σ.*

1. *If $t <_s u$ then $t <_{\text{lpo}} u$, for $t, u \in \mathcal{T}(\Sigma, V)$.*
2. *If $v \in \text{var}(t)$ then $t >_{\text{lpo}} v$ 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 $F(t_1, \ldots, t_n) >_{\text{lpo}} u$.*

Proof

1. Induction on the proof of $t <_s u$.

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

   Case 2: $F(t_1, \dots, t_n) <_s F(t_1, \dots, t_{i-1}, u, t_{i+1}, \dots, t_n)$, *because* $t_i <_s u$ *for* $1 \leq i \leq n$. By the induction hypothesis we have $t_i <_{\text{lpo}} u$. Thus we have $F(t_1, \dots, t_n) <_{\text{lpo}} F(t_1, \dots, t_{i-1}, u, t_{i+1}, \dots, t_n)$.

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

2. Structural induction on t.

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

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

3. Structural induction on u.

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

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

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

   Case $F \rhd G$. By definition $F(t_1, \dots, t_n) >_{\text{lpo}} u$. □

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 $\mathcal{D}, \mathcal{C}$ be disjoint S-sorted signatures. Let $\rhd$ be a binary relation on $\mathcal{D}$.*

1. *If $\rhd$ is a strict partial order then $\rhd_{\mathcal{C}}$ is a strict partial order.*

2. *If $\rhd$ is well-founded, then $\rhd_{\mathcal{C}}$ is well-founded.*

Proof

1. Assume $\rhd$ is a strict partial order. We have $C \not\rhd_{\mathcal{C}} C$, for each $C \in \mathcal{C}$. For each $F \in \mathcal{D}$ we have $F \not\rhd_{\mathcal{C}} F$, because $\rhd$ is irreflexive. Thus $\rhd_{\mathcal{C}}$ is irreflexive.
   Assume $F_1 \rhd_{\mathcal{C}} F_2$. Then $F_1 \in \mathcal{D}$. If $F_2 \in \mathcal{C}$, then $F_2 \not\rhd_{\mathcal{C}} F_1$. If $F_2 \in \mathcal{D}$, then $F_2 \not\rhd_{\mathcal{C}} F_1$ because $\rhd$ is anti-symmetric. Thus $\rhd_{\mathcal{C}}$ is anti-symmetric.
   Assume $F_1 \rhd_{\mathcal{C}} F_2$ and $F_2 \rhd_{\mathcal{C}} F_3$. Then $F_1, F_2 \in \mathcal{D}$. If $F_3 \in \mathcal{C}$, then $F_1 \rhd_{\mathcal{C}} F_3$. If $F_3 \in \mathcal{D}$, then $F_1 \rhd_{\mathcal{C}} F_3$, because $\rhd$ is transitive. Thus $\rhd_{\mathcal{C}}$ is transitive.
   Thus $\rhd_{\mathcal{C}}$ is a strict partial order.

2. Assume we have an infinite sequence $F_1 \rhd_{\mathcal{C}} F_2 \rhd_{\mathcal{C}} F_3 \rhd_{\mathcal{C}} \ldots$. We must have $F_i \in \mathcal{D}$ for all i, as $C \not\rhd_{\mathcal{C}} F$, for $C \in \mathcal{C}$. But then we have $F_1 \rhd F_2 \rhd F_3 \rhd \ldots$. □

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 $(\mathcal{D}, \mathcal{C}, R)$ be an S-sorted Constructor System. Assume $\rhd$ is a strict partial order on $\mathcal{D}$. If a rule $l \to r \in R$ is argument decreasing and is $\rhd_{\mathcal{C}}$-bounded then $l >_{\text{lpo}} r$.*

Proof
Assume $F(t_1, \ldots, t_n) \to r \in R$ is argument decreasing and is $\rhd_{\mathcal{C}}$-bounded by F. Then r is argument decreasing for $F(t_1, \ldots, t_n)$, and r is $\rhd_{\mathcal{C}}$-bounded by F. By Lemma 5.2.14 we obtain $F(t_1, \ldots, t_n) >_{\text{lpo}} r$, as $\text{var}(r) \subseteq F(t_1, \ldots, t_n)$. □

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

Proof
Assume $\rhd$ is a well-founded strict partial order on $\mathcal{D}$. By Lemma 5.2.15 $\rhd_{\mathcal{C}}$ is a well-founded strict partial order. If $(\mathcal{D}, \mathcal{C}, R)$ is $\rhd_{\mathcal{C}}$-bounded then by Proposition 5.2.16 we have $l >_{\text{lpo}} r$ for all rules $r \in R$. Thus by Theorem 2.2.23 we obtain that $(\mathcal{D}, \mathcal{C}, R)$ 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 $\text{CS}(\mathcal{D}, \mathcal{C}, R)$ and a strict partial order $>$ on R. We will write $\text{PCS}(\mathcal{D}, \mathcal{C}, R, >)$.

Example 5.3.2 Priority Constructor Systems are well suited for defining equality:

```
sort  bool, nat
cons  True:                         bool
      False:                        bool
      0:                            nat
      S:                  nat →     nat
def   Eq:           nat × nat →     bool
rule  Eq(0,0)              → True
   >  Eq(S(w),S(x))        → Eq(w,x)
   >  Eq(y,z)              → False
```

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 $(\mathcal{D}, \mathcal{C}, R)$ 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 $C_1, C_2 \in \mathcal{C}$ and $m, n \geq 0$, and either
   (a) $C_1 \neq C_2$, or
   (b) i. $C_1 = C_2$, and
       ii. $\exists i \leq m \; t_i \#_s u_i$.
2. Two terms t, u *have a strongly incompatible argument*, denotation $t \#_s^{\mathrm{Arg}} u$, if $t = F(t_1, \ldots, t_n)$, $u = F(u_1, \ldots, u_n)$, and $\exists i \leq n \; t_i \#_s u_i$, for some $F \in \mathcal{D}$.

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)`$\#_s^{\mathrm{Arg}}$`Eq(0,0)`, and `Eq(S(y),0)`$\#_s^{\mathrm{Arg}}$`Eq(S(w),S(x))`, but *not* `Eq(y,0)`$\#_s^{\mathrm{Arg}}$`Eq(0,0)`.

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

1. Let $F \in \mathcal{D}$. Let $l_1 \to r_1 \in R$ be an F-rule. A redex $C[l_1^\sigma]$ is *enabled*, if $l_2 \#_s^{\mathrm{Arg}} l_1^\sigma$, for each F-rule $l_2 \to r_2 \in R$, such that $l_2 \to r_2 > l_1 \to r_1$.
2. A reduction step $C[l^\sigma] \to C[r^\sigma]$ is *enabled*, if the redex $C[l^\sigma]$ is enabled. We denote this by $C[l^\sigma] \to_e C[r^\sigma]$. The reflexive transitive closure of $\to_e$ is denoted by $\twoheadrightarrow_e$.

Example 5.3.6 Enabled reduction steps of the PCS of Example 5.3.2 are:

$$\begin{array}{rcl} \texttt{Eq(0,0)} & \rightarrow_e & \texttt{True}, \\ \texttt{Eq(S(}x\texttt{),0)} & \rightarrow_e & \texttt{False}, \\ \texttt{Eq(S(0),S(0))} & \rightarrow_e & \texttt{Eq(0,0)}. \end{array}$$

The reduction step `Eq(x,0)` $\rightarrow$ `False` is *not* enabled, because x and `0` are not strongly incompatible.

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

1. *If $t \#_s u$ then $t^\sigma \#_s u$, for each substitution σ.*
2. *If $t \#_s u$ then t and u are not unifiable.*

Proof

1. Induction on the proof of $t \#_s u$.

   Base case: $C(t_1, \ldots, t_m) \#_s C'(u_1, \ldots, u_n)$, because $C \neq C'$, for $C, C' \in \mathcal{C}$. Then also $C(t_1, \ldots, t_m)^\sigma \#_s C'(u_1, \ldots, u_n)$.

   Induction step: $C(t_1, \ldots, t_n) \#_s C(u_1, \ldots, u_n)$, because $t_i \#_s u_i$ for some $i \leq n$ and $C \in \mathcal{C}$. 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 $t \#_s u$.

   Base case: $C(t_1, \ldots, t_m) \#_s C'(u_1, \ldots, u_n)$, because $C \neq C'$, for $C, C' \in \mathcal{C}$. We have $C(t_1, \ldots, t_m)^\sigma = C(t_1^\sigma, \ldots, t_m^\sigma) \neq C'(u_1^\sigma, \ldots, u_n^\sigma) = C(u_1, \ldots, u_m)^\sigma$.

   Induction step: $C(t_1, \ldots, t_n) \#_s C(u_1, \ldots, u_n)$, because $t_i \#_s u_i$ for some $i \leq n$ and $C \in \mathcal{C}$. 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. □

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:

rule	`Eq(0,0)`	$\rightarrow$ `True`
	`Eq(S(w),S(x))`	$\rightarrow$ `Eq(w,x)`
	`Eq(0,S(y))`	$\rightarrow$ `False`
	`Eq(S(z),0)`	$\rightarrow$ `False`

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,..., 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 existance 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 normaling 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.

$$\begin{tikzcd}
\text{IncPat} \arrow[r] & \text{IncLhs} \arrow[r] & \text{todo}^F \arrow[r] & \text{new} \arrow[r] & \text{subs} \arrow[r] & \text{newrules} \arrow[r] & \text{transform} \\
 & & & & & \text{precrules} \arrow[ur] &
\end{tikzcd}$$

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 Σ 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 Ω-terms. By t_Ω we indicate the Ω-term obtained from a term t by replacing each variable with Ω. Let W be a finite subset of V and u an Ω-term. *fresh*(W, u) indicates the term obtained from u by replacing each Ω with a different variable not in W. Let $F \in \Sigma$, $\tau(F) = s_1 \times \ldots \times s_n \to s$, $t_i \in \mathcal{T}(\Sigma, V)_{s_i}$. $\Omega(F, i, t_i)$ denotes the Ω-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 Ω-terms. t and u are *compatible*, denoted by $t \uparrow u$, if there exists some Ω-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 Ω-terms t and u is denoted by $t \sqcup u$.

Remark 5.3.10 For each Ω-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 Ω-term `Eq(0,0)` is incompatible with `Eq(S(`Ω`),S(`Ω`))`. The Ω-terms `Eq(0,`Ω`)` and `Eq(`Ω`,S(`Ω`))` are compatible and have least upper bound `Eq(0,S(`Ω`))`.

Lemma 5.3.12 *Let Σ be an S-sorted signature.*

1. *If $t \# u$ and $w \succeq u$, then $t \# w$.*

2. *If $t \# u$ and $w \uparrow u$, then $t \# w \sqcup u$.*

3. *$\exists i \leq n$ $t_i \# u_i \Leftrightarrow F(t_1, \ldots, t_n) \# F(u_1, \ldots, u_n)$, 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 $i \leq n$.
□

In Constructor Systems the notions 'strongly incompatible' for patterns and 'incompatible' for Ω-terms of patterns are equivalent. We will use this fact to work with Ω-patterns in the algorithm for transforming a PCS into an equivalent CS.

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

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 $C(p_1, \ldots, p_n)\#_s C'(q_1, \ldots, q_m)$ and $C(p_1, \ldots, p_n)_\Omega \# C'(q_1, \ldots, q_m)$.

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

A term is a redex of a left-linear rule, if and only if it is an upper bound of the Ω-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* Σ *be an* S*-sorted signature.*

1. *If* u *is linear and* $t \succeq u_\Omega$ *then* $u^\sigma = t$, *for some substitution* σ.

2. *If* $u^\sigma = t$ *then* $t \succeq u_\Omega$.

Proof

1. Proof with structural induction on u.

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

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

2. Proof with structural induction on u. □

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

Definition 5.3.15 Let $\mathrm{CS}(\mathcal{D},\mathcal{C},R)$ be an S-sorted Constructor System.

1. (a) IncPat($C(t_1,\ldots,t_n)$) $= \{C'(\Omega,\ldots,\Omega) | C' \in \mathcal{C}, \text{sort}(C') = s, C \neq C'\} \cup$ $\{\Omega(C,i,u) | 1 \leq i \leq n, u \in \text{IncPat}(t_i)\}$, for $C \in \mathcal{C}$, with $\tau(C) = s_1 \times \ldots \times s_n \to s$.

   (b) IncPat(v) $= \emptyset$, for $v \in V$.

2. Let $F \in \mathcal{D}$, $\tau(F) = s_1 \times \ldots \times s_n \to s$. Let $p_1,\ldots,p_n \in \mathcal{T}(\mathcal{C},V)$ $(1 \leq n)$. 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:
IncPat(`0`)= {`S(`Ω`)`}.
Every pattern that is strongly incompatible with `0` is of the form `S(t)`, because *nat* has two constructors `0`,`S`, and moreover `S(t)` $\succeq$ `S(`Ω`)`.

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

We can compute IncLhs for the left-hand sides of the rules for `Eq`:

$$\begin{aligned} \text{IncLhs}(\texttt{Eq(0,0)}) &= \{\texttt{Eq}(\texttt{S}(\Omega),\Omega), \texttt{Eq}(\Omega,\texttt{S}(\Omega))\} \\ \text{IncLhs}(\texttt{Eq}(\texttt{S}(w),\texttt{S}(x))) &= \{\texttt{Eq}(\texttt{0},\Omega), \texttt{Eq}(\Omega,\texttt{0})\} \\ \text{IncLhs}(\texttt{Eq}(y,z)) &= \emptyset. \end{aligned}$$

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

In the next lemma we show that IncPat and IncLhs compute Ω-terms that are incompatible with the Ω-terms of their arguments.

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

1. $\forall t \in$ IncPat(p) $p_\Omega \# t$, *for* $p \in \mathcal{T}(\mathcal{C},V)$.

2. $\forall t \in$ IncLhs($F(p_1,\ldots,p_n)$) $F(p_1,\ldots,p_n)_\Omega \# t$.

Proof

1. Structural induction on p.

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

   Case $p = C(p_1,\ldots,p_n)$. Let $t \in$ IncPat($C(p_1,\ldots,p_n)$).

   Either $t = C'(\Omega,\ldots,\Omega)$ *and* $C' \in \mathcal{C}, C \neq C'$. Then $C(p_1,\ldots,p_n) \# C'(\Omega,\ldots,\Omega)$.

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

2. By definition $t = \Omega(F,i,u)$ for some $u \in$ IncPat(p_i). Thus $(p_i)_\Omega \# u$ and therefore by Lemma 5.3.12 $F(p_1,\ldots,p_n)_\Omega \# \Omega(F,i,u)$. □

In the following lemma we show that all terms that IncPat and IncLhs compute the most general Ω-terms that are incompatible with the Ω-terms of their arguments.

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

1. *If* $p \#_s t$ *then* $t \succeq u$ *for some* $u \in \mathrm{IncPat}(p)$*, for* $p \in \mathcal{T}(\mathcal{C}, V)$.
2. *If* $F(p_1, \ldots, p_n) \#_s^{\mathrm{Arg}} t$ *then* $t \succeq u$ *for some* $u \in \mathrm{IncLhs}(F(p_1, \ldots, p_n))$.

Proof

1. Structural induction on p.

   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 $C' \in \mathcal{C}$.

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

   Case $C = C'$. We have $p_i \#_s q_i$, for some i. By the induction hypothesis there exists a $u \in \mathrm{IncPat}(p_i)$ that $q_i \succeq u$. Thus $C(q_1, \ldots, q_n) \succeq \Omega(C, i, u)$. By definition $\Omega(C, i, u) \in \mathrm{IncPat}(C(p_1, \ldots, p_n))$.

2. Assume $F(p_1, \ldots, p_n) \#_s^{\mathrm{Arg}} t$. 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 $u \in \mathrm{IncPat}(p_i)$ we have $t_i \succeq u$. Thus $\Omega(F, i, u) \in \mathrm{IncLhs}(F(p_1, \ldots, p_n))$ and $F(t_1, \ldots, t_n) \succeq \Omega(F, i, u)$. □

Now we will define a function 'todo^F' that, given a sequence $\vec{l}$ of application of F to patterns, computes Ω-terms that are incompatible with all Ω-terms $(l_i)_\Omega$. This definition is based on 'IncLhs'.

Using the definition of 'todo^F' we will define a function 'new' that, given a sequence of applications $\vec{l}$ of a defined symbol F to patterns, and an application m of F to patterns, computes upper bounds of m_Ω that are incompatible with all Ω-terms $(l_i)_\Omega$.

Finally we will define a function 'subs' that, given a sequence of applications $\vec{l}$ of a defined symbol F to patterns, and an application m of F to pattern, computes substitutions σ, such that $l_i \#_s^{\mathrm{Arg}} m^\sigma$.

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) $\mathrm{todo}^F(\epsilon) = \{F(\Omega, \ldots, \Omega)\}$.

   (b) i. $\mathrm{td}^F(l_1, \ldots, l_{k+1}) = \{t \in \mathrm{todo}^F(l_1, \ldots, l_k) | (l_{k+1})_\Omega \# t\} \cup \{u \sqcup t | u \in \mathrm{IncLhs}(l_{k+1}), t \in \mathrm{todo}^F(l_1, \ldots, l_k), (l_{k+1})_\Omega \uparrow t, u \uparrow t\}$;

   ii. $\mathrm{todo}^F(l_1, \ldots, l_{k+1}) = \{t \in \mathrm{td}^F(l_1, \ldots, l_{k+1}) | t \not\succ \mathrm{td}^F(l_1, \ldots, l_{k+1})\}$.

2. Let $\vec{l}$ be a sequence of applications of F to patterns. Let m be an application of F to patterns.

   (a) $\mathrm{nw}(\vec{l}, m) = \{m_\Omega \sqcup t | t \in \mathrm{todo}^F(\vec{l}), m_\Omega \uparrow t\}$;

   (b) $\mathrm{new}(\vec{l}, m) = \{t \in \mathrm{nw}(\vec{l}, m) | t \not\succ \mathrm{nw}(\vec{l}, m)\}$.

3. Let $\vec{l}$ be a sequence of applications of F to patterns. Let m be an application of F to patterns.
   $\mathrm{subs}(\vec{l}, m) = \{\mathrm{mgu}(m, \mathrm{fresh}(\mathrm{var}(m), u)) | u \in \mathrm{new}(\vec{l}, m)\}$.

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

$$\begin{array}{rcl}
\mathrm{todo}^{Eq}(\mathtt{Eq(0,0)}) & = & \{\mathtt{Eq}(\mathtt{S}(\Omega), \Omega), \mathtt{Eq}(\Omega, \mathtt{S}(\Omega))\}; \\
\mathrm{todo}^{Eq}(\mathtt{Eq}(0,0), \mathtt{Eq}(\mathtt{S}(w), \mathtt{S}(x))) & = & \{\mathtt{Eq}(\mathtt{S}(\Omega), 0), \mathtt{Eq}(0, \mathtt{S}(\Omega))\}; \\
\mathrm{todo}^{Eq}(\mathtt{Eq}(0,0), \mathtt{Eq}(\mathtt{S}(w), \mathtt{S}(x)), \mathtt{Eq}(y, z)) & = & \emptyset.
\end{array}$$

Computation of new for the left-hand sides of the rules for Eq:

$$\begin{array}{rcl}
\mathrm{new}(\epsilon, \mathtt{Eq(0,0)}) & = & \{\mathtt{Eq(0,0)}\}; \\
\mathrm{new}(\langle \mathtt{Eq(0,0)} \rangle, \mathtt{Eq(S(w),S(x))}) & = & \{\mathtt{Eq}(\mathtt{S}(\Omega), \mathtt{S}(\Omega))\}; \\
\mathrm{new}(\langle \mathtt{Eq(0,0)}, \mathtt{Eq}(\mathtt{S}(w), \mathtt{S}(x)) \rangle, \mathtt{Eq}(y, z)) & = & \{\mathtt{Eq}(\mathtt{S}(\Omega), 0), \mathtt{Eq}(0, \mathtt{S}(\Omega)) y\}.
\end{array}$$

The following technical lemma is necessary to show that applying a substitution of 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 $(\mathcal{D}, \mathcal{C}, R)$ be an S-sorted Constructor System. Let $F \in \mathcal{D}$. Let $\vec{l}$ be a sequence of linear applications of F to patterns. Let m be a linear application of F to patterns.*

1. $\forall t \in \mathrm{todo}^F(\vec{l})\ \exists t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C} \cup \{\Omega\})\ t = F(t_1, \ldots, t_n)$.

2. $\forall t \in \mathrm{new}(\vec{l}, m)\ \exists t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C} \cup \{\Omega\})\ t = F(t_1, \ldots, t_n)$.

3. $\forall \sigma \in \mathrm{subs}(\vec{l}, m)\ \mathrm{linear}(m^\sigma) \wedge \exists p_1, \ldots, p_n \in \mathcal{T}(\mathcal{C}, V)\ m^\sigma = F(p_1, \ldots, p_n)$.

Proof

1. Induction on length of $\vec{l}$.

   Case $|\vec{l}| = 0$. Trivial.

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

   Either $t \in \mathrm{todo}^F(l_1, \ldots, l_i)$ and $(l_{i+1})_\Omega \# t$. By the induction hypothesis we have $t = F(t_1, \ldots, t_n)$ for some $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C} \cup \{\Omega\})$.

Or $t = s \sqcup u$ *with* $s \in \mathrm{IncLhs}((l_{i+1})_\Omega), u \in todo^F(l_1, \ldots, l_i)$ *such that* $(l_{i+1})_\Omega \uparrow u$ *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 $s \sqcup u = F(s_1 \sqcup u_1, \ldots, s_n \sqcup u_n)$, and $s_i \sqcup u_i \in \mathcal{T}(\mathcal{C} \cup \{\Omega\})$ $(i \leq n)$. Thus $t = F(t_1, \ldots, t_n)$ for some $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C} \cup \{\Omega\})$.

2. Let $t \in \mathrm{new}(\vec{l}, m)$. Then $t = m_\Omega \sqcup t'$ with $t' \in \mathrm{todo}^F(\vec{l})$. 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 $m_\Omega \sqcup t' = F((p_1)_\Omega \sqcup t_1, \ldots, (p_n)_\Omega \sqcup t_n)$ and $(p_i)_\Omega \sqcup t_i \in \mathcal{T}(\mathcal{C} \cup \{\Omega\})$ $(i \leq n)$.

3. Let $\sigma \in \mathrm{subs}(\vec{l}, m)$. Then $\sigma = \mathrm{mgu}(m, \mathrm{fresh}(\mathrm{var}(m), u))$ for some $u \in \mathrm{new}(\vec{l}, m)$. Let $u' = \mathrm{fresh}(\mathrm{var}(m), u))$. Let $\sigma = \mathrm{mgu}(m, u')$. Because m and u' are linear and $\mathrm{var}(m) \cap \mathrm{var}(u') = \emptyset$, m^σ 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 σ we obtain that $m^\sigma = F(q_1, \ldots, q_n)$, for some patterns $q_1, \ldots, q_n$. □

The following lemma shows that *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. $\forall t \in \mathrm{todo}^F(\vec{l})\ \forall j \leq |\vec{l}|\ (l_j)_\Omega \# t$.

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

3. $\forall \sigma \in \mathrm{subs}(\vec{l}, m)\ \forall j \leq |\vec{l}|\ l_j \#_s^{\mathrm{Arg}} m^\sigma$.

Proof

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

   Case $|\vec{l}| = 0$. Trivial.

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

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

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

2. Let $t \in \mathrm{new}(\vec{l}, m)$. Then $t = (m)_\Omega \sqcup t'$ with $t' \in \mathrm{todo}^F(\vec{l})$. Thus we have $(l_j)_\Omega \# t'$ $(j \leq |\vec{l}|)$. The result follows with properties of $\sqcup$ in Lemma 5.3.12.

3. Let $\sigma \in \mathrm{subs}(\vec{l}, m)$. Then $\sigma = \mathrm{mgu}(m, \mathrm{fresh}(\mathrm{var}(m), u))$ for some $u \in \mathrm{new}(\vec{l}, m)$. Thus $(l_j)_\Omega \# u$ $(j \leq |\vec{l}|)$. Therefore $l_j \#_s^{\mathrm{Arg}} u$, and $l_j \#_s u^\sigma = m^\sigma$ $(j \leq |\vec{l}|)$. □

The following lemma shows that *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 $(\mathcal{D}, \mathcal{C}, R)$ 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. Let t be an application of F to terms.*

1. $(\forall j \leq |\vec{l}|\ l_j \#_s^{\mathrm{Arg}} t) \Rightarrow \exists u \in \mathrm{todo}^F(\vec{l})\ t \succeq u$.
2. $(\forall j \leq |\vec{l}|\ l_j \#_s^{\mathrm{Arg}} m^\sigma) \Rightarrow \exists u \in \mathrm{new}(\vec{l}, m)\ m^\sigma \succeq u$, *for any substitution σ.*
3. $(\forall j \leq |\vec{l}|\ l_j \#_s^{\mathrm{Arg}} t) \wedge (\exists \tau\ m^\tau = t) \Rightarrow \exists \sigma \in \mathrm{subs}(\vec{l}, m)\ \exists \sigma'\ (m^\sigma)^{\sigma'} = t$.

Proof

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

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

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

   Either $(l_{i+1})_\Omega \# u$ *and* $u \in td^F(\vec{l})$. Then $u \succeq u'$ for some $u' \in \mathrm{todo}^F(l_1, \ldots, l_{i+1})$. Thus also $t \succeq u'$.

   Or $(l_{i+1})_\Omega \uparrow u$. As $l_{i+1} \#_s^{\mathrm{Arg}} t$ we have $t \succeq w$ for some $w \in \mathrm{IncLhs}(l_{i+1})$. Therefore we have $w \uparrow u$ and thus $w \sqcup u \in td^F(\vec{l})$. Thus we have $w \sqcup u \succeq t'$ for some $t' \in \mathrm{todo}^F(l_1, \ldots, l_{i+1})$. Moreover we have $t \succeq w \sqcup u$ and thus $t \succeq t'$.

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

3. Assume $l_j \#_s^{\mathrm{Arg}} t$ $(j \leq |\vec{l}|)$ and $m^\tau = t$. Then $m^\tau \succeq u$ for some $u \in \mathrm{new}(\vec{l})$. Let $w = \mathrm{fresh}(\mathrm{var}(m), u)$. Then we have $m^{\tau'} = t = w^{\tau'}$ for some substitution τ', because $\mathrm{var}(m) \cap \mathrm{var}(w) = \emptyset$. By definition we have $(m^{\mathrm{mgu}(m,w)})^{\sigma'} = m^\tau = t$ for some σ' and $\mathrm{mgu}(m, w) \in \mathrm{subs}(\vec{l}, m)$. □

We will use the definition of *subs* to define a function 'newrules', that given a sequence $\vec{l}$ 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 $(\mathcal{D},\mathcal{C},R)$ be an S-sorted Constructor System. Let $F \in \mathcal{D}$. Let $l_1 \rightarrow r_1, \ldots, l_k \rightarrow r_k, m \rightarrow t$ be F-rules.
$\text{newrules}(\langle l_1 \rightarrow r_1, \ldots, l_k \rightarrow r_k\rangle,\ m \rightarrow t) = \{m^\sigma \rightarrow t^\sigma | \sigma \in \text{subs}(\langle l_1, \ldots, l_k\rangle, m)\}$.

Example 5.3.25 Let $(\mathcal{D},\mathcal{C},R,>)$ 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(ϵ, `Eq(0,0)` $\rightarrow$ `True`)= {`Eq(0,0)` $\rightarrow$ `True`};
> newrules($\langle$`Eq(0,0)` $\rightarrow$ `True`$\rangle$, `Eq(S(w),S(x))` $\rightarrow$ `Eq(w,x)`) =
> {`Eq(S(a),S(b))` $\rightarrow$ `Eq(a,b)`};
> newrules($\langle$`Eq(0,0)` $\rightarrow$ `True`,`Eq(S(w),S(x))` $\rightarrow$ `Eq(w,x)`$\rangle$, `Eq(y,z)` $\rightarrow$ `False`) =
> {`Eq(S(c),0)` $\rightarrow$ `False`,`Eq(0,S(d))` $\rightarrow$ `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 $\vec{l}$ 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 $(\mathcal{D},\mathcal{C},R)$ be an S-sorted Priority Constructor System. Let $F \in \mathcal{D}$. Let $l_1 \rightarrow r_1, \ldots, l_{k+1} \rightarrow r_{k+1}$ be F-rules.*

1. $\forall l \rightarrow r \in \text{newrules}(\langle l_1 \rightarrow r_1, \ldots, l_k \rightarrow r_k\rangle, l_{k+1} \rightarrow r_{k+1})\ \forall j \leq k\ l_j \#_s^{\text{Arg}} l.$
2. *If $l_j \#_s^{\text{Arg}} t$, for all $j \leq k$, and $l_{k+1}^\sigma = t$ then t is an r-redex for some* $r \in \text{newrules}(\langle l_1 \rightarrow r_1, \ldots, l_k \rightarrow r_k\rangle, l_{k+1} \rightarrow r_{k+1})$.

Proof

1. Let $r \in \text{newrules}(\langle l_1 \rightarrow r_1, \ldots, l_k \rightarrow r_k\rangle, l_{k+1} \rightarrow r_{k+1})$. By definition, for some $\sigma \in \text{subs}(\langle l_1, \ldots, l_k\rangle, l_{k+1})$ $r = l_{k+1}^\sigma \rightarrow r_{k+1}^\sigma$. Thus $\forall j \leq k\ l_j \#_s^{\text{Arg}} l_{k+1}^\sigma$ by Lemma 5.3.22.
2. Assume $\forall j \leq k\ l_j \#_s^{\text{Arg}} t$ and $\exists \sigma\ l_{k+1}^\sigma = F(t_1, \ldots, t_n)$. By Lemma 5.3.23 we have $(l_{k+1}^\tau)^{\tau'} = F(t_1, \ldots, t_n)$, for some $\tau \in \text{subs}(\langle l_1, \ldots, l_k\rangle, l_{k+1})$, and substitution τ'. By definition we have $l_{k+1}^\tau \rightarrow r_{k+1}^\tau \in \text{newrules}(\langle l_1, \ldots, l_k\rangle, l_{k+1})$ and $F(t_1, \ldots, t_n)$ is an $l_{k+1}^\tau \rightarrow r_{k+1}^\tau$-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 $(\mathcal{D},\mathcal{C},R,>)$ be an S-sorted Priority Constructor System with finitely many rules.

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

2. $\text{transform}(\mathcal{D}, \mathcal{C}, R, >) = (\mathcal{D}, \mathcal{C}, \bigcup_{r \in R} \text{newrules}(\text{precrules}(R, >, r), r))$.

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`:

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

Using the results of the previous example we get:

$$\begin{array}{l}\text{transform}(\mathcal{D}, \mathcal{C}, R, >) = (\mathcal{D}, \mathcal{C},\\ \{\texttt{Eq(0,0)} \rightarrow \texttt{True}, \texttt{Eq(S(a),S(b))} \rightarrow \texttt{Eq(a,b)},\\ \texttt{Eq(S(c),0)} \rightarrow \texttt{False}, \texttt{Eq(0,S(d))} \rightarrow \texttt{False}\}).\end{array}$$

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* σ *be a substitution.*
$l_1^\sigma \rightarrow_e r_1^\sigma \Leftrightarrow \exists l_2 \rightarrow r_2 \in \text{newrules}(\text{precrules}(R, >, l_1 \rightarrow r_1), l_1 \rightarrow r_1)\ \exists \tau\ l_1^\sigma = l_2^\tau \wedge r_1^\sigma = r_2^\tau$.

Proof Let $l_1 \rightarrow r_2 \in R$. Let σ be a substitution.

$\Rightarrow$. Assume $l_1^\sigma \rightarrow_e r_1^\sigma$. Then we have $l' \#_s^{\text{Arg}} l_1^\sigma$ 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^σ is an $l_2 \rightarrow r_2$-redex 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 $l_2 \rightarrow r_2 = l_1^\tau \rightarrow r_1^\tau$ for some substitution τ.

$\Leftarrow$. Assume $l_1^\sigma = l_2^\tau$ and $r_1^\sigma = r_2^\tau$ 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 $l' \#_s^{\text{Arg}} l_2$ and thus $l' \#_s^{\text{Arg}} l_2^\tau$, for all $l' \rightarrow r' \in R$. By the definition of newrules we have $l_2 \rightarrow r_2 = l_1^{\tau'} \rightarrow r_1^{\tau'}$ for some substitution τ'. Thus we have $l_1^\sigma = (l_1^{\tau'})^\tau \rightarrow_e (r_1^{\tau'})^\tau = r_1^\sigma$. □

Theorem 5.3.30 (reduction preservation) *Let* $(\mathcal{D}, \mathcal{C}, R, >)$ *be an S-sorted Priority Constructor System with finitely many rules. Then* $\text{transform}(\mathcal{D}, \mathcal{C}, R, >)$ *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 $\text{PCS}(\mathcal{D}, \mathcal{C}, R, >)$ 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 $\text{transform}(\mathcal{D}, \mathcal{C}, R, >)$. □

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 Σ an S-sorted signature. Let R be a set rewrite rules over Σ 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 $\langle$`True`, `False`$\rangle$ (as `Eq(0,0)` is a redex for the first and third rule), and $\langle$`Eq(x,y)`, `False`$\rangle$ (as `Eq(S(x),S(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.

```
sort  bool
cons  True:                      bool
      False:                     bool
def   Choice:  bool × bool →     bool
rule  Choice(False,x)   → False
      Choice(y,True)    → True
```

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 Σ be an S-sorted signature. Let $l_1, l_2 \in \mathcal{T}(\Sigma, V)$. Let σ be a substitution.*

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

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

Proof

1. Assume τ is a unifier for l_1^σ and l_2. We define a substitution σ' as follows:
$\sigma'(v) = \begin{cases} v & (\text{if } v \in \mathrm{var}(l_2)) \\ \sigma(v) & (\text{otherwise}) \end{cases}$. We have $l_1^\sigma = l_1^{\sigma'}$ and $l_2^{\sigma'} = l_2$.
Thus $\tau \circ \sigma'$ is a unifier for l_1 and l_2.

2. Assume that (possibly after renaming variables in $l_2 \to r_2$) $\mathrm{var}(l_1^\sigma) \cap \mathrm{var}(l_2) = \emptyset = \mathrm{var}(l_1) \cap \mathrm{var}(l_2)$. Assume l_1^σ and l_2 are unifiable. We define a substitution σ' as follows: $\sigma'(v) = \begin{cases} v & (\text{if } v \in \mathrm{var}(l_2)) \\ \sigma(v) & (\text{otherwise}) \end{cases}$. We have $l_1^\sigma = l_1^{\sigma'}$, $r_1^\sigma = r_1^{\sigma'}$, $l_2^{\sigma'} = l_2$, and $r_2^{\sigma'} = r_2$. Let τ be a unifier for l_1^σ 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 τ is a unifier for r_1^σ and r_2. □

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 $(\mathcal{D}, \mathcal{C}, R, >)$ be an S-sorted* PCS *with finitely many rules. If $>$ prevents conflicts then* transform$(\mathcal{D}, \mathcal{C}, R, >)$ *is weakly orthogonal, and hence confluent.*

Proof
Assume PCS$(\mathcal{D}, \mathcal{C}, R, >)$ is finite, and $>$ prevents conflicts. Let $(\mathcal{D}, \mathcal{C}, R') =$ transform$(\mathcal{D}, \mathcal{C}, R, >)$. By the theorem $(\mathcal{D}, \mathcal{C}, R')$ 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 $\langle r_1^\sigma, r_2^\sigma \rangle$ is a critical pair obtained from unification of the left-hand sides of two rules $l_1 \to r_1, l_2 \to r_2 \in R'$. By definition there are rules $l_3 \to r_3, l_4 \to r_4 \in R$, such that $l_1 \to r_1 \in$ newrules(precrules$(R, >, l_3 \to r_3), l_3 \to r_3)$ and $l_2 \to r_2 \in$ newrules(precrules$(R, >, l_4 \to r_4), l_4 \to r_4)$. By Lemma 5.3.33 l_3 and l_4 are unifiable. As $>$ prevents conflicts we either have $r_3^\tau = r_4^\tau$ ($\tau = \mathrm{mgu}(l_3, l_4)$), or $l_3 \to r_3$ and $l_4 \to r_4$ are comparable by $>$. In the first case we have $r_1^\sigma = r_2^\sigma$ by Lemma 5.3.33. In the second case by Lemma 5.3.7 and 5.3.26 we obtain $l_1 \#_s^{\mathrm{Arg}} l_2$ which contradicts the unifiability of l_1 and l_2. □

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 Ω-term of the left-hand side of a left-linear rule or is a redex for that rule.

Lemma 5.3.35 *Let Σ 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 $t_\Omega = t = fresh(W, t)$.*
2. *If $u \succeq t$ then $u = t$.*
3. *If $t \uparrow u$ then $t \succeq u$.*
4. *If u is linear and $t \uparrow u_\Omega$ then $u^\sigma = t$, for some substitution σ.*

Proof

1. Because t does not contain variables we have $t_\Omega = t$, and as t does not contain Ω we have $\text{fresh}(W, t) = t$.

2. Induction on the proof of $u \succeq t$.

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

   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. □

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 σ, 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 $(l')^\tau \neq F(t_1, \ldots, t_n)$ for all substitutions τ. Thus by Lemma 5.3.35 $l'_\Omega \# F(t_1, \ldots, t_n)$ and by Lemma 5.3.12 and 5.3.13 we have $l' \#_s^{\text{Arg}} F(t_1, \ldots, t_n)$. 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$(\mathcal{D}, \mathcal{C}, R, >)$. □

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 $(\mathcal{D}, \mathcal{C}, R)$. □

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 Σ be a finite S-sorted algebraic data type-signature with function symbols $\mathcal{F}$ and typing function $\tau : \mathcal{F} \longrightarrow S^+$, that is specified using the method of Example 2.2.4. The sets $V, \mathcal{C}, S, \mathcal{F}$ are assumed to be pairwise disjoint. The syntax of *pseudo terms* $\mathcal{T}_\pi$ is defined as follows:

$$\mathcal{T}_\pi = V \mid \mathcal{C} \mid S \mid \mathcal{F} \mid (\mathcal{T}_\pi\ \mathcal{T}_\pi) \mid \lambda V{:}\mathcal{T}_\pi.\mathcal{T}_\pi \mid \Pi V{:}\mathcal{T}_\pi.\mathcal{T}_\pi$$

The constants in the set $\mathcal{C}\backslash$Universes will be used as names of function definitions. We assume that this set is split into disjoint infinite subsets $\mathcal{C}_s$ for each type universe mbox$s \in$ Universes. From now on we will assume that S contains the sorts Bool, Nat, List, and that Σ 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:

sort	Bool, Nat, List		
func	True:		Bool
	False:		Bool
	0:		Nat
	S:	Nat $\longrightarrow$	Nat
	Nil:		List
	Cons:	Nat $\times$ List $\longrightarrow$	List

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 β-reduction for pseudo terms ($\in \mathcal{T}_\pi$) is defined as follows:

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

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 $f \notin V$.

2. The set of *rules* $\mathcal{R}$ is the smallest set such that: if $F \in \mathcal{C}$, $p_1, \ldots, p_n \in \mathcal{P}$, $t \in \mathcal{T}_\pi$, 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 $l_2 \Rightarrow r_2$ 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 $y \notin FV(l_1) \cup BV(r_1)$. 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 $F \in \mathcal{C}$, $t \in \mathcal{T}_\pi$, $\vec{r}$ 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 Leq:

$$\boxed{\begin{array}{ll} \multicolumn{2}{l}{\mathsf{Leq{:}Nat} \to \mathsf{Nat} \to \mathsf{Bool} =} \\ \mathsf{Leq\ O}\ y & \Rightarrow \mathsf{True}, \\ \mathsf{Leq\ (S}\ x\mathsf{)\ O} & \Rightarrow \mathsf{False}, \\ \mathsf{Leq\ (S}\ x\mathsf{)\ (S}\ y\mathsf{)} & \Rightarrow \mathsf{Leq}\ x\ y \end{array}}$$

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, Leq (S O) O=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 $\mathcal{X}_\pi$ of *pseudo contexts* contains all finite sequence of context items.

3. Let $\Gamma \in \mathcal{X}_\pi$ be a pseudo context. The set *fundefs*(Γ) contains the function definitions of Γ, and the set *consts*(Γ) contains the defined constants of the function definitions in fundefs(Γ). By $FV(\Gamma)$ we denote the set of subjects of the variable declarations of Γ.

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 $v_i \in V, t_i \in \mathcal{T}_\pi$. The substitution sequence $\vec{\sigma}$ is *standard* if $FV(t_i) \cap \{v_1, \ldots, v_n\} = \emptyset$ 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) $\text{subst}(t, \epsilon) = t$.

   (b) $\text{subst}(t, \langle (v_1, u_1), \vec{\sigma} \rangle) = \text{subst}(t[v_1 := u_1], \vec{\sigma})$.

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 $l \Rightarrow t \in \mathcal{R}$. 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 $u_1, u_2 \in \mathcal{T}_\pi$.

2. We define a ternary relation $\rightarrow_{\pi_0}$ on a pseudo context and two pseudo terms as: $\Gamma_1, F{:}t = \vec{r}, \Gamma_2 \vdash C[u_1] \rightarrow_{\pi_0} C[u_2]$, if $u_1 \Rightarrow u_2$ is an r_i-reduction step, $C[\,] \in \mathcal{H}_\pi$.

Example 6.1.14 Let Δ_{Leq} be the function definition of Example 6.1.10.
We have $\Delta_{\mathsf{Leq}} \vdash \mathsf{Leq}\ (\mathsf{S}\ \mathsf{O})\ (\mathsf{S}(\mathsf{S}\ \mathsf{O})) \rightarrow_{\pi_0} \mathsf{Leq}\ \mathsf{O}\ (\mathsf{S}\ \mathsf{O})$. But *not* $\epsilon \vdash \mathsf{Leq}\ \mathsf{O}\ n \rightarrow_{\pi_0} \mathsf{True}$, as the empty context does not contain a definition for Leq.

Remark 6.1.15 If $t_1 \Rightarrow t_2$ is a $l \Rightarrow r$-reduction step and $FV(t_1) \cap FV(l) = \emptyset$ then there exists a standard substitution sequence σ 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 $FV(t_1) \cap (FV(l') \cup BV(r')) = \emptyset$.