Pol, Jan Cornelis van de

Termination of Higher-order Rewrite Systems /Jan Cornelis van de Pol
- Utrecht: Universiteit Utrecht, Faculteit Wijsbegeerte.
- (Quaestiones infinitae, ISSN 0927-3395; volume 16)
- Proefschrift Universiteit Utrecht.
- Met literatuur opgave.
- Met samenvatting in het Nederlands.
ISBN 90-393-1357-1

Termination of Higher-order Rewrite Systems

Terminatie van hogere-orde herschrijfsystemen

(met een samenvatting in het Nederlands)

Proefschrift ter verkrijging van de graad van doctor
aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. dr. J.A. van Ginkel
ingevolge het besluit van het College van Decanen
in het openbaar te verdedigen
op woensdag 11 december 1996 des ochtends om 10.30 uur
door

Jan Cornelis van de Pol

geboren op 6 april 1969, te Barneveld

promotor: Prof. Dr. J.A. Bergstra (Faculteit der Wijsbegeerte)
co-promotor: Dr. M.A. Bezem (Faculteit der Wijsbegeerte)

Een deel van het onderzoek werd verricht aan het Mathematisches Institut van de Ludwig–Maximilians–Universität te München. Dit betreft de Hoofdstukken 5.3–5.5, de aanzet tot Hoofdstuk 6 en de Appendix. Dit deel werd gefinancierd door de Europese Unie als Science Twinning Contract SC1*–CT91–0724.

Preface

This thesis has not been written in isolation; it could not have been. I needed nice people to be friends with, to chat to, to listen to, to learn from and to become inspired by.

I am indebted to my supervisors Jan Friso Groote, who initiated my research, and Marc Bezem, who read my papers and directed me during writing this thesis with many valuable remarks and wise lessons. He also is my co-promotor. I am very grateful to my promotor, Jan Bergstra. His influence is not measurable, but large. He was always willing to give advice, asked and unasked. He stimulated and enabled various escapades outside my specialistic research and forced me to think about "what next".

I thank all the other colleagues at the Department of Philosophy of the Utrecht University, both the scientific and administrative staff and the system managers. I especially mention my old roommates Jan Springintveld and Alex Sellink, as well as my currently most direct colleagues, Inge Bethke, Wan Fokkink, Marco Hollenberg, Kees Vermeulen, Albert Visser and Bas van Vlijmen.

I was given to spend a considerable amount of time at the Mathematical Institute of the Munich University. My host, Helmut Schwichtenberg, enabled a very intensive research. He had great influence on the direction of this thesis and was a pleasant co-author. Special attention deserves Ulrich Berger for his friendly hospitality and many useful technical and non-technical discussions. With great pleasure I recall my roommate Robert Stärk and all the other colleagues, that without exception tolerated my poor German. Felix Joachimski, Ralph Matthes and Karl-Heinz Niggl commented on parts of the text.

The members of the reading committee, Dirk van Dalen, Jan Willem Klop, Vincent van Oostrom, Helmut Schwichtenberg and Hans Zantema, were as kind as to read the manuscript of this thesis, and to give their judgement and comments.

Hans Zantema also was my teacher in term rewriting and the supervisor of my Master's thesis. I learned much from him and will always see him amidst green foliage, in his office with the door always wide open.

Finally, I am grateful to my family. My parents and grandmother for always supporting my study with their interest. Corrie, my wife, for her good-humored love and for giving me Ella and Jan. For all these gifts I thank God.

Daarom kwelt het verstand zich bij het onderzoeken van overtollige en nietswaardige dingen met een belachelijke nieuwsgierigheid.

JOHANNES CALVIJN, INSTITUTIE II, II, 12

Deze moeilijke bezigheid heeft God de kinderen der mensen gegeven om zich daarin te bekommeren.

PREDIKER I, 13

Contents

Chapter 1

Introduction

Rewriting and Termination

The word *rewriting* suggests a process of computation. Typically, the objects of computation are syntactic expressions in some formal language. A rewrite system consists of a collection of rules (the program). A computation step is performed by replacing a part of an expression by another expression, according to the rules. The resulting expression may be rewritten again and again, giving rise to a reduction sequence. Such a sequence can be seen as a computation.

Certain expressions are considered as results, or *normal forms*. A computation terminates successfully when such a normal form has been reached. This situation can be recognized by the fact that no rewrite rule is applicable. It may also happen that a computation does not end in a normal form, but goes on and on forever. This is unfortunate, because such computations yield no result.

Suppose that a computation runs on a computer for a long time without yielding a result. In that case, it may be difficult to decide whether it is wise to wait just a bit longer, or to abort the computation by switching the computer off. Therefore, given a rewrite system it is an interesting question whether its rules admit infinite computations, or not. We call a rewrite system *terminating* if all reduction sequences supported by it are finite. In general, it is difficult to prove termination of rewrite systems. The example above already shows that termination cannot be decided in finite time by just testing the program.

This whole thesis is devoted to the termination problem for a particular kind of rewrite systems, namely higher-order rewrite systems. These are a particular combination of first-order term rewriting systems and lambda calculus. Before these systems will be described in more detail, some application areas of rewriting are identified and the importance of termination in these areas is explored.

Rewrite systems have been recognized as a fundamental tool in both computer science and logic. The applications below have in common that a rewrite system is used to transform certain expressions into *equivalent* expressions of a nice form. The

rewrite rules can be understood as directed equations. As equations they ensure that the initial expression and the result of the computation are equivalent. The rules are directed, because one form is preferred to another one. Rewrite systems are used for the following purposes:

- Prototyping functional languages;
- Describing transformations on programs;
- Implementing abstract data types;
- Automated theorem proving, especially for equational logic;
- Proving completeness of axiomatizations for algebras;
- Proving consistency of proof calculi for logics.

These tasks can be divided into practical and theoretical applications. In theoretical applications it is enough to know that in principle, expressions can be reduced to normal form. The practical tasks are performed by actually computing such a normal form. For some applications, it is required that the normal form is unique.

We now describe certain desirable properties of rewrite systems. To this end, the following notation is convenient. For objects s and t, we write $s \to t$ if s can be rewritten to t in one step (t need not be unique). If s rewrites to t in zero, one or more steps, we say that s *reduces* to t and we write $s \twoheadrightarrow t$.

We already encountered termination. A rewrite system is *terminating* if all rewrite sequences $s_0 \to s_1 \to s_2 \to \cdots$ are finite. Termination is often called *strong normalization.* A rewrite system is *weakly normalizing* if for all objects s there exists a normal form t, such that $s \twoheadrightarrow t$. This is not equivalent to termination, because in a weakly normalizing rewrite system infinite computations may exist too. Clearly, strong normalization implies weak normalization.

Another important question is, whether the normal form of an expression is uniquely determined. Weak normalization still admits that a term reduces to two different normal forms. A rewrite system is *confluent* if for all objects r, s, t such that both $r \twoheadrightarrow s$ and $r \twoheadrightarrow t$, there exists a u such that $s \twoheadrightarrow u$ and $t \twoheadrightarrow u$. In words: two diverging computations can always flow together. In a confluent and weakly normalizing rewrite system, every object is guaranteed to have a unique normal form.

A rewrite system is *locally confluent* if for all objects r, s, t such that both $r \to s$ and $r \to t$, there exists a u such that $s \twoheadrightarrow u$ and $t \twoheadrightarrow u$. The difference with confluence is that the common successor is only guaranteed after a one step divergence. Local confluence is weaker than confluence and does not imply uniqueness of normal forms. But due to its local nature it is easier to detect.

The question arises why termination is an issue. Is weak normalization not sufficient? First of all, termination implies weak normalization, so its importance is inherited from weak normalization. As advantages of the latter we mention:

- For programs, weak normalization guarantees that a result exists.

- For function definitions, weak normalization is needed for totality of the function.
- For a decision procedure of equational logic, weak normalization guarantees that both sides of a true equation can be brought into normal form.
- In completeness and consistency proofs, weak normalization guarantees that every formula or proof can be transformed into an equivalent one of a certain nice shape.

In fact, weak normalization is not sufficient in practical applications. In order to actually compute a normal form, a *strategy* is needed in addition, which in each situation prescribes which step must be chosen next. Without a normalizing strategy the normal form can be missed by getting involved in an infinite computation. It is tempting to see the strategy as part of the rules. In that view, the rewrite system becomes strongly normalizing. As arguments in favor of proving termination we mention:

- Termination implies weak normalization;
- A terminating rewrite system doesn't need a strategy, because there is no danger of infinite computations;
- Termination and local confluence together imply confluence (local confluence is often easier to prove than confluence);
- Termination means that the rewrite relation is well-founded, which yields a strong induction principle.

This provides practical and theoretical evidence that termination is an interesting property. After all, it is quite natural to ask whether *all* reduction sequences eventually lead to a normal form.

Higher-order Rewrite Systems

Higher-order rewrite systems combine first-order term rewriting systems and simply-typed lambda calculus in a special way. We first introduce the latter two formalisms. The formalisms are characterized by the objects of computation.

Term rewriting systems. In term rewriting, the objects are *first-order terms*. Such terms are built from a number of function symbols, each expecting a fixed number of arguments. Consider the symbols $\{0, s, a, q\}$, where 0 (zero) has no arguments, s (successor) and q (square) are unary function symbols, and a (addition) is binary. Using these symbols, we can built the natural numbers, e.g. 3 is represented by the term $s(s(s(0)))$, because 3 is the third successor of 0. We can also form more complex terms, like $q(a(s(0), q(s(s(0)))))$, which is interpreted as $(1 + 2^2)^2$.

Furthermore, a term may contain variables, which are place holders for arbitrary terms. The fact that variables range over terms only — and not over e.g. function symbols — explains the adjective *first-order*.

So far, a and q are idle symbols. By giving rewrite rules, we can make them compute. Consider the following rewrite system:

$$\text{(I)} \qquad \begin{cases} a(X,0) & \mapsto & X \\ a(X,s(Y)) & \mapsto & s(a(X,Y)) \\ q(0) & \mapsto & 0 \\ q(s(X)) & \mapsto & s(a(q(X),a(X,X))) \end{cases}$$

Here X and Y are variables, representing arbitrary terms. If a certain term contains a subterm that matches the left-hand side of one of these rules, then that subterm can be replaced by the corresponding instance of the right-hand side. This constitutes one rewrite step. The subterm that is replaced is called the redex. As an example we show that $2^2 \twoheadrightarrow 4$. In each step, the redex has been underlined:

$$\begin{array}{rcl} \underline{q(s(s(0)))} & \to & s(a(q(s(0)),\underline{a(s(0),s(0))})) \\ & \to & s(a(q(s(0)),s(\underline{a(s(0),0)}))) \\ & \to & s(\underline{a(q(s(0)),s(s(0)))}) \\ & \to & s(s(\underline{a(q(s(0)),s(0))})) \\ & \to & s(s(s(\underline{a(q(s(0)),0)}))) \\ & \to & s(s(s(\underline{q(s(0))}))) \\ & \to & s(s(s(s(\underline{a(q(0),\underline{a(0,0)})})))) \\ & \to & s(s(s(s(\underline{a(q(0),0)})))) \\ & \to & s(s(s(s(\underline{q(0)})))) \\ & \to & s(s(s(s(0)))) \end{array}$$

The latter term contains no redex, so it is a normal form.

There are two limitations of first-order term rewriting that we wish to overcome. The first one is that there are no variables for function symbols. Functional programming languages show that this would be a convenient construction. Consider e.g. the following definition of *twice*, which applies its first argument (a function!) on the second one twice.

$$\text{(II)} \qquad \mathit{twice}(F,X) \quad \mapsto \quad F(F(X))$$

The other limitation is that first-order terms don't support the construct of bound — or local — variables. This feature exists in most programming languages and also in proofs and formulae of predicate logic. Bound variables occur quite naturally, e.g. in the following rules, which define the sum of a certain expression in i, for all $0 \le i \le n$:

$$\text{(III)} \qquad \begin{cases} \sum_{i \le 0} E & \mapsto & E[i := 0] \\ \sum_{i \le s(n)} E & \mapsto & a(\sum_{i \le n} E, E[i := s(n)]) \end{cases}$$

Here E denotes an expression that may contain i, and $E[i := n]$ denotes the same expression in which n is substituted for all occurrences of i. As reduction sequence we could have for instance:

$$\begin{array}{lcl} \sum_{i \leq s(0)} s(i) & \to & a(\sum_{i \leq 0} s(i), s(s(0))) \\ & \to & a(s(0), s(s(0))) \\ & \to & s(a(s(0), s(0))) \\ & \to & s(s(a(s(0), 0))) \\ & \to & s(s(s(0))) \end{array}$$

Simply-typed lambda calculus. We now describe lambda calculus, which is a calculus of functions and has a notion of bound variables. The two basic operations to construct lambda terms are function application and lambda abstraction. A function can be *used* by applying it to an argument. Application is written as juxtaposition. A function can be *introduced* by giving a law for it. The function that maps x to $E(x)$ is written $\lambda x.E(x)$. Pure lambda calculus has no function symbols. Variables are place holders for arbitrary functions. E.g. the term $\lambda x.(xy)$ represents the function that takes an x and yields (xy), i.e. x applied to y. This is quite different from $\lambda y.(xy)$, which takes an arbitrary y and applies the function x to it. The function *twice* can be represented by the lambda term $\lambda f.\lambda x.f(fx)$.

The lambda calculus has only one rewrite rule, called the β-rule. This rule expresses that applying a function $\lambda x.M$ to an argument N yields M, in which all occurrences of x are replaced by N. In symbols:

$$\text{(IV)} \qquad (\lambda x.M)N \quad \mapsto \quad M[x := N]$$

In the *simply-typed* lambda calculus that we consider, term formation is subject to type restrictions. Types are constructed from the base type o, by repeatedly applying the binary type constructor $\to$. Typically, o denotes the set of natural numbers, and $\sigma \to \tau$ the set of functions from σ into τ. In this way a hierarchy of functions is introduced. Functions of type $o \to o$ act on basic objects; higher-order functions (also called functionals) act on functions of a lower type. The typing rules ensure that functions of type $\sigma \to \tau$ can be applied only to functions of type σ, yielding a result of type τ.

As an example, let x, z be variables of type o, f of type $o \to o$ and g of type $o \to o \to o$. Then omitting a number of parentheses, $(\lambda f.f(fx))(\lambda z.gzz)$ is a term of type o, which reduces to $g(gxx)(gxx)$:

$$\begin{array}{lcl} (\lambda f.f(fx))(\lambda z.gzz) & \to & (\lambda z.gzz)((\lambda z.gzz)x) \\ & \to & (\lambda z.gzz)(gxx) \\ & \to & g(gxx)(gxx) \end{array}$$

Simply-typed lambda calculus can be combined with a term rewriting system, by giving the function symbols of the latter a type, e.g. $0 : o$; $q, s : o \to o$; $a : o \to o \to o$. The rule for *twice* can now also be incorporated, by giving *twice* the type $(o \to o) \to$

$o \to o$. Using the rules introduced so far, it can be verified that in the combined system, $\mathit{twice}(\lambda x.s(a(x,x)),0) \twoheadrightarrow s(s(s(0)))$.

In this combined system, we have both named and nameless functions. Function definitions can use both pattern matching (inherited from term rewriting) and parameter passing (inherited from lambda calculus). Therefore, this is the basis for a powerful programming language. Nevertheless, the rules for the $\sum$-operator are not well-formed in this system, because they contain a new binder. To amend this, we will use higher-order rewrite systems.

Higher-order rewrite systems. In higher-order rewrite systems, the λ remains the only binder. The key observation is that other binders, like the $\sum$-operator, can be represented by a higher-order function symbol. Assign the type $o \to (o \to o) \to o$ to $\sum$. Then we can write e.g. $\sum(n, \lambda i.a(i,i))$, which in the previous notation reads $\sum_{i \le n} a(i,i)$.

Thus, the objects of a higher-order rewrite system are lambda terms that may contain function symbols of any type. The lambda calculus also takes care of substitution, and technical matters like the scope of local variables. Instead of $E[i := n]$, we can now write $(\lambda i.E)n$; the actual substitution can be performed by a β-rewrite step. The translated sum-rules now read:

$$\text{(V)} \qquad \begin{cases} \sum(0, \lambda i.F(i)) & \mapsto & F(0) \\ \sum(s(n), \lambda i.F(i)) & \mapsto & a(\sum(n, \lambda i.F(i)), F(s(n))) \end{cases}$$

In higher-order rewriting, it is ensured that β-reduction is performed immediately, after applying a rule. In technical terms, this means that the rewrite relation is generated *modulo* β. We clarify this with an example. In the plain combination of rewriting and β-reduction, we would have the following reduction sequence:

$$\begin{array}{lll} \sum(s(0), \lambda i.s(i)) & \to & a(\sum(0, \lambda i.s(i)), (\lambda i.s(i))s(0)) \\ & \to_\beta & a(\sum(0, \lambda i.s(i)), s(s(0))) \\ & \to & a((\lambda i.s(i))0, s(s(0))) \\ & \to_\beta & a(s(0), s(s(0))) \\ & \to & s(a(s(0), s(0))) \\ & \to & s(s(a(s(0), 0))) \\ & \to & s(s(s(0))) \end{array}$$

By rewriting modulo β, the β-steps are performed immediately, so they become invisible. We then get the following reduction sequence, which corresponds better with the sequence displayed earlier in connection with the rules (III):

$$\begin{array}{lll} \sum(s(0), \lambda i.s(i)) & \to & a(\sum(0, \lambda i.s(i)), s(s(0))) \\ & \to & a(s(0), s(s(0))) \\ & \to & s(a(s(0), s(0))) \\ & \to & s(s(a(s(0), 0))) \\ & \to & s(s(s(0))) \end{array}$$

Substitution only occurs in the right-hand side of the sum-rules. This corresponds with invisible β-reductions. Substitution may also occur on the left-hand side of the higher-order rules, which corresponds with invisible β-expansions. Because of the possibility of β-expansions, higher-order rewriting is more complex than the plain combination of lambda calculus and term rewriting.

In the following chapters, several examples of higher-order rewriting occur. It appears that the lambda calculus itself can be understood as a higher-order rewrite system. The procedure to find the prenex-normal form of first-order formulae can be viewed as a higher-order rewrite system. Also proof normalization in arithmetic based on natural deduction can be seen as a higher-order rewrite system.

As a merit of the complex formalism of higher-order rewriting, we see that it has all the above mentioned systems as instances. This makes it possible to study general notions, like termination, in a common framework.

The Semantical Approach to Termination

Why does the symbol a mean addition, s the successor function, q the square and so on? This particular meaning is supported by the fact, that we get true equations if we *interpret* the rules in this way. Writing $[\![t]\!]$ for the interpretation of t, we get for instance:

$$[\![a(x, s(y))]\!] = x + (1 + y) = 1 + (x + y) = [\![s(a(x, y))]\!].$$

In other words, addition, successor and so on form a *model*. Of course, there may be more models. Similarly, lambda calculus is about functions, because the β-rule is true in the model of functions.

What we will propose, is to use a variant of this semantics in termination proofs. Instead of a model in which the rules correspond to true equalities, we look for a termination model, in which the rules are true *inequalities*. This is not a new idea; what is new, is that we make this technique available for higher-order rewrite systems. This appears to be a non-trivial extension of similar methods in first-order term rewriting and lambda calculus.

The semantical method will be supported by a theorem, which states that if a higher-order rewrite system has a termination model, then it is a terminating system. For first-order term rewriting, a termination model has the following ingredients:

1. A set equipped with a well-founded partial order $>$.
2. For each function symbol, a strictly monotonic (= increasing) function on this set. Roughly speaking, f is strictly monotonic if whenever $x > y$ holds, $f(\cdots x \cdots) > f(\cdots y \cdots)$.
3. It must hold that for every rule $l \mapsto r$ in the rewrite system, $[\![l]\!] > [\![r]\!]$.

By (3) and (2), if $s \to t$, then $[\![s]\!] > [\![t]\!]$. Here (2) is needed, because the rule may be used to replace a *part* of s; strict monotonicity of all function symbols guarantees

that the surrounding context preserves the decrease in order. Hence any reduction sequence can be interpreted as a descending chain in $>$ of the same length. By (1), this chain is finite, so the reduction sequence is finite. Therefore, the system is terminating (qed).

We illustrate this with an example. For the term rewriting system (I), defining a and q, we take for (1) the natural numbers ($\mathbb{N}$) with the usual *greater-than* relation. For (2), we put:

$$\begin{array}{rcl} [\![0]\!] & := & 1 \\ [\![s(x)]\!] & := & x+1 \\ [\![a(x,y)]\!] & := & x+2y \\ [\![q(x)]\!] & := & 3x^2 \end{array}$$

These functions are strictly monotonic. Now we verify (3):

$$[\![a(x,0)]\!] = x+2 > x = [\![x]\!]$$

$$[\![a(x,s(y))]\!] = x+2(y+1) > x+2y+1 = [\![s(a(x,y))]\!]$$

$$[\![q(0)]\!] = 3 > 1 = [\![0]\!]$$

$$[\![q(s(x))]\!] = 3(x+1)^2 = 3x^2+6x+3 > 3x^2+6x+1 = [\![s(a(q(x),a(x,x)))]\!]$$

We have found a termination model satisfying (1), (2) and (3), hence the term rewriting system (I) is terminating.

The generalization of termination models to higher-order rewrite systems is quite technical. The natural numbers with the usual greater-than relation can be taken as a partial order on the base type. Terms of type $o \to o$ are interpreted as functions from $\mathbb{N}$ to $\mathbb{N}$. We define $f > g$ to be: For all x, $f(x) > g(x)$. The partial order and the notion *strictly monotonic* have to be generalized to functions of higher types. We also have to deal with function variables. In the termination models these variables range over *weakly* monotonic functions (= non-decreasing). In particular, constant functions are weakly monotonic. For details we refer to Chapter 4 and 5.

As an example, we show how a termination model for the system of sum-rules looks like. We suggest the following interpretation for the sum-symbol:

$$[\![\textstyle\sum(n,f)]\!] := n + 2f(0) + \cdots + 2f(n)$$

We take for granted that this function is strictly monotonic both in f and in n. We can now verify that the rules correspond with a decrease in order.

$$[\![\textstyle\sum(0,\lambda i.f(i))]\!] = 1 + 2f(0) + 2f(1) > f(1) = [\![f(0)]\!]$$

$$\begin{array}{rcl} [\![\sum(s(n),\lambda i.f(i))]\!] & = & n+1+2f(0)+\cdots+2f(n)+2f(n+1) \\ & > & n+2f(0)+\cdots+2f(n)+2f(n+1) \\ & \geq & n+2f(0)+\cdots+2f(n)+2f(n) \\ & = & [\![a(\sum(n,\lambda i.f(i)),f(n))]\!] \end{array}$$

Here $\geq$ holds, because f is at least weakly monotonic. By the main result of this thesis, the calculations above imply that the higher-order rewrite system (V) is terminating.

For the sake of completeness, we mention that putting $[\![twice(F,X)]\!] := F(FX) + X + 1$ yields a termination model for the single-rule system (II). It is well-known that in simply-typed lambda calculus the β-rule (IV) is also terminating. By another result of this thesis, we can conclude that the plain combination of the β-rule with the rule defining *twice* also terminates.

Contents of the Remaining Chapters

In Chapter 2, we formally introduce term rewriting systems, lambda calculus and higher-order rewrite systems. Since these systems have a lot in common, the chapter starts with an introduction to abstract reduction systems. This chapter contains no new results.

Chapter 3 is a quite extensive summary of the semantical approach to termination proofs. We explain how this technique works for term rewriting systems, and for lambda calculus. These are existing techniques. In Section 3.4, it is explained why these methods can not be immediately used for proving termination of higher-order rewrite systems; we sketch how a modification and integration of the two methods should look like. This section also contains an overview of related work about proving termination of higher-order rewrite systems.

Chapters 4 and 5 form the core of the thesis. The theoretical basis is established in Chapter 4. This chapter can be read quite independently of the exact formulation of higher-order rewriting. It is devoted to the extensions of *strictly monotonic* and *weakly monotonic* to functions of all types. We propose a notion of *strict* functions, which are strictly monotonic in a certain sense, even in the presence of weakly monotonic functions.

In Chapter 5 the theory on weakly monotonic and strict functionals is applied to derive a semantical method for proving termination of higher-order rewrite systems. The method is applied to many examples. Most notably are Gödel's T — a system that extends simply-typed lambda calculus with higher-order primitive recursion — and a rewrite system that normalizes proofs in natural deduction. The latter system is complicated by the presence of the so-called permutative reductions. We also identified computation rules for functionals and methods to find strict functionals. These make it easier to fulfill the requirements of the method that we propose, thus supporting the application of our method.

Chapter 6 can be read independently of the rest of this thesis. In this chapter, we compare the semantical approach to termination proofs, with a more traditional approach that emerged from lambda calculus, and which uses strong computability predicates. Although the two proofs seem completely unrelated, we found a remarkable connection.

The idea to reveal this connection is as follows: We start with a proof based on strong computability predicates. This proof is decorated with information on the length of reduction sequences. After this, we extract the computational content of

this proof, by using the modified realizability interpretation. In this way, we find a program that given a term, estimates an upper bound for the length of reduction sequences starting in it. It turns out that this upper bound coincides with the function that is used in semantical termination proofs. This scheme is carried out for simply-typed lambda calculus and Gödel's T.

Finally, the Appendix contains a reproduction of Prawitz's proof based on a variant of strong computability. We added it because there are a number of connections with our work. Our reproduction is denser than the text in [Pra71].

Contribution and Related Work.

The major contribution of this thesis is a general method to prove termination of higher-order rewrite systems. Although the method is not complete, it covers a lot of examples, as will be extensively shown. Easy application of it is supported by providing computation rules for functionals and ways to find strict functionals.

We show that our method can deal with non-trivial examples. For the normalization of natural deductions, including permutative conversions, we present the first semantical termination proof. A proof using a variant of the strong computability predicates already existed. It is reproduced in the Appendix.

As a corollary we prove that adding a terminating term rewriting system to the simply-typed lambda calculus preserves termination. This modularity result is already known, but we present a new proof of it. We also generalize this result for a particular kind of higher-order rewrite rules. In the latter case, we have to know the termination model of the rewrite rules. Under certain conditions on this termination model, the plain combination of the higher-order rules with β-reduction is terminating.

Finally, the last chapter of this thesis shows how to compare two existing proof methods, that could not be compared before. By program extraction from strong computability proofs, we find programs that play a crucial role in semantical termination proofs. This recipe can probably be applied to other systems. Note that termination, as well as realizability and semantic models play an important role in consistency proofs of logical systems. It is interesting to connect these notions. Whether the connection we found has logical consequences remains open.

Most results of this thesis have been published in conference proceedings. In [Pol94] a description of the semantical proof method is given. In [PS95] several computation rules are given and the method is applied to larger examples. The former two papers form the basis of Chapter 4 and 5. Chapter 6 is the full version of [Pol96]. The modularity results have not been published before.

For pointers to related work we refer to Section 2.5.3, Chapter 3 (especially the last section), the introduction to Chapter 6 and the Bibliography.

Chapter 2

The Systems

This chapter is devoted to the syntactical introduction of several systems. The most advanced systems are the higher-order term rewriting systems (HRS). Because these systems can be understood as a combination of usual (first-order) term rewriting systems (TRS) and simply-typed lambda calculus ($\lambda^{\rightarrow}$), we first introduce the latter two systems. The three systems are introduced in Section 2.3, 2.4 and 2.5, respectively.

Because the three systems that we introduce have a lot in common, we start off with the introduction of *abstract* reduction systems (ARS, Section 2.2). The particular systems that we already mentioned can be seen as specializations of ARSs.

Before doing anything, some general terminology and handy notation will be fixed.

2.1 Preliminary Terminology and Notation

This section gives an overview of a number of general notions and notation, that will be used. We tried to keep notation close to mathematical conventions, by using "naive set theory".

The natural numbers are $0, 1, 2, \ldots$. The set of natural numbers is denoted by $\mathbb{N}$. We will use $+$ for the binary addition function and $>$ for the binary greater-than-relation. With $\mathbb{N}_{\geq n}$ we denote the set of natural numbers that are greater than or equal to n.

For sets A and B we write $A \cup B$ for the union and $A \cap B$ for the intersection of A and B. The element-relationship is denoted by $x \in A$ (x is an element of A), the subset relation with $A \subseteq B$ (A is a subset of B). A and B are disjoint, if they have no elements in common. If A and B are disjoint, their union may be written as $A \uplus B$. With $A \times B$ we write the cartesian product of A and B. A pair (x, y) is in $A \times B$ if $x \in A$ and $y \in B$. If $z = (x, y)$, then $\pi_0(z)$ denotes x and $\pi_1(z)$ denotes y.

With $A \Rightarrow B$ we denote the set of all functions from A to B. A is called the domain of these functions, and B the co-domain. If $f \in A \Rightarrow B$ and $x \in A$, then we write $f(x)$ for the result of applying f to x; of course $f(x) \in B$. If $h \in A \Rightarrow (B \Rightarrow C)$, we will write $h(x, y)$ for the application $h(x)(y)$, as if h were in $(A \times B) \Rightarrow C$.

Two functions $f, g \in A \Rightarrow B$ are equal, if for all $x \in A$, $f(x) = g(x)$. Hence a function can be defined by specifying its input/output behavior, which can be conveniently expressed by using an abstraction: if $E[x]$ is an expression, possibly containing x, then we write $\boldsymbol{\lambda} x \in A.E[x]$ for the function that maps each $a \in A$ to $E[a]$ (the result of substituting a for x in E). This notation is borrowed from the lambda calculus, which will be introduced later. For $f \in A \Rightarrow B$ and $g \in B \Rightarrow C$, we write $g \circ f \in A \Rightarrow C$ for the composition of f and g. The composition can be defined as $\boldsymbol{\lambda} x \in A.g(f(x))$.

A sequence of objects $x_1, x_2, \ldots, x_n$ is often abbreviated as $\vec{x}_n$, or $\vec{x}$ when the length is unknown or not important. Conversely, if a sequence $\vec{y}_n$ is given, then we write y_1 for the first element, y_2 for the second, etcetera. With ε we denote the empty sequence (i.e. n=0).

If A is a set and for each $i \in A$, X_i is some object, we write $(X_i)_{i \in A}$ for an A-indexed family. This is equivalent to the function $\boldsymbol{\lambda} i \in A.X_i$. If the index set is clear from the context, the family can be abbreviated with X. If each X_i is itself a set, then $\bigcup_{i \in A}(X_i)$ denotes the union of all sets in the family. If the X_i are pairwise disjoint, we will confuse X with its disjoint union. If an A-indexed family X is given, then we write X_a for the a-th element of X, i.e. the element with index a.

Given a set A, a binary relation on it is a subset of $A \times A$. With xRy we denote that R holds between x and y. A relation R is reflexive if xRx holds for all $x \in A$; it is irreflexive if for no $x \in A$, xRx holds. R is symmetric if for all $x, y \in A$, xRy implies yRx; it is transitive if for all $x, y, z \in A$ with xRy and yRz, xRz holds. Finally, R is anti-symmetric if for all $x, y \in A$, xRy and yRx imply that $x = y$.

With R^+ we denote the transitive closure of R, i.e. it is the smallest set that contains R and is transitive. With R^* its reflexive-transitive closure. With R^{-1} we denote the inverse of R: $xR^{-1}y$ holds if and only if yRx.

An *equivalence relation* is a binary relation that is reflexive, symmetric and transitive. An equivalence relation $\approx$ on A, generates a set of equivalence classes. The equivalence class of x consists of the elements $y \in A$ such that $x \approx y$. The equivalence classes are pairwise disjoint.

A (strict) *partial order* is a binary relation that is transitive and irreflexive. We use symbols $>$ and $\succ$ for partial orders. Many authors define partial orders to be reflexive, anti-symmetric and transitive. Our choice is more convenient in termination proofs. We always mean strict partial order, when we say "partial order" or just "order". A partial order is *well-founded* if there is no infinite decreasing sequence $x_0 > x_1 > x_2 > \cdots$.

Given partial orders $(B, >_B)$, $(A_1, >_1), \ldots, (A_n, >_n)$ we call the function $f \in (A_1 \times \cdots \times A_n) \Rightarrow B$ *strictly monotonic* if for each $1 \leq i \leq n$, and for all $x_1 \in A_1, \ldots, x_n \in A_n$ and $y \in A_i$,

$$x_i >_i y \Rightarrow f(x_1, \ldots, x_i, \ldots, x_n) >_B f(x_1, \ldots, y, \ldots, x_n) \ .$$

A *pre-order* (also called quasi-order) is a binary relation that is reflexive and transitive. A pre-order $\geq$ generates an equivalence relation and a partial order as follows: $x \approx y$ if and only if $x \geq y$ and $y \geq x$; $x > y$ if and only if $x \geq y$ but not $y \geq x$. The latter partial order is said to be generated by the pre-order.

2.2 Abstract Reduction Systems

In the subsequent sections, we will encounter several reduction systems. A number of definitions and lemmas are not special for one of these systems in particular, but have a general nature. To this end, we introduce the well known notion of an *abstract reduction system* (ARS). An ARS is of the form (A, R), where R is a binary relation on A.

So far, there is nothing special about ARSs. In fact the distinguishing feature of ARSs comes with their use. We think of A as a set of objects, and of R as a reduction relation, or as computation steps. That is, given an object $a \in A$, it can be "computed" by performing steps, that is finding $b, c, \ldots$ such that $aRbRc\cdots$, until no step can be done anymore. In that case we have reached a so-called normal form.

To stress that an ARS is about transformations, we often denote the relation by $\to$. The reflexive-transitive closure of $\to$ is denoted by $\twoheadrightarrow$. The reflexive-symmetric-transitive closure is denoted by $=$. Syntactic equality on objects is denoted by $\equiv$. If $a \to b$, then b is called a *one-step reduct* of a. The *reducts* of a are those b such that $a \twoheadrightarrow b$. We write $\to_1 \cdot \to_2$ for the composition of $\to_1$ and $\to_2$. Furthermore, $\overset{n}{\to}$ denotes the n-fold composition of $\to$.

Several questions about the computations may arise: Can any object be computed, i.e. can we find a reduction sequence to normal form? Is the result of a computation uniquely determined, i.e. independent of the steps that we choose to perform? or: Does any rewrite sequence eventually terminate in a normal form? These natural questions give rise to several definitions.

Definition 2.2.1 *Let $\mathcal{A} = (A, \to)$ be an ARS. Let $a, b \in A$ be given.*

- *a is a normal form, if for no $x \in A$, $a \to x$.*
- *a is* weakly normalizing *(WN) if it has a normal form, i.e. if for some $x \in A$, $a \twoheadrightarrow x$ and x is a normal form.*
- *A reduction sequence from a is a (finite or infinite) sequence $a \equiv a_0 \to a_1 \to a_2 \to \cdots$.*
- *a is* strongly normalizing *(SN) or* terminating *if every reduction sequence from a is finite.*
- *a is strongly normalizing in at most n steps,* $\mathrm{SN}(a, n)$*, if every reduction sequence from a has at most n steps.*
- *a is* confluent *or* Church-Rosser *(CR) if for all $b, c \in A$ such that $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$, there exists a d such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.*
- *a is* weakly Church-Rosser *or* locally confluent *(WCR) if for all $b, c \in A$ such that $a \to b$ and $a \to c$, there exists a d such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.*
- *$\mathcal{A}$ is WN, SN, WCR or CR, if for all $x \in A$, x is WN, SN, WCR or CR.*

- $\mathcal{A}$ *is* finitely branching, *if for all* $x \in A$, *the set* $\{y \mid x \to y\}$ *is finite. This is sometimes called locally finite.*

Weak normalization is a desirable property, because it ensures that every object has a normal form, so it can be computed. It is also desirable that the answer is unique. This is ensured by confluence: If a reduces to two normal forms b and c, then by confluence $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$ for some d. Because b and c are normal, it must be the case that $b \equiv c \equiv d$. So weak normalization and confluence together ensure that any object can be computed in a unique way. In the presence of these properties, we write $a{\downarrow}$ for the unique normal form of a.

In fact, weak normalization is a bit too weak. We only have that there exists a reduction to normal form. In order to really compute, we should also have a good strategy to find that reduction. Termination (or strong normalization) is convenient, because in that case we need not care about a reduction strategy: Every reduction eventually leads to a normal form. (A strategy may become important if we also take efficiency into consideration.)

Confluence and termination are often difficult to prove. We refer to [Oos94] for a recent study on confluence proofs. It is often more easy to prove local confluence. The difference is that for any object, we only have to prove something for its one-step reducts. This check can often be automated, especially in case $\to$ is generated by a finite number of rules. We can now give a second reason to be interested in strong normalization:

Lemma 2.2.2 *[New42] If* $\mathcal{A}$ *is SN and WCR, then it is CR.*

The only properties that we have not yet motivated are the binary SN-predicate and the finite branching. If we know $\mathrm{SN}(a, n)$, we have an upper bound on the longest reduction sequence from a. It may be convenient to know the resources that are needed to perform a computation. Having a function f such that for all $a \in A$, $\mathrm{SN}(a, f(a))$ even gives a uniform upper bound to reduction sequences in an ARS.

However, the main motivation for the binary SN-predicate is a methodological one. The binary predicate gives more information. Different methods to prove termination may yield different upper bounds. We can compare the methods by inspecting which bound is sharper. In Chapter 6 we will compare different SN-proofs with respect to the upper bounds they impose on the length of reduction sequences. There also exist SN-proofs that give no indication about the length of reduction sequences.

The relationship between the unary and the binary SN-predicate is given by the following lemma, which is an immediate consequence of König's Lemma.

Lemma 2.2.3 *If* $\mathcal{A}$ *is finitely branching, then* $\mathrm{SN}(\mathcal{A})$ *holds if and only if* $\forall a{\in}A.\exists n{\in}\mathbb{N}.\mathrm{SN}(a, n)$.

Proof: $\Rightarrow$: Consider the reduction tree from a. This tree is finitely branching by assumption, and all paths in it have finite length, by SN. By König's Lemma, the number of nodes in the tree is finite, so the number of paths in it is also finite. Hence we can take a path with the greatest length. This length is the required n.

$\Leftarrow$: Immediate. This part doesn't use the finite branching. $\boxtimes$

2.3 First-order Term Rewriting Systems

The first instances of ARSs that we encounter are the TRSs. Here the objects are first-order terms in some signature. The reduction relation is generated by closing a set of rewrite rules under substitution and context.

First order term rewriting can be seen as the proof theory that comes with equational logic. Alternatively, it can be seen as the operational semantics of abstract data types. The study of TRSs yields a lot of insights in functional programming languages. Standard texts on term rewriting are [HO80, DJ90, Klo92].

Definition of TRSs. A *first-order signature* is a tuple $(\mathcal{F}, \mathcal{V})$, where $\mathcal{F}$ is the set of function symbols and $\mathcal{V}$ is a set of variables. It is assumed that $\mathcal{F} \cap \mathcal{V} = \varnothing$. Associated to $\mathcal{F}$ is a function $\mathit{arity} : \mathcal{F} \Rightarrow \mathbb{N}$, which gives each $f \in \mathcal{F}$ its arity.

From now on, we fix a signature $\Sigma = (\mathcal{F}, \mathcal{V})$. We can define the set of terms $\mathcal{T}(\Sigma)$ inductively as follows: if $x \in \mathcal{V}$, then $x \in \mathcal{T}(\Sigma)$; and if $f \in \mathcal{F}$, $\mathit{arity}(f) = n$ and $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$, then $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)$. The variables $x, y, z, \ldots$ (possibly subscripted) will range over $\mathcal{V}$; $r, s, t, \ldots$ are reserved for elements in $\mathcal{T}(\Sigma)$. With $\mathit{Var}(t)$ we denote the set of variables that occur in t.

A term t is closed if $\mathit{Var}(t) = \varnothing$. The closed terms are built by leaving out the first clause in the definition of $\mathcal{T}(\Sigma)$. The only way to start building closed terms is with a constant, i.e. a function symbol with arity 0. If there are no constants, then the set of closed terms is empty.

A *substitution* is a function in $\mathcal{V} \Rightarrow \mathcal{T}(\Sigma)$. It is extended to terms in a homomorphic way. For substitutions we use $\theta, \theta_1, \ldots$. The result of applying substitution θ to term t is denoted by t^θ. Thus $f(t_1, \ldots, t_n)^\theta \equiv f(t_1^\theta, \ldots, t_n^\theta)$.

A context over Σ is a term t in $\mathcal{T}(\mathcal{F}, \mathcal{V} \cup \Box)$, such that $\Box$ occurs exactly once in t. Contexts are regarded as terms with a hole in it (namely $\Box$), and are often called $C[\,]$ and $D[\,]$. The result of filling the hole in $C[\,]$ with t is denoted by $C[t]$.

A rewrite rule is a pair $l \mapsto r$, such that $l, r \in \mathcal{T}(\Sigma)$, $l \notin \mathcal{V}$ and $\mathit{Var}(r) \subseteq \mathit{Var}(l)$. A rewrite rule $l \mapsto r$ is called *duplicating*, if some variable x occurs more often in r than in l. It is called *collapsing* if r is a variable.

A *term rewriting system* (TRS) is a tuple (Σ, R), where Σ is a signature and R a set of rewrite rules.

The rewrite relation generated by a TRS $\mathcal{R} = (\Sigma, R)$ is denoted by $\to_{\mathcal{R}}$, and is defined as follows:

$$s \to_{\mathcal{R}} t :\Longleftrightarrow \text{ there are } C[\,], \theta \text{ and } l \mapsto r \text{ such that } s \equiv C[l^\theta] \wedge t \equiv C[r^\theta] \ .$$

Disjunctive normal form. As an example, we consider a TRS to find the disjunctive normal form of propositional formulae. We only consider the following connec-

tives: $\wedge$ and $\vee$ (binary, written infix) and $\neg$ (unary). The rules are

$$\begin{array}{rcl} x \wedge (y \vee z) & \mapsto & (x \wedge y) \vee (x \wedge z) \\ (y \vee z) \wedge x & \mapsto & (y \wedge x) \vee (z \wedge x) \\ \neg(x \wedge y) & \mapsto & \neg x \vee \neg y \\ \neg(x \vee y) & \mapsto & \neg x \wedge \neg y \\ \neg\neg x & \mapsto & x \end{array}$$

It is clear that the normal forms of this system correspond to disjunctive normal forms (i.e. disjunctions of conjunctions of positive and negative literals). The left hand side of each rule is logically equivalent to the corresponding right hand side. So if this system is WN, then every formula can be written in a logically equivalent disjunctive normal form. In Section 3.1, we show as an illustration that this system is SN.

We remark that this system is not confluent, which corresponds to the fact that the disjunctive normal form is not uniquely determined. Consider for instance the following diverging reductions:

$$\begin{array}{lcl} (v \vee w) \wedge (x \vee y) & \to_{\mathcal{R}} & ((v \vee w) \wedge x) \vee ((v \vee w) \wedge y) \\ & \overset{2}{\to}_{\mathcal{R}} & ((v \wedge x) \vee (w \wedge x)) \vee ((v \wedge y) \vee (w \wedge y)) \ , \end{array}$$

and

$$\begin{array}{lcl} (v \vee w) \wedge (x \vee y) & \to_{\mathcal{R}} & (v \wedge (x \vee y)) \vee (w \wedge (x \vee y)) \\ & \overset{2}{\to}_{\mathcal{R}} & ((v \wedge x) \vee (v \wedge y)) \vee ((w \wedge x) \vee (w \wedge y)) \ . \end{array}$$

Both results are in normal form, so they cannot be brought together by performing more reductions.

2.4 Simply-typed Lambda Calculus

In this section, we will introduce the *simply-typed lambda calculus*. This is another instance of ARSs. The objects of this ARS are those lambda terms that can be assigned a simple type. We will introduce two reduction relations on terms, $\to_\beta$ and $\to_{\beta\overline{\eta}}$. Section 2.4.1 is on the static part: terms, types and a lot of terminology. In Section 2.4.2 we will introduce the rewrite relations.

Simply-typed lambda calculus was introduced by Church in [Chu40]. For general notions and notation in lambda calculus, see [Bar84]. For an overview of typed lambda calculi the reader is referred to [Bar92].

2.4.1 Terms and Types

In fact we should speak about simply-typed lambda *calculi*. We allow several parameters to vary, namely the base types, the constants and the variables that are used. This information is stored in a signature.

Types. The simple types are constructed from a set of base types. Other types can be formed by repeatedly applying the binary type operator $\to$. Later on, other calculi will be introduced that have more type forming operators (e.g. product types). To distinguish the various calculi, we will denote the types with $\mathbb{T}^{\to}(\mathcal{B})$, the *simple types* over a set of base types $\mathcal{B}$. This set is formally defined as the smallest set satisfying

- $\mathcal{B} \subseteq \mathbb{T}^{\to}(\mathcal{B})$
- If $\sigma, \tau \in \mathbb{T}^{\to}(\mathcal{B})$ then also $\sigma \to \tau \in \mathbb{T}^{\to}(\mathcal{B})$.

In the sequel, the variables $\rho, \sigma, \tau, \ldots$ (possibly subscripted) will range over $\mathbb{T}^{\to}(\mathcal{B})$; we use ι, κ as metavariables over $\mathcal{B}$. Intuitively, $\sigma \to \tau$ is the type of functions that can be applied to input of type σ, yielding output of type τ. To save brackets, we follow the convention that $\to$ associates to the right. I.e. $\rho \to \sigma \to \tau$ denotes the type $\rho \to (\sigma \to \tau)$. We also write $\vec{\rho}_n \to \sigma$ for the type $\rho_1 \to \cdots \to \rho_n \to \sigma$.

Every type can be uniquely written in the form $\vec{\sigma}_n \to \iota$, where ι is a base type. If $\sigma = \vec{\sigma}_n \to \iota$, then n is called the *arity* of σ, which corresponds to the number of arguments that functions of this type expect. In this case we also write $fac(\sigma)$ for $\vec{\sigma}$, the *factors* of σ and $res(\sigma)$ for ι, the *result type* of σ. The latter can be recursively defined by

- For $\iota \in \mathcal{B}$, $res(\iota) = \iota$.
- $res(\sigma \to \tau) = res(\tau)$.

If two types have the same factors, then they accept the same sequence of arguments. We write $\sigma \approx \tau$ in that case. This equivalence relation can be inductively defined as follows:

- $\iota \approx \kappa$, for $\iota, \kappa \in \mathcal{B}$.
- If $\sigma \approx \tau$, then $\rho \to \sigma \approx \rho \to \tau$.

As a measure of complexity, we introduce the notion of *type level*. This is defined with induction over the types:

- For $\iota \in \mathcal{B}$ we put $\mathrm{TL}(\iota) = 0$.
- $\mathrm{TL}(\sigma \to \tau) = \max(\mathrm{TL}(\sigma) + 1, \mathrm{TL}(\tau))$.

Thus nesting arrows to the left is considered complex.

Signature. In pure lambda calculus, terms are constructed from typed variables by the operators of abstraction and application. These operators get their meaning by the rules for β- or $\beta\eta$-equality. In our setting, additional constants are allowed, which will get their meaning by higher-order rewrite rules. These rules are not discussed in this section.

The set of simply-typed lambda terms is parametrized by a set of base types, sets of typed constants and sets of typed variables. These three sets will be combined in a signature. A *higher-order signature* is a triple $(\mathcal{B}, \mathcal{C}, \mathcal{V})$, where

- $\mathcal{B}$ is a set, which serves as the set of base types.
- $\mathcal{C}$ is a family $(\mathcal{C}_\sigma)_{\sigma \in \mathbb{T}^\to(\mathcal{B})}$, where the $\mathcal{C}_\sigma$ are pairwise disjoint. Elements of $\mathcal{C}_\sigma$ are called constants of type σ.
- $\mathcal{V}$ is a family $(\mathcal{V}_\sigma)_{\sigma \in \mathbb{T}^\to(\mathcal{B})}$, where the $\mathcal{V}_\sigma$ are countably infinite and pairwise disjoint. Elements of $\mathcal{V}_\sigma$ are called variables of type σ.
- $\mathcal{B}$, $\bigcup \mathcal{C}$ and $\bigcup \mathcal{V}$ are pairwise disjoint.

Symbols c, d, f, g typically range over $\mathcal{C}$. Symbols x, y, z are reserved for variables in $\mathcal{V}$. We write x^σ to stress that (the variable) x has type σ.

Terms. Given a signature $\mathcal{F} = (\mathcal{B}, \mathcal{C}, \mathcal{V})$ we define the set of *simply-typed lambda terms*, $\Lambda^\to(\mathcal{F})$, as the union of the sets of terms of type τ, for each $\tau \in \mathbb{T}^\to(\mathcal{B})$. These sets, written $\Lambda^\to_\tau(\mathcal{F})$, are defined simultaneously as the smallest sets satisfying

- $\mathcal{V}_\tau \subseteq \Lambda^\to_\tau(\mathcal{F})$.
- $\mathcal{C}_\tau \subseteq \Lambda^\to_\tau(\mathcal{F})$,
- If $M \in \Lambda^\to_{\sigma\to\tau}(\mathcal{F})$ and $N \in \Lambda^\to_\sigma(\mathcal{F})$, then $MN \in \Lambda^\to_\tau(\mathcal{F})$. This is called application of M to N.
- If $x \in \mathcal{V}_\sigma$ and $M \in \Lambda^\to_\tau(\mathcal{F})$ then $\lambda x.M \in \Lambda^\to_{\sigma\to\tau}(\mathcal{F})$. This is called abstraction of x in M.

We put $\Lambda^\to(\mathcal{F}) := \bigcup_{\tau \in \mathbb{T}^\to(\mathcal{B})} \Lambda^\to_\tau(\mathcal{F})$, the set of simply-typed lambda terms over the signature $\mathcal{F}$.

Often the signature is known from the context, in which case it can be omitted by simply writing $\mathbb{T}^\to$ and $\Lambda^\to$. In the sequel of this section, we assume a fixed signature $\mathcal{F} = (\mathcal{B}, \mathcal{C}, \mathcal{V})$. With terms we will mean simply-typed lambda terms in this signature. Instead of $M \in \Lambda^\to_\tau$ we will also write $M : \tau$.

The following notational conventions will be used. The symbols M, N, P (possibly subscripted) are reserved for terms. We write $\lambda xyz.M$ for $\lambda x.\lambda y.\lambda z.M$. Application associates to the left and binds stronger than abstraction. So NMP means $(NM)P$ and $\lambda x.yx$ denotes $\lambda x.(yx)$. Finally, we write $M\vec{P}_n$ as a shorthand for $MP_1 \cdots P_n$ and $\lambda\vec{x}_n.M$ for $\lambda x_1 x_2 \cdots x_n.M$.

Bound and free variables. One can think about lambda terms as a notation for functions. These functions can be applied to each other, provided the types fit. The type of a term determines to which other terms it can be applied. Abstraction is the "inverse" of application. With abstraction one can construct functions. The λx in $\lambda x.yx$ expresses that this term has to be interpreted as a function in x (and y is a free parameter of it). Applying this function to a yields ya. This is quite different from applying $\lambda y.yx$ to a. Other examples of lambda terms are $\lambda x.x$ (the identity function); $\lambda x.\lambda y.x$ (left projection) and $\lambda x.y$ (the constant y function). This intuition

underlies the standard model of simply-typed lambda calculus, which we will discuss in Section 3.2.

The λx part in $\lambda x.M$ is said to *bind* the occurrences of variable x in M; x is a *bound variable*. Variables that occur outside the scope of any binder are called free. In principle, a variable can occur both bound and free in one term. Consider e.g. the term $(cx)(\lambda x.x)$, which is well-typed in an appropriate signature. Formally, we define the set of free variables of a simply-typed term M, denoted by $\mathrm{FV}(M)$, by induction on M as follows:

- $\mathrm{FV}(x) = \{x\}$ for $x \in \mathcal{V}$
- $\mathrm{FV}(c) = \varnothing$ for $c \in \mathcal{C}$
- $\mathrm{FV}(MN) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$
- $\mathrm{FV}(\lambda x.M) = \mathrm{FV}(M) \setminus \{x\}$

If $\mathrm{FV}(M) = \varnothing$, we say that M is a *closed* term.

Alpha-conversion, substitution, variable convention. Note that $\lambda x.x$ and $\lambda y.y$ both denote identity. The bound variable has a local nature, so the name that is chosen for it is not important. Changing the names of bound variables is known as α-conversion. From now on, we will *identify* terms that only differ in the names of the bound variables. So in fact a lambda term should be viewed as an equivalence class of α-convertible terms. The terms that are written down are arbitrary representatives of this equivalence class. We write $M \equiv N$ if M and N are α-convertible. E.g. $\lambda x.\lambda x.x \equiv \lambda x.\lambda y.y$. The latter is more intelligible, because distinct variables have different names.

A *substitution* is a mapping from variables to terms of the same type. More precisely, it is a finite function $\{x_1 \mapsto M_1, \ldots, x_n \mapsto M_n\}$, where for each $1 \leq i \leq n$, there exists a τ such that $x_i \in \mathcal{V}_\tau$ and $M_i \in \Lambda^{\rightarrow}_\tau$.

Substitutions are extended to homomorphisms on terms. We write M^θ for the application of θ to M, i.e. the simultaneous substitution of the x_i by the M_i in M. If $\theta = \{x \mapsto N\}$, we also write $M[x := N]$. M^θ can be defined with induction on M as follows:

- $x^\theta \equiv \theta(x)$ if x is in the domain of θ.
- $x^\theta \equiv x$, if $x \in \mathcal{V}$ but not in the domain of θ.
- $c^\theta \equiv c$, if $c \in \mathcal{C}$.
- $(MN)^\theta \equiv M^\theta N^\theta$
- $(\lambda x.M)^\theta \equiv \lambda x.(M^\theta)$, provided no name clashes occur.

A name clash can occur in two ways:

1. x occurs in the domain of θ. In that case, a substitution would rename a bound variable, which is unintended.

2. x occurs free in the co-domain of θ. More precisely, for some y, $x \in \mathrm{FV}(\theta(y))$. In that case, there is an unintended capture of that x by the λx.

Although there is a proviso, the definition of M^θ is total, because we can always rename the bound variables such that the proviso is met. Consequently, substitution is only defined up to $\equiv$. For example, applying the substitution $\{y \mapsto (ax)\}$ to the term $(\lambda x.yx)$ should yield $\lambda z.axz$, not the erroneous $\lambda x.axx$.

In the sequel, we tacitly assume that necessary renamings will be performed automatically. Moreover, in concrete situations, we assume that renamings are not necessary, due to a convenient choice of the names of the bound variables. Such assumptions are often referred to as the *variable convention*.

Note that this convention only works in a mathematical text. In a computer program, or in formalized proofs, one cannot escape from renaming the variables explicitly. To make the choice of variable names systematic, De Bruijn indices (i.e. numbers referring to the binders) can be used instead of named variables [Bru72]. See [Bar84, p. 26], where a more severe variable convention is introduced.

Restricted classes of terms. A subclass of lambda terms, the so called *λI-terms*, is obtained by restricting the formation of $\lambda x.M$ to the case that $x \in \mathrm{FV}(M)$. We denote this class by $\Lambda^{\to}$-I. This class plays an important rôle in Gandy's proof that β-reduction in simply-typed lambda calculus terminates (Section 3.3). The term $\lambda x.\lambda y.x$ for instance is not in $\Lambda^{\to}$-I, because the argument y is not used.

An other restriction of lambda terms is formed by the so called *patterns*. Given a term M, it can be written as $\lambda\vec{x}.N$, with N not a lambda abstraction. Now M is a pattern, if all variables $y \in \mathrm{FV}(N)$ occur in a position $yz_1 \ldots z_n$ and the $z_1, \ldots, z_n$ are pairwise distinct free variables. This class is important for higher-order matching [Mil91].

2.4.2 β- and η-Reduction

Equations. So far, we have no calculus yet. We will now introduce two schematic equations, traditionally called β and η. The β equality expresses what happens if we apply a term $\lambda x.M$ to a term N. The result will be M, with the formal parameter x instantiated by the argument N. Hence we get (for every x, M and N such that the following schema is well-typed)

$$(\beta) \qquad (\lambda x.M)N = M[x := N] \ .$$

The equality $=_\beta$ is defined as the closure of the rule (β) under the usual equational laws of reflexivity, symmetry, transitivity and compatibility. With the latter we mean that if $M =_\beta N$, then $MP =_\beta NP$, $PM =_\beta PN$ and $\lambda x.M =_\beta \lambda x.N$.

There is a second equation schema: For each M with $x \notin \mathrm{FV}(M)$,

$$(\eta) \qquad \lambda x.Mx = M \ .$$

This schema can be justified by noting that the left and right hand side yield the same result when applied to a term N. The equality $=_{\beta\eta}$ is defined as the closure of the rules (β) and (η) under the usual equational laws.

The equational theory thus obtained can be turned into a reduction system by directing the equations (β) and/or (η). We will be mostly concerned by the reduction system based on the (β)-rule only. This reduction system will be called $\lambda^{\rightarrow}_{\beta}$.

β-Reduction. We write $\rightarrow_\beta$ for the compatible closure of the rewrite schema

$$(\beta) \qquad (\lambda x.M)N \quad \mapsto \quad M[x := N] \ .$$

The reflexive-transitive closure is written $\twoheadrightarrow_\beta$. The chosen direction is the intuitive one. It corresponds to function calls in many programming languages. For this reason, $\lambda^{\rightarrow}_{\beta}$ is *the* prototype to study parameter passing in functional languages.

The system $\lambda^{\rightarrow}_{\beta}$ has been studied extensively. As a reduction system, it has many nice properties. We only list a few of them.

Theorem 2.4.1 *[CR36] $\lambda^{\rightarrow}_{\beta}$ is confluent.*

Confluence does not depend on typing information. The untyped lambda calculus is already confluent. Proofs of this fact are due to Church and Rosser (in fact this proof is in the setting of the untyped λI-calculus), and Martin-Löf and Tait. The proofs can be found in e.g. [Bar84].

Theorem 2.4.2 *$\lambda^{\rightarrow}_{\beta}$ is weakly normalizing.*

By the previous theorems, every simply-typed lambda term has a unique β-normal form. It is not difficult to characterize this normal form.

Lemma 2.4.3 *M is in normal form, if and only if it is of the form $\lambda\vec{x}_n.yN_1,\ldots,N_m$, where $y \in \mathcal{V} \cup \mathcal{C}$, and for each $1 \leq i \leq m$, N_i is in β-normal form again.*

Finally,

Theorem 2.4.4 *$\lambda^{\rightarrow}_{\beta}$ is strongly normalizing.*

There are many proofs of the last fact [Gan80, Tai67, Tro73]. We will present Gandy's proof in Section 3.3. Tait's proof is presented in Section 6.1. Chapter 6 is devoted to the comparison of these two proofs.

Normalization essentially relies on typing information. In the untyped lambda calculus, WN does not hold. In this calculus, application and abstraction are not restricted by any type constraints. In this way one can construct more terms, among which the famous $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$. It is easily verified that $\Omega \rightarrow_\beta \Omega$, which destroys strong normalization. As Ω is the only successor of Ω, it has no normal form, so weak normalization is violated too.

There are much more strongly normalizing lambda terms than those typable in $\Lambda^{\rightarrow}$. It is undecidable whether an untyped lambda term is strongly normalizing or

not. Stronger typing systems have been studied, that capture more and more strongly normalizing terms, most notably System F (capturing second order polymorphism) [Gir72], System F_ω (featured by unlimited polymorphism) and $\lambda\cap^-$ (incorporating intersection types). In the latter system, strong normalization and typability coincide [Bak92]. Of course, in this system it is undecidable whether a term is typable or not.

Long normal forms. At first sight, the most reasonable direction of the equation schema (η) is from left to right, for only in that direction the terms get smaller. We therefore define $\to_\eta$ as the compatible closure of the relation $\lambda x.Mx \mapsto M$, provided $x \notin \mathrm{FV}(M)$. We put $\to_{\beta\eta} = \to_\beta \cup \to_\eta$. This relation has nice rewrite properties, for it is strongly normalizing and confluent.

It is also possible to choose the opposite direction, which is more convenient in higher-order rewriting and also for higher-order matching. However, we cannot allow unrestricted η-expansion, because it has undesirable rewrite properties. First of all, normal forms would not exist: For any $M : o \to o$ we have

$$M \leftarrow_\eta \lambda x.Mx \leftarrow_\eta \lambda x.(\lambda y.My)x \leftarrow_\eta \cdots \ .$$

In the presence of β, another infinite reduction exists:

$$MN \leftarrow_\eta (\lambda x.Mx)N \to_\beta MN \leftarrow_\eta \cdots \ .$$

Note that also in the first reduction sequence, β-redexes are created. We therefore define *restricted η-expansion* as follows:

$$M \to_{\overline{\eta}} N :\iff M \leftarrow_\eta N \text{ without creating a } \beta\text{-redex.}$$

With $\to_{\beta\overline{\eta}}$ we denote $\to_\beta \cup \to_{\overline{\eta}}$. We remark that a subterm M of P may be expanded to $\lambda x.Mx$ if the following conditions hold:

- M has an arrow type;
- $x \notin \mathrm{FV}(M)$;
- M is not of the form $\lambda x.N$ already; and
- M does not occur in a context MN within P.

With $\lambda^{\to}_{\beta\overline{\eta}}$ we denote the ARS $(\Lambda^{\to}, \to_{\beta\overline{\eta}})$.

Proposition 2.4.5 *$\lambda^{\to}_{\beta\overline{\eta}}$ is strongly normalizing and confluent.*

Proofs of this fact can be found in [CK94, Aka93, Dou93]. Therefore, each lambda term has a unique $\beta\overline{\eta}$-normal form, which is denoted by $M\downarrow_{\beta\overline{\eta}}$. Furthermore, every $\beta\overline{\eta}$-normal form M is of the form $\lambda\vec{x}_n.(aN_1 \ldots N_m)$, where $(aN_1 \ldots N_m)$ is of base type, $a \in \mathcal{V} \cup \mathcal{C}$, n is the arity of the type of M and each N_i is in $\beta\overline{\eta}$-normal form again. One reason to work with restricted η-expansion is that $\beta\overline{\eta}$-normal forms have this nice structure. Each term starts with a number of lambdas, reflecting how many arguments it expects, followed by an expression of base type, which in turn is an

atomic term applied on similar terms. Lambda terms in $\beta\overline{\eta}$-normal form are much like first order terms.

The following lemma shows why it is also technically convenient to work with η-expansions instead of η-reduction: the $\overline{\eta}$-normal form is preserved under substitution and β-reduction.

Lemma 2.4.6 *Let M and N be $\overline{\eta}$-normal. Then*

- *$M[x := N]$ is $\overline{\eta}$-normal.*
- *If $M \to_\beta M'$ then M' is $\overline{\eta}$-normal.*

By definition of $\overline{\eta}$, no β redexes can emerge during η-expansion. The previous lemma says that no $\overline{\eta}$-redexes can emerge in an $\overline{\eta}$-normal form after β-reduction. Hence the $\beta\overline{\eta}$-normal form can be found among others via $\twoheadrightarrow_\beta \cdot \twoheadrightarrow_{\overline{\eta}}$ as well as via $\twoheadrightarrow_{\overline{\eta}} \cdot \twoheadrightarrow_\beta$.

2.5 Higher-order Term Rewriting

In this section, we introduce *higher-order term rewriting*. Higher-order term rewriting is a combination of first-order term rewriting and simply-typed lambda calculus. Higher-order rewriting can be viewed as "rewriting with functions" as opposed to rewriting with first-order objects.

The main motivation for extending first-order term rewriting is to enhance first-order terms with *bound variables*. Many formal languages have a construct of bound — or local — variables, e.g. lambda calculus, first order logic, Pascal programs. The claim is, that objects of such languages (programs, formulae, proofs) can be faithfully represented by lambda terms containing (higher-order) constants. This point of view is not new at all. In [CFC58, p. 85] e.g. the following is stated (and proved): *"Any binding operation can in principle be defined in terms of functional abstraction and an ordinary operation"*. The modern slogan for this attempt is *"higher-order syntax"*.

The main reasons for extending simply-typed lambda calculus is to enlarge the expressive power and to add abstract data types. The resulting formalism inherits the notions of bound variables, substitution and parameter passing of the lambda calculus, and it inherits pattern matching and function definition by equations from term rewriting.

In this section we will first motivate the use of higher-order syntax. Then we give the formal definition of higher-order term rewriting. We then give a short discussion and provide links to related formalisms. Finally we give some examples.

2.5.1 Substitution Calculus

In the view of higher-order syntax, there is only one binder, the λ of some lambda calculus. This calculus is used as a metalanguage. All other binding mechanisms are formulated via this metalanguage. This guarantees that all binding mechanisms

are treated in a uniform way. Hence matters of scope, renaming of local variables, substitution, free parameter provisos etc. have to be dealt with only once. Following [Nip91, Nip93], we will take the simply-typed lambda calculus with $\beta\overline{\eta}$-reduction as a metalanguage. Following [Oos94, Raa96] we will refer to the metalanguage as the *substitution calculus*.

As a typical example, let us consider finding the prenex normal form of first-order logical formulae. In this normal form, the quantifiers only occur at the outside of the formula. The standard way to prove that each formula is logically equivalent to a prenex normal form, is by giving rules that push the quantifiers outside step by step, and prove that applying these rules repeatedly, will terminate eventually. One of these rules could be:

$$\varphi \wedge \forall x.\psi \quad \mapsto \quad \forall x.(\varphi \wedge \psi) \ ,$$

provided that x does not occur free in φ. If x happens to occur free in φ, the bound variable x must be renamed before the rule can be applied. E.g. $p(x) \wedge \forall x.\exists y.r(x,y)$ must be renamed to $p(x) \wedge \forall z.\exists y.r(z,y)$, before it can be rewritten with the first rule to $\forall z.(p(x) \wedge \exists y.r(z,y))$.

In the point of view of higher-order syntax, the ψ formula in the rule above is a function that depends on x. This formalizes the traditional informal notation $\psi[x]$, which means that x may occur in ψ. Instead of $\psi[x]$, we will write $\lambda x.\psi[x]$, or equivalently, $\lambda x.\psi$, thus expressing the dependency on x more precisely.

Assuming two base types, ι for individuals and o for formulae, the type of $\lambda x.\psi$ will be $\iota \to o$. This is the type that the $\forall$-quantifier expects as argument. Now the $\forall$-quantifier does not bind variables, because they are already bound by the λ. Instead, it is viewed as an ordinary constant of type $(\iota \to o) \to o$.

In the traditional informal notation, after introducing a formula like $\psi[x]$, many authors write $\psi[t]$ to denote ψ with all free occurrences of x replaced by the term t. We can now write $\psi[t]$ as an application, viz. $(\lambda x.\psi)t$. Note that β-reduction is necessary to actually perform the substitution. So β-reduction is a component of the substitution calculus. It is needed to animate the metalanguage. Remember that the definition of β-reduction takes care that name clashes are avoided by renaming bound variables.

The rewrite rule written above can now be formulated in the substitution calculus as follows:

$$P \wedge \forall(\lambda x.Qx) \quad \mapsto \quad \forall\lambda x.(P \wedge (Qx)),$$

where P^o and $Q^{\iota\to o}$ are formal variables. P can be instantiated by a concrete φ; Q can be instantiated by something like $\lambda x.\psi[x]$. In the rule above, we wrote the $\beta\overline{\eta}$-normal form of Q, viz. $\lambda x.Qx$.

The proviso that x is not allowed to occur in φ is now captured by the usual variable convention. On the other hand, x may appear in ψ. This is because we can instantiate $Q^{\iota\to o}$ with $\lambda z.\psi[z]$ (which doesn't contain x free) such that (Qx) reduces to $\psi[x]$ in the substitution calculus.

The substitution calculus will get one more task, namely the instantiation of the rules. The rewrite rule above is in fact a schema, for which concrete φ and ψ for P^o and $Q^{\iota\to o}$ have to be substituted. To shift the burden of this substitution to the substitution calculus, we will write the rule as

$$\lambda P^o.\lambda Q^{\iota\to o}.P \wedge \forall(\lambda x.Qx) \quad \mapsto \quad \lambda P^o.\lambda Q^{\iota\to o}.\forall\lambda x.(P \wedge (Qx)) \ .$$

Applying the left hand side and the right hand side to e.g. (py) and $\lambda z.(qzy)$ and computing the β-normal form of both sides, will bind P to (py) and (Qx) to (qxy), yielding the correct instance

$$(py) \wedge \forall\lambda x.(qxy) \quad \mapsto \quad \forall\lambda x.(py) \wedge (qxy) \ .$$

This process of rewriting will be more formally defined in the next section.

2.5.2 Higher-order Rewrite Systems

We now define what a higher-order rewrite system is. To this end, we have to know how the rules may look like. An HRS is then a set of higher-order rules in a certain simply-typed signature. After this, we will define which ARS is induced by such a system. Its objects will be simply-typed lambda terms in $\beta\overline{\eta}$-normal form. The rewrite relation is generated by a number of higher-order rules, that are understood modulo $=_{\beta\eta}$, to deal with substitution.

Definition 2.5.1

- *Given a signature $\mathcal{F} = (\mathcal{B}, \mathcal{C}, \mathcal{V})$ (in the sense of Section 2.4.1), a* higher-order rewrite rule *is a pair, written $L \mapsto R$, such that for some $\tau \in \mathbb{T}^{\to}(\mathcal{B})$, both $L : \tau$ and $R : \tau$, and L and R are closed $\beta\overline{\eta}$-normal forms.*
- *A* higher-order rewrite system (HRS) *is a tuple $(\mathcal{F}, \mathcal{R})$, where $\mathcal{F}$ is a signature and $\mathcal{R}$ is a set of higher-order rewrite rules in this signature.*

Usually, there are several restrictions on the rules. We have deliberately chosen to keep these restrictions outside the definition of HRSs.

In order to define the rewrite relation generated by an HRS, we have to close the relation under contexts. Because the rules contain no free variables, a closure under substitution is not necessary. Instead of closing under substitution, we let matching take place modulo the substitution calculus. We first define what a context is.

A *context* over a signature $(\mathcal{B}, \mathcal{C}, \mathcal{V})$ is a term C in $\Lambda^{\to}(\mathcal{B}, \mathcal{V} \cup \{\Box\}, \mathcal{C})$ such that $\Box$ occurs free in C at exactly one position. Note that the number of occurrences of $\Box$ is not invariant under β-reduction. This is not problematic, because we will only use contexts that are in β-normal form. Just as in the first-order case we write arbitrary contexts as $C[]$; $C[M]$ denotes the term M in context $C[]$. It is obtained by replacing $\Box$ by M in $C[]$.

Definition 2.5.2 *Given an HRS $(\mathcal{F}, \mathcal{R})$, we define the rewrite relation $\to_{\mathcal{R}}$ as follows: $M \to_{\mathcal{R}} N$ if and only if there is a context $C[]$ in $\beta\overline{\eta}$-normal form, and a rule $(L \mapsto R) \in \mathcal{R}$, such that $M \equiv C[L]{\downarrow_{\beta\overline{\eta}}}$ and $N \equiv C[R]{\downarrow_{\beta\overline{\eta}}}$.*

Thus an HRS $(\mathcal{F}, \mathcal{R})$ induces the ARS $(\{M\downarrow_{\beta\overline{\eta}} \mid M \in \Lambda^{\rightarrow}(\mathcal{F})\}, \rightarrow_{\mathcal{R}})$.

Because L and R are closed terms, putting them into a context does not bind free variables. Because C and L are in $\overline{\eta}$-normal form, $C[L]$ is in $\overline{\eta}$-normal form too. Moreover, β-reduction respects $\overline{\eta}$-normal forms, so we can find M by $\twoheadrightarrow_\beta$ from $C[L]$. So a rewrite step from M to N consists of β-expansions, followed by a literal replacement, followed by β-reduction to normal form, as in

$$M \twoheadleftarrow_\beta C[L] \quad \mapsto \quad C[R] \twoheadrightarrow_\beta N \ .$$

We now illustrate how a rewrite step is performed, using the following rule:

$$\lambda P^o.\lambda Q^{\iota\rightarrow o}.P \wedge \forall(\lambda z^\iota.Qz) \quad \mapsto \quad \lambda P^o.\lambda Q^{\iota\rightarrow o}.\forall(\lambda z^\iota.P \wedge (Qz)) \ .$$

Take as context $C \equiv (qx) \vee (\Box(px)(\lambda x.\exists\lambda y.(rxy)))$. Then plugging in the left hand side yields

$$C[\lambda P^o.\lambda Q^{\iota\rightarrow o}.P \wedge \forall\lambda z^\iota.Qz] \twoheadrightarrow_\beta (qx) \vee ((px) \wedge \forall\lambda z.(\exists\lambda y.(rzy))) \ .$$

For the right hand side we have

$$C[\lambda P^o.\lambda Q^{\iota\rightarrow o}.\forall\lambda z^\iota.P \wedge (Qz)] \twoheadrightarrow_\beta (qx) \vee \ \forall\lambda z^\iota.((px) \wedge \exists\lambda y.(rzy)) \ .$$

In traditional notation, this corresponds to the intended rewrite step

$$q(x) \vee (p(x) \wedge \forall z.\exists y.r(z,y)) \quad \mapsto \quad q(x) \vee \ \forall z.(p(x) \wedge \exists y.r(z,y)) \ .$$

2.5.3 Remarks and Related Work

This section consists of a set of general remarks regarding higher-order rewriting. A short overview of related formalisms is given.

Loose practice. In examples we will write down HRSs whose rules are not closed. In this case we will write the free variables with capitals. This abuse of language is not problematic, because the rules can always be closed by prefixing the left- and right hand side with $\lambda\vec{X}$, where $\vec{X}$ are the variables occurring free in the rule. We call the result of this operation the *closure* of the rule. It is defined modulo the order of the free variables. Note that the rewrite relation does not depend on the choice of the order of these variables. So such ill-formed rules can easily be turned into well-formed rules. The fact that the rules are closed is convenient in the definition of higher-order rewriting and in proofs because they make substitutions superfluous, but counter intuitive in examples. This is the reason that we allow the loose notation.

HRSs based on β. As we noticed before, although the substitution calculus for HRSs is $\lambda^{\rightarrow}_{\beta\overline{\eta}}$, only $=_\beta$ plays a rôle in the computation of the rewrite relation. It is also possible to introduce HRSs with $\lambda^{\rightarrow}_{\beta}$ as a substitution calculus, which we will call β-HRSs in the sequel. The only difference is that the rules and contexts are only required to be in β-normal form. So a β-HRS is a bit more liberal.

For most second order HRSs, the $\beta\overline{\eta}$-discipline is more intuitive. It is for example quite strange to distinguish between $\forall P$ and $\forall\lambda x.Px$. Working modulo $\beta\overline{\eta}$ makes these two equal. However, for HRSs with arbitrary high type it is more intuitive to admit terms that are not $\beta\overline{\eta}$-normal form. Not only because η-expansions are rather space consuming, but it is quite impossible to write down the η-expansion of e.g. the first higher-order recursion rule schema $R_\sigma fx0 \mapsto x$, because its form depends on σ.

For the termination method that we will present, it is immaterial whether we work with $\beta\overline{\eta}$-HRSs or with β-HRSs. The reason is that ordinary HRSs can be seen as a restriction on β-HRSs, for which we will prove correctness of our method.

In Section 4.5 HRSs based on $\lambda^{\times}_{\beta}$ will be introduced. This is a real extension.

Decidability and patterns. Note that the question whether a certain rule L is applicable to a certain term M amounts to finding a suitable context $C[]$; this is equivalent to the higher-order matching problem $\exists X.M =_{\beta\overline{\eta}} (XL)$? Decidability of higher-order matching is still an open question. For this reason, the left hand sides of the rules are often restricted to patterns, for which higher-order matching is decidable. An additional advantage of using patterns is that it guarantees that the matching problem above has at most one solution.

We also admit HRSs with non-pattern left hand sides. The reason for this is that using patterns is only a sufficient and not a necessary condition for decidability and uniqueness of matching problems. We will encounter several unproblematic HRSs that have non-pattern left hand sides.

Other approaches. There are many different definitions of higher-order (term) rewrit(e)(ing) in recent literature. We only mention a few mile stones. For a historical overview of the several formats and a technical comparison between several of them, see e.g. [OR94, Oos94, Raa96].

In 1980 Klop introduced *combinatory reduction systems* (CRS) [Klo80, KOR93]. This is the first systematic study of TRSs with bound variables (lambda calculus with particular extensions had already been studied before; Aczel [Acz78] already considered general extensions of lambda calculus). In combinatory reduction systems, untyped lambda calculus is used as substitution calculus. Instead of reduction to normal form (which is impossible in untyped lambda calculus) developments are used. The left hand sides of the rules are restricted to patterns.

The systems in this thesis are inspired by and very similar to Nipkow's higher-order rewrite systems [Nip91, Nip93]. The main difference is that Nipkow builds in the restriction to patterns. This is however inspired by his work on confluence, and plays a less important rôle for termination. Minor differences are that Nipkow requires the rules to be of base type (where we require them to be closed) and defines the rewrite relation in terms of contexts, substitutions and $\beta\overline{\eta}$-reduction.

Wolfram studies higher-order term rewriting systems [Wol93]. These systems are the same as the HRSs that we study, up to minor differences. Wolfram requires the rules of base type, and also defines a rewrite step in terms of contexts, substitutions and $\beta\overline{\eta}$-reduction, but these differences can be seen as presentation matters only.

We also mention Van Oostrom [Oos94] and Van Raamsdonk [Raa96], who introduced the so called Higher-Order Rewriting Systems (HORS), meant to generalize all existing formalisms. HORSs parametrize over the substitution calculus that is used. Our HRSs, β-HRSs and the HRSs based on $\lambda^{\times}_{\beta}$ are instances of Van Oostrom's HORSs.

Quite another approach can be found in [Bre88, Dou92], where the typed lambda calculus is not used as a substitution calculus. Instead of this, a first-order TRS is combined with the β-reduction from the simply-typed, or even polymorphic lambda calculus. Each reduction step is either a TRS-step (performed on lambda-terms) or a β-reduction step. The reduction relation is much simpler, because matching is not done modulo a theory, but it is just a syntactic matter. In [JO91] this is extended to higher-order rules of a certain format. Also higher type systems can be used, e.g. in [BFG94] the combination of the Calculus of Constructions with a set of higher-order rewrite rules is studied.

In the approach using lambda calculus as substitution calculus it is possible to formulate beta-reduction on the rewrite level. Therefore HRSs subsume the direct combination of higher-order rewrite rules with lambda calculus.

2.5.4 Examples of Higher-order Rewrite Systems

We will now give some examples of HRSs, to give an idea what can be done with them. We first give an HRS that finds prenex normal forms. In this HRS substitution plays a marginal rôle. We also show how the untyped lambda calculus (with β- and η-reduction) can be presented as an HRS. Much more involved HRSs can be found in Section 5.2 and 5.5. Some of these examples have as substitution calculus $\lambda^{\times}_{\beta}$, simply typed lambda calculus extended with product types.

2.5.4.1 Finding the prenex-normal form

Let us consider finding the prenex normal form of first-order logical formulae. In this normal form, the quantifiers only occur at the outside of the formula. We deal with formulae of the following form (where $t_1, \ldots, t_n$ are arbitrary first-order terms):

$$p(t_1, \ldots, t_n) \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi \mid \forall x.\varphi \mid \exists x.\varphi \ .$$

The quantifiers $\forall$ and $\exists$ act as binders; we let the connectives $\wedge$ and $\vee$ bind stronger than the quantifiers; $\neg$ binds the strongest.

The higher-order signature for formulae has base types o for formulae and ι for individuals. The constants are:

$p : \vec{\iota}_n \to o$	for each n-ary predicate symbol p.
$\wedge : o \to o \to o$	
$\vee : o \to o \to o$	
$\neg : o \to o$	
$\forall : (\iota \to o) \to o$	
$\exists : (\iota \to o) \to o$	

Every logical formula can be represented by a $\beta\overline{\eta}$-normal form of type o in this signature, whose free variables have type ι. Conversely, each such term corresponds directly to a logical formula.

A classical result is, that each formula is logically equivalent to a formula in prenex normal form, that is, a formula of the form $Q_1 x_1, \ldots, Q_n x_n.\varphi$, where the Q_i are the quantifiers $\forall$ or $\exists$ and φ doesn't contain any of these quantifiers. The standard way to prove that each formula has a logically equivalent prenex normal form is by giving a method to push the quantifiers outside step by step. The following collection of rules suffices:

$$\begin{array}{rcl}
P \wedge \forall\lambda x.(Q\,x) & \mapsto & \forall\lambda x.P \wedge (Q\,x) \\
(\forall\lambda x.(Q\,x)) \wedge P & \mapsto & \forall\lambda x.(Q\,x) \wedge P \\
P \vee \forall\lambda x.(Q\,x) & \mapsto & \forall\lambda x.P \vee (Q\,x) \\
(\forall\lambda x.(Q\,x)) \vee P & \mapsto & \forall\lambda x.(Q\,x) \vee P \\
P \wedge \exists\lambda x.(Q\,x) & \mapsto & \exists\lambda x.P \wedge (Q\,x) \\
(\exists\lambda x.(Q\,x)) \wedge P & \mapsto & \exists\lambda x.(Q\,x) \wedge P \\
P \vee \exists\lambda x.(Q\,x) & \mapsto & \exists\lambda x.P \vee (Q\,x) \\
(\exists\lambda x.(Q\,x)) \vee P & \mapsto & \exists\lambda x.(Q\,x) \vee P \\
\neg\forall\lambda x.(Q\,x) & \mapsto & \exists\lambda x.\neg(Q\,x) \\
\neg\exists\lambda x.(Q\,x) & \mapsto & \forall\lambda x.\neg(Q\,x)
\end{array}$$

Here P^o and $Q^{\iota\to o}$ are variables. Remember that the actual rules, as used in the rewrite relation as defined in Definition 2.5.2, are closed forms of these rules. We put $\mathcal{H}_{\text{pnf}}$ the HRS $(\mathcal{C}, \mathcal{V}, R)$, where $\mathcal{C}$ are the constants and R the rules as indicated above; $\mathcal{V}$ is a sufficiently large set of variables.

It is easy to see, that if a formula does not contain one of the left hand sides as a subterm, then it is in prenex normal form. If a formula contains an occurrence of the left hand sides, then it cannot be in prenex normal form. In this case, we replace that occurrence by the corresponding right hand side. When this process terminates, a prenex normal form is reached. This termination issue will be deferred to Chapter 5.

2.5.4.2 From TRS to HRS

Any TRS $(\mathcal{F}, \mathcal{V}, \mathcal{R})$ can be converted into an HRS. This is done by currying the signature and the rules, and then closing the rules.

Put $\mathcal{B} := \{o\}$, the sort of first-order terms. For any $f \in \mathcal{F}$ with arity n, we include a constant symbol $f' : \underbrace{o \to \cdots \to o}_{n \text{ times}} \to o$ in $\mathcal{F}'$. $\mathcal{V}'$ is a sufficiently large set of typed variables that contains $\mathcal{V}$ as variables of type o. Now terms in $(\mathcal{F}, \mathcal{V})$ can be curried as follows. We write $\langle t \rangle$ for the curried version of t; it will be of type o.

$$\begin{array}{rcl}
\langle x \rangle & \equiv & x \\
\langle f(t_1, \ldots, t_n) \rangle & \equiv & (f'\,\langle t_1 \rangle \cdots \langle t_n \rangle)
\end{array}$$

For any rule $l \mapsto r \in \mathcal{R}$, we include the rule $\lambda\vec{x}.\langle l \rangle \mapsto \lambda\vec{x}.\langle r \rangle$ in $\mathcal{R}'$, where $\vec{x}$ are the free variables that occur in l. By the definition of first-order rule, r does not contain

additional variables. We call the HRS $(\{o\}, \mathcal{F}', \mathcal{V}', \mathcal{R}')$ the *curried version* of the TRS $(\mathcal{F}, \mathcal{V}, \mathcal{R})$.

The curried version of a TRS contains more typable terms than the original TRS. In particular, it may contain lambdas. Note however that we require all terms in the rewrite relation to be in β-normal form, so the lambdas can only occur at "non-interesting" places. The curried version will be used later in order to add higher-order rules to an arbitrary TRS, e.g. β-reduction on the rewrite level. The same translation occurs in [Oos94].

2.5.4.3 Untyped lambda calculus

Another typical example of HRSs is the untyped lambda calculus, with β and η-reduction. This HRS has a signature with as only base type o, the type of untyped lambda terms. There are two constants, for application and lambda abstraction:

$$\begin{array}{l} \mathsf{app} : o \to o \to o \\ \mathsf{abs} : (o \to o) \to o \ . \end{array}$$

Note that it is possible to construct untypable terms, like $(\lambda x.xx)(\lambda x.xx)$ in this signature. The latter term is encoded as:

$$\mathsf{app}\,(\mathsf{abs}\,\lambda x.(\mathsf{app}\,xx))(\mathsf{abs}\,\lambda x.(\mathsf{app}\,xx)) \ .$$

The HRS of β- and η-reduction has the following rules:

$$\begin{array}{rcl} \mathsf{app}\,(\mathsf{abs}\,\lambda x^o.Fx)Y & \mapsto & (FY) \\ \mathsf{abs}\,\lambda x^o.(\mathsf{app}\,Yx) & \mapsto & Y \ . \end{array}$$

Now $F^{o\to o}$ and Y^o are free variables, to be abstracted in order to obtain the proper rewrite rules.

Note that the substitution needed to perform the β-reduction is now performed by the substitution calculus. Furthermore, the proviso that is usually attached to the η-rule is now implicit, due to the variable convention.

As an example, it is now shown how the term given above, rewrites to itself. This rewrite step uses the context $(\Box(\lambda x.(\mathsf{app}\,xx))(\mathsf{abs}\,\lambda x.(\mathsf{app}\,xx)))$. We now get β-expansion, literal replacement and β-reduction as follows:

$$\begin{array}{rl} & \mathsf{app}\,(\mathsf{abs}\,\lambda x.(\mathsf{app}\,xx))(\mathsf{abs}\,\lambda x.(\mathsf{app}\,xx)) \\ \twoheadleftarrow_\beta & (\lambda F^{o\to o}\lambda Y^o.\mathsf{app}\,(\mathsf{abs}\,\lambda x^o.Fx)Y)\ \underline{(\lambda x.(\mathsf{app}\,xx))}\ \underline{(\mathsf{abs}\,\lambda x.(\mathsf{app}\,xx))} \\ \mapsto & (\lambda F^{o\to o}\lambda Y^o.(FY))\ \underline{(\lambda x.(\mathsf{app}\,xx))}\ \underline{(\mathsf{abs}\,\lambda x.(\mathsf{app}\,xx))} \\ \twoheadrightarrow_\beta & \mathsf{app}\,(\mathsf{abs}\,\lambda x.(\mathsf{app}\,xx))(\mathsf{abs}\,\lambda x.(\mathsf{app}\,xx)) \ . \end{array}$$

For clarity, we underlined the arguments corresponding to F and Y.

Although we only needed a second-order signature, this example exhibits the advantages of the HRS-framework. First, lambda-calculus can be expressed in simple rules, without any side conditions. This enables the study of lambda calculus in a general setting. E.g. results on critical pairs to prove local confluence for HRSs can be directly applied to lambda calculus [Nip91].

A second advantage is that we can add more rules to this system. This admits a uniform approach for the study of definitional extensions of lambda calculus. In Section 5.2.1 we will see how to encode simply typed lambda calculus as an HRS and we will study extensions of it, like Gödel's T (Section 5.3).

Chapter 3

The Semantical Approach to Termination Proofs

In the previous chapter, higher-order term rewriting was introduced as a combination of first-order term rewriting and simply-typed lambda calculus. This chapter serves as an illustration how the *semantics* of such systems can be used to obtain termination proofs for them.

The usual semantics of a TRS is an algebra, which is a set with certain functions. The terms can be interpreted in this set; the algebra is a model if the rewrite rules, interpreted as equations, hold. The standard semantics for simply typed lambda calculus consists of a type structure, more particularly the functionals of finite type. Each lambda term can be interpreted as a functional, in such a way that the (β)- and (η)-equalities hold.

The structures that are used in termination proofs are no models in this equational sense. We again take an algebra or the type structure of functionals and interpret the terms in it. But now the interpretation is non-standard. Instead of being equal, the left and right hand side of the rules are ordered in some well-founded partial order. Under certain extra conditions such an interpretation is called a *termination model*. Every rewrite step gives rise to a decrease of the corresponding values in the termination model. Hence the existence of a termination model ensures termination.

In Section 3.1 we recapitulate how termination proofs for TRSs can use monotone algebras [Zan94]. In Section 3.2 we show the standard model of functionals of finite type for the simply-typed lambda calculus and we show in Section 3.3 how a subset of them, the hereditarily monotonic functionals can be used to prove termination of this calculus [Gan80]. Finally, in Section 3.4, we investigate how similar techniques can be used for termination proofs of higher-order term rewriting systems. The latter section also contains a short overview of other approaches to termination proofs of higher-order rewrite systems.

This chapter contains no new results, but it is an introduction to the theory and methods in Chapter 4 and 5. In Chapter 6 we analyze the proof introduced in

Section 3.3.

3.1 Monotone Algebras for Termination of TRSs

Given a signature $\Sigma = (\mathcal{F}, \mathcal{V})$, an algebra is a structure $(A, (f_A)_{f \in \mathcal{F}})$, such that whenever $f \in \mathcal{F}$ and $arity(f) = n$, $f_A \in A^n \Rightarrow A$. Given such an algebra, we can interpret any closed term in a canonical way. To interpret open terms, we have to know the value of the variables. A valuation is a function in $\mathcal{V} \Rightarrow A$. We write $[\![s]\!]^{\mathcal{A}}_{\alpha}$ for the interpretation of s in the algebra $\mathcal{A}$, relative to the valuation α. If $\mathcal{A}$ is clear from the context, we write $[\![s]\!]_{\alpha}$. This notion is defined with induction on s as follows:

$$\begin{array}{rcl} [\![x]\!]_{\alpha} & = & \alpha(x) \\ [\![f(t_1, \ldots, t_n)]\!]_{\alpha} & = & f_A([\![t_1]\!]_{\alpha}, \ldots, [\![t_n]\!]_{\alpha}) \end{array}$$

Such an algebra is a model of a TRS (Σ, R), if for each rule $l \mapsto r \in R$, and for all valuations α, $[\![l]\!]_{\alpha} = [\![r]\!]_{\alpha}$. That is, all equations hold.

3.1.1 Monotone Algebras

For termination proofs, we are interested in a *termination model*. The algebra is extended with a partial order $>$. We say that a rule $l \mapsto r \in R$ is decreasing, if for any valuation α, $[\![l]\!]_{\alpha} > [\![r]\!]_{\alpha}$. In that case, we also write $l > r$.

If a rule $l \mapsto r$ is decreasing, then the quantification over all valuations above guarantees that for all substitutions θ, $l^{\theta} > r^{\theta}$. If moreover all function symbols are interpreted in a strictly monotonic way, then we obtain for any context $C[\,]$, $C[l^{\theta}] > C[r^{\theta}]$. Thus, any rewrite sequence gives rise to a decreasing sequence in the algebra. This ensures termination, provided the partial order $>$ is well-founded. This informal explanation justifies the following definitions and theorem.

Recall that given a partial order $(A, >)$, we call $f \in A^n \Rightarrow A$ *strictly monotonic* if for all $x, y \in A$, with $x > y$ we have $f(\ldots, x, \ldots) > f(\ldots, y, \ldots)$ (also for all possible arguments on the $\ldots$).

Definition 3.1.1 *A* monotone algebra *for* $\Sigma = (\mathcal{F}, \mathcal{V})$ *is a structure* $(A, >, (f_A)_{f \in \mathcal{F}})$, *such that*

1. $(A, >)$ *is a non-empty partial order; and*
2. *For each* $f \in \mathcal{F}$, *with* $arity(f) = n$, f_A *is a strictly monotonic function in* $A^n \Rightarrow A$.

The monotone algebra is well-founded, *if* $>$ *is well-founded.*

A rule $l \mapsto r$ is called *decreasing* in $\mathcal{A}$ if for any valuation α, $[\![l]\!]^{\mathcal{A}}_{\alpha} > [\![r]\!]^{\mathcal{A}}_{\alpha}$.

Definition 3.1.2 *A* termination model *for a TRS* (Σ, R) *is a well-founded monotone algebra for* Σ, *such that each rule in* R *is decreasing.*

Theorem 3.1.3 ([HO80, Zan94]) *Any TRS has a termination model if and only if it is terminating.*

Proof: The *only-if* part has been sketched above. The *if* can be seen as follows: take $\mathcal{A} = (\mathcal{T}(\Sigma), \to_{\mathcal{R}}^+, (f_A)_{f\in\mathcal{F}})$, where $f_A(t_1, \ldots t_n) = f(t_1, \ldots, t_n)$. This is the open term model.

This is a monotone algebra, because every function symbol f is strictly monotonic, for if $s \to_{\mathcal{R}}^+ t$ then $f(\ldots, s, \ldots) \to_{\mathcal{R}}^+ f(\ldots, t, \ldots)$. Also, $\to_{\mathcal{R}}^+$ is well-founded, because $\to_{\mathcal{R}}$ is terminating by assumption. The algebra is a termination model, because for any rule $l \mapsto r$ and substitution θ (that now plays the rôle of a valuation), $l^\theta \to_{\mathcal{R}}^+ r^\theta$. ⊠

The theorem provides for a complete method to prove termination. Note that it doesn't give an algorithm. A clever guess of the partial order and the interpretation has to be made. The test whether a given algebra is monotone, or whether a monotone algebra is a termination model is undecidable too. Nevertheless, the method turns out to be helpful to find many termination proofs. Completeness of the method can be used to prove several modularity results (see e.g. Section 5.2.3).

Disjunctive normal form.

To illustrate the use of Theorem 3.1.3, we provide a termination model for the example of Section 2.3, computing the disjunctive normal form of propositional formulae. We decide to try (a subset of) the natural numbers as domain, because this makes the computations much easier.

Note that taking multiplication for $\wedge$ and addition for $\vee$, would yield true equations for the first two rules. To get a decreasing rule, we have to add a slight distortion. Hence we put:

$$\begin{aligned} F_\wedge(x, y) &= x \cdot y \\ F_\vee(x, y) &= x + y + 1 \end{aligned}$$

Clearly $F_\vee$ is strictly monotonic. However, $F_\wedge$ is only strictly monotonic if we restrict the domain to $\mathbb{N}_{\geq 1}$. We can now verify that the first two rules are decreasing. Let α be an arbitrary valuation; for the moment we do not distinguish between x and $\alpha(x)$, just as in ordinary high-school algebra. The left- and right hand side of the first rule can be computed (the first two rules are similar, because $F_\wedge$ and $F_\vee$ are symmetric):

$$[\![x \wedge (y \vee z)]\!] = x \cdot (y + z + 1) = x \cdot y + x \cdot z + x \ , \text{ and}$$

$$[\![(x \wedge y) \vee (x \wedge z)]\!] = x \cdot y + x \cdot z + 1$$

If we restrict our domain to $\mathbb{N}_{\geq 2}$, then the former is greater than the latter.

We still have to provide an interpretation of $\neg$, such that the last three rules are decreasing. The fourth is the most problematic one, for we need $F_\neg(x + y + 1) >$

$F_\neg(x) \cdot F_\neg(y)$. So a sum in the arguments has to overcome a product in the result. This requires an exponential function. Hence we put:

$$F_\neg(x) = 2^x \ .$$

We can now verify that the last three rules are decreasing in $\mathbb{N}_{\geq 2}$:

$$[\![\neg(x \wedge y)]\!] = 2^{x \cdot y} > 2^x + 2^y + 1 = [\![\neg x \vee \neg y]\!]$$

$$[\![\neg(x \vee y)]\!] = 2^{x+y+1} = 2^x \cdot 2^y \cdot 2 > 2^x \cdot 2^y = [\![\neg x \wedge \neg y]\!]$$

$$[\![\neg\neg x]\!] = 2^{2^x} > 2^x > x = [\![x]\!]$$

So in the well-founded monotone algebra $(\mathbb{N}_{\geq 2}, >, F_\wedge, F_\vee, F_\neg)$ all five rules are decreasing, so it is a termination model. By Theorem 3.1.3 this proves termination of the TRS in Section 2.3 that computes the disjunctive normal form.

Remark. We can now also find a monotone algebra based on $\mathbb{N}$, by translating the functions by 2. We then find as interpretations:

$$\begin{array}{rcl} G_\wedge(x,y) & = & (x+2) \cdot (y+2) - 2 \\ G_\vee(x,y) & = & x + y + 2 \\ G_\neg(x) & = & 2^{x+2} - 2 \end{array}$$

3.1.2 More on Termination

Termination of TRSs is a quite well-studied topic. Huet and Lankford proved that termination of TRSs with finitely many rules is undecidable [HL78]. Therefore any method to prove termination is bound to be a semi-procedure. It is either limited to a subclass of TRSs, or the method itself cannot be automated.

Semi-decision procedures of the former kind are various forms of lexicographic and recursive path orders, see e.g. [Der82, Der87]. The approach using monotone algebras is an example of a method that cannot be automated itself, as it is complete. Such methods only give heuristics how to tackle termination proofs. Various combinations of these two approaches have been proposed, e.g. [KL80, Ges94, Zan95].

Another indication that termination is a difficult problem is that termination is not a modular property. This means that there are terminating TRSs, with disjoint sets of function symbols, whose union is not terminating. So a termination problem cannot be tackled by dividing a TRS into small parts, even not when these parts have nothing in common. The following example is due to Toyama [Toy87]:

$$\begin{array}{rcl} f(0,1,x) & \mapsto & f(x,x,x) \\ g(x,y) & \mapsto & x \\ g(x,y) & \mapsto & y \end{array}$$

Although the rule defining f is terminating, and the rules defining g are terminating too, there exists an infinite reduction in the combined system, starting with the term $f(0,1,g(0,1))$.

Rusinowitch [Rus87] and Middeldorp [Mid89] proved that it is essential for all such counter examples that one of the systems has a duplicating rule, and the other a collapsing rule. In other cases, termination is preserved by the disjoint union. As a consequence, if the TRS $(\mathcal{F}, \mathcal{V}, R)$ is terminating, then so is the TRS $(\mathcal{F} \uplus \mathcal{G}, \mathcal{V}, R)$, because $(\mathcal{G}, \mathcal{V}, \varnothing)$ has no rules, so in particular neither duplicating nor collapsing rules. Another (incomparable) modularity result is the following:

Proposition 3.1.4 *Let $\mathcal{R} = (\mathcal{F}, \mathcal{V}, R)$ be a terminating TRS. Then the TRS $\mathcal{R}' = (\mathcal{F} \uplus \{g\}, \mathcal{V}, R \uplus g(x, y) \mapsto x)$ is also terminating.*

This is a corollary of a more general theorem in [MOZ96]. The proof uses semantic self-labeling. Notice that adding the second projection too does not always preserve termination, as is witnessed by Toyama's example above. The previous proposition will be used in Section 5.2.3.

3.2 Functionals of Finite Type

As noted earlier (Section 2.4.1) the main intuition about simply-typed lambda calculus is, that the objects are functions that can be applied to each other in a type respecting way. This intuition is made formal, by designating a standard model for this calculus. This model will be the topic of the current section. The types are interpreted as sets of functions. The terms of a certain type as elements in the interpretation of that type.

Interpretation of types. In the standard model, types will be interpreted by sets. In particular, the type forming operator $\to$ will be interpreted by the full function space between its arguments. The interpretation of the basic types has to be specified in each model separately. We will call this choice the *interpretation key for the basic types*. Given a set of base types $\mathcal{B}$, such a key is a family of non-empty sets $(\mathcal{I}_\iota)_{\iota \in \mathcal{B}}$. Now the interpretation of types can be inductively defined as an extension of the interpretation key $\mathcal{I}$. We define $\mathcal{T}_\sigma$, the *set of functionals of type σ* by the following clauses:

$$\begin{aligned} \mathcal{T}_\iota &= \mathcal{I}_\iota \\ \mathcal{T}_{\sigma\to\tau} &= \mathcal{T}_\sigma \Rightarrow \mathcal{T}_\tau. \end{aligned}$$

Recall that $A \Rightarrow B$ denotes the set-theoretic function space between A and B. The set $\mathcal{T}_\sigma$ serves as the interpretation of type σ. The collection $(\mathcal{T}_\sigma)_{\sigma \in \mathbb{T}^\to(\mathcal{B})}$ will serve as the domain of the standard model. The objects in this domain are known as the *functionals of finite type*.

It is not necessary to take the full function space. An objection could be that the cardinality of $\mathcal{T}_\sigma$ grows with the complexity for σ. In the typical case of one base type o, interpreted by the natural numbers $\mathbb{N}$, the set $\mathcal{T}_{o\to o}$ already is uncountable. Smaller models have been studied extensively. These can be constructed by restricting

the function space to e.g. the computable functions. More general notions of type structures can be obtained by allowing an arbitrary interpretation of application. We restrict attention to full function spaces.

One important property of the model based on full function spaces is that equality is *extensional*. Extensionality means that for all types σ and functions $f, g \in \mathcal{T}_{\sigma\to\tau}$, we have the following: if for all $x \in \mathcal{T}_\sigma$, $f(x) = g(x)$ then $f = g$. The reason is that in set theory a function is identified by its graph, so if two functions agree on all arguments, they must be the same. This property will be used later on when proving that (η) holds in the type structure $\mathcal{T}$.

Interpretation of terms. We now switch from the interpretation of the types to that of terms. Again, we have to make a choice for the basic entities, being the constants and the variables. Given a signature $(\mathcal{B}, \mathcal{C}, \mathcal{V})$ and a base type interpretation key $\mathcal{I}$, an *interpretation key for the constants* is a family of functions $(\mathcal{J}_\sigma)_{\sigma\in\mathbb{T}^\to(\mathcal{B})}$, such that for each type σ, $\mathcal{J}_\sigma \in \mathcal{C}_\sigma \Rightarrow \mathcal{T}_\sigma$.

A choice for the value of the variables is called a *valuation*. It is a type respecting function from variables into the standard model. More formally, a valuation for a signature $(\mathcal{B}, \mathcal{C}, \mathcal{V})$ with base type interpretation $\mathcal{I}$ is a family of functions $(\alpha_\sigma)_{\sigma\in\mathbb{T}^\to(\mathcal{B})}$, such that for each type σ, $\alpha_\sigma \in \mathcal{V}_\sigma \Rightarrow \mathcal{T}_\sigma$.

Given a valuation α, we write $\alpha[x := b]$ for the valuation that equals α on all variables except for x, where it takes the value b.

We are now able to define the interpretation of terms in a fixed signature $\mathcal{F} = (\mathcal{B}, \mathcal{C}, \mathcal{V})$. An interpretation key $\mathcal{I}$ for the types acts as implicit parameter. A valuation and an interpretation key for the constants is provided as explicit parameter. For each type $\sigma \in \mathbb{T}^\to(\mathcal{B})$, we denote the interpretation of a term $M : \sigma$ under the valuation α and constant interpretation $\mathcal{J}$ by $[\![M]\!]_{\alpha,\mathcal{J}} \in \mathcal{T}_\sigma$. The definition is by induction on M:

$$\begin{aligned}
[\![x]\!]_{\alpha,\mathcal{J}} &= \alpha(x) \text{ if } x \in \mathcal{V} \\
[\![c]\!]_{\alpha,\mathcal{J}} &= \mathcal{J}(c) \text{ if } c \in \mathcal{C} \\
[\![MN]\!]_{\alpha,\mathcal{J}} &= [\![M]\!]_{\alpha,\mathcal{J}}([\![N]\!]_{\alpha,\mathcal{J}}) \\
[\![\lambda x^\rho.M]\!]_{\alpha,\mathcal{J}} &= \boldsymbol{\lambda} a \in \mathcal{T}_\rho.[\![M]\!]_{\alpha[x:=a],\mathcal{J}}
\end{aligned}$$

The interpretation of constants is often fixed. The variables on the other hand have no fixed meaning, which is of course the essence of their use. We mostly suppress the subscript $\mathcal{J}$ and write $[\![M]\!]_\alpha$. If M is closed, we also write $[\![M]\!]$. This notation is justified by the fact that the denotation of closed terms M does not depend on a valuation. By induction on M one can prove the stronger statement, that the value of α on the free variables of M completely determines $[\![M]\!]_\alpha$.

Given a substitution θ and a valuation α, we write $\alpha \circ \theta$ for the new valuation that sends x to $[\![x^\theta]\!]_\alpha$ (given a fixed constant interpretation $\mathcal{J}$). Note that the valuation $\alpha \circ \{x \mapsto N\}$ equals $\alpha[x := [\![N]\!]_\alpha]$.

Equations β and η. We have prematurely called the type structure $\mathcal{T}$ the standard model. We still have to show that the equations β and η are satisfied. For η we can

use that equality is extensional. To show that β holds, we need the following technical lemma, which shows that substitutions and valuations commute as expected.

Lemma 3.2.1 (Substitution Lemma) *Let θ be a substitution, α a valuation and $\mathcal{J}$ an interpretation of the constants. Then for each term $M \in \Lambda^{\to}$ we have $[\![M^\theta]\!]_{\alpha,\mathcal{J}} = [\![M]\!]_{\alpha\circ\theta,\mathcal{J}}$.*

Proof: The proof is by induction on the structure of M. For the λ-case it is important that the induction hypothesis holds for arbitrary α.

- $M \equiv x \in \mathcal{V}$: by definition of $\alpha \circ \theta$.
- $M \equiv c \in \mathcal{C}$: $[\![c^\theta]\!]_{\alpha,\mathcal{J}} = \mathcal{J}(c) = [\![c]\!]_{\alpha\circ\theta,\mathcal{J}}$.
- $M \equiv NP$: Using the induction hypothesis for N and P, we have

$$[\![M^\theta]\!]_\alpha \;=\; [\![N^\theta]\!]_\alpha([\![P^\theta]\!]_\alpha) \;=\; [\![N]\!]_{\alpha\circ\theta}([\![P]\!]_{\alpha\circ\theta}) \;=\; [\![M]\!]_{\alpha\circ\theta} \;\;.$$

- $M \equiv \lambda x^\sigma.N$: Let $a \in \mathcal{T}_\sigma$. We first prove the following statement,

$$(*) \qquad \alpha[x := a] \circ \theta = (\alpha \circ \theta)[x := a] \;\;.$$

  By the variable convention, we may assume that the name x has been chosen such, that it is not in the domain or co-domain of θ, so $x^\theta \equiv x$, and for all y, $x \notin \mathrm{FV}(y^\theta)$. We now check the statement for every variable y. If $y \equiv x$, we have:

$$(\alpha[x := a] \circ \theta)(x) = [\![x^\theta]\!]_{\alpha[x:=a]} = a = ((\alpha \circ \theta)[x := a])(x)$$

  And for $y \not\equiv x$, we have

$$(\alpha[x := a] \circ \theta)(y) = [\![y^\theta]\!]_{\alpha[x:=a]} = [\![y^\theta]\!]_\alpha = (\alpha \circ \theta)(y) = ((\alpha \circ \theta)[x := a])(y)$$

  Having this, we proceed by the following calculation:

$$\begin{array}{lcll} [\![M^\theta]\!]_\alpha & = & \boldsymbol{\lambda} a \in \mathcal{T}_\sigma.([\![N^\theta]\!]_{\alpha[x:=a]}) & \\ & = & \boldsymbol{\lambda} a \in \mathcal{T}_\sigma.([\![N]\!]_{\alpha[x:=a]\circ\theta}) & \text{(by i.h.)} \\ & = & \boldsymbol{\lambda} a \in \mathcal{T}_\sigma.([\![N]\!]_{(\alpha\circ\theta)[x:=a]}) & \text{(by } *\text{)} \\ & = & [\![M]\!]_{\alpha\circ\theta} \;\;. & \end{array}$$

$\boxtimes$

Now we easily obtain the following proposition:

Proposition 3.2.2 *$\mathcal{T} \vDash \beta$ and $\mathcal{T} \vDash \eta$.*

Proof: We will show that both equalities hold for arbitrary open terms and for all valuations. Let an arbitrary valuation α be given.

We first prove that $[\![(\lambda x.M)N]\!]_\alpha = [\![M[x := N]]\!]_\alpha$. Using the definition of interpretation the left hand side can be computed and equals $[\![M]\!]_{\alpha[x:=[\![N]\!]_\alpha]}$. By Lemma 3.2.1 this equals $[\![M[x := N]]\!]_\alpha$. This shows that (β) holds.

Now we show that $[\![\lambda x.(Mx)]\!]_\alpha = [\![M]\!]_\alpha$. The left hand side is a function that, applied to arbitrary a, yields $[\![M]\!]_\alpha(a)$, as x does not occur free in M. So does the right hand side. By extensionality, (η) holds. ⊠

3.3 Monotonic Functionals for Termination of $\lambda^{\rightarrow}_{\beta}$

We now sketch Gandy's proof of strong normalization of the simply-typed lambda calculus. Gandy defines a subset of the full type structure: the hereditarily monotonic functionals. He observes that every λI-term is hereditarily monotonic. A translation of lambda terms into λI-terms gives the required strong normalization result.

This section is meant to give insight in the concepts used by Gandy. Our proposal to prove termination of HRSs (Chapter 4, 5) can be seen as a modification of these concepts. We will only give definitions, statements and some proof sketches to illustrate certain points. For detailed proofs and extensions to other systems, we refer to [Gan80].

For simplicity we assume that there is only one base type, called o. Furthermore, we assume that there exist constants $0 : o$, $Succ : o \to o$ and $+ : o \to o \to o$. So we work in a signature $(\{o\}, \mathcal{V}, \{0, Succ, +\})$ for some set of variables $\mathcal{V}$. We will use the standard model, with interpretation $\mathbb{N}$ for o and the usual zero, successor and addition functions on natural numbers for 0, *Succ* and $+$. Finally, with $>$ we denote the usual "greater than" relation on natural numbers.

3.3.1 Hereditarily Monotonic Functionals

The order $(\mathbb{N}, >)$ plays the same rôle for lambda terms of base type, as the monotone algebra in termination proofs of TRSs. For lambda terms of other types, new partial orders will be defined, by lifting $>$ to higher types. Simultaneously, a notion of monotonicity is defined on functionals in $\mathcal{T}_\sigma$, the *hereditarily monotonic functionals*.

The sets $\mathcal{HM}_\sigma$ (hereditarily monotonic functionals) and partial order ${}_{\mathrm{hm}}{>}_\sigma$ (greater than on hereditarily monotonic input) are defined by induction on σ.

Definition 3.3.1 *The sets $\mathcal{HM}_\sigma \subseteq \mathcal{T}_\sigma$ and the relations ${}_{\mathrm{hm}}{>}_\sigma \subseteq \mathcal{HM}_\sigma \times \mathcal{HM}_\sigma$ on them are defined by simultaneous induction on σ as follows:*

$$
\begin{array}{rcl}
\mathcal{HM}_o & = & \mathbb{N} \\
x \ {}_{\mathrm{hm}}{>}_o \ y & \Longleftrightarrow & x > y \\
\mathcal{HM}_{\sigma\to\tau} & = & \{f \in \mathcal{T}_{\sigma\to\tau} \mid \forall x, y \in \mathcal{HM}_\sigma. \\
 & & \qquad f(x) \in \mathcal{HM}_\tau \wedge (x \ {}_{\mathrm{hm}}{>}_\sigma \ y \Rightarrow f(x) \ {}_{\mathrm{hm}}{>}_\tau \ f(y))\} \\
f \ {}_{\mathrm{hm}}{>}_{\sigma\to\tau} \ g & \Longleftrightarrow & \forall x \in \mathcal{HM}_\sigma . f(x) \ {}_{\mathrm{hm}}{>}_\tau \ g(x)
\end{array}
$$

These notions depend on each other in the following way. Functionals in $\mathcal{HM}$ preserve the order $_{\rm hm}>$; on the other hand, $f\ _{\rm hm}> g$ holds if the functions f and g are pointwise related, but only on arguments in $\mathcal{HM}$.

By induction on the types it can be shown that $_{\rm hm}>$ is indeed a strict partial order. On type level 1, hereditary monotonicity coincides with the usual definition of strict monotonicity on number theoretic functions. As an example of type level 2, we remark that $\boldsymbol{\lambda} f \in \mathbb{N}\Rightarrow\mathbb{N}.f(1)\ _{\rm hm}>\ \boldsymbol{\lambda} f \in \mathbb{N}\Rightarrow\mathbb{N}.f(0)$, because for strict monotonic number theoretic functions f, $f(1) > f(0)$. Both functionals are hereditarily monotonic, for if f is pointwise greater than g, then $f(0) > g(0)$ and $f(1) > g(1)$.

Not every closed lambda term denotes a hereditarily monotonic functional; consider e.g. $\lambda xy.x$, the left projection. However, it can be proved that each closed λI-term denotes a hereditarily monotonic functional. Recall that in each subterm of a λI-term of the form $\lambda x.M$, x occurs free in M. A valuation α is called hereditarily monotonic, if for each variable x, $\alpha(x) \in \mathcal{HM}$. The following lemma can be proved straightforwardly, with simultaneous induction on the type of M.

Lemma 3.3.2 (Gandy)

1. *For each $M \in \Lambda^{\rightarrow}$-$I$ and hereditarily monotonic valuation α, $[\![M]\!]_\alpha \in \mathcal{HM}$.*
2. *If the variable x occurs free in term $M \in \Lambda^{\rightarrow}$-$I$, and the valuations α and β agree on all variables except x and $\alpha(x)\ _{\rm hm}>\ \beta(x)$ then $[\![M]\!]_\alpha\ _{\rm hm}>\ [\![M]\!]_\beta$.*

One can define an order on lambda terms in the following way:

$$M\ _{\rm hm}> N \iff \text{for all hereditarily monotonic valuations } \alpha, [\![M]\!]_\alpha\ _{\rm hm}>\ [\![N]\!]_\alpha.$$

As a corollary of the previous lemma, we obtain that λI-contexts preserve the order.

Corollary 3.3.3 *For any terms M and N and contexts $C[\,]$, if both $C[M]$ and $C[N]$ are λI-terms, $M\ _{\rm hm}> N$ then $C[M]\ _{\rm hm}> C[N]$.*

Proof: With $C[M]$ and $C[N]$, also their subterms M and N are λI-terms, hence their denotation is hereditarily monotonic by the first part of the Lemma 3.3.2. It follows from Lemma 3.3.2.(2), that for λI-contexts $D[\,]$, $D[M]\ _{\rm hm}> D[N]$. (Use that that for any valuation α, $[\![D[M]]\!]_\alpha = [\![D[\Box]]\!]_{\alpha[\Box:=[\![M]\!]_\alpha]}$).

However, $C[\,]$ is not necessarily a λI-term, for there can occur binders λx in $C[\,]$ such that x does not occur free in the corresponding subterm. Because $C[M]$ and $C[N]$ are λI-terms, such x's must occur free in M and N. Let $\vec{x}$ be the sequence of all such variables, then $C[\Box\vec{x}]$, $\lambda\vec{x}.M$ and $\lambda\vec{x}.N$ are λI-terms. Moreover, we have that $\lambda\vec{x}.M\ _{\rm hm}>\ \lambda\vec{x}.N$. Hence we may apply Lemma 3.3.2.(2), which yields

$$C[M] =_\beta C[(\lambda\vec{x}M)\vec{x}]\ _{\rm wm}> C[(\lambda\vec{x}N)\vec{x}] =_\beta C[N]\ .$$

$\boxtimes$

The proof is somewhat simpler than the corresponding one in [Gan80, 1.4]. Note that the corollary heavily depends on the restriction to λI-terms. Such terms denote hereditarily monotonic functionals, so they preserve the order in the desired way. This is not the case for arbitrary lambda terms: Although $M \ {}_{\mathrm{hm}}\!> N$, it cannot be the case that $(\lambda x.0)M \ {}_{\mathrm{hm}}\!> (\lambda x.0)N$, because both sides denote the same number.

The corollary is an important step in the proof. In the sequel, a translation M^* from $\Lambda^{\rightarrow}$ to $\Lambda^{\rightarrow}$-I will be given, for which the β-rule is decreasing. The previous corollary then ensures that a rewrite step in any λI-context corresponds to a decreasing step. Before doing so, it is shown that any type contains functionals in $\mathcal{HM}$.

3.3.2 Special Hereditarily Monotonic Functionals

On types of level 1, hereditary monotonicity coincides with the usual notion of strict monotonicity. So *Succ* and $+$, the usual successor and addition function are in $\mathcal{HM}$. These functions are lifted up through the type structure, by giving suitable λI-expressions, in the signature $(o, \mathcal{V}, \{0, \mathit{Succ}, +\})$. Because 0, *Succ* and $+$ are hereditarily monotonic, all λI-terms over these functions are in $\mathcal{HM}$ as well, by Lemma 3.3.2.

The pointwise extension of *Succ* and $+$ to higher types yields functions $\mathit{Succ}_\sigma \in \mathcal{T}_{\sigma\to\sigma}$ and $\oplus_\sigma \in \mathcal{T}_{\sigma\to\sigma\to\sigma}$ (written in infix notation) and are defined as follows:

$$\begin{aligned}\mathit{Succ}_{\sigma\to\tau} &= \lambda f^{\sigma\to\tau}.\lambda x^{\sigma}.\mathit{Succ}_\tau(fx)\\ \oplus_{\sigma\to\tau} &= \lambda f^{\sigma\to\tau}.\lambda g^{\sigma\to\tau}.\lambda x^{\sigma}.((fx)\oplus_\tau (gx))\ .\end{aligned}$$

In order to ensure that elements of $\mathcal{HM}$ exist for all types, Gandy defines the functionals $\mathsf{L}_\sigma \in \mathcal{HM}_\sigma$, based on 0 and $\oplus_\sigma$:

$$\begin{aligned}\mathsf{L}_o &= 0\\ \mathsf{L}_{o\to o} &= \lambda x^o.x\\ \mathsf{L}_{(\sigma\to\tau)\to o} &= \lambda f^{\sigma\to\tau}.\mathsf{L}_{\tau\to o}(f\mathsf{L}_\sigma)\\ \mathsf{L}_{\sigma\to(\tau\to\rho)} &= \lambda x^{\sigma}.\lambda y^{\tau}.(\mathsf{L}_{\sigma\to\rho}x)\oplus_\rho(\mathsf{L}_{\tau\to\rho}y)\end{aligned}$$

By Lemma 3.3.2 for each type σ, $\mathsf{L}_\sigma \in \mathcal{HM}_\sigma$. We let α_{L} be the valuation mapping each x^σ to L_σ.

3.3.3 Termination of (β).

At this point, we are quite near a termination argument of β-reduction in the simply-typed lambda calculus. To this end, a mapping of λ-terms to λI-terms is given. This translation meets the additional requirement that the translated left hand side of the β-rule is greater (in the sense of ${}_{\mathrm{hm}}\!>$) than the translated right hand side. The translation of M is denoted by M^*.

Definition 3.3.4

$$\begin{aligned}x^* &\equiv x\\ (MN)^* &\equiv M^*N^*\\ (\lambda x^\sigma.M^\tau)^* &\equiv \lambda x^\sigma.\mathit{Succ}_\tau(M^*\oplus_\tau(\mathsf{L}_{\sigma\to\tau}x))\end{aligned}$$

Here the $+(\mathsf{L}x)$ part is needed to ensure that M^* is a λI-term. The $Succ_\tau$-function is needed to achieve that the β-rule is decreasing. We indeed have

$$((\lambda x^\sigma.M^\tau)N)^* =_\beta Succ_\tau(M^*[x := N^*] \oplus_\tau (\mathsf{L}_{\sigma\to\tau}N^*)) \ {}_{\mathrm{hm}}{>}_\tau \ M^*[x := N^*] \ .$$

Theorem 3.3.5 *β-reduction terminates in the simply-typed lambda calculus. In particular, given a lambda term $M : \tau$, the length of each reduction sequence from M is at most $[\![\mathsf{L}_{\tau\to o}M^*]\!]_{\alpha_\mathsf{L}}$*

Proof: Let a reduction sequence $M_0 \to_\beta M_1 \to_\beta M_2 \to_\beta \cdots M_n$ of type τ be given. Take some $0 \leq i < n$. Let M_{i+1} be obtained from M_i by replacing a redex P by Q. As shown before, $P^* \ {}_{\mathrm{hm}}{>} \ Q^*$. Because both M_i^* (with subterm P^*) and M_{i+1}^* (with subterm Q^*) are λI-terms, Corollary 3.3.3 may be applied, yielding that $M_i^* \ {}_{\mathrm{hm}}{>} \ M_{i+1}^*$. By monotonicity of the L-functionals, also $\mathsf{L}_{\tau\to o}M_i^* \ {}_{\mathrm{hm}}{>} \ \mathsf{L}_{\tau\to o}M_{i+1}^*$. This means that for all hereditarily monotonic valuations the corresponding inequality holds, especially for the valuation α_L, so $[\![\mathsf{L}_{\tau\to o}M_i^*]\!]_{\alpha_\mathsf{L}} > [\![\mathsf{L}_{\tau\to o}M_{i+1}^*]\!]_{\alpha_\mathsf{L}}$. This inequality between natural numbers holds for any $0 \leq i < n$, so the length of the reduction sequence, n, is at most $[\![\mathsf{L}_{\tau\to o}M_0^*]\!]_{\alpha_\mathsf{L}}$. ⊠

3.4 Towards Termination of Higher-order Rewrite Systems

We have set ourselves the goal to develop theory that can serve as a tool to find termination proofs for higher-order rewrite systems. In the previous sections we recapitulated successful semantical approaches to termination proofs for first-order term rewriting and for lambda calculus. As higher-order rewriting systems form a combination of these two formalisms, we will also propose a semantical approach to prove their termination.

We will generalize the approach using monotone algebras of Section 3.1. In this method a termination proof for some TRS consists of an interpretation of the constants as strictly monotonic functions in some partial order, such that the rewrite rules are decreasing. These ingredients can be lifted to higher-order rewriting, by lifting partial orders through the types, and by defining an appropriate notion of strict monotonicity for functionals. The first idea could now be, to apply the work of Gandy, presented in Section 3.3. That is, strictly monotonic functions are generalized to hereditarily monotonic functionals, and any order on the base types can be lifted in the same way as $(\mathbb{N}, >)$ was lifted to ${}_{\mathrm{hm}}{>}$.

Unfortunately, this idea does not work immediately. This is mainly due to two reasons. First, Gandy's method is designed to prove termination of *β-reduction*, whereas in higher-order rewriting, both $\to_\beta$ and $\leftarrow_\beta$ are used. Therefore, we will look at interpretations that are invariant under β-reduction, instead of decreasing for β-reduction. The other reason is that Gandy's ordering is not really closed under taking contexts. It is only closed under taking λI-contexts.

So our task will be to design a strict partial order and a good notion of monotonicity together with an interpretation on lambda terms, such that the following requirements are met:

- the interpretation of β-equal terms is identical,
- the order must be closed under taking arbitrary context, provided the free variables and constants are interpreted in a certain monotonic sense.

Unfortunately, these requirements together are highly problematic. Assume that $a > b$ holds, for a and b of base type. By the second requirement, $(\lambda x.c)a$ must be greater than $(\lambda x.c)b$. However, by the first requirement, both terms are interpreted by the value of c.

We can escape from this problem, by weakening the second requirement. Inspection of Definition 2.5.2 reveals that it suffices to require that the order is closed under contexts that are in $\beta\overline{\eta}$-normal form, and the context $(\lambda x.c)\Box$ that we used above contains a β-redex.

However, still assuming $a > b$, by weakening the second requirement accordingly, we should at least have that $\lambda x.xa > \lambda x.xb$. But now we take as context $\Box(\lambda z.c)$, and again the first and second requirement together yield the problematic $c > c$.

As a solution we will propose the use of two distinct orders. For a rule $L \mapsto R$, we will require that $L >_1 R$. This should imply that for arbitrary $\beta\overline{\eta}$-normal contexts C, $C[L] >_2 C[R]$. Furthermore, both $>_1$ and $>_2$ will be invariant under β-reduction.

In the next chapter, we develop a theory concerning weakly monotonic and strict functionals. This study will yield the orders $>_1$ and $>_2$ with the required properties. In Chapter 5 we show how these orders can be used in termination proofs for higher-order rewrite systems.

Existing work on termination. Confluence for higher-order rewriting systems is rather well-studied. Klop (in the context of combinatory reduction systems), Nipkow and Van Oostrom are mainly concerned with confluence. Some of the main results are that orthogonal CRSs are confluent [Klo80], weakly orthogonal HRSs are confluent [Oos94, Raa96] and a critical pair lemma for HRSs [Nip91].

Remarkably, termination for higher-order rewriting is much less studied. Van Raamsdonk [Raa96] proves that outermost-fair rewriting is a normalizing strategy for orthogonal higher-order term rewriting. But note that termination (or strong normalization) requires that *all* reduction sequences end in a normal form.

Furthermore, for *orthogonal* CRSs, Klop [Klo80] gives a method to reduce strong normalization to weak normalization, which is often easier to prove. This work is preceded by [Ned73] and succeded by [Sør96].

As far as we know, [Pol94] provides the first method to prove termination of arbitrary HRSs, by using a semantical approach. This work is based on similar work for TRSs [Zan94] and for simply-typed lambda calculus [Gan80].

Kahrs [Kah95] shows that it is possible to use the hereditarily monotonic functionals in termination proofs for HRSs. He avoids constant functions by systematically

translating lambda terms to λI-terms (using the translation M^*). The price to be paid is that β-equal terms are not interpreted equally. As a consequence, testing whether a rule is decreasing involves a syntactical analysis of the applied substitution. His method is tailored to proving termination of extensions of simply-typed lambda calculus, the main example being a calculus with products and co-products. See also [Kah96].

Another approach to termination of HRSs can be found in [LS93, Lor94, LP95], where lexicographic path orders have been generalized from the first-order to the higher-order case. These approaches have a first-order flavor, because they generalize a method from first-order term rewriting. Their methods are restricted to second-order HRSs, with patterns in the left hand sides of the rules. Very recently, [JR96] defined a recursive path order on arbitrary $\beta\overline{\eta}$-normal forms.

Finally, we mention some work on termination for the direct combination of lambda calculus with rewrite rules. In [Bre88] it is proved that the combination of a terminating TRS with the simply-typed lambda calculus is still terminating. See Theorem 5.2.6 for an alternative proof. In [BG90] this result is extended to the polymorphic lambda calculus. In [JO91] a kind of primitive recursive format for higher-order rules is given, which combined with polymorphic lambda calculus guarantees termination. This is extended by [BFG94] to the calculus of constructions. In all these cases, the proofs are based on strong computability arguments, so essentially a method from the lambda calculus is used.

Chapter 4

Weakly Monotonic and Strict Functionals

In this chapter two classes of functionals are studied: the weakly monotonic and the strict functionals. The main motivation for introducing them is that they are useful in termination proofs for higher-order rewrite systems. This chapter is mainly based on [Pol94, PS95].

In Section 3.4 we pointed out why the hereditarily monotonic functionals are less suitable for termination proofs. Recall that all λI-terms are hereditarily monotonic (Lemma 3.3.2). In order to capture all lambda terms, we introduce the class of *weakly monotonic functionals*. In particular, a constant function is weakly monotonic. In Section 4.1 we define the weakly monotonic functionals, and prove that all lambda terms denote weakly monotonic functionals.

Section 4.2 is devoted to what we call ordered domains. These are collections of partial orders, for which the presence of binary strictly monotonic functions is guaranteed. The canonical example will be addition on natural numbers. We also show in this section that lifting addition to higher types in the usual way, does not destroy arithmetical laws, like associativity.

In Section 4.3, we define the set of *strict functionals*. This set will be a proper subset of the weakly monotonic functionals. It will also have more severe requirements than the hereditarily monotonic functions. In particular, the set of strict functionals is closed under application to weakly monotonic arguments. In this way, the presence of the weakly monotonic functionals is compensated. We will also provide some sufficient conditions, to prove that a given functional is strict.

In many applications, there is only one base type, which is interpreted by the natural numbers. We devote Section 4.4 to the particular case of the functionals based on natural numbers. We prove certain properties for this special case, that do not hold in all type structures.

The theory concerning weakly monotonic and strict functionals can be extended by considering product types. This extension is necessary in the example from proof

theory that will be treated in Section 5.5. The extension to product types is carried out in Section 4.5

To help the reader, we end this introduction with an overview of the different classes of order preserving function(al)s that have been and will be introduced.

strictly monotonic: (cf. Section 2.1) This is the notion the reader is likely familiar with. We only use it for functions of type level 1 (number theoretic functions) and not for functionals of higher type.

hereditarily monotonic: (cf. Definition 3.3.1) This generalizes "strict monotonicity" to higher types. The notion is due to Gandy, who used it for a termination proof of the simply-typed lambda calculus.

weakly monotonic: (cf. Definition 4.1.1) This is a generalization of the notion "non-decreasing" for number theoretic functions to higher types. It differs from hereditarily monotonic by including constant functions. It is also defined hereditarily.

strict: (cf. Definition 4.3.1) This notion is the most restrictive notion. Strict functionals preserve the order in a strict way, *even in the presence of weakly monotonic functionals.*

Hence the first notion is the familiar one, the second is only needed in Section 3.3. The latter two notions are involved in the method to prove termination that will be introduced in Chapter 5.

4.1 Weakly Monotonic Functionals

Let a set of base types $\mathcal{B}$ be given. Let each base type ι be interpreted by a partial order $(\mathcal{I}_\iota, >_\iota)$. Recall that the carrier sets of these partial orders generate the full type structure $(\mathcal{T}_\sigma)_{\sigma\in\mathbb{T}^\to(\mathcal{B})}$, as defined in Section 3.2. We will write $x \geq_\iota y$ for the reflexive closure of $>_\iota$, i.e. $x \geq_\iota y$ if and only if $x >_\iota y$ or $x = y$. In many examples, $\mathcal{B} = \{o\}$, in which case o is interpreted as $(\mathbb{N}, >)$.

We will now define the subset of *weakly monotonic functionals* and simultaneously, the preorder ${}_{\mathrm{wm}}\geq$ on them, to be pronounced as *greater than or equal on weakly monotonic arguments.*

Definition 4.1.1 *For any type $\sigma \in \mathbb{T}^\to(\mathcal{B})$, we define the set $\mathcal{WM}_\sigma \subseteq \mathcal{T}_\sigma$ and a relation ${}_{\mathrm{wm}}\geq_\sigma \subseteq \mathcal{WM}_\sigma \times \mathcal{WM}_\sigma$ by simultaneous induction on σ as follows:*

- $\mathcal{WM}_\iota = \mathcal{I}_\iota$ *for* $\iota \in \mathcal{B}$.
- $x \,{}_{\mathrm{wm}}\geq_\iota y$ *if and only if* $x \geq_\iota y$.
- $f \in \mathcal{WM}_{\sigma\to\tau}$ *if and only if*
  - $f \in \mathcal{T}_{\sigma\to\tau}$, *and*
  - $\forall x \in \mathcal{WM}_\sigma . f(x) \in \mathcal{WM}_\tau$, *and*

– $\forall x, y \in \mathcal{WM}_\sigma . x \ {}_{\rm wm}{\geq_\sigma}\ y \Rightarrow f(x) \ {}_{\rm wm}{\geq_\tau}\ f(y)$.

- $f \ {}_{\rm wm}{\geq_{\sigma\to\tau}}\ g$ *if and only if* $\forall x \in \mathcal{WM}_\sigma . f(x) \ {}_{\rm wm}{\geq_\tau}\ g(x)$.

The only difference with the hereditarily monotonic functionals defined in Section 3.3 is that $\geq_\iota$ is used on base types instead of $>_\iota$. This results in the fact that constant functions are weakly monotonic, although they are not hereditarily monotonic.

Another related notion is *strong majorization* [Bez86]. The (highly) recursive clause for the latter notion is: f majorizes g if and only if for all x, y that can be majorized, if x majorizes y then $f(x)$ majorizes both $f(y)$ and $g(y)$. This is defined simultaneously with MAJ, the set of those functionals that can be majorized by some functional. At first sight it appears that "f is weakly monotonic" and "f majorizes itself" are related, but the two are incomparable.

Note that ${}_{\rm wm}{\geq}$ is only a preorder and not a partial order, because anti-symmetry fails. It may happen that f and g coincide on weakly monotonic arguments (so that $f \ {}_{\rm wm}{\geq}\ g$ and $g \ {}_{\rm wm}{\geq}\ f$), but differ on other arguments, i.e. $f = g$ does not hold.

Having the weakly monotonic functionals at our disposal, we can now define the strict partial order ${}_{\rm wm}{>}$, *greater than on weakly monotonic input.*

Definition 4.1.2 *For each type σ the relation ${}_{\rm wm}{>_\sigma} \subseteq \mathcal{WM}_\sigma \times \mathcal{WM}_\sigma$ is defined by induction on the types as follows:*

- $x \ {}_{\rm wm}{>_\iota}\ y$ *if and only if* $x >_\iota y$.
- $f \ {}_{\rm wm}{>_{\sigma\to\tau}}\ g$ *if and only if* $\forall x \in \mathcal{WM}_\sigma . f(x) \ {}_{\rm wm}{>_\tau}\ g(x)$.

Note that although $\lambda x.x \ {}_{\rm wm}{\geq_{o\to o}}\ \lambda x.0$, it is neither the case that $\lambda x.x \ {}_{\rm wm}{>}\ \lambda x.0$, nor that $\lambda x.x = \lambda x.0$. So ${}_{\rm wm}{\geq}$ is not the reflexive closure of ${}_{\rm wm}{>}$, nor is ${}_{\rm wm}{>}$ the partial order generated by ${}_{\rm wm}{\geq}$. The following lemmata shed some light on the relationship between ${}_{\rm wm}{\geq}$ and ${}_{\rm wm}{>}$.

Lemma 4.1.3 *Let $f, g \in \mathcal{T}_{\sigma\to\tau}$ and $x, y \in \mathcal{T}_\sigma$ for arbitrary $\sigma, \tau \in \mathbb{T}^{\to}(\mathcal{B})$. We have*

1. *If $f \in \mathcal{WM}$ and $x \in \mathcal{WM}$ then $f(x) \in \mathcal{WM}$;*
2. *If $f \ {}_{\rm wm}{\geq}\ g$ and $x \in \mathcal{WM}$ then $f(x) \ {}_{\rm wm}{\geq}\ g(x)$;*
3. *If $f \ {}_{\rm wm}{>}\ g$ and $x \in \mathcal{WM}$ then $f(x) \ {}_{\rm wm}{>}\ g(x)$;*
4. *If $f \in \mathcal{WM}$ and $x \ {}_{\rm wm}{\geq}\ y$ then $f(x) \ {}_{\rm wm}{\geq}\ f(y)$.*

Proof: By definition. ⊠

Lemma 4.1.4 *For all $\sigma \in \mathbb{T}^{\to}(\mathcal{B})$ and $x, y, z \in \mathcal{WM}_\sigma$, we have the following:*

1. *If $x \ {}_{\rm wm}{>}\ y$ or $x = y$ then $x \ {}_{\rm wm}{\geq}\ y$.*
2. *If $x \ {}_{\rm wm}{\geq}\ y$ and $y \ {}_{\rm wm}{\geq}\ z$ then $x \ {}_{\rm wm}{\geq}\ z$.*

3. *If* $x \,{}_{\mathrm{wm}}\!\geq y$ *and* $y \,{}_{\mathrm{wm}}\!> z$ *then* $x \,{}_{\mathrm{wm}}\!> z$.

4. *If* $x \,{}_{\mathrm{wm}}\!> y$ *and* $y \,{}_{\mathrm{wm}}\!\geq z$ *then* $x \,{}_{\mathrm{wm}}\!> z$.

5. *If* $x \,{}_{\mathrm{wm}}\!> y$ *and* $y \,{}_{\mathrm{wm}}\!> z$ *then* $x \,{}_{\mathrm{wm}}\!> z$.

Proof: All five statements can be proved straightforwardly by induction on σ. In fact, (1) and (3) imply (5). ⊠

By (1) and (2), ${}_{\mathrm{wm}}\!\geq$ is a preorder. Anti-symmetry of ${}_{\mathrm{wm}}\!>_\sigma$ can also be shown by induction on σ, which together with (5) implies that ${}_{\mathrm{wm}}\!>$ is a strict partial order.

The following example shows that ${}_{\mathrm{wm}}\!\geq_{(o\to o)\to o}$ is not a partial order, but only a preorder:

EXAMPLE. Let $\mathcal{B} = \{o\}$, $\mathcal{I}_o = \mathbb{N}$ and $>_o\, = >$, the usual order on natural numbers. Let $F, G \in ((\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N})$ be the functionals defined as follows:

$$F(f) = \begin{cases} 1 \text{ if } f(0) > f(1), \\ 0 \text{ otherwise,} \end{cases}$$

and $G(f) = 0$ for all $f \in (\mathbb{N} \Rightarrow \mathbb{N})$. Clearly, $F \,{}_{\mathrm{wm}}\!\geq G$. Note that if $f(0) > f(1)$, then $f \notin \mathcal{WM}_{o\to o}$. Hence, also $G \,{}_{\mathrm{wm}}\!\geq F$. Nevertheless, F and G are not equal, so ${}_{\mathrm{wm}}\!\geq_{(o\to o)\to o}$ is not anti-symmetric. ⊠

Lambda terms are weakly monotonic. We will now prove that *all* lambda terms denote weakly monotonic functionals, provided the variables and constants denote weakly monotonic functionals. This should be compared with Lemma 3.3.2, stating that λI-terms are hereditarily monotonic.

From now on, we assume a fixed signature $\mathcal{F} = (\mathcal{B}, \mathcal{C}, \mathcal{V})$, interpreted in $(\mathcal{I}_\iota, >_\iota)_{\iota\in\mathcal{B}}$. We also assume a fixed constant interpretation $\mathcal{J}$. We will call a valuation α weakly monotonic, if $\alpha(x^\sigma) \in \mathcal{WM}_\sigma$, for all $x \in \mathcal{V}$. We write $\alpha \,{}_{\mathrm{wm}}\!\geq \beta$ for valuations α and β, if for all $x \in \mathcal{V}$, $\alpha(x) \,{}_{\mathrm{wm}}\!\geq \beta(x)$. The constant interpretation $\mathcal{J}$ is weakly monotonic if for all c, $\mathcal{J}(c) \in \mathcal{WM}$. Recall that $[\![M]\!]_{\alpha,\mathcal{J}}$ denotes the interpretation of M under the valuation α, relative to $\mathcal{J}$.

Proposition 4.1.5 *Let $\mathcal{J}$ be weakly monotonic. For each $M \in \Lambda^{\to}(\mathcal{F})$ and weakly monotonic valuations α, β, we have:*

1. $[\![M]\!]_{\alpha,\mathcal{J}} \in \mathcal{WM}$.

2. *If* $\alpha \,{}_{\mathrm{wm}}\!\geq \beta$ *then* $[\![M]\!]_{\alpha,\mathcal{J}} \,{}_{\mathrm{wm}}\!\geq [\![M]\!]_{\beta,\mathcal{J}}$.

Proof: (induction on the structure of M, for all valuations α and β.)
If $M \equiv x \in \mathcal{V}$:

1. $[\![M]\!]_{\alpha,\mathcal{J}} = \alpha(x)$ is weakly monotonic by assumption.

2. Let $\alpha \,{}_{\mathrm{wm}}\!\geq \beta$. Then $[\![M]\!]_{\alpha,\mathcal{J}} = \alpha(x) \,{}_{\mathrm{wm}}\!\geq \beta(x) = [\![M]\!]_{\beta,\mathcal{J}}$.

If $M \equiv c \in \mathcal{C}$:

1. $[\![M]\!]_{\alpha,\mathcal{J}} = \mathcal{J}(c)$ is weakly monotonic by assumption.

2. $[\![M]\!]_{\alpha,\mathcal{J}} = \mathcal{J}(c) = [\![M]\!]_{\beta,\mathcal{J}}$, so by Lemma 4.1.4.(1) $[\![M]\!]_{\alpha,\mathcal{J}} \mathrel{\mathrm{wm}}\geq [\![M]\!]_{\beta,\mathcal{J}}$.

If $M \equiv PQ$:

1. By induction hypothesis (1) both $[\![P]\!]_\alpha$ and $[\![Q]\!]_\alpha$ are weakly monotonic. Then by Lemma 4.1.3.(1), also $[\![M]\!]_\alpha = [\![P]\!]_\alpha([\![Q]\!]_\alpha)$ is weakly monotonic.

2. By induction hypothesis (2) $[\![P]\!]_\alpha \mathrel{\mathrm{wm}}\geq [\![P]\!]_\beta$. By induction hypothesis (1) $[\![Q]\!]_\alpha$ is weakly monotonic, so we have, with Lemma 4.1.3.(2), that $[\![P]\!]_\alpha([\![Q]\!]_\alpha) \mathrel{\mathrm{wm}}\geq [\![P]\!]_\beta([\![Q]\!]_\alpha)$. We also get from the induction hypotheses (1,2) that $[\![Q]\!]_\alpha \mathrel{\mathrm{wm}}\geq [\![Q]\!]_\beta$ and $[\![P]\!]_\beta$ is weakly monotonic. Therefore, with Lemma 4.1.3.(4) we get that, $[\![P]\!]_\beta([\![Q]\!]_\alpha) \mathrel{\mathrm{wm}}\geq [\![P]\!]_\beta([\![Q]\!]_\beta)$. Now by transitivity, we have $[\![M]\!]_\alpha \mathrel{\mathrm{wm}}\geq [\![M]\!]_\beta$.

If $M \equiv \lambda x.N$: Say $x \in \mathcal{V}_\sigma$

1. First, choose $a \in \mathcal{WM}_\sigma$, then $[\![\lambda x.N]\!]_\alpha(a) = [\![N]\!]_{\alpha[x:=a]}$. This is weakly monotonic by induction hypothesis (1). Furthermore, if $a \mathrel{\mathrm{wm}}\geq_\sigma b$, then $\alpha[x := a] \mathrel{\mathrm{wm}}\geq \alpha[x := b]$, so by induction hypothesis (2) $[\![N]\!]_{\alpha[x:=a]} \mathrel{\mathrm{wm}}\geq [\![N]\!]_{\alpha[x:=b]}$. This is equivalent to $[\![M]\!]_\alpha(a) \mathrel{\mathrm{wm}}\geq [\![M]\!]_\alpha(b)$, so $[\![M]\!]_\alpha$ is weakly monotonic.

2. Let $a \in \mathcal{WM}_\sigma$. Because $\alpha \mathrel{\mathrm{wm}}\geq \beta$, we have $\alpha[x := a] \mathrel{\mathrm{wm}}\geq \beta[x := a]$. So using induction hypothesis (2) we can compute: $[\![\lambda x.N]\!]_\alpha(a) = [\![N]\!]_{\alpha[x:=a]} \mathrel{\mathrm{wm}}\geq [\![N]\!]_{\beta[x:=a]} = [\![\lambda x.N]\!]_\beta(a)$. So indeed, $[\![M]\!]_\alpha \mathrel{\mathrm{wm}}\geq [\![M]\!]_\beta$. ⊠

This lemma can also be used to prove that certain functionals are weakly monotonic, namely if they can be expressed as lambda terms over known weakly monotonic functionals. The following is an immediate consequence of Proposition 4.1.5.(1)

Corollary 4.1.6 *Let $f_1, \ldots, f_n$ be weakly monotonic, and let α be a valuation, mapping x_i to f_i for $1 \leq i \leq n$. If g can be written as $[\![M]\!]_\alpha$, with $\mathrm{FV}(M) \subseteq \{x_1, \ldots, x_n\}$, then g is weakly monotonic.*

EXAMPLE. Multiplication ($*$) on natural numbers is weakly monotonic. The corollary can be used to show that taking squares is weakly monotonic, for the squaring function can be written as $\lambda x.(x * x)$. Similarly, the function mapping each $f \in (\mathbb{N} \Rightarrow \mathbb{N})$ to $f(f(0) * f(1))$ is weakly monotonic, because it can be written as $\lambda f.f(f(0) * f(1))$, where 0, 1 and $*$ are weakly monotonic. ⊠

This illustrates how weak monotonicity can be proved quite easily for a wide class of functionals.

4.2 Addition on Functionals

In Section 4.3.2, we will need that $(\mathcal{I}_\iota, >_\iota)_{\iota\in\mathcal{B}}$ admits enough strictly monotonic functions. We can construct strictly monotonic functions of all kinds, if for any combination $\iota, \kappa \in \mathcal{B}$, there exists a strictly monotonic function $+_{\iota,\kappa} \in \mathcal{T}_{\iota\to\kappa\to\iota}$. In this case, we will speak of an ordered domain.

Definition 4.2.1 *An* ordered domain *for a set of base types $\mathcal{B}$ is a structure $((\mathcal{I}_\iota, >_\iota, 0_\iota)_{\iota\in\mathcal{B}}, (+_{\iota,\kappa})_{\iota,\kappa\in\mathcal{B}})$, such that for each $\iota, \kappa \in \mathcal{B}$*

- *$(\mathcal{I}_\iota, >_\iota)$ is a partial order;*
- *$0_\iota \in \mathcal{I}_\iota$;*
- *$+_{\iota,\kappa} \in \mathcal{I}_\iota \Rightarrow \mathcal{I}_\kappa \Rightarrow \mathcal{I}_\iota$ is strictly monotonic.*

A well-founded domain *is an ordered domain, such that for each $\iota \in \mathcal{B}$, $>_\iota$ is well-founded.*

The existence of binary strictly monotonic functions excludes finite sets A with a non-empty order, for there is no binary strictly monotonic function in $A \Rightarrow A \Rightarrow A$. (Consider e.g. $A = \{0, 1\}$, then a binary strictly monotonic function cannot exist, for we need at least three values $0 + 0 < 0 + 1 < 1 + 1$.)

Requiring strictly monotonic functions implies that the sets $(\mathcal{I}_\iota)_\iota$ all have the same order type, which must be a power of ω, i.e. ω^γ for some ordinal γ (for the latter see [Fer95, Thm. 5.25]).

In the rest of this section, we work in a fixed ordered domain of the form $((\mathcal{I}_\iota, >_\iota, 0_\iota)_{\iota\in\mathcal{B}}, (+_{\iota,\kappa})_{\iota,\kappa\in\mathcal{B}})$, unless stated otherwise. We lift elements of $\mathcal{I}_\iota$ and the $+$ function to higher types in the standard way, by putting for each $n \in \mathcal{I}_\iota$,

$$\begin{array}{rcl} \underline{n}_\iota & = & n \\ \underline{n}_{\sigma\to\tau} & = & \boldsymbol{\lambda} x^\sigma . \underline{n}_\tau \end{array}$$

$$\begin{array}{rcl} x \oplus_{\iota,\kappa} y & = & x +_{\iota,\kappa} y \\ f \oplus_{\sigma\to\rho,\sigma\to\tau} g & = & \boldsymbol{\lambda} x^\sigma . f(x) \oplus_{\rho,\tau} g(x) \end{array}$$

More schematically, $(f \oplus g)(\vec{x}) = f(\vec{x}) + g(\vec{x})$, and $\underline{n}(\vec{x}) = n$. Note that $\oplus_{\sigma,\tau}$ is only defined if $\sigma \approx \tau$, i.e. σ and τ have the same factors. In that case, we have $\oplus_{\sigma,\tau} : \sigma \to \tau \to \sigma$. The lifted $\underline{n}_\sigma$ only exists if $res(\sigma) = \iota$.

Lemma 4.2.2 *Let $\sigma, \tau \in \mathbb{T}^\to(\mathcal{B})$ with $\sigma \approx \tau$; let $n \in \mathcal{I}_{res(\sigma)}$. Then we have*

1. *$\underline{n}_\sigma \in \mathcal{WM}_\sigma$;*
2. *$\oplus_{\sigma,\tau} \in \mathcal{WM}_{\sigma\to\tau\to\sigma}$.*

Proof: Immediate by Corollary 4.1.6. ⊠

Computation rules. To assist simple computations involving $\oplus$ and $\underline{n}$, we derive some properties. These properties of $\oplus$, $\underline{n}$, ${}_{\mathrm{wm}}\geq$ and ${}_{\mathrm{wm}}>$, rely on the corresponding properties for the underlying $+$, n, $\geq$ and $>$. We first concentrate on some equalities in the ordered domain $(\mathbb{N}, >, 0, +)$.

Lemma 4.2.3 *In the ordered domain $(\mathbb{N}, >, 0, +)$, the following statements hold, for all $\sigma \in \mathbb{T}^{\rightarrow}(\{o\})$ and $x, y, z \in \mathcal{T}_\sigma$ and $m, n \in \mathbb{N}$,*

1. $x \oplus y = y \oplus x$;
2. $(x \oplus y) \oplus z = x \oplus (y \oplus z)$;
3. $\underline{0} \oplus x = x \oplus \underline{0} = x$;
4. $\underline{m+n}_\sigma = \underline{m}_\sigma \oplus \underline{n}_\sigma$.

Proof: Straightforward induction on σ. We only treat (1). For $\sigma = o$, the statement is a true arithmetic equation. For $\sigma = \rho \to \tau$, let $a \in \mathcal{T}_\rho$ be given. Then $(x \oplus_{\sigma,\sigma} y)(a) = x(a) \oplus_{\tau,\tau} y(a)$. By induction hypothesis, the latter equals $y(a) \oplus x(a) = (y \oplus x)(a)$. This holds for any $a \in \mathcal{T}$, so $x \oplus y = y \oplus x$. ⊠

In general, given an ordered base type interpretation $(\mathcal{I}, >, 0, +)$ and a true equation in its language, then the equation obtained by replacing $+$ by $\oplus_{\sigma,\sigma}$ and 0 by $\underline{0}_\sigma$ is true in $\mathcal{T}_\sigma$. A similar preservation result holds if there is more than one base type, and if additional domain constants are used.

We now concentrate on inequalities.

Lemma 4.2.4 *Let $\sigma, \tau \in \mathbb{T}^{\rightarrow}(\mathcal{B})$ with $\sigma \approx \tau$; let $f, g \in \mathcal{WM}_\sigma$ and $h \in \mathcal{WM}_\tau$ and $m, n \in \mathcal{I}_\iota$, where $\iota = res(\sigma)$. We have*

1. *If $f \,{}_{\mathrm{wm}}> g$ then $f \oplus h \,{}_{\mathrm{wm}}> g \oplus h$;*
2. *If $f \,{}_{\mathrm{wm}}> g$ then $h \oplus g \,{}_{\mathrm{wm}}> h \oplus f$;*
3. *If $m >_\iota n$ then $\underline{m}_\sigma \,{}_{\mathrm{wm}}> \underline{n}_\sigma$.*

Proof: Straightforward by induction on σ. We only carry out the proof of (1). For base type, the proposition follows from strict monotonicity of $+_{\iota,\kappa}$.

Now assume that (1) holds for type τ. Let $f \,{}_{\mathrm{wm}}>_{\sigma\to\tau} g$ and $x \in \mathcal{WM}_\sigma$. By Lemma 4.1.3.(3), $f(x) \,{}_{\mathrm{wm}}> g(x)$. Using the induction hypothesis and the definition of $\oplus$ we get:

$$(f \oplus h)(x) = f(x) \oplus h(x) \,{}_{\mathrm{wm}}> g(x) \oplus h(x) = (g \oplus h)(x) \ .$$

This holds for arbitrary weakly monotonic x, so $f \oplus h \,{}_{\mathrm{wm}}> g \oplus h$. ⊠

Preservation statement. More generally, in any ordered domain $(\mathcal{I}, >, 0, +)$, if P is a true Horn clause in the language $\{+, 0, >, \geq\}$, then for any σ, the formula obtained by replacing each occurrence of $+$ by $\oplus_{\sigma,\sigma}$, 0 by $\underline{0}_\sigma$, $>$ by ${}_{\mathrm{wm}}{>}_\sigma$ and $\geq$ by ${}_{\mathrm{wm}}{\geq}_\sigma$ is a true formula in $\mathcal{WM}_\sigma$.

We will not give a formal formulation of the preservation statement, for it would be quite tedious to make it precise. The proof of the preservation statement is by induction on σ. On base type, we get a true equation by assumption. On type $\sigma \to \tau$, the statement reduces to an instance of the same statement on type τ, which is true by induction hypothesis.

The reason that the preservation statement holds, is that by definition, $f \,{}_{\mathrm{wm}}{\geq}\, g$ holds, if and only if for all $x \in \mathcal{WM}$, $f(x) \,{}_{\mathrm{wm}}{\geq}\, g(x)$, and similarly for ${}_{\mathrm{wm}}{>}$. In fact, Lemma 4.1.4.(2)–(5) are instances of this general preservation statement.

We give some examples on natural numbers that will be used later on.

Lemma 4.2.5 *In the ordered domain $(\mathbb{N}, >, 0, +)$ we have for any $\sigma \in \mathbb{T}^{\to}(\{o\})$, $m, n \in \mathbb{N}$ and $x, y \in \mathcal{WM}_\sigma$:*

1. $x \oplus y \,{}_{\mathrm{wm}}{\geq}\, x$;
2. $x \oplus \underline{1}_\sigma \,{}_{\mathrm{wm}}{>}\, x$.

Proof: By the preservation statement. ⊠

The preservation statement still holds when we allow equality in the hypotheses of the Horn clause, so that e.g. Lemma 4.1.4.(1) becomes a corollary. But allowing equality in the conclusion destroys the preservation statement, as is witnessed by the non-theorem $f \,{}_{\mathrm{wm}}{\geq}\, g \wedge g \,{}_{\mathrm{wm}}{\geq}\, f \Rightarrow f = g$, which is only true on base type.

4.3 Strict Functionals

We now introduce the strict functionals. Their distinguishing property is that they preserve the order in their arguments in a very strict way, namely even in the presence of weakly monotonic arguments. Consider the following two expressions:

$$f\vec{M}7\vec{N} \quad \text{and} \quad f\vec{M}3\vec{N},$$

with $\vec{M}$ and $\vec{N}$ arbitrary lambda terms. It is understood that $7 > 3$. For termination proofs, it is important that such an inequality is preserved by any lambda context. So the question becomes: for which functions f can we guarantee that the left expression above is greater than the right hand one? It is not surprising that we can find weakly monotonic f, such that the two expressions become equal, as constant functions are weakly monotonic.

However, also for hereditarily monotonic functionals, the required inequality may fail. This is due to the fact that $\vec{M}$ and $\vec{N}$ need not be λI-terms. So it is possible that $f\vec{M}$ is not hereditarily monotonic, although f is. Consequently, the inequality $7 > 3$ is not reflected by the context.

The set of strict functionals (denoted by $\mathcal{ST}$) will be defined such that they preserve the order. In particular, it will be closed under application to weakly monotonic arguments. We also define an order $\ {}_{\mathrm{st}}{>}$ which means *greater than on strict arguments*. We will be able to prove, that if $L\ {}_{\mathrm{wm}}{>}\ R$ and if all constants and variables are interpreted by strict functionals, then for any context C, $C[L]\ {}_{\mathrm{st}}{>}\ C[R]$.

In Section 4.3.1, we give a formal definition of the strict functionals and prove some properties about them. It is also proved there that contexts in $\beta\overline{\eta}$-normal form are order preserving in some way. In Section 4.3.2 some important examples of strict functionals are given. An easy method to prove strictness for a wide class of functionals is also provided there. Finally, the latter section includes a number of handy computation rules.

4.3.1 Definition and Properties

Let a set $\mathcal{B}$ of base types be given, together with an ordered type interpretation $(\mathcal{I}_\iota, >_\iota)_{\iota\in\mathcal{B}}$. We now define the set $\mathcal{ST}$ of *strict functionals* and the partial order $\ {}_{\mathrm{st}}{>}$ *greater than on strict arguments*, motivated before.

Definition 4.3.1 *For any type $\sigma \in \mathbb{T}^{\to}(\mathcal{B})$, we define the set $\mathcal{ST}_\sigma \subseteq \mathcal{WM}_\sigma$ and a relation $\ {}_{\mathrm{st}}{>}_\sigma \subseteq \mathcal{WM}_\sigma \times \mathcal{WM}_\sigma$ by simultaneous induction on σ as follows:*

- $\mathcal{ST}_\iota = \mathcal{I}_\iota$ *for* $\iota \in \mathcal{B}$.
- $x\ {}_{\mathrm{st}}{>}_\iota\ y$ *if and only if* $x >_\iota y$.
- $f \in \mathcal{ST}_{\sigma\to\tau}$ *if and only if*
  - $f \in \mathcal{WM}_{\sigma\to\tau}$, *and*
  - $\forall x \in \mathcal{WM}_\sigma . f(x) \in \mathcal{ST}_\tau$, *and*
  - $\forall x, y \in \mathcal{WM}_\sigma . x\ {}_{\mathrm{st}}{>}_\sigma\ y \Rightarrow f(x)\ {}_{\mathrm{wm}}{>}_\tau\ f(y)$.
- $f\ {}_{\mathrm{st}}{>}_{\sigma\to\tau}\ g$ *if and only if* $f\ {}_{\mathrm{wm}}{\geq}\ g$ *and* $\forall x \in \mathcal{ST}_\sigma . f(x)\ {}_{\mathrm{st}}{>}_\tau\ g(x)$.

We say that $f : \vec{\sigma}_n \to \iota$ is strict in its i-th argument, if for all $x_1 \in \mathcal{WM}_{\sigma_1}, \ldots, x_n \in \mathcal{WM}_{\sigma_n}$ and $y \in \mathcal{WM}_{\sigma_i}$, we have

$$x_i\ {}_{\mathrm{st}}{>}\ y \Rightarrow f(x_1, \ldots, x_i, \ldots, x_n) >_\iota f(x_1, \ldots, y, \ldots, x_n) \ .$$

Notice that f is strict if and only if it is strict in all its arguments. We will prove (Proposition 4.3.6) that the sets $\mathcal{ST}_\sigma$ are non-empty.

The following relationships hold by definition:

Lemma 4.3.2 *Let $f, g \in \mathcal{T}_{\sigma\to\tau}$ and $x, y \in \mathcal{T}_\sigma$ for arbitrary $\sigma, \tau \in \mathbb{T}^{\to}(\mathcal{B})$. Then we have*

1. *If $x \in \mathcal{ST}$ then $x \in \mathcal{WM}$, but not conversely;*
2. *If $f \in \mathcal{ST}$ and $x \in \mathcal{WM}$ then $f(x) \in \mathcal{ST}$;*

3. *If* $f \in \mathcal{ST}$ *and* $x \ {}_{\mathrm{st}}\!>\ y$ *then* $f(x) \ {}_{\mathrm{wm}}\!>\ f(y)$;

4. *If* $x \in \mathcal{ST}$ *and* $f \ {}_{\mathrm{st}}\!>\ g$ *then* $f(x) \ {}_{\mathrm{st}}\!>\ g(x)$;

5. *If* $x \ {}_{\mathrm{st}}\!>\ y$ *then* $x \ {}_{\mathrm{wm}}\!\geq\ y$.

Ad (1), we already noted that $\underline{0}_{o\to o}$ is weakly monotonic. It is clearly not strict, because e.g. for any two natural numbers m and n, $\underline{0}(m) = \underline{0}(n)$.

We also have the following properties:

Lemma 4.3.3 *Let* $x, y, z \in \mathfrak{T}_\sigma$ *for arbitrary* $\sigma \in \mathbb{T}^{\to}(\mathcal{B})$. *Then we have*

1. *If* $x \ {}_{\mathrm{wm}}\!>\ y$ *then* $x \ {}_{\mathrm{st}}\!>\ y$;

2. *If* $x \ {}_{\mathrm{wm}}\!\geq\ y$ *and* $y \ {}_{\mathrm{st}}\!>\ z$ *then* $x \ {}_{\mathrm{st}}\!>\ z$;

3. *If* $x \ {}_{\mathrm{st}}\!>\ y$ *and* $y \ {}_{\mathrm{wm}}\!\geq\ z$ *then* $x \ {}_{\mathrm{st}}\!>\ z$.

Proof: In all three cases, the main idea is, that if x is greater than y on weakly monotonic input, then on strict input this must be the case too, because $\mathcal{ST} \subseteq \mathcal{WM}$.

1. Induction on σ. For base type ι, both ${}_{\mathrm{wm}}\!>$ and ${}_{\mathrm{st}}\!>$ coincide with $>_\iota$. Assume that (1) holds for τ. Assume $f \ {}_{\mathrm{wm}}\!>_{\sigma\to\tau}\ g$ holds. By Lemma 4.1.4.(1), $f \ {}_{\mathrm{wm}}\!\geq\ g$. We show that for all strict x, $f(x) \ {}_{\mathrm{st}}\!>\ g(x)$. Let $x \in \mathcal{ST}_\sigma$. Then $x \in \mathcal{WM}_\sigma$ by Lemma 4.3.2.(1), so $f(x) \ {}_{\mathrm{wm}}\!>\ g(x)$ by Lemma 4.1.3.(3). By the induction hypothesis, $f(x) \ {}_{\mathrm{st}}\!>\ g(x)$.

2. Also induction on σ. On base type, the relationship holds. Now assume that (2) holds on type τ. Assume $f \ {}_{\mathrm{wm}}\!\geq_{\sigma\to\tau}\ g$ and $g \ {}_{\mathrm{st}}\!>_{\sigma\to\tau}\ h$. By Lemma 4.3.2.(5) and Lemma 4.1.4.(2), we have $f \ {}_{\mathrm{wm}}\!\geq\ h$. Now we show that for arbitrary $x \in \mathcal{ST}_\sigma$, $f(x) \ {}_{\mathrm{st}}\!>_\tau\ h(x)$. Assume $x \in \mathcal{ST}_\sigma$, then by Lemma 4.3.2.(1), $x \in \mathcal{WM}$; by Lemma 4.1.3.(2), $f(x) \ {}_{\mathrm{wm}}\!\geq\ g(x)$. By Lemma 4.3.2.(4), $g(x) \ {}_{\mathrm{st}}\!>\ h(x)$. Hence by induction hypothesis, $f(x) \ {}_{\mathrm{st}}\!>\ h(x)$.

3. Similar to (2). ⊠

From now on, applications of Lemma 4.1.3, 4.1.4, 4.3.2 and Lemma 4.3.3 will not always be mentioned. From Lemma 4.3.2.(5) and 4.3.3.(1) it is clear that ${}_{\mathrm{st}}\!>$ lies between ${}_{\mathrm{wm}}\!\geq$ and ${}_{\mathrm{wm}}\!>$. In fact, the part $f \ {}_{\mathrm{wm}}\!\geq\ g$ in Definition 4.3.1 was added in order to obtain Lemma 4.3.2.(5), which eventually leads to Lemma 4.3.7. The latter lemma is desirable because it provides many strict functionals.

Main theorem. The strict functionals are defined in order to have the following theorem, stating that if $L \ {}_{\mathrm{wm}}\!>\ R$, then for any context C in $\beta\overline{\eta}$-normal form, $C[L] \ {}_{\mathrm{st}}\!>\ C[R]$, provided the constants and variables are interpreted strictly. Consider the context $cM(\lambda f.f(\Box N))P$. Using the previous lemmata, we can make the following deduction. This can be seen as an illustration of the proof of the theorem.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{L \;{}_{\mathrm{wm}}\!> R \quad N \in \mathcal{WM}}{(LN) \;{}_{\mathrm{wm}}\!> (RN)}\,4.1.3.(3)}{f \in \mathcal{ST} \qquad (LN) \;{}_{\mathrm{st}}\!> (RN)}\,4.3.3.(1)}{(f(LN)) \;{}_{\mathrm{wm}}\!> (f(RN))}\,4.3.2.(3)}{(f(LN)) \;{}_{\mathrm{st}}\!> (f(RN)) \text{ (for arbitrary } f \in \mathcal{ST})}\,4.3.3.(1)}{\lambda f.(f(LN)) \;{}_{\mathrm{st}}\!> \lambda f.(f(RN))}\,\text{Def. of }{}_{\mathrm{st}}\!>}{cM(\lambda f.f(LN)) \;{}_{\mathrm{wm}}\!> cM(\lambda f.f(RN))}\,4.3.2.(3) \qquad P \in \mathcal{WM}}{cM(\lambda f.f(LN))P \;{}_{\mathrm{wm}}\!> cM(\lambda f.f(RN))P}\,4.1.3.(3)}{cM(\lambda f.f(LN))P \;{}_{\mathrm{st}}\!> cM(\lambda f.f(RN))P}\,4.3.3.(1)$$

$$\dfrac{c \in \mathcal{ST} \quad M \in \mathcal{WM}}{(cM) \in \mathcal{ST}}$$

We assume a fixed signature $\mathcal{F} = (\mathcal{B}, \mathcal{C}, \mathcal{V})$, with base type interpretation $\mathcal{I}$ and constant interpretation $\mathcal{J}$. $\mathcal{J}$ is called strict, if for each $c \in \mathcal{C}$, $\mathcal{J}(c) \in \mathcal{ST}$. Similarly, a valuation α is called strict, if for each $x \in \mathcal{V}$, $\alpha(x) \in \mathcal{ST}$.

Theorem 4.3.4 *Let $\mathcal{J}$ be a strict constant interpretation, and α a strict valuation. For any closed $M, N \in \Lambda^{\rightarrow}(\mathcal{F})$, and context $C[\,]$ in β-normal form we have, if $[\![M]\!]_{\mathrm{wm}}\!> [\![N]\!]$ then $[\![C[M]]\!]_{\alpha,\mathcal{J}\;{}_{\mathrm{st}}}\!> [\![C[N]]\!]_{\alpha,\mathcal{J}}$.*

Proof: The proof is by induction on $C[\,]$. We use that by Lemma 2.4.3 any β-normal form, so in particular $C[\,]$, is of the form $\lambda\vec{x}.y\vec{P}$, where $y \in \mathcal{C} \cup \mathcal{V} \cup \{\Box\}$. We distinguish cases, whether $y \equiv \Box$ or the $\Box$ is contained in on of the P_i. In the latter case, P_i is again a context in β-normal form, so the theorem already holds for it, by induction hypothesis. We now give the proof in detail.

Assume that $[\![M]\!]_{\mathrm{wm}}\!> [\![N]\!]$ and let α be a strict valuation, so in particular it is weakly monotonic. Let $\vec{a} \in \mathcal{ST}$ be given. Let β be the valuation $\alpha[\vec{x} := \vec{a}]$, which clearly is strict. In view of Lemma 4.3.3.(1), it is sufficient to prove that $[\![(y\vec{P})[M]]\!]_{\beta\;{}_{\mathrm{wm}}}\!> [\![(y\vec{P})[N]]\!]_{\beta}$. We distinguish

- Case $y \equiv \Box$. Then $(y\vec{P})[M] \equiv M\vec{P}$. Because $[\![M]\!]_{\mathrm{wm}}\!> [\![N]\!]$ and each $[\![P_i]\!]_{\beta} \in \mathcal{WM}$, we have $[\![M\vec{P}]\!]_{\beta\;{}_{\mathrm{wm}}}\!> [\![N\vec{P}]\!]_{\beta}$.
- Case $y \in \mathcal{V} \cup \mathcal{C}$. In this case, one of the P_i's contains the $\Box$, so by induction hypothesis, $[\![P_i[M]]\!]_{\beta\;{}_{\mathrm{st}}}\!> [\![P_i[N]]\!]_{\beta}$. If $y \in \vec{x}$, then its value is decided by $\vec{a}$; if y is a free variable in $C[\,]$, then its value is decided by α; if y is a constant, it gets its value from $\mathcal{J}$. In all these cases, $[\![y]\!]_{\beta} \in \mathcal{ST}$. Then also $[\![(yP_1 \ldots P_{i-1})]\!]_{\beta} \in \mathcal{ST}$, so $[\![(yP_1 \ldots P_{i-1}P_i[M])]\!]_{\beta\;{}_{\mathrm{wm}}}\!> [\![(yP_1 \ldots P_{i-1}P_i[N])]\!]_{\beta}$. Also the remaining $[\![P_j]\!]_{\beta}$ are weakly monotonic, so $[\![(y\vec{P})[M]]\!]_{\beta\;{}_{\mathrm{wm}}}\!> [\![(y\vec{P})[N]]\!]_{\beta}$. ⊠

In fact a slightly stronger theorem holds. For strict α and β and if $f \ {}_{\mathrm{wm}}{>}\ g$, then for any β-normal term C (playing the rôle of a context) with $x \in \mathrm{FV}(C)$, $[\![C]\!]_{\alpha[x:=f]} \ {}_{\mathrm{st}}{>}\ [\![C]\!]_{\alpha[x:=g]}$. This is stronger in two respects. First, f and g are arbitrary weakly monotonic, instead of restricted to denotations of closed lambda terms. In the second place, the x may appear more than once in M. The proof can be found in [PS95] and is similar to the one given here. We will only need the theorem in the form stated above.

4.3.2 The Existence of Strict Functionals

It is not immediately clear that strict functionals exist for all types. Below we will define functionals S_0^σ of type σ, that turn out to be strict. However, some conditions on the ordered type interpretation have to be made, namely that it admits enough strictly monotonic functions on type level 1. This is captured by the notion of an ordered domain (Section 4.2).

We define the identity on type σ, as follows:

$$\mathsf{I}_\sigma = \boldsymbol{\lambda} x^\sigma.x$$

One might think that I_σ is a strict function. However, this is not the case. Although I is hereditarily monotonic by Lemma 3.3.2.(2), it is not strict. This is seen as follows. $\mathsf{I}(\underline{0}) = \underline{0}$ is a constant function, so it cannot be strict. By Lemma 4.3.2.(2), strictness is preserved under application on weakly monotonic arguments, so I itself is not strict. The same applies to $\oplus$, because $\oplus\underline{0} = \mathsf{I}$ (in the standard model based on natural numbers).

4.3.2.1 The strict functionals S and M.

We now define for any type σ with $res(\sigma) = \iota$, functionals $\mathsf{M}_\sigma : \sigma \to \iota$ and $\mathsf{S}_\sigma : \iota \to \sigma$. These functionals are presented as a series of lambda terms, defined by simultaneous recursion on σ, based on constants 0_κ and $+_{\iota,\kappa}$. In the recursive clause, we assume w.l.o.g. that $res(\sigma) = \kappa$ and $res(\tau) = \iota$.

Definition 4.3.5 *Let $((\mathcal{I}_\iota, >_\iota, 0_\iota)_{\iota\in\mathcal{B}}, (+_{\iota,\kappa})_{\iota,\kappa\in\mathcal{B}})$ be an ordered domain. Then we define*

$$\begin{array}{lcl}
\mathsf{M}_\iota & = & \boldsymbol{\lambda} x^\iota.x \\
\mathsf{M}_{\sigma\to\tau} & = & \boldsymbol{\lambda} f^{\sigma\to\tau}.\mathsf{M}_\tau(f(\mathsf{S}_\sigma(0_\kappa))) \\
\mathsf{S}_\iota & = & \boldsymbol{\lambda} x^\iota.x \\
\mathsf{S}_{\sigma\to\tau} & = & \boldsymbol{\lambda} x^\iota.\boldsymbol{\lambda} f^\sigma.\mathsf{S}_\tau(x +_{\iota,\kappa} \mathsf{M}_\sigma(f))
\end{array}$$

Given σ with $res(\sigma) = \iota$, we write S_0^σ for the functional $\mathsf{S}_\sigma(0_\iota)$.

The intuition is that M_σ serves as a *measure* on functionals. A functional is mapped to some base type, by applying it to strict arguments. Its counterpart S is called so, because it is a canonical *strict* functional. It is minimal in a certain sense (see Proposition 4.4.4). It works by summing up the measures of its arguments. The

first argument of S plays the rôle of a storage cell, keeping the sum of the arguments already processed.

We proceed by showing that M and S are strict functionals. We then have strict functionals of any type, because S_0^σ is of type σ. Note that by Corollary 4.1.6, S_σ and M_σ are weakly monotonic already.

Proposition 4.3.6 *For any $\sigma \in \mathbb{T}^{\to}(\mathcal{B})$ with $res(\sigma) = \iota$, we have $\mathsf{M}_\sigma \in \mathcal{ST}_{\sigma\to\iota}$ and $\mathsf{S}_\sigma \in \mathcal{ST}_{\iota\to\sigma}$.*

Proof: (Simultaneous induction on σ). Base: This is trivial. Let $x >_\iota y$, then $\mathsf{M}_\iota(x) = \mathsf{S}_\iota(x) = x >_\iota y = \mathsf{S}_\iota(y) = \mathsf{M}_\iota(y)$.

Induction step: Assume that S_σ, S_τ, M_σ and M_τ are strict. Furthermore, let $res(\tau) = \iota$ and $res(\sigma) = \kappa$. To prove that $\mathsf{S}_{\sigma\to\tau} \in \mathcal{ST}$ we have to prove for arbitrary $x, y \in \mathcal{I}_\iota$ and $f, g \in \mathcal{WM}_\sigma$:

1. if $x >_\iota y$ then $\mathsf{S}(x,f) \;{}_{\mathrm{wm}}{>}_\tau\; \mathsf{S}(y,f)$;
2. if $f \;{}_{\mathrm{st}}{>}_\sigma\; g$ then $\mathsf{S}(x,f) \;{}_{\mathrm{wm}}{>}_\tau\; \mathsf{S}(x,g)$;
3. $\mathsf{S}(x,f) \in \mathcal{ST}_\tau$.

ad 1: Assume $x >_\iota y$ and $f \in \mathcal{WM}_\sigma$. By strictness of $+$, $x +_{\iota,\kappa} \mathsf{M}_\sigma(f) >_\iota y +_{\iota,\kappa} \mathsf{M}_\sigma(f)$. By strictness of S_τ, $\mathsf{S}_\tau(x +_{\iota,\kappa} (\mathsf{M}_\sigma f)) \;{}_{\mathrm{wm}}{>}\; \mathsf{S}_\tau(y +_{\iota,\kappa} (\mathsf{M}_\sigma f))$.

ad 2: Assume $x \in \mathcal{I}_\iota$ and $f \;{}_{\mathrm{st}}{>}\; g$. By strictness of M_σ, $\mathsf{M}_\sigma(f) >_\kappa \mathsf{M}_\sigma(g)$. By strictness of $+$ and S_τ, $\mathsf{S}_\tau(x +_{\iota,\kappa} \mathsf{M}_\sigma(f)) \;{}_{\mathrm{wm}}{>}_\tau\; \mathsf{S}_\tau(x +_{\iota,\kappa} \mathsf{M}_\sigma(g))$.

ad 3: Assume $x \in \mathcal{I}_\iota$ and $f \in \mathcal{WM}_\sigma$. Note that $x +_{\iota,\kappa} \mathsf{M}_\sigma(f)$ is of base type. By induction hypothesis, S_τ is strict. By Lemma 4.3.2.(2) $\mathsf{S}_\tau(x +_{\iota,\kappa} \mathsf{M}_\sigma(f)) \in \mathcal{ST}_\tau$.

To prove that $\mathsf{M}_{\sigma\to\tau} \in \mathcal{ST}$, we have to show, that for arbitrary $f, g \in \mathcal{WM}_{\sigma\to\tau}$,

1. If $f \;{}_{\mathrm{st}}{>}\; g$ then $\mathsf{M}_{\sigma\to\tau}(f) \;{}_{\mathrm{wm}}{>}\; \mathsf{M}_{\sigma\to\tau}(g)$.
2. $\mathsf{M}_{\sigma\to\tau}(f) \in \mathcal{ST}_\tau$

ad 1. Let $f \;{}_{\mathrm{st}}{>}_{\sigma\to\tau}\; g$. By induction hypothesis, S_σ is strict, so S_0^σ is strict too. Therefore $f(\mathsf{S}_0^\sigma) \;{}_{\mathrm{st}}{>}_\tau\; g(\mathsf{S}_0^\sigma)$. By strictness of M_τ we get $\mathsf{M}_\tau(f(\mathsf{S}_0^\sigma)) >_\iota \mathsf{M}_\tau(g(\mathsf{S}_0^\sigma))$.

ad 2. This is trivial, because $\mathsf{M}(f) \in \mathcal{I}_\iota = \mathcal{ST}_\iota$. ⊠

Having these strict functionals, it is easy to construct more strict functionals. The following lemma gives an easy method to prove strictness of a certain class of functionals.

Lemma 4.3.7 *Let $\sigma, \tau \in \mathbb{T}^{\to}(\mathcal{B})$ with the same factors be given. Let $f \in \mathcal{ST}_\sigma$ and $g \in \mathcal{WM}_\tau$, then $f \oplus_{\sigma,\tau} g \in \mathcal{ST}_\sigma$ and $g \oplus_{\tau,\sigma} f \in \mathcal{ST}_\tau$.*

Proof: We only prove that $f \oplus g \in \mathcal{ST}_\sigma$; the other proposition can be proved similarly. The proof is by induction on the types σ. The base case of the induction is trivial, because $f \oplus g \in \mathcal{I}_\iota = \mathcal{ST}_\iota$.

Now assume that the theorem holds for types σ and τ. Let $f \in \mathcal{ST}_{\rho\to\sigma}$ and $g \in \mathcal{WM}_{\rho\to\tau}$. Let $x, y \in \mathcal{WM}_\sigma$ be given. We have to prove:

1. If $x \ {}_{\mathrm{st}}{>}\ y$ then $(f \oplus g)(x) \ {}_{\mathrm{wm}}{>}\ (f \oplus g)(y)$.
2. $(f \oplus g)(x) \in \mathcal{ST}_\sigma$.

ad 1. Let $x \ {}_{\mathrm{st}}{>}\ y$. By Lemma 4.3.2.(3), $f(x) \ {}_{\mathrm{wm}}{>}\ f(y)$. By Lemma 4.3.2.(5), $x \ {}_{\mathrm{wm}}{\geq}\ y$ and by Lemma 4.3.2.(1) $f \in \mathcal{WM}$, so by Lemma 4.1.3.(4), $f(x) \ {}_{\mathrm{wm}}{\geq}\ g(x)$.

Using Lemma 4.2.4, we get

$$(f \oplus g)(x) = f(x) \oplus g(x) \ {}_{\mathrm{wm}}{>}\ f(y) \oplus g(x) \ {}_{\mathrm{wm}}{\geq}\ f(y) \oplus g(y) = (f \oplus g)(y) \ .$$

The required inequality now follows from Lemma 4.1.4.(4).

ad 2. We have $(f \oplus g)(x) = f(x) \oplus g(x)$. As $f \in \mathcal{ST}$ and $x \in \mathcal{WM}$ it follows that $f(x) \in \mathcal{ST}$. Moreover, as $g \in \mathcal{WM}$, we have $g(x) \in \mathcal{WM}$. By the induction hypothesis, $f(x) \oplus g(x) \in \mathcal{ST}$. ⊠

4.3.2.2 Computation rules for S and M.

In a concrete termination proof, several calculations involving M and S have to be made. Therefore, it is convenient to have some computation rules. These computation rules depend on the choice of the functions $+_{\iota,\kappa}$ and 0_κ. We will say that *0 is the identity element*, if for any $\iota, \kappa \in \mathcal{B}$, $+_{\iota,\kappa}$ has a left identity element 0_ι and a right identity element 0_κ; i.e. $0 + x = x$ and $y + 0 = y$.

The most surprising is the following, which states that in fact $\mathsf{S}_0^{\sigma\to\iota}$ and M coincide.

Lemma 4.3.8 *If 0 is the identity element, then for any $\sigma \in \mathbb{T}^{\to}(\mathcal{B})$, with $res(\sigma) = \iota$, we have $\mathsf{S}_0^{\sigma\to\iota} = \mathsf{M}_\sigma$*

Proof: This can be verified by a simple calculation. For all $f \in \mathcal{T}_\sigma$ we have:

$$\mathsf{S}(0_\iota, f) = \mathsf{S}_\iota(0_\iota +_{\iota,\iota} \mathsf{M}_\sigma(f)) = \mathsf{M}_\sigma(f) \ .$$

We used that S_ι is the identity function and 0_ι is the left identity element of $+$. ⊠

The reason that we defined two functionals that eventually turn out to be the same is manifold. In the first place, by defining S and M together, it becomes clearer how the recursion over the types runs. This can be seen by comparing the definition with that of Gandy's L-functionals (Section 3.3). In Section 4.4 it will turn out that the S-functionals are more efficient than the L-ones. Finally, the definition that we give can be extracted from a traditional SN-proof for the simply-typed lambda calculus in a very canonical way. This will be demonstrated in Chapter 6.

Lemma 4.3.9 *If 0 is the identity element, then for any $\sigma \in \mathbb{T}^{\to}(\mathcal{B})$ with $res(\sigma) = \iota$ and $x \in \mathcal{I}_\iota$, we have $\mathsf{M}_\sigma(\mathsf{S}_\sigma(x)) = x$.*

Proof: Induction on σ. For base type, $\mathsf{M}_\iota(\mathsf{S}_\iota(x)) = x$ by definition of M and S. Now let $res(\sigma) = \kappa$ and $res(\tau) = \iota$, and assume as induction hypothesis that $\mathsf{M}_\sigma(\mathsf{S}_\sigma(y)) = y$

and $\mathsf{M}_\tau(\mathsf{S}_\tau(x)) = x$ for all $y \in \mathcal{I}_\kappa$ and $x \in \mathcal{I}_\iota$. We can now compute for arbitrary $x \in \mathcal{I}_\iota$:

$$\begin{array}{lcll} \mathsf{M}_{\sigma\to\tau}(\mathsf{S}_{\sigma\to\tau}(x)) & = & \mathsf{M}_\tau(\mathsf{S}_{\sigma\to\tau}(x, \mathsf{S}_\sigma(0_\kappa))) & \\ & = & \mathsf{M}_\tau(\mathsf{S}_\tau(x +_{\iota,\kappa} \mathsf{M}_\sigma(\mathsf{S}_\sigma(0_\kappa)))) & \\ & = & x +_{\iota,\kappa} 0_\kappa & \text{induction hypotheses} \\ & = & x & \text{by identity 0.} \end{array}$$

$\boxtimes$

From the previous lemma and Definition 4.3.5, it follows that $\mathsf{S}_0^{\sigma\to\tau}(\mathsf{S}_0^\sigma) = \mathsf{S}_0^\tau$, provided 0 is the identity element. Finally, we have the following lemma.

Lemma 4.3.10 *If 0 is the identity element, then for any $\rho, \sigma, \tau \in \mathbb{T}^\to(\mathcal{B})$, we have $\mathsf{S}_0^{\sigma\to\tau} \circ \mathsf{S}_0^{\rho\to\sigma} = \mathsf{S}_0^{\rho\to\tau}$.*

Proof: Let $f \in \mathcal{T}_\rho$ be given. The following calculation suffices:

$$\begin{array}{lcll} (\mathsf{S}_0^{\sigma\to\tau} \circ \mathsf{S}_0^{\rho\to\sigma})(f) & = & \mathsf{S}_0^{\sigma\to\tau}(\mathsf{S}_\sigma(\mathsf{M}_\rho(f))) & \text{using that 0 is the identity} \\ & = & \mathsf{S}_\tau(\mathsf{M}_\sigma(\mathsf{S}_\sigma(\mathsf{M}_\rho(f)))) & \text{using that 0 is the identity} \\ & = & \mathsf{S}_\tau(\mathsf{M}_\rho(f)) & \text{by Lemma 4.3.9} \\ & = & \mathsf{S}_0^{\rho\to\tau}(f) & \text{using that 0 is the identity} \end{array}$$

$\boxtimes$

4.4 Functionals over the Natural Numbers

In many cases, there is only one base type, interpreted by the natural numbers, with the usual order $>$. It is therefore quite helpful to study this particular case in some detail. The S and M functions are parametrized by $+_{o,o}$ and 0_o, for which we choose the usual addition and 0 of the natural numbers. Because 0 is the identity element of $+$, Lemmata 4.3.8, 4.3.9 and 4.3.10 hold in this case.

We first give some examples of the strict functionals S_0:

$$\begin{array}{rcl} \mathsf{S}_0^o & = & 0 \\ \mathsf{S}_0^{o\to o}(x) & = & x \\ \mathsf{S}_0^{(o\to o)\to o}(f) & = & f(0) \\ \mathsf{S}_0^{((o\to o)\to o)\to o}(F) & = & F(\boldsymbol{\lambda} f.f(0)) \\ \mathsf{S}_0^{(o\to o)\to o\to o}(f, x) & = & f(0) + x \end{array}$$

Sometimes the following equality is handy:

Lemma 4.4.1 *For any $\sigma \in \mathbb{T}^\to(\{o\})$, $\mathsf{S}_\sigma(n) = \underline{n}_\sigma \oplus \mathsf{S}_0^\sigma$.*

Proof: Induction on σ. If $\sigma = o$, then $\mathsf{S}_\sigma(n) = n = n + 0 = \underline{n}_o \oplus \mathsf{S}_0^o$. Assume that the equality holds for type τ. Then we have for any $x \in \mathcal{T}_\rho$,

$$\begin{array}{rcl} \mathsf{S}_{\rho\to\tau}(n)(x) & = & \mathsf{S}_\tau(n + \mathsf{M}(x)) \\ & = & \underline{n + \mathsf{M}(x)}_\tau \oplus \mathsf{S}_0^\tau \text{ by induction hypothesis} \\ & = & \underline{n}_\tau \oplus (\underline{\mathsf{M}(x)}_\tau \oplus \mathsf{S}_0^\tau) \text{ using Lemma 4.2.3.(2),(4)} \\ & = & \underline{n} \oplus \mathsf{S}_\tau(\underline{\mathsf{M}(x)}) \text{ by induction hypothesis} \\ & = & \underline{n} \oplus \mathsf{S}_0^{\rho\to\tau}(x) \end{array}$$

⊠

In general, given $\sigma = \vec{\sigma}_n \to o$,

$$\mathsf{S}_0^\sigma(\vec{x}_n) = \sum_{1\le i\le n} \mathsf{M}_{\sigma_i\to o}(x_i) \ .$$

As Gandy remarks, his L-functionals have as characteristic equation

$$\mathsf{L}_\sigma(\vec{x}_n) = \sum_{1\le i\le n} p_i \cdot \mathsf{L}_{\sigma_i\to o}(x_i) \ ,$$

where $p_1 = 1$, and for $2 \le i \le n$, $p_i = 2^{i-2}$.

So it is clear that the S_0 functionals are smaller than the L functionals. In fact, when we replace L by S_0 in the proof of Theorem 3.3.5, we obtain a sharper upper bound for the longest β-reduction sequences.

One might wonder, whether there are even smaller strict functionals. However, as the proposition below expresses, S_0 is the smallest strict functional in a certain sense. We will need an auxiliary lemma, and a generalized predecessor functional.

Lemma 4.4.2 *For all $n \in \mathbb{N}$, for all types τ and $x \in \mathcal{WM}_\tau$, if $x \,{}_{\mathrm{wm}}{>}_\tau \mathsf{S}_\tau(n)$ then $x \,{}_{\mathrm{wm}}{\ge} \mathsf{S}_\tau(n+1)$.*

Proof: Induction on τ. On type o, the lemma reduces to the true implication: If $x > n$ then $x \ge n + 1$.

Assume that the lemma holds already for type τ. Let $f \in \mathcal{WM}_{\sigma\to\tau}$. Assume $f \,{}_{\mathrm{wm}}{>}\, \mathsf{S}(n)$. Let $x \in \mathcal{WM}_\sigma$, then $f(x) \,{}_{\mathrm{wm}}{>}\, \mathsf{S}(n,x) = \mathsf{S}_\tau(n + \mathsf{M}(x))$. Using the induction hypothesis,

$$f(x) \,{}_{\mathrm{wm}}{\ge} \mathsf{S}_\tau((n + \mathsf{M}(x)) + 1) = \mathsf{S}_\tau((n+1) + \mathsf{M}(x)) = \mathsf{S}_{\sigma\to\tau}(n+1, x) \ .$$

So for any $x \in \mathcal{WM}_\sigma$, $f(x) \,{}_{\mathrm{wm}}{\ge} \mathsf{S}(n+1)(x)$, hence $f \,{}_{\mathrm{wm}}{\ge} \mathsf{S}(n+1)$. ⊠

The generalized predecessor function is defined as follows.

$$\begin{array}{rcl} \mathit{Pred}_o(x) & = & \begin{cases} x & \text{if } x = 0 \\ x - 1 & \text{otherwise} \end{cases} \\ \mathit{Pred}_{\sigma\to\tau}(f) & = & \boldsymbol{\lambda} x^\sigma . \mathit{Pred}_\tau(fx) \end{array}$$

Lemma 4.4.3 *For all types σ and $x \in \mathcal{WM}_\sigma$, we have*

1. $\mathsf{M}_\sigma(\mathit{Pred}_\sigma(x)) = \mathit{Pred}_o(\mathsf{M}_\sigma(x))$.
2. $x \,{}_{\mathrm{wm}}{\geq_\sigma}\, \mathit{Pred}_\sigma(x)$.
3. $\mathit{Pred}_\sigma \in \mathcal{WM}_{\sigma\to\sigma}$.

Proof:

1. Induction on σ. For $\sigma = o$, note that M_o is the identity. Assume that the statement holds for τ. Then we compute for $\sigma \to \tau$:

$$\begin{array}{lcl} \mathsf{M}_{\sigma\to\tau}(\mathit{Pred}_{\sigma\to\tau}(x)) & = & \mathsf{M}_\tau(\mathit{Pred}_{\sigma\to\tau}(x)(\mathsf{S}_0^\sigma)) \\ & = & \mathsf{M}_\tau(\mathit{Pred}_\tau(x(\mathsf{S}_0^\sigma))) \\ & = & \mathit{Pred}_o(\mathsf{M}_\tau(x(\mathsf{S}_0^\sigma))) \text{ by induction hypothesis} \\ & = & \mathit{Pred}_o(\mathsf{M}_{\sigma\to\tau}(x)) \end{array}$$

2. Straightforward induction on σ (analogous to the preservation statement).
3. Pred_o is non-decreasing, hence weakly monotonic. Pred_σ is a lambda expression in Pred_o, so by Corollary 4.1.6 it is weakly monotonic too.

⊠

Proposition 4.4.4 *For any $\sigma \in \mathbb{T}^{\to}(\{o\})$ and $f \in \mathcal{ST}_\sigma$, we have $f \,{}_{\mathrm{wm}}{\geq}\, \mathsf{S}_0^\sigma$.*

Proof: We prove by simultaneous induction on σ the following two propositions.

1. If $f \in \mathcal{ST}_\sigma$ then $f \,{}_{\mathrm{wm}}{\geq}\, \mathsf{S}_0^\sigma$.
2. If $f \in \mathcal{WM}_\sigma$ and $\mathsf{M}_\sigma(f) \geq 1$ then $f \,{}_{\mathrm{st}}{>}\, \mathit{Pred}(f)$.

For $\sigma = o$, we have for any $f \in \mathbb{N}$, $f \geq \mathsf{S}_0 = 0$, proving (1). For (2), note that $\mathsf{M}_o(f) = f \geq 1$ by assumption, so $\mathit{Pred}(f) = f - 1$.

Now, let $f \in \mathcal{WM}_{\sigma\to\tau}$ and assume that both properties hold on types σ and τ.

- For (1) let $f \in \mathcal{ST}_{\sigma\to\tau}$. We have to prove that for all $x \in \mathcal{WM}_\sigma$, $f(x) \,{}_{\mathrm{wm}}{\geq}\, \mathsf{S}_0(x)$. Note that $\mathsf{S}_0(x) = \mathsf{S}_\tau(\mathsf{M}_\sigma(x))$. The inequality is proved with induction on $\mathsf{M}(x)$, which is a natural number.

  If $\mathsf{M}(x) = 0$, then by the *main* induction hypothesis (1), $f(x) \,{}_{\mathrm{wm}}{\geq}\, \mathsf{S}_0^\tau = \mathsf{S}_0(x)$. This hypothesis is applicable because by Lemma 4.3.2.(2), $f(x) \in \mathcal{ST}$.

  If $\mathsf{M}(x) = n + 1$, then we can assume that for all y with $\mathsf{M}(y) = n$, $f(y) \,{}_{\mathrm{wm}}{\geq}\, \mathsf{S}_\tau(\mathsf{M}(y))$. By Lemma 4.4.3.(1), $\mathsf{M}(\mathit{Pred}(x)) = n$, so by the *inner* induction hypothesis, $f(\mathit{Pred}(x)) \,{}_{\mathrm{wm}}{\geq}\, \mathsf{S}_\tau(n)$. By the *main* induction hypothesis (2), $x \,{}_{\mathrm{st}}{>}\, \mathit{Pred}(x)$, so by strictness of f, $f(x) \,{}_{\mathrm{wm}}{>}\, f(\mathit{Pred}(x))$. By Lemma 4.1.4.(4), $f(x) \,{}_{\mathrm{wm}}{>}\, \mathsf{S}_\tau(n)$, hence by Lemma 4.4.2, $f(x) \,{}_{\mathrm{wm}}{\geq}\, \mathsf{S}_\tau(n+1) = \mathsf{S}_0(x)$.

- For (2), assume $\mathsf{M}_{\sigma\to\tau}(f) \geq 1$. Let $x \in \mathcal{ST}_\sigma$, then by induction hypothesis (1), $x \,{}_{\mathrm{wm}}{\geq_\sigma}\, \mathsf{S}_0$, so using weak monotonicity of f and M, $\mathsf{M}_\tau(f(x)) \geq \mathsf{M}_\tau(f(S_0^\sigma)) = \mathsf{M}_{\sigma\to\tau}(f) \geq 1$. By induction hypothesis (2), $f(x) \,{}_{\mathrm{st}}{>}\, \mathit{Pred}(f(x)) = \mathit{Pred}(f, x)$. This holds for any strict x and $f \,{}_{\mathrm{wm}}{\geq}\, \mathit{Pred}(f)$ by Lemma 4.4.3.(2), so $f \,{}_{\mathrm{st}}{>}\, \mathit{Pred}(f)$. ⊠

4.5 Extension to Product Types

The previous sections were based on simply-typed lambda calculus, with $\rightarrow$ as the only type constructor. In Section 5.5, we also need product types. The goal of the current section, is to add product types to $\lambda^{\rightarrow}_{\beta}$. The resulting system will be $\lambda^{\times}_{\beta}$. The theory developed so far, will be extended in order to be applicable to HRSs with $\lambda^{\times}_{\beta}$ as substitution calculus.

In [PS95] product types were incorporated. However, we have changed some definitions. The main difference is that in that paper, the definition of HRSs is taken modulo the $\beta\pi$-calculus, i.e. also projections are performed implicitly. We have now decided to add only product *types* without changing the terms. The constants for pair formation and projections and the corresponding rewrite rules can be expressed by ordinary HRS-rules (cf. 5.4). This decision makes the theory slightly more elegant. Especially, the computation rules from Section 4.3.2.2 still hold in the resulting theory.

4.5.1 HRSs based on $\lambda^{\times}_{\beta}$

We simply add a new binary type forming operator, $\times$, representing the cartesian product. Given a set of base types $\mathcal{B}$, the set of product types, $\mathbb{T}^{\times}(\mathcal{B})$ is defined inductively as the least set satisfying

- $\mathcal{B} \subseteq \mathbb{T}^{\times}(\mathcal{B})$
- If $\sigma, \tau \in \mathbb{T}^{\times}(\mathcal{B})$ then also $\sigma \rightarrow \tau \in \mathbb{T}^{\times}(\mathcal{B})$.
- If $\sigma, \tau \in \mathbb{T}^{\times}(\mathcal{B})$, then $\sigma \times \tau \in \mathbb{T}^{\times}(\mathcal{B})$.

By convention, $\times$ binds stronger than $\rightarrow$, so e.g. $o \times o \rightarrow o$ denotes the type $(o \times o) \rightarrow o$.

The notions of *arity* and *factors* will not be generalized to product types. The *result* of a type, the *type level* and the congruence on types $\approx$ will be generalized by adding the following clauses to the corresponding definitions in Section 2.4.1:

- $res(\sigma \times \tau) = res(\sigma)$.
- $\mathrm{TL}(\sigma \times \tau) = \max(\mathrm{TL}(\sigma), \mathrm{TL}(\tau))$.
- $\sigma \times \tau \approx \rho \times \nu$ if and only if $\sigma \approx \rho$ and $\tau \approx \nu$.

The choice for $res(\sigma \times \tau)$ is not canonical, but choosing for $res(\sigma)$ turns out to be more convenient in the definition of $\mathsf{M}_{\sigma \times \tau}$.

We will *not* add new term forming constructors for pairing and projections. This is not necessary, because we parametrized lambda calculus with a signature. Pairing and projection symbols can be added as constants. The reduction relation for pairing and projection can be defined by an HRS. This will be illustrated in Section 5.4.

The notions of higher-order signature, terms and substitutions and the various reduction relations can be generalized by replacing $\mathbb{T}^{\rightarrow}$ by $\mathbb{T}^{\times}$ everywhere. We write $\Lambda^{\times}(\mathcal{F})$ for terms involving product types. The resulting ARS is denoted by $\lambda^{\times}_{\beta}$ and is of the form $(\Lambda^{\times}, \rightarrow_{\beta})$.

In the sequel of this chapter we consider HRSs that are based on $\lambda^{\times}_{\beta}$ as substitution calculus.

4.5.2 Weakly Monotonic and Strict Pairs

We will shortly describe how the considerations on weakly monotonic and strict functionals (Section 4.1–4.4) extends to the product types. We extend the definition of the various notions in such a way, that the lemmata and theorems of the previous chapters (apart from Proposition 4.4.4) still hold in the extended theory.

4.5.2.1 Weakly monotonic and strict functionals

First, the interpretation of a product type is recursively defined as follows. Here for each base type $\iota \in \mathcal{B}$, $\mathcal{I}_\iota$ is a fixed interpretation for it.

$$\begin{array}{rcl} \mathcal{T}_\iota & = & \mathcal{I}_\iota \\ \mathcal{T}_{\sigma\to\tau} & = & \mathcal{T}_\sigma \Rightarrow \mathcal{T}_\tau. \\ \mathcal{T}_{\sigma\times\tau} & = & \mathcal{T}_\sigma \times \mathcal{T}_\tau \end{array}$$

In the following definitions, we will not repeat the clauses for the base case and function case. We just list the clauses for the product case. This means that this clause has to be added to the existing recursive definition, and that the types σ, τ in the function clause range over product types.
Extend Definition 4.1.1 with

- $\mathcal{WM}_{\sigma\times\tau} := \mathcal{WM}_\sigma \times \mathcal{WM}_\tau$.
- $(u,v)\ {}_{\mathrm{wm}}{\geq}_{\sigma\times\tau}\ (x,y) :\Longleftrightarrow\ u\ {}_{\mathrm{wm}}{\geq}_\sigma\ x \wedge v\ {}_{\mathrm{wm}}{\geq}_\tau\ y$.

Extend Definition 4.1.2 with

- $(u,v)\ {}_{\mathrm{wm}}{>}_{\sigma\times\tau}\ (x,y) :\Longleftrightarrow\ (u,v)\ {}_{\mathrm{wm}}{\geq}_{\sigma\times\tau}\ (x,y) \wedge (u\ {}_{\mathrm{wm}}{>}_\sigma\ x \vee v\ {}_{\mathrm{wm}}{>}_\tau\ y)$.

Extend Definition 4.3.1 with

- $\mathcal{ST}_{\sigma\times\tau} := \mathcal{ST}_\sigma \times \mathcal{ST}_\tau$.
- $(u,v)\ {}_{\mathrm{st}}{>}_{\sigma\times\tau}\ (x,y) :\Longleftrightarrow\ (u,v)\ {}_{\mathrm{wm}}{\geq}_{\sigma\times\tau}\ (x,y) \wedge (u\ {}_{\mathrm{st}}{>}_\sigma\ x \vee v\ {}_{\mathrm{st}}{>}_\tau\ y)$.

The products are ordered in such a way, that a pair decreases if one component decreases and the other does not increase. The latter condition is needed to preserve well-foundedness.

Lemma 4.1.3, Lemma 4.1.4, Lemma 4.3.2 and Lemma 4.3.3 are also valid when we let σ, τ range over $\mathbb{T}^{\times}(\mathcal{B})$. The proofs can be easily extended. Also Proposition 4.1.5 and Theorem 4.3.4 remain valid in the presence of product types. Their proofs do not require any change.

4.5.2.2 M- and S-functionals

We now extend Definition 4.3.5. Recall that if $res(\sigma) = \iota$, then $\mathsf{S}_\sigma : \iota \to \sigma$ and $\mathsf{M}_\sigma : \sigma \to \iota$. In the clauses below, we assume that $res(\sigma) = \iota$ and $res(\tau) = \kappa$. We put

- $\mathsf{M}_{\sigma\times\tau}(x,y) = \mathsf{M}_\sigma(x) +_{\iota,\kappa} \mathsf{M}_\tau(y)$
- $\mathsf{S}_{\sigma\times\tau}(x^\iota) = (\mathsf{S}_\sigma(x), \mathsf{S}_\tau(0_\kappa))$.

Recall that (x,y) denotes the "mathematical" ordered pair of x and y. Some examples may illustrate (o is a base type):

$$\begin{array}{rcl}
\mathsf{S}_{o\times o}(a) & = & (a,0) \\
\mathsf{M}_{o\times o}(x,y) & = & x+y \\
\mathsf{S}_{o\times o\to o}(a)(x,y) & = & a+(x+y) \\
\mathsf{S}_{o\times o\to o\times o}(a)(x,y) & = & (a+(x+y),0) \\
\mathsf{M}_{o\times o\to o\times o}(f) & = & \pi_1(f(0,0)) + \pi_2(f(0,0))
\end{array}$$

We first extend the proof of Proposition 4.3.6, stating that M and S are strict.

Proposition 4.5.1 *For any $\sigma \in \mathbb{T}^\times(\mathcal{B})$, M_σ and S_σ are strict.*

Proof: The proof extends the inductive proof of Proposition 4.3.6 with the product cases. Let $res(\sigma) = \iota$ and $res(\tau) = \kappa$. Assume that S_σ, S_τ, M_σ, M_τ are all strict.

- For $\mathsf{S}_{\sigma\times\tau}$ we have to prove
  1. If $x >_\iota y$ then $\mathsf{S}_{\sigma\times\tau}(x) \,{}_{\mathrm{wm}}{>}\, \mathsf{S}_{\sigma\times\tau}(y)$. Let $x >_\iota y$. Then by induction hypothesis, $\mathsf{S}_\sigma(x) \,{}_{\mathrm{wm}}{>}\, \mathsf{S}_\sigma(y)$. This implies, by the definition of ${}_{\mathrm{wm}}{>}$ on pairs $(\mathsf{S}_\sigma(x), \mathsf{S}_\tau(0_\kappa)) \,{}_{\mathrm{wm}}{>}\, (\mathsf{S}_\sigma(y), \mathsf{S}_\tau(0_\kappa))$.
  2. For any x, $\mathsf{S}_{\sigma\times\tau}(x)$ is strict. By induction hypothesis for S_σ and S_τ and Lemma 4.3.2.(2), $\mathsf{S}_\sigma(x)$ and $\mathsf{S}_\tau(0)$ are strict; by definition of $\mathcal{ST}$ on pairs, the pair $(\mathsf{S}_\sigma(x), \mathsf{S}_\tau(0))$ is also strict.
- For $\mathsf{M}_{\sigma\times\tau}$ we have to prove: If $(u,v) \,{}_{\mathrm{st}}{>}_{\sigma\times\tau}\, (x,y)$, then $\mathsf{M}(u,v) >_\iota \mathsf{M}(x,y)$. Let $(u,v) \,{}_{\mathrm{st}}{>}\, (x,y)$, then either $u \,{}_{\mathrm{st}}{>}\, x$ and $v \,{}_{\mathrm{wm}}{\geq}\, y$, or $v \,{}_{\mathrm{st}}{>}\, y$ and $u \,{}_{\mathrm{wm}}{\geq}\, x$. We now assume that the first case applies, the other case goes similarly. By induction hypothesis, $\mathsf{M}_\sigma(u) >_\iota \mathsf{M}_\sigma(x)$ and $\mathsf{M}_\tau(v) \geq_\kappa \mathsf{M}_\tau(y)$. Then (by strictness of $+_{\iota,\kappa}$) $\mathsf{M}_\sigma(u) +_{\iota,\kappa} \mathsf{M}_\tau(v) >_\iota \mathsf{M}_\sigma(x) +_{\iota,\kappa} \mathsf{M}_\tau(v) \geq_\iota \mathsf{M}_\sigma(x) +_{\iota,\kappa} \mathsf{M}_\tau(y)$. Using transitivity of $>_\iota$ yields that $\mathsf{M}_{\sigma\times\tau}(u,v) >_\iota \mathsf{M}_{\sigma\times\tau}(x,y)$.

⊠

Put

$(x,y) \oplus_{\sigma\times\tau,\rho\times\nu} (u,v) := (x \oplus_{\sigma,\rho} u, y \oplus_{\tau,\nu} v)$.

$\underline{n}_{\sigma\times\tau} := (\underline{n}_\sigma, \underline{n}_\tau)$.

Then for natural numbers, Lemma 4.2.3 and 4.2.5 remain true for the product case. For arbitrary ordered domain, we keep Lemma 4.2.2, 4.2.4. Also Lemma 4.3.7, which generates many strict functionals, holds for all types $\sigma, \tau \in \mathbb{T}^\times(\mathcal{B})$. The inductive proofs of all these lemmata can easily be extended.

4.5.2.3 Computation rules for S and M

Also the computation rules for M and S remain to hold, under assumption that 0 is the identity, i.e. $0_\iota +_{\iota,\kappa} x = x$ and $y +_{\iota,\kappa} 0_\kappa = y$ for all $\iota, \kappa \in \mathcal{B}$, $x \in \mathcal{I}_\kappa$ and $y \in \mathcal{I}_\iota$. Lemma 4.3.8 is independent of the rules for the product type. Also Lemma 4.3.9 remains valid:

Lemma 4.5.2 *If 0 is the identity, then for any $\sigma \in \mathbb{T}^\times(\mathcal{B})$ with $res(\sigma) = \iota$ and $x \in \mathcal{I}_\iota$, we have $\mathsf{M}_\sigma(\mathsf{S}_\sigma(x)) = x$.*

Proof: We extend the inductive proof of Lemma 4.3.9 with the product case. Assume that $res(\sigma) = \iota$ and $res(\tau) = \kappa$. By induction hypothesis, $\mathsf{M}_\sigma(\mathsf{S}_\sigma(x)) = x$ and $\mathsf{M}_\tau(\mathsf{S}_\tau(y)) = y$, for any $x \in \mathcal{I}_\iota$ and $y \in \mathcal{I}_\kappa$. Then we can compute:

$$\begin{array}{lcll} \mathsf{M}_{\sigma\times\tau}(\mathsf{S}_{\sigma\times\tau}(x)) & = & \mathsf{M}_{\sigma\times\tau}(\mathsf{S}_\sigma(x), \mathsf{S}_\tau(0)) & \\ & = & \mathsf{M}_\sigma(\mathsf{S}_\sigma(x)) + \mathsf{M}_\tau(\mathsf{S}_\tau(0)) & \\ & = & x + 0 & \text{by induction hypothesis} \\ & = & x & \text{because 0 is the identity} \end{array}$$

⊠

Also Lemma 4.3.10 remains true. In the proof the application of Lemma 4.3.9 can be replaced by an application of Lemma 4.5.2.

We also have a new computation rule. This depends on an extra assumption of the $+_{\iota,\kappa}$ functionals. We say that $+$ is associative, if for any combination $\iota, \kappa, \lambda \in \mathcal{B}$ and $x \in \mathcal{I}_\iota$, $y \in \mathcal{I}_\kappa$ and $z \in \mathcal{I}_\lambda$ the following holds: $(x +_{\iota,\kappa} y) +_{\iota,\lambda} z = x +_{\iota,\lambda} (y +_{\kappa,\lambda} z)$.

Lemma 4.5.3 *If $+$ is associative, then for any $\rho, \sigma, \tau \in \mathbb{T}^\times(\mathcal{B})$, with $res(\rho) = \iota$, $x \in \mathcal{I}_\iota$, $f \in \mathcal{T}_\sigma$ and $g \in \mathcal{T}_\tau$, we have $\mathsf{S}_{\sigma\times\tau\to\rho}(x)(f,g) = \mathsf{S}_{\sigma\to\tau\to\rho}(x)(f)(g)$.*

Proof: The proof is by mere calculation:

$$\begin{array}{lcll} \mathsf{S}_{\sigma\times\tau\to\rho}(x^\iota)(f,g) & = & \mathsf{S}_\rho(x + \mathsf{M}_{\sigma\times\tau}(f,g)) & \\ & = & \mathsf{S}_\rho(x + (\mathsf{M}_\sigma(f) + \mathsf{M}_\tau(g))) & \\ & = & \mathsf{S}_\rho((x + \mathsf{M}_\sigma(f)) + \mathsf{M}_\tau(g)) & \text{by associativity of } + \\ & = & \mathsf{S}_{\tau\to\rho}(x + \mathsf{M}_\sigma(f))(g) & \\ & = & \mathsf{S}_{\sigma\to\tau\to\rho}(x)(f)(g) & \end{array}$$

⊠

Proposition 4.4.4 does *not* remain valid in the presence of product types. This is easy to see. We have that $\mathsf{S}_0^{o\to(o\times o)}(x) = (x, 0)$. Using symmetry considerations, we have that also f defined by $f(x) = (0, x)$ is a strict functional. However, the two are not related. So we loose the property that S_0^σ is the smallest strict functional for any σ.

Chapter 5

Termination of Higher-order Rewrite Systems

In this chapter we take the profit of the theory developed in Chapter 4. The theory on weakly monotonic and strict functionals quite easily provides for a method to prove termination of HRSs, in the same way as monotone algebras give a method to prove termination of TRSs. We present this method in Section 5.1.1. Section 5.1.2 is devoted to some straightforward applications of the proof method. This part is mainly based on [Pol94].

We then investigate how the scope of our method can be extended. To this end, we internalize the simply-typed lambda calculus (Section 5.2). With this we mean that its terms and the β-rule are encoded in an HRS. The encoding will be performed in such a way, that the thus obtained $\mathcal{H}_{\text{lam}}$ can be easily combined with other systems. After the encoding, the combination of an arbitrary first-order TRS with lambda calculus can be described; we can even add higher-order rules of a restricted format. As an easy consequence of our termination method, we derive modularity properties for these combinations (Section 5.2.3). Especially, we prove that the combination of a TRS with β-reduction terminates if and only if the TRS terminates. This result is not new, but we have a completely different proof.

We proceed with some larger examples (Sections 5.3–5.5). Using the method of Section 5.1.1, we prove termination of Gödel's T, simply-typed lambda calculus with surjective pairing and strong normalization for natural deduction trees in first-order logic. In the latter case, the rules are formed by the proper reduction rules, that remove detours, and the permutative conversions. These examples are based on [PS95].

Finally, we give an example of an HRS that cannot be proved terminating by the method. We will also sketch a possible modification of the method, to tackle at least this kind of examples. It is not clear whether the modified method is stronger than the original method, let alone if it is a complete method. We also mention possible extensions to other type disciplines.

5.1 Higher-order Monotone Algebras for Termination Proofs

We give the definition of a higher-order monotone algebra, and present a method to prove termination of HRSs in Section 5.1.1. Section 5.1.2 is devoted to some straightforward applications of the proof method. We give three examples: finding the prenex normal form, a specification of the disjoint union of two sorts and an example from process algebra with an infinite choice operator over data elements. We start the application section with a specialization of the method to second-order HRSs.

5.1.1 A Method for Proving Termination

Recall that an ordered domain is a collection of partial orders $(\mathcal{I}_\iota, >_\iota)$ for each base type ι, together with inhabitants $0_\iota \in \mathcal{I}_\iota$ and for each two base types ι and κ, a strictly monotonic function of type $\mathcal{I}_\iota \Rightarrow \mathcal{I}_\kappa \Rightarrow \mathcal{I}_\iota$. A monotone higher-order algebra is obtained by extending an ordered domain with strict functionals for all constants.

Definition 5.1.1 *Given a higher-order signature $\mathcal{F} = (\mathcal{B}, \mathcal{C}, \mathcal{V})$, a* monotone higher-order algebra *is a structure $(\mathcal{D}, (\mathcal{J}_c)_{c\in\mathcal{C}})$, such that*

1. *$\mathcal{D}$ is an ordered domain, and*
2. *For each $\sigma \in \mathbb{T}^{\rightarrow}(\mathcal{B})$ and $c \in \mathcal{C}_\sigma$, $\mathcal{J}(c) \in \mathcal{ST}_\sigma$.*

A monotone higher-order algebra is well-founded *if the underlying ordered domain is.*

Definition 5.1.2 *A rule $L \mapsto R$ is* decreasing *in the monotone higher-order algebra, if and only if $[\![L]\!] \ _{\mathrm{wm}}\!> [\![R]\!]$.*

Note that rules are closed, so their interpretation is regardless of valuations. But in practical examples, we write open terms, with capitals for the free variables. In that case, a rule is decreasing, if for all weakly monotonic valuations α, $[\![L]\!]_\alpha \ _{\mathrm{wm}}\!> [\![R]\!]_\alpha$. This holds if and only if the closure of the rule is decreasing.

Definition 5.1.3 *A* termination model *for a higher-order term rewriting system $(\mathcal{F}, \mathcal{R})$, is a well-founded monotone higher-order algebra, in which each rule of $\mathcal{R}$ is decreasing.*

Theorem 5.1.4 *If an HRS $(\mathcal{F}, \mathcal{R})$ has a termination model, then it is terminating.*

Proof: Let $(\mathcal{F}, \mathcal{R})$ have a termination model. We first consider one rewrite step: let $M \to_{\mathcal{R}} N$ be a rewrite step of type τ, with $res(\tau) = \iota$. Then, by the definition of a rewrite step, there exists a context $C[\,]$ in β-normal form and a rule $L \mapsto R \in \mathcal{R}$, such that $C[L] \twoheadrightarrow_\beta M$ and $C[R] \twoheadrightarrow_\beta N$. Because the monotone algebra is based upon an ordered domain, the functionals S_0^σ can be constructed as in Definition 4.3.5; they are strict by Proposition 4.3.6. Let α_{S} be the strict valuation that maps each x^σ to S_0^σ.

The second condition of a monotone higher-order algebra ensures that the constants are interpreted in a strict way. The rules are decreasing, so $[\![L]\!]_{\text{wm}} > [\![R]\!]$. Hence we can apply Theorem 4.3.4 and we have:

$$[\![M]\!]_{\alpha_S} = [\![C[L]]\!]_{\alpha_S} \,_{\text{st}}{>}_\tau\, [\![C[R]]\!]_{\alpha_S} = [\![N]\!]_{\alpha_S} \ .$$

By Proposition 4.3.6, $\mathsf{M}_\tau \in \mathcal{ST}_{\tau\to\iota}$, hence by Lemma 4.3.2.(3), $\mathsf{M}_\tau([\![M]\!]_{\alpha_S}) >_\iota \mathsf{M}_\tau([\![N]\!]_{\alpha_S})$.

So any rewrite sequence corresponds to a decreasing sequence in $(\mathcal{I}_\iota, >_\iota)$. This sequence is finite by well-foundedness of the algebra. Hence the HRS is terminating. ⊠

The theorem is not specific about the underlying substitution calculus. It simply holds for all three cases that we encountered, so it can be used in the setting of HRSs based on $\lambda^{\rightarrow}_{\beta\overline{\eta}}$, $\lambda^{\rightarrow}_{\beta}$ or $\lambda^{\times}_{\beta}$.

Note that, contrary to Theorem 3.1.3, we have Theorem 5.1.4 only in one direction. The converse implication does not hold, as will be shown in Section 5.6. So our method is incomplete.

Helpful hints. Theorem 5.1.4 supports the following method to prove termination of an HRS:

1. For the base types ι, choose well-suited partial orders $(\mathcal{I}_\iota, >_\iota)$ that are non-empty and well-founded and have strictly monotonic functions $+_{\iota,\kappa} \in \mathcal{I}_\iota \Rightarrow \mathcal{I}_\kappa \Rightarrow \mathcal{I}_\iota$.
2. Find an appropriate strict interpretation for the constant symbols.
3. Show for any rule $L \mapsto R$ that $[\![L]\!]_{\text{wm}} > [\![R]\!]$.

After step (1) we have a well-founded ordered domain, in step (2) we obtain a higher-order monotone algebra; in step (3) we know that the algebra is a termination model.

In most applications step (1) is quite easy, step (2) requires some intuition (sometimes ingenuity) and step (3) is rather straightforward. However, in some cases it is not obvious how to choose (1), and given (1) and (2), testing (3) is not decidable in general.

For (1) we will often choose the ordered domain $(\mathbb{N}, >, 0, +)$. It is well-known that even for first-order term rewriting this is not enough. There exist terminating TRSs for which a non-total order is necessary. See [Zan94, Fer95] for a hierarchy of underlying partial orders.

There is no general recipe how to find a suitable strict interpretation (2). There are two starting points. The first starting point is the standard interpretation of the rewrite system as an equational theory, which yields a model. A modification of this model can yield a termination model. The second starting point is an intuitive measure that decreases in every rewrite step. Often such a measure can be expressed in terms of a strict interpretation. Experience with the corresponding method for first-order term rewriting systems surely helps.

Another helpful hint to find a strict interpretation (2), is the following way to find more and more strict functionals. First, by Proposition 4.3.6 we obtain strict functionals S_0^σ for each type σ. Then, by Lemma 4.3.7, the sum of a strict functional and a weakly monotonic functional is again strict. To find weakly monotonic functionals, Corollary 4.1.6 can be used, which states that anything that can be written as a lambda term is weakly monotonic. The last helpful hint is that all strictly monotonic functions in a first-order signature are strict functionals of type level 1.

Finally, in order to verify that (3) holds we refer the reader to the various computation rules that have been proved, especially in Section 4.2, Section 4.3.2.2 and Section 4.5.2.3. The preservation statements, formulated in Section 4.2, can be used to lift computation rules that hold on base types to higher types.

The next section is devoted to several illustrations of the method we propose.

5.1.2 Second-order Applications

We will now provide some examples, to illustrate the use of higher-order monotone algebras for termination proofs. The first example is the HRS $\mathcal{H}_{\text{pnf}}$, finding prenex normal forms, introduced in Section 2.5.4. Then we show termination of a specification of the surjective disjoint union. This example is typical, as it contains a non-pattern left hand side. Finally, we show an example from process algebra with a choice-operator over data. In fact this latter example was the motivation to start working on termination of higher-order term rewriting.

Most higher-order rewrite systems are in fact second-order systems, in the sense that the function symbols have type level at most 2, and the variables have a type of level at most 1. It is therefore advantageous to inspect how the various partial orders and notions of monotonicity behave on the lower type levels.

The base types have type level 0. On this level, several notions coincide. For ι with $\mathrm{TL}(\iota) = 0$, we have

- $x \,{}_{\text{st}}{>}_\iota\, y \iff x \,{}_{\text{wm}}{>}_\iota\, y \iff x >_\iota y$,
- $x \,{}_{\text{wm}}{\geq}_\iota\, y \iff (x >_\iota y \vee x = y)$,
- $\mathcal{I}_\iota = \mathcal{T}_\iota = \mathcal{WM}_\iota = \mathcal{ST}_\iota$.

Type level 1 contains the usual functions. On this level, the usual notions of strict and weak monotonicity apply. The different orders collapse to the pointwise ordering on functions, defined as

$f > g$ if and only if for all arguments $\vec{x}, f(\vec{x}) > g(\vec{x})$.

Likewise, we define

$f \geq g$ if and only if for all arguments $\vec{x}, f(\vec{x}) \geq g(\vec{x})$.

We have for σ with $\mathrm{TL}(\sigma) = 1$,

- $f \in \mathcal{ST}_\sigma$ if f is strictly monotonic in the usual sense.

- $f \in \mathcal{WM}_\sigma$ if f is weakly monotonic in the usual sense.
- $f \ {}_{\mathrm{wm}}{>}_\sigma \ g$ and $f \ {}_{\mathrm{st}}{>}_\sigma \ g$ are equivalent to $f > g$.
- $f \ {}_{\mathrm{wm}}{\geq}_\sigma \ g$ is equivalent to $f \geq g$.

On type level 2, the various notions diverge. We note that for σ with $\mathrm{TL}(\sigma) = 2$, we have

- $F \in \mathcal{ST}_\sigma$ if and only if for all weakly monotonic arguments $\vec{f}, g_1, g_2, \vec{h}$ (of type level at most 1), if $g_1 > g_2$ then $F(\vec{f}, g_1, \vec{h}) >_\iota F(\vec{f}, g_2, \vec{h})$.

5.1.2.1 Prenex Normal Form

The first example proves termination of the HRS $\mathcal{H}_{\mathrm{pnf}}$, computing the prenex normal form of first order formulae (introduced in Section 2.5.4.1). Because the system is rather symmetric, we only repeat the first and the last rule. Recall that the free variables, written with capitals, should be abstracted in the left- and right hand sides to obtain the proper rewrite rules.

$$\begin{array}{rcl} P \wedge \forall\lambda x.(Qx) & \mapsto & \forall\lambda x.P \wedge (Qx) \\ \neg\exists\lambda x.(Qx) & \mapsto & \forall\lambda x.\neg(Qx) \end{array}$$

We interpret both base types o and ι by the well-founded ordered domain $(\mathbb{N}, >, 0, +)$. As interpretation for the constants, we put:

$$\begin{array}{rcl} \mathcal{J}(\wedge) = \mathcal{J}(\vee) & = & \boldsymbol{\lambda} x, y \in \mathbb{N}.(2 \cdot x + 2 \cdot y) \\ \mathcal{J}(\neg) & = & \boldsymbol{\lambda} x \in \mathbb{N}.(2 \cdot x) \\ \mathcal{J}(\forall) = \mathcal{J}(\exists) & = & \boldsymbol{\lambda} f \in \mathbb{N} \Rightarrow \mathbb{N}.(f(0) + 1) \end{array}$$

The first two functions are strictly monotonic, hence strict. The second function is also strict, for if $f \ {}_{\mathrm{st}}{>} \ g$, then $f(0) + 1 > g(0) + 1$. Finally, we have to check that the rules are decreasing. Because $\mathcal{J}(\forall) = \mathcal{J}(\exists)$ and $\mathcal{J}(\vee) = \mathcal{J}(\wedge)$ is symmetric, the only rules from Section 2.5.4.1 we have to prove decreasing are the two mentioned above. The calculations are simple. Let an arbitrary weakly monotonic valuation α be given. We write $[\![M]\!]$ for the interpretation of M under this valuation.

$$\begin{array}{rl} & [\![P \wedge \forall\lambda x.(Qx)]\!] \\ = & 2 \cdot [\![P]\!] + 2 \cdot ([\![Q]\!](0) + 1) \\ > & 2 \cdot [\![P]\!] + 2 \cdot [\![Q]\!](0) + 1 \\ = & \boldsymbol{\lambda} x.(2 \cdot [\![P]\!] + 2 \cdot [\![Q]\!](x))(0) + 1 \\ = & [\![\forall\lambda x.P \wedge (Qx)]\!] \end{array}$$

and

$$\begin{array}{rl} & [\![\neg\exists\lambda x.(Qx)]\!] \\ = & 2 \cdot ([\![Q]\!](0) + 1) \\ > & 2 \cdot [\![Q]\!](0) + 1 \\ = & \boldsymbol{\lambda} x.(2 \cdot [\![Q]\!](x))(0) + 1 \\ = & [\![\exists\lambda x.\neg(Qx)]\!] \quad . \end{array}$$

We have found a termination model for $\mathcal{H}_{\mathrm{pnf}}$, so by Theorem 5.1.4, it is terminating.

Note that in this example, the fact that $[\![Q]\!]$ is weakly monotonic is not important. The system is fairly simple, because both the left- and the right hand side are only patterns, so β-reduction and expansion play a marginal rôle.

5.1.2.2 Surjective Disjoint Union

The next example is a specification of the disjoint union of two base types A and B. The union is itself a base type, U. The function symbols consist of two constructors, the left and right injections, and a case distinction for each base type. The latter construct has a type of level 2. The function symbols have the following types (where $\iota \in \{A, B, U\}$):

$$\begin{array}{rcl} case_\iota & : & U \to (A \to \iota) \to (B \to \iota) \to \iota \\ inl & : & A \to U \\ inr & : & B \to U \end{array}$$

All function symbols have type level ≤ 2. There are nine rules, namely for each base type ι an instance of the schemas below. The first two are the ordinary reduction rules for case distinction; the last one expresses that disjoint union is surjective (it expresses that Z must be either a left- or a right injection). The following symbols are used as free variables: $\{X^A, Y^B, Z^U, F^{A\to\iota}, G^{B\to\iota}, H^{U\to\iota}\}$.

$$\begin{array}{rcl} case_\iota(inl(X), F, G) & \mapsto & F(X) \\ case_\iota(inr(Y), F, G) & \mapsto & G(Y) \\ case_\iota(Z, \lambda x^A.H(inl(x)), \lambda y^B.H(inr(y))) & \mapsto & H(Z) \end{array}$$

Note that this example does not fit in the framework of Nipkow [Nip91, p. 347], because the left hand side of the last rule is not a pattern (the argument of H is not a bound variable). Termination for this example is less trivial than for the prenex normal form, because there is an application of two free variables. Therefore, it is not the case that the number of *case* occurrences decreases in each step: If X contains a *case* occurrence, then $F(X)$ can generate many copies of it in the right hand side of the first rule.

Nevertheless, the interpretation in a termination model is smooth. Take $\mathcal{I}(\iota) = (\mathbb{N}, >, 0, +)$, for each $\iota \in \{A, B, U\}$. Interpret the constants as follows.

$$\begin{array}{c} \mathcal{J}(inl) = \mathcal{J}(inr) = \boldsymbol{\lambda} a.a \\ \mathcal{J}(case) = \boldsymbol{\lambda} a \in \mathbb{N}.\boldsymbol{\lambda} f, g \in \mathbb{N} \Rightarrow \mathbb{N}.f(a) + g(a) + a + 1 \end{array}$$

We first show that these functions are strict. This is only non-trivial for the last function. We check strictness of $\mathcal{J}(case)$ in all arguments. First, let $a > b$ and let $f, g \in \mathbb{N} \Rightarrow \mathbb{N}$ be weakly monotonic. Then $f(a) + g(a) \geq f(b) + g(b)$, so $f(a) + g(a) + a + 1 > f(b) + g(b) + b + 1$. Hence $\mathcal{J}(case)$ is monotonic in a.

Now let $f_1 \ {}_{\mathrm{st}}{>} \ f_2$ and let $a \in \mathbb{N}$ and $g \in \mathbb{N} \Rightarrow \mathbb{N}$ be weakly monotonic. Then $f_1(a) > f_2(a)$, and $g(a) \geq g(b)$, hence $f_1(a) + g(a) + a + 1 > f_2(a) + g(a) + a + 1$. This

shows strictness in f. Strictness in g can be checked similarly. Hence $(\mathbb{N}, >, 0, +, \mathcal{J})$ is a higher-order monotone algebra.

Finally, we verify that the rules are decreasing. To this end, we compute the interpretations of the left- and right hand sides of the rules, for arbitrary valuation.

Left hand side	Right hand side
$[\![F]\!]([\![X]\!]) + [\![G]\!]([\![X]\!]) + [\![X]\!] + 1$	$[\![F]\!]([\![X]\!])$
$[\![F]\!]([\![Y]\!]) + [\![G]\!]([\![Y]\!]) + [\![Y]\!] + 1$	$[\![G]\!]([\![Y]\!])$
$2 \cdot [\![H]\!]([\![Z]\!]) + [\![Z]\!] + 1$	$[\![H]\!]([\![Z]\!])$

The left hand sides are clearly greater than the right hand sides. So the higher-order monotone algebra indicated above is a termination model. By Theorem 5.1.4, the system for the "surjective disjoint union" is terminating.

To get full disjunction, we have to add a union operator as type forming constructor, and we have to add case distinctions for arbitrary types. See [Gan80, Kah95] for a semantical termination proof of the resulting system. Moreover, rules for permutative conversions have to be added to find nice normal forms. In Section 5.5 we show that permutative conversions for existential quantifiers can be dealt with.

5.1.2.3 Process Algebra with Data

The final second-order application comes from process algebra [BK84], or better μCRL, which extends process algebra with abstract data types [GP90, GP94]. We only concentrate on the fragment with non-deterministic choice ($+$), sequential composition ($\cdot$), deadlock (δ) and the data dependent choice (Σ) from μCRL. The Process Algebra part can be formulated in a first order Term Rewriting System (see for instance [AB91]). The rules for the Sum-operator require higher-order rewrite rules to deal with the bound variables. A similar formulation of μCRL can be found in [Sel94].

There are two base types: $\{Proc, Data\}$. Furthermore, here is a list of the function symbols with their types:

$$\begin{array}{lcl} + & : & Proc \to Proc \to Proc \\ \cdot & : & Proc \to Proc \to Proc \\ \delta & : & Proc \\ \sum & : & (Data \to Proc) \to Proc \end{array}$$

$\{X, Y, Z, P, Q, D\}$ are used as free variables. Now we have the following set of rules, with the binary function symbols written infix. Note that the left hand side of rule

Sum3 is not a pattern, and that Sum1 and Sum5 have an implicit side condition.

$$
\begin{array}{rrcl}
\text{A3:} & X + X & \mapsto & X \\
\text{A4:} & (X + Y) \cdot Z & \mapsto & (X \cdot Z) + (Y \cdot Z) \\
\text{A5:} & (X \cdot Y) \cdot Z & \mapsto & X \cdot (Y \cdot Z) \\
\text{A6:} & X + \delta & \mapsto & X \\
\text{A7:} & \delta \cdot X & \mapsto & \delta \\
\text{Sum1:} & \Sigma(\lambda d^{Data}.X) & \mapsto & X \\
\text{Sum3:} & \Sigma(\lambda d^{Data}.(Pd)) + (PD) & \mapsto & \Sigma(\lambda d^{Data}.(Pd)) \\
\text{Sum4:} & \Sigma(\lambda d^{Data}.(Pd) + (Qd)) & \mapsto & \Sigma(\lambda d^{Data}.(Pd)) + \Sigma(\lambda d^{Data}.(Qd)) \\
\text{Sum5:} & \Sigma(\lambda d^{Data}.(Pd)) \cdot X & \mapsto & \Sigma(\lambda d^{Data}.(Pd) \cdot X)
\end{array}
$$

To prove termination of this system we interpret both base types $Data$ and $Proc$ by $(\mathbb{N}_{\geq 1}, >, 1, +)$, with the usual meaning. This is a well-founded ordered domain. The function symbols are interpreted in the following way:

$$
\begin{array}{rcl}
[\![+]\!] & = & \boldsymbol{\lambda} a.\boldsymbol{\lambda} b.a + b + 1 \\
[\![\cdot]\!] & = & \boldsymbol{\lambda} a.\boldsymbol{\lambda} b.a \cdot (b + 1) \\
[\![\delta]\!] & = & 1 \\
[\![\sum]\!] & = & \boldsymbol{\lambda} f.(3 \cdot f(1) + 1)
\end{array}
$$

In the right hand sides of these equations, $\cdot$ denotes multiplication and $+$ denotes addition on natural numbers. This is an extension of the interpretation in [AB91] for the Process Algebra part of the system. The first three functions are strictly monotonic in $\mathbb{N}_{\geq 1}$, hence strict. The last functional is also strict, for if $f > g$ (pointwise), then $f(1) > g(1)$, hence $3 \cdot f(1) + 1 > 3 \cdot g(1) + 1$. Now we compute the values of the left hand sides and right hand sides.

	interpretation of the left hand side	interpretation of the right hand side
A3	$2 \cdot [\![X]\!] + 1$	$[\![X]\!]$
A4	$([\![X]\!] + [\![Y]\!] + 1) \cdot ([\![Z]\!] + 1)$	$([\![X]\!] + [\![Y]\!]) \cdot ([\![Z]\!] + 1) + 1$
A5	$[\![X]\!] \cdot ([\![Y]\!] + 1) \cdot ([\![Z]\!] + 1)$	$[\![X]\!] \cdot ([\![Y]\!] \cdot [\![Z]\!] + [\![Y]\!] + 1)$
A6	$[\![X]\!] + 2$	$[\![X]\!]$
A7	$[\![X]\!] + 1$	1
Sum1	$3 \cdot [\![X]\!] + 1$	$[\![X]\!]$
Sum3	$3 \cdot [\![P]\!](1) + [\![PD]\!] + 2$	$3 \cdot [\![P]\!](1) + 1$
Sum4	$3 \cdot ([\![P]\!](1) + [\![Q]\!](1)) + 4$	$3 \cdot ([\![P]\!](1) + [\![Q]\!](1)) + 3$
Sum5	$(3 \cdot [\![P]\!](1) + 1) \cdot ([\![X]\!] + 1)$	$3 \cdot [\![P]\!](1) \cdot ([\![X]\!] + 1) + 1$

It is easy to verify that in $\mathbb{N}_{\geq 1}$, the interpretation of the left hand side is greater than the interpretation of the right hand side on each line in the table. Hence we have found a termination model, and by Theorem 5.1.4 this system of Process Algebra and Sum rules is terminating. Because commutativity and associativity (AC) of $+$ holds in the termination model, also termination modulo AC can be proved in this way.

5.2 Internalizing Simply-typed Lambda Calculus

5.2.1 Encoding the Simply-typed Lambda Calculus

In Section 2.5.4 an HRS representing the untyped lambda calculus was presented. In this section we encode the simply-typed lambda calculus $\lambda^{\rightarrow}_{\beta}$ as an HRS. We assume for convenience that there is only one base type, called o.

It may be confusing that the simply-typed lambda calculus now exists at two levels. It underlies the formalism of higher-order term rewriting as substitution calculus. On the other hand, simply-typed lambda calculus is just a particular HRS.

The advantage of the encoding of $\lambda^{\rightarrow}_{\beta}$ as an HRS is that we can now deal with extensions of lambda calculus in a uniform framework. One such extension will be Gödel's T. This enables us to give more modular termination proofs, because β-reduction is now an ordinary rewrite rule.

In principle, the presence of two levels results in two forms of abstraction and application and also in two different β-rules. But a more leisure encoding is obtained by identifying the types of the HRS and the calculus that we encode. We also identify the application symbols of the two levels. To encode typed lambda abstraction we introduce infinitely many constants.

The β-HRS $\mathcal{H}_{\mathrm{lam}}$ is defined as follows. As function symbols it has for each type $\sigma, \tau \in \mathbb{T}^{\rightarrow}(\{o\})$,

$$\mathsf{abs}_{\sigma,\tau} : (\sigma \to \tau) \to \sigma \to \tau \ .$$

The β-reduction rule can now be represented by the higher-order rewrite rule schema (for each type σ, τ)

$$(\mathtt{beta}) \qquad \mathsf{abs}_{\sigma,\tau}(\lambda x^{\sigma}.Fx)X \mapsto (FX) \ .$$

We write $\to_{\mathtt{beta}}$ for the rewrite relation induced by $\mathcal{H}_{\mathrm{lam}}$.

We can represent any lambda term as β-normal form in the signature of $\mathcal{H}_{\mathrm{lam}}$ as follows. We write $\langle M \rangle$ for the representation of M.

$$\begin{array}{rcl} \langle x^{\sigma} \rangle & \equiv & x^{\sigma} \\ \langle MN \rangle & \equiv & (\langle M \rangle \, \langle N \rangle) \\ \langle \lambda x^{\sigma}.M^{\tau} \rangle & \equiv & (\mathsf{abs}_{\sigma,\tau} \, \lambda x^{\sigma}.\langle M \rangle) \end{array}$$

The following lemma expresses a partial correctness of the encoding of $\lambda^{\rightarrow}_{\beta}$ into $\mathcal{H}_{\mathrm{lam}}$.

Lemma 5.2.1 *For all terms M and N (of the appropriate type)*

1. $\langle M \rangle [x := \langle N \rangle] \equiv \langle M[x := N] \rangle$.
2. *If $M \to_{\beta} N$ then $\langle M \rangle \to_{\mathtt{beta}} \langle N \rangle$.*

Proof: (1) Straightforward by induction on M. (2) Induction on the definition of $\to_\beta$. We only do the base case (outermost reduction). The other cases are immediate. Let $M = (\lambda x.P)Q$ and $N = P[x := Q]$. Then

$$\begin{array}{rcll} \langle M\rangle & \equiv & \langle(\lambda x.P)Q\rangle & \\ & \equiv & (\mathsf{abs}\,(\lambda x.\langle P\rangle)\langle Q\rangle) & \\ & \mapsto_{\mathtt{beta}} & (\lambda x.\langle P\rangle)\langle Q\rangle & \\ & \to_\beta & \langle P\rangle[x := \langle Q\rangle] & \\ & \equiv & \langle P[x := Q]\rangle & \text{By (1)} \\ & \equiv & \langle N\rangle & \end{array}$$

⊠

As a consequence, each β-reduction sequence in $\lambda^{\to}_{\beta}$ maps to a `beta`-reduction sequence in $\mathcal{H}_{\mathrm{lam}}$ of equal length. Hence termination of $\mathcal{H}_{\mathrm{lam}}$ implies termination of $\lambda^{\to}_{\beta}$. However, in the definition of a higher-order rewriting step, it is already used that unique β-normal forms exist, so in fact weak normalization and confluence of $\lambda^{\to}_{\beta}$ is already presumed.

Remark. In fact, too much terms are well-typed in $\mathcal{H}_{\mathrm{lam}}$, because $\langle\_\rangle$ is not surjective. An example of a typable term in $\mathcal{H}_{\mathrm{lam}}$ is $(\mathsf{abs}\,\lambda F.F(\lambda x.x))(\lambda g.gy)$. Note that this term reduces in a single `beta`-step to y. This illustrates that the reduction relation of $\mathcal{H}_{\mathrm{lam}}$ is more complex than that of $\lambda^{\to}_{\beta}$.

5.2.2 Termination Models for $\mathcal{H}_{\mathrm{lam}}$

We now show that many ordered domains can be extended to a termination model for $\mathcal{H}_{\mathrm{lam}}$. The additional information that we need is that the ordered domain admits a strict projection.

Definition 5.2.2 *Given a partial order $(A, >)$, a binary function $\otimes$ in $A \Rightarrow A \Rightarrow A$ is* projective *if it is strictly monotonic and for all $x, y \in A$, $x \otimes y > x$.*

Examples are addition in $(\mathbb{N}_{\geq 1}, >)$, multiplication in $(\mathbb{N}_{\geq 2}, >)$ and the function $\boldsymbol{\lambda} x, y.x+y+1$ in $(\mathbb{N}, >)$. With $\otimes_\sigma$ we denote the pointwise extension of $\otimes$ to arguments of type σ (analogously to $\oplus_{\sigma,\tau}$). By straightforward induction on σ one proves that for all $x, y \in \mathcal{WM}_\sigma$, $x \otimes_\sigma y \;{}_{\mathrm{wm}}{>}\; x$.

Theorem 5.2.3 *Let $\mathcal{A}$ be a well-founded ordered domain with a projective $\otimes$. Then there exists a constant interpretation $\mathcal{J}$, such that $(\mathcal{A}, \mathcal{J})$ is a termination model for $\mathcal{H}_{\mathrm{lam}}$.*

Proof: Let $\mathcal{A} = \{A, >, 0, +\}$. The only function symbols to be interpreted are $\mathsf{abs}_{\sigma,\tau}$. We put

$$\mathcal{J}(\mathsf{abs}_{\sigma,\tau}) = \mathsf{I}_{\sigma\to\tau} \otimes_{(\sigma\to\tau)\to\sigma\to\tau} \mathsf{S}_0^{(\sigma\to\tau)\to\sigma\to\tau}.$$

This is strict, by Lemma 4.3.7 (the fact that we use $\otimes$ instead of $\oplus$ is inessential). We now show that the `beta`-rule is decreasing: Let α be a weakly monotonic valuation; we write $[\![M]\!]$ for the interpretation under α. We have

$$\begin{array}{rl} & [\![\mathsf{abs}_{\sigma,\tau}(\lambda x.Fx)X]\!] \\ = & [\![F]\!]([\![X]\!]) \otimes \mathsf{S}_0([\![F]\!],[\![X]\!]) \\ \underset{\mathrm{wm}}{>} & [\![F]\!]([\![X]\!]) \quad \text{because } \otimes \text{ is projective} \\ = & [\![(FX)]\!] \end{array}$$

Hence $(\mathcal{A},\mathcal{J})$ is a termination model. ⊠

Corollary 5.2.4 *$\lambda^{\rightarrow}_{\beta}$ terminates.*

Proof: The function $\boldsymbol{\lambda} x,y.x+y+1$ is projective in the well-founded ordered domain $(\mathbb{N},>,0,+)$. By Theorem 5.2.3 there is a termination model for $\mathcal{H}_{\mathrm{lam}}$. By Theorem 5.1.4 this implies termination of $\mathcal{H}_{\mathrm{lam}}$, which by Lemma 5.2.1.(2) implies termination of $\lambda^{\rightarrow}_{\beta}$.

Remark. In the proof of Corollary 5.2.4, the interpretation of abstraction obeys the following equation:

$$\mathcal{J}(\mathsf{abs}_{\sigma,\tau})(f,x) = f(x) \oplus_{\tau} \mathsf{S}_0(f,x) \oplus_{\tau} \underline{1}_{\tau} \ .$$

This example is quite typical for (small extensions of) various lambda calculi. The $f(x)$ is inspired by the standard model for the `beta`-rule; this part ensures that the left hand side is not smaller than the right hand side. To make $\mathcal{J}(\mathsf{abs})$ strict the S_0-part is added. The $\oplus\underline{1}$ was added to ensure that the rule really gets decreasing.

Using this interpretation, an upper bound on the length of the longest reduction from any term M can be given, namely $\mathsf{M}([\![\langle M\rangle]\!]_{\alpha_{\mathsf{S}}})$. The most striking difference with the interpretation that Gandy found (namely $[\![\mathsf{L}_{\tau\rightarrow o}M^*]\!]_{\alpha_{\mathsf{L}}}$, see Section 3.3.3) is that f occurs as argument of the S_0-part. This is necessary to make $\mathcal{J}(\mathsf{abs})$ strict. From Theorem 3.3.5 and the remark at the beginning of Section 4.4, it appears that this occurrence of f is in fact superfluous in the upper bound expression.

The reason for the coarse upper bound is that too much terms are well typed in $\mathcal{H}_{\mathrm{lam}}$ and consequently its reduction relation is more complex than $\lambda^{\rightarrow}_{\beta}$.

5.2.3 Modularity of Termination

In higher-order rewriting, a set of higher-order rules is turned into a reduction system, by closing the rules under context and computing modulo a substitution calculus, like $\lambda^{\rightarrow}_{\beta}$. Following e.g. [Bre88, Dou92, JO91], it is also possible to add β-reduction as an ordinary rewrite rule. Given a set of rewrite rules, the reduction relation is then obtained by closing the rules under the β-rule, substitution and context. The one step rewrite relation is easier to compute, because it is not generated modulo a theory.

First-order rules.

A particular situation arises when the rewrite rules form a (first-order) TRS. It is proved in [Bre88], that if the TRS $\mathcal{R}$ is terminating, then the combination $\mathcal{R} \cup \beta$ terminates too. A similar modularity result holds for confluence.

The preservation of termination is a non-trivial modularity result. We have no restrictions on the TRS, so in particular it can have duplicating and collapsing rules. Also β-reduction may duplicate and collapse arguments. Hence the situation seems quite bad in view of Toyama's example, given in Section 3.1.2. The combination is even not completely disjoint, because both components share the same application symbols.

The proof in [Bre88] is based on the SN-proof for simply-typed lambda calculus that uses computability predicates. We will give an alternative proof of this fact. Our proof is more or less a corollary of the semantical proof of termination of simply-typed lambda calculus. We proved that every projective well-founded ordered domain can be used to prove termination of simply-typed lambda calculus. Given a terminating TRS, its term algebra gives rise to a well-founded ordered domain. This need not be projective, but it is if we add the rule $g(x, y) \mapsto x$ to the original TRS.

Definition 5.2.5 *Let $\mathcal{R}$ be a TRS. Then we define the HRS $\mathcal{H}_{\mathcal{R}\beta}$ as the union of the curried version of $\mathcal{R}$ and $\mathcal{H}_{\mathrm{lam}}$.*

Theorem 5.2.6 *Let $\mathcal{R} = (\mathcal{F}, \mathcal{V}, \mathcal{R})$ be a terminating TRS. Then the HRS $\mathcal{H}_{\mathcal{R}\beta}$ is terminating too.*

Proof: Let c be a new constant and g be a fresh binary function symbol. By Proposition 3.1.4, $(\mathcal{F} \uplus \{c, g\}, \mathcal{R} \uplus \{g(x, y) \mapsto x\})$ is still terminating. By Theorem 3.1.3, there exists a first-order termination model $(A, >, (f_A)_{f \in \mathcal{F} \cup \{c,g\}})$. Now $(A, >, c_A, g_A)$ is a well-founded ordered domain, with a projective g_A. Each f_A is strictly monotonic and its type is of type level at most 1, so it is strict. So $(A, >, c_A, g_A, (f_A)_{f \in \mathcal{F} \cup \{c,g\}})$ is a higher-order termination model for the curried version of $\mathcal{R}$. By Theorem 5.2.3 it can be extended to a termination model of $\mathcal{H}_{\mathcal{R}\beta}$. ⊠

Higher-order rules.

What happens when we combine higher-order rules with β-reduction? In [JO91] a kind of primitive recursive format occurs. It is proved there that the rewrite system composed from a set of higher-order rules in this format, combined with an arbitrary terminating TRS and β-reduction is still terminating. The following rules, with constant symbol $K : (o \to o) \to (o \to o) \to (o \to o)$ are outside this primitive recursive format:

$$\begin{array}{rcl} Kfgx & \mapsto & fx \\ K(Kf)h & \mapsto & Kf(Kgh) \end{array}$$

It is not difficult to give a termination model for this system. This proves termination of the K-rules *modulo* β. However, this does not immediately imply termination of the K-rules combined with β-reduction. E.g. the reduction $(\lambda z.a)(Kfgx) \to$

$(\lambda z.a)(fx) \to a$ lives completely inside one β-equivalence class. Therefore, reduction sequences in the combined system cannot be mapped to reduction sequences in the system modulo β.

The idea now is that the translation $\langle \_ \rangle$ of Section 5.2.1, maps reductions in the combination of the K-rules with β to reductions in the HRS $K \cup \mathcal{H}_{\mathrm{lam}}$. The problematic reduction above becomes

$$(\mathsf{abs}\,\lambda z.a)(Kfgx) \to (\mathsf{abs}\,\lambda x.a)(fx) \to a \ .$$

This idea only works when the original set of rules contains no lambdas. It is often the case that the higher-order rewrite rules do not contain lambdas. This is for instance the case in Gödel's T and in the system of K-rules above. For such systems, we derive a partial modularity result, based on Theorem 5.2.3.

Theorem 5.2.7 *Let $\mathcal{R}$ be a set of higher-order rules without lambdas. If there is a termination model for $\mathcal{R}$ that has a projective function, then the combination $\mathcal{R} \cup \beta$ terminates.*

Proof: Assume that $\mathcal{R}$ has a termination model, in which $\otimes$ is projective. A reduction step $M \to N$ in the combined system is either an $\mathcal{R}$-step or a β step. We apply the translation into $\mathcal{H}_{\mathrm{lam}}$, $\langle \_ \rangle$. If $M \to_\beta N$, then $\langle M \rangle \to_{\mathtt{beta}} \langle N \rangle$ by Lemma 5.2.1.(2). If $M \to N$, using the rewrite rule $L \mapsto R$, then for some context $C[\,]$ and substitution θ, $M \equiv C[L^\theta]$. Put $C'[\,] := \langle C[\,] \rangle$ and $\theta'(x) := \langle \theta(x) \rangle$. Because L contains no lambdas, $\langle L \rangle \equiv L$. Using this and Lemma 5.2.1.(1), we get:

$$\langle M \rangle \equiv \langle C[L^\theta] \rangle \equiv C'[\langle L \rangle^{\theta'}] \equiv C'[L^{\theta'}] \ .$$

Similarly, $\langle N \rangle \equiv C'[R^{\theta'}]$. Hence, we have that $\langle M \rangle \to \langle N \rangle$ using the rule $L \mapsto R$.

This shows that any reduction in $\mathcal{R} \cup \beta$ can be mapped to a reduction in the HRS $\mathcal{R} \cup \mathcal{H}_{\mathrm{lam}}$ of equal length. By assumption, $\otimes$ is projective in the termination model of $\mathcal{R}$, so by Theorem 5.2.3 this model can be extended to a termination model of $\mathcal{R} \cup \mathcal{H}_{\mathrm{lam}}$. By Theorem 5.1.4 this combined system terminates, hence $\mathcal{R} \cup \beta$ terminates. ⊠

This is only a partial modularity result, because the theorem in fact relies on a particular termination proof for the higher-order rules. Hence we have not proved modularity of termination for the combination of higher-order rules with β-reduction, but nevertheless we provide a modular approach to termination proofs of such systems.

As an example, we give a termination model for the K-rules given before. These rules contain no lambdas, hence by Theorem 5.2.7, the combination of the K-rules with β-reduction terminates.

We take the ordered domain $(\mathbb{N}, >, 0, +)$ with as interpretation

$$\mathcal{J}(K)(f, g, x) := 2 \cdot f(x) + g(x) + x + 1$$

This is clearly strict. Moreover, writing f, g, h, x for the interpretation of f, g, h, x under an arbitrary weakly monotonic valuation, we compute

$$\begin{array}{rcl} [\![Kfgx]\!] & = & 2 \cdot f(x) + g(x) + x + 1 \\ & > & f(x) \\ & = & [\![fx]\!] \end{array}$$

and for arbitrary $a \in \mathbb{N}$,

$$\begin{array}{rcl} [\![K(Kfg)h]\!](a) & = & 2 \cdot \mathcal{J}(K)(f,g,a) + h(a) + a + 1 \\ & = & 2 \cdot (2 \cdot f(a) + g(a) + a + 1) + h(a) + a + 1 \\ & > & 2 \cdot f(a) + (2 \cdot g(a) + h(a) + a + 1) + a + 1 \\ & = & 2 \cdot f(a) + \mathcal{J}(K)(g,h,a) + a + 1 \\ & = & [\![Kf(Kgh)]\!](a) \ . \end{array}$$

Hence both rules are decreasing in $(\mathbb{N}, >, 0, +, \mathcal{J})$.

Map and append.

We end this section with an example involving map and append on lists. Again, the intended rewrite relation is obtained by combining the higher-order rules with β-reduction.

Consider terms built up from the constants

$$\begin{array}{rclcrcl} \mathsf{nil} & : & o & & \mathsf{append} & : & o \to o \to o \\ \mathsf{cons} & : & o \to o \to o & & \mathsf{map} & : & (o \to o) \to o \to o. \end{array}$$

The types are chosen such that e.g. $\mathsf{map}(\lambda x.\mathsf{append}(x,x), \ell)$ is well typed. Terms of type o represent finite lists of lists of $\cdots$. The functions map and append are defined via the following rewrite rules

$$\begin{array}{rcll} \mathsf{append}(\mathsf{nil}, \ell) & \mapsto & \ell & \text{(i)} \\ \mathsf{append}(\mathsf{cons}(k, \ell), m) & \mapsto & \mathsf{cons}(k, \mathsf{append}(\ell, m)) & \text{(ii)} \\ \mathsf{map}(f, \mathsf{nil}) & \mapsto & \mathsf{nil} & \text{(iii)} \\ \mathsf{map}(f, \mathsf{cons}(k, \ell)) & \mapsto & \mathsf{cons}(f(k), \mathsf{map}(f, \ell)) & \text{(iv)} \\ \mathsf{append}(\mathsf{append}(k, \ell), m) & \mapsto & \mathsf{append}(k, \mathsf{append}(\ell, m)) & \text{(v)} \\ \mathsf{map}(f, \mathsf{append}(\ell, k)) & \mapsto & \mathsf{append}(\mathsf{map}(f, \ell), \mathsf{map}(f, k)) & \text{(vi)} \end{array}$$

The rules fit in the schema of [JO91], for the rules of append form a terminating TRS and the rules for map are primitive recursive.

The rules contain no lambdas, hence we can also apply Theorem 5.2.7. This reduces termination of the combination of these rules with β-reduction to the task of finding a termination model for the six rules above.

As ordered domain, we choose $(\mathbb{N}, >, 0, +)$, which also has a strictly monotonic projection (e.g. $\lambda xy.x + y + 1$). The interpretation of the constants is defined as follows:

$$\begin{array}{rcl} \mathcal{J}(\mathsf{nil}) & := & 1 \\ \mathcal{J}(\mathsf{cons})(m,n) & := & m + n + 1 \\ \mathcal{J}(\mathsf{append})(m,n) & := & 2m + n + 2 \\ \mathcal{J}(\mathsf{map})(f,n) & := & \sum_{i=0}^{n} f(i) + 3n + 1 \end{array}$$

The interpretations of nil, cons and append are obviously strict. Strictness of $\mathcal{J}(\mathsf{map})$ follows e.g. by Lemma 4.3.7, if we write its definition as

$$\big(f(0) + n\big) + \big(\sum_{i=1}^{n} f(i) + 2n + 1\big) \ .$$

Hence we have a higher-order monotone algebra. We still have to check that the rules are decreasing. In the sequel k, ℓ, m, f are arbitrary values for the corresponding variables. Note that f ranges over weakly monotonic functionals. For rule (v) e.g. the check boils down to the true inequality $2 \cdot (2\ell + k + 2) + m + 2 > 2\ell + (2k + m + 2) + 2$. We don't present all calculations here, but let us yet verify the most difficult one, rule (vi):

$$\begin{aligned}
& [\![\mathsf{map}(f, \mathsf{append}(\ell, k))]\!] \\
= & \sum_{i=0}^{2\ell+k+2} f(i) + 3 \cdot (2\ell + k + 2) + 1 \\
= & \sum_{i=0}^{\ell} f(i) + \sum_{i=\ell+1}^{2\ell+1} f(i) + \sum_{i=2\ell+2}^{2\ell+k+2} f(i) + 6\ell + 3k + 7 \\
> & \sum_{i=0}^{\ell} f(i) + \sum_{i=0}^{\ell} f(i) + \sum_{i=0}^{k} f(i) + 6\ell + 3k + 5 \quad \text{because } f \text{ is monotonic} \\
= & \; 2 \cdot (\sum_{i=0}^{\ell} f(i) + 3\ell + 1) + (\sum_{i=0}^{k} f(i) + 3k + 1) + 2 \\
= & \; [\![\mathsf{append}(\mathsf{map}(f, \ell), \mathsf{map}(f, k))]\!]
\end{aligned}$$

For all rules, this relation between left- and right hand side holds. Therefore the combination of the rules for map and append with the β-reduction rule terminates.

5.3 Example: Gödel's T

We now apply our method to prove termination of Gödel's T. This system extends simply-typed lambda calculus with higher-order primitive recursion operators R_ρ of type $\rho \to (o \to \rho \to \rho) \to o \to \rho$, for any type ρ. Here o is a base type, for natural numbers, which comes with the constants 0^o and $\mathsf{S}^{o \to o}$. The usual rules expressing recursion are:

$$\begin{aligned}
\mathsf{R}_\rho\, gh0 &\;\mapsto\; g \;, \\
\mathsf{R}_\rho\, gh(\mathsf{S}x) &\;\mapsto\; hx(\mathsf{R}_\rho\, ghx) \;.
\end{aligned}$$

By Theorem 5.2.7, termination of the whole system (including β-reduction) follows if we find a termination model for the two schemas above. We work in the ordered domain $(\mathbb{N}, >, 0, +)$. The symbol 0 is interpreted by 0. The symbol S is interpreted as the successor function.

Furthermore, we define recursively

$$\begin{aligned}
\mathcal{J}(\mathsf{R}_\rho)(g, h, 0) &:= g \oplus_\rho \mathsf{S}_0(g, h) \oplus \underline{1}_\rho \;, \\
\mathcal{J}(\mathsf{R}_\rho)(g, h, n+1) &:= h(n, \mathcal{J}(\mathsf{R}_\rho)(g, h, n)) \oplus_\rho g \oplus_\rho \mathsf{S}_0(g, h) \oplus \underline{n+2}_\rho \;.
\end{aligned}$$

The intuition behind this interpretation is as follows: The first summand of both rules is inspired by the standard interpretation of the recursor operator, as dictated

by the rewrite rules. The addition of 1 makes the first rule decreasing. The addition of $\mathsf{S}_0(g,h)$ makes $\mathcal{J}(\mathsf{R})$ strict in g and h. Finally, to ensure strictness in the third argument, g and n have been added to the second defining clause. Finally, adding 2 ensures that the second rule is decreasing, and that the $n+1$-case is always greater than the 0-case, which is needed for strictness.

First, we show that the rules are decreasing. Write $[\![M]\!]$ for the denotation of M under an arbitrary weakly monotonic valuation α, then we have (for any ρ)

$$[\![\mathsf{R}_\rho\, gh0]\!] = [\![g]\!] \oplus \mathsf{S}_0([\![g]\!],[\![h]\!]) \oplus \underline{1} \;{}_{\mathrm{wm}}{>}\; [\![g]\!]$$

and

$$[\![\mathsf{R}_\rho\, gh(\mathsf{S}x)]\!] \;{}_{\mathrm{wm}}{>}\; [\![h]\!]([\![x]\!], \mathcal{J}(\mathsf{R}_\rho)([\![g]\!],[\![h]\!],[\![x]\!])) = [\![hx(\mathsf{R}_\rho\, ghx)]\!]$$

So the rules are decreasing.

We now show that $\mathcal{J}(\mathsf{R}_\rho)$ is strict for any ρ. Let for $n \in \mathbb{N}$, R^n denote the functional $\boldsymbol{\lambda} g,h.\mathcal{J}(\mathsf{R}_\rho)(g,h,n)$. We first show by induction on n that R^n is strict. After that, we show that if $m > n$ then $\mathsf{R}^m \;{}_{\mathrm{wm}}{>}\; \mathsf{R}^n$. This is also proved by induction on n. Together these two statements imply strictness of $\mathcal{J}(\mathsf{R}_\rho)$

- To prove: R^n is strict. If $n = 0$ then $\mathsf{R}^n = \big(\boldsymbol{\lambda} g,h.\, g \oplus \underline{1}\big) \oplus \mathsf{S}_0$, which is strict by Lemma 4.3.7. Assume now that R^n is strict, then it is also weakly monotonic. Note that

  $$\mathsf{R}^{n+1} = \big(\boldsymbol{\lambda} g,h.\, h(n, \mathsf{R}^n(g,h)) \oplus g \oplus \underline{n+2}\big) \oplus \mathsf{S}_0 \;\;,$$

  so also R^{n+1} is strict by Lemma 4.3.7.

- Let $m > n$, put $m' := m-1$. To prove: $\mathsf{R}^m \;{}_{\mathrm{wm}}{>}\; \mathsf{R}^n$. Let $g,h \in \mathcal{WM}$. It suffices to prove $\mathsf{R}^m(g,h) \;{}_{\mathrm{wm}}{>}\; \mathsf{R}^n(g,h)$. If $n = 0$, then we have

  $$\begin{array}{rcl} \mathsf{R}^m(g,h) & {}_{\mathrm{wm}}{\geq} & g \oplus \mathsf{S}_0(g,h) \oplus \underline{m'+2} \\ & {}_{\mathrm{wm}}{>} & g \oplus \mathsf{S}_0(g,h) \oplus \underline{1} \\ & = & \mathsf{R}^n(g,h). \end{array}$$

  Now for $n = n'+1$, we have $m' > n'$. By induction hypothesis, $\mathsf{R}^{m'}(g,h) \;{}_{\mathrm{wm}}{>}\; \mathsf{R}^{n'}(g,h)$. By weak monotonicity of h, we obtain

  $$h(m', \mathsf{R}^{m'}(g,h)) \;{}_{\mathrm{wm}}{\geq}\; h(n', \mathsf{R}^{n'}(g,h)) \;\;.$$

  Clearly $\underline{m'+2} \;{}_{\mathrm{wm}}{>}\; \underline{n'+2}$. Hence also in this case, $\mathsf{R}^m(g,h) \;{}_{\mathrm{wm}}{>}\; \mathsf{R}^n(g,h)$

So $\mathcal{J}(\mathsf{R}_\rho)$ is indeed strict.

We have proved all ingredients of the statement that $(\mathbb{N}, >, 0, +, \mathcal{J})$ is a termination model. The function $\boldsymbol{\lambda} x.\boldsymbol{\lambda} y.x{+}y{+}1$ is projective in this model, so with Theorem 5.2.7, we conclude that Gödel's T is strongly normalizing.

5.4 Example: Surjective Pairing

As illustration we show how surjective pairing can be specified as a terminating HRS. We prove termination by the method of Theorem 5.1.4.

We will use only the base type o. Furthermore, for any type $\sigma, \tau \in \mathbb{T}^{\times}(\{o\})$ we have the following constants:

$$\begin{array}{lcl} \Pi^0_{\sigma,\tau} & : & \sigma \times \tau \to \sigma \\ \Pi^1_{\sigma,\tau} & : & \sigma \times \tau \to \tau \\ \Pi_{\sigma,\tau} & : & \sigma \to \tau \to \sigma \times \tau \end{array}$$

The first two are the left- and right projection, respectively. The last one is the pairing operator.

As higher-order rewrite rules we introduce all well-typed instances of the following schemas:

$$\begin{array}{rcl} \Pi^0(\Pi xy) & \mapsto & x \\ \Pi^1(\Pi xy) & \mapsto & y \\ \Pi(\Pi^0 x, \Pi^1 x) & \mapsto & x \end{array}$$

To prove termination, we apply the general recipe. As ordered domain, we choose $(\mathbb{N}, >, 0, +)$. We interpret Π^0, Π^1 and Π as follows:

$$\begin{array}{rcl} \mathcal{J}(\Pi^0_{\sigma,\tau})(x,y) & = & x \oplus \mathsf{S}_0^{\sigma\times\tau\to\sigma}(x,y) \\ \mathcal{J}(\Pi^1_{\sigma,\tau})(x,y) & = & y \oplus \mathsf{S}_0^{\sigma\times\tau\to\tau}(x,y) \\ \mathcal{J}(\Pi_{\sigma,\tau})(x)(y) & = & (x,y) \oplus \mathsf{S}_0^{\sigma\to\tau\to\sigma\times\tau}(x)(y) \oplus \underline{1}_{\sigma\times\tau} \end{array}$$

It can easily be verified that this is a termination model for surjective pairing, hence the system terminates. Because the ordered domain is projective and the HRS contains no lambdas, also the combination of surjective pairing with β-reduction is terminating (Theorem 5.2.7).

5.5 Example: Permutative Conversions in Natural Deduction

The next example comes from proof theory in the style of Prawitz. In [Pra71] proofs are formalized by natural deduction trees. Several reduction rules on those trees are given, to bring them into a certain normal form. These reductions are divided in *proper reductions* and *permutative conversions*. Strong normalization is then proved via a refined notion of strong computability, called *strong validity* (see the Appendix for a reproduction of this proof).

In [Gan80] also examples taken from proof theory occur. There a normalization proof is given via hereditarily monotonic functionals, but the permutative conversions are not dealt with. Girard gives another adaptation of Gandy's approach, which can be extended to the full calculus, including permutative conversions (see [Gir87, Exc. 2.C.10]). Instead of bounding reduction lengths by functionals, Girard uses the

length of a specific reduction path, given by a weak normalization theorem for the full calculus.

We present a termination proof for the whole calculus, including the permutative conversions. We start with introducing a linear notation for natural deduction trees, derivation terms (Section 5.5.1). Then we translate the calculus of derivation terms into the HRS $\mathcal{H}_\exists$, which is based on $\lambda^\times_\beta$ (Section 5.5.2). In Section 5.5.3 we prove termination of $\mathcal{H}_\exists$, using the method of Section 5.1.1. The translation into $\mathcal{H}_\exists$ will be such that strong normalization of the calculus of derivation terms immediately follows.

5.5.1 Proof Normalization in Natural Deduction

The set of first-order formulae that we work with, is defined inductively as follows.

$$P(r_1, \ldots, r_n) \;\mid\; A \to B \;\mid\; A \wedge B \;\mid\; \exists x.A \;\mid\; \forall x.A$$

Here and in the sequel $A, B, C, \ldots$ denote arbitrary formulae. Metavariables $r, s, t, \ldots$ range over first-order terms. Atomic formulae are of the first form in the definition above, where P is an n-ary predicate symbol. The set of free variables in a formula (denoted by $\mathrm{FV}(A)$) is defined as usually, i.e. $\exists x$ and $\forall x$ above bind precisely the free occurrences of x in A.

Disjunction is not included, to avoid that we have to extend the previous theory with coproduct types. This extension is possible (actually [Gan80, Kah95] treat coproduct types), but not necessary for our purpose, namely to show that the semantical proof method can deal with permutative conversions. Also negation is absent, so we work in *minimal* logic, where negation plays no special rôle ($\bot$ may be present as 0-ary function symbol).

We now introduce derivation terms. They can be seen as linear notations for natural deduction trees (cf. [Pra71]), with assumptions labeled by variables. This corresponds to the Curry-Howard isomorphism, but now we add also existential quantification. Metavariables d, e, f range over derivation terms. Simultaneously with derivations, the set of free assumptions in a derivation (denoted $\mathrm{FA}(d)$) is defined. With d^A we denote that d derives the statement A, from the assumptions $\mathrm{FA}(d)$. Finally, u, v denote arbitrary assumption variables, x, y, z denote object variables and r, s, t range over object terms.

Definition 5.5.1 *Derivation terms are defined inductively, and simultaneously with the set of free assumption variables (*$\mathrm{FA}(d)$*) occurring in them.*

1	u^A	$\mathrm{FA}(u) = u$
2	$(\lambda u^A.d^B)^{A\to B}$	$\mathrm{FA}(\lambda u.d) = \mathrm{FA}(d) \setminus \{u\}$
3	$(d^{A\to B}e^A)^B$	$\mathrm{FA}(de) = \mathrm{FA}(d) \cup \mathrm{FA}(e)$
4	$\langle d^A, e^B\rangle^{A\wedge B}$	$\mathrm{FA}(\langle d, e\rangle) = \mathrm{FA}(d) \cup \mathrm{FA}(e)$
5	$\pi_0(d^{A\wedge B})^A$	$\mathrm{FA}(\pi_0(d)) = \mathrm{FA}(d)$
6	$\pi_1(d^{A\wedge B})^B$	$\mathrm{FA}(\pi_1(d)) = \mathrm{FA}(d)$
7	$(\lambda x.d^A)^{\forall x.A}$, *provided (1)*	$\mathrm{FA}(\lambda x.d) = \mathrm{FA}(d)$
8	$(d^{\forall x.A(x)}r)^{A(r)}$	$\mathrm{FA}(dr) = \mathrm{FA}(d)$
9	$\exists^+[r; d^{A(r)}]^{\exists x.A(x)}$	$\mathrm{FA}(\exists^+[r; d]) = \mathrm{FA}(d)$
10	$\exists^-[d^{\exists x.A(x)}; y; u^{A(y)}; e^B]^B$, *provided (2)*	$\mathrm{FA}(\exists^-[d; y; u; e]) = \mathrm{FA}(d) \cup (\mathrm{FA}(e) \setminus \{u\})$

The provisos are:

1. $x \notin \mathrm{FV}(B)$ *for any* $u^B \in \mathrm{FA}(d)$.

2. $y \notin \mathrm{FV}(B)$ *and* $y \notin \mathrm{FV}(C)$ *for any* $v^C \in \mathrm{FA}(e) \setminus \{u\}$.

Using the first clause, assumption variables can be introduced. Clause (2), (4), (7) and (9) are called *introduction principles*, because they introduce a connective. The remaining rules, (3), (5), (6), (8) and (10) are the *elimination principles*. The last construction, $\exists^-[d^{\exists x.A(x)}; y; u^{A(y)}; e^B]^B$ can be depicted more traditionally as follows:

$$\dfrac{\exists x.A(x) \qquad \begin{array}{c}[u : A(y)]\\ \vdots\ (e)\\ B\end{array}}{B}\,[u]$$

In $\lambda u.d$, the free occurrences of assumption u of d are bound. The same happens with the free assumption u of e in the derivation $\exists^-[d; y; u; e]$; note that in the latter derivation, the occurrences of u in d remain free. (This corresponds with the fact that in the picture above the assumption $A(y)$ is only discarded in the second subtree). Also object variables can be bound. This happens in $\lambda x.d$ (which binds the free x's of d) and in $\exists^-[d; y; u; e]$ (which binds the free y's of e). This gives rise to the notion $\mathrm{FV}(d)$, the free object variables in d, which can be defined inductively.

The following conversion rules are taken from [Pra71]. The first five are the *proper reductions*, the last five are called *permutative conversions*. These are necessary to

obtain normal forms with a nice structure (i.e. with the subformula property).

$$\begin{array}{rcl}
(\lambda u.d)e & \mapsto & d[u := e] \\
\pi_0\langle d, e\rangle & \mapsto & d \\
\pi_1\langle d, e\rangle & \mapsto & e \\
(\lambda x.d)r & \mapsto & d[x := r] \\
\exists^-[\exists^+[r; d]; x; u; e] & \mapsto & e[x, u := r, d] \\
 & & \\
(\exists^-[d; x; u; e])f & \mapsto & \exists^-[d; x; u; (ef)] \\
\pi_0(\exists^-[d; x; u; e]) & \mapsto & \exists^-[d; x; u; \pi_0(e)] \\
\pi_1(\exists^-[d; x; u; e]) & \mapsto & \exists^-[d; x; u; \pi_1(e)] \\
(\exists^-[d; x; u; e])r & \mapsto & \exists^-[d; x; u; (er)] \\
\exists^-[\exists^-[d; x; u; e]; y; v; f] & \mapsto & \exists^-[d; x; u; \exists^-[e; y; v; f]]
\end{array}$$

(renaming of bound variables may be needed to avoid unintended name conflicts.)

The proper reductions remove direct detours in a proof. The permutative conversions are needed, because also indirect detours exist. Such a hidden detour is made direct by a number of permutative conversions, and then eliminated by a proper reduction. There is a permutative conversion for every elimination principle. As an example, consider the following two-step reduction, which uses the first permutative conversion followed by the first proper reduction.

$$(\exists^-[d; y; u; \lambda v.e])f \;\to\; \exists^-[d; y; u; (\lambda v.e)f] \;\to\; \exists^-[d; y; u; e[v := f]] \;.$$

The first rewrite step is depicted below in the traditional notation. Note that the resulting proof contains an ordinary detour, which was hidden before applying the permutative conversion and could therefore not be eliminated immediately.

$$\dfrac{\exists x.A(x) \qquad \dfrac{\dfrac{[u:A(y)]\ [v:B]}{\vdots\,(e)\ C}}{B \to C}\,[v] \qquad \vdots\,(f)\ B}{B \to C}\,[u] \qquad B}{C}$$

$$\to$$

$$\dfrac{\exists x.A(x) \qquad \dfrac{\dfrac{[u:A(y)]\ [v:B]}{\vdots\,(e)\ C}}{B \to C}\,[v] \qquad B}{C}\,[u] \qquad \vdots\,(f)\ B}{C}$$

5.5.2 Encoding Natural Deduction into an HRS

In this section we will encode natural deduction, with proper and permutative conversions, into the HRS $\mathcal{H}_\exists$. The translation proceeds as follows: first the formulae are mapped onto types. Then derivation terms are translated into terms of a certain signature. Finally, the reduction rules are translated into HRS rules.

Transforming the formulae to types is done by the well known technique of removing the dependencies on object terms, also called *collapsing*. The collapse of A will be denoted by A^*. In the following definition, P is an n-ary predicate symbol.

$$\begin{array}{rcl} P(r_1,\ldots,r_n)^* & = & o \\ (A \to B)^* & = & A^* \to B^* \\ (A \wedge B)^* & = & A^* \times B^* \\ (\exists x.A)^* & = & o \times A^* \\ (\forall x.A)^* & = & o \to A^* \end{array}$$

Clearly, for any formula A we have that $A^* \in \mathbb{T}^\times(\{o\})$. The distinction between implication and universal quantification disappears. Existential quantifiers and conjunctions are translated into product types. The distinction between individuals and atomic formulae could still be made, but is not needed. Both kinds of entities are represented by one base type o.

The derivation terms are translated too. To this end we introduce a series of constant symbols for the derivation tree forming constructions. The signature of $\mathcal{H}_\exists$ contains for any $\sigma,\tau \in \mathbb{T}^\times(\{o\})$ the following constants. Most notable is the type of $\exists^-$.

$$\begin{array}{rcl} \mathsf{app}_{\sigma,\tau} & : & (\sigma \to \tau) \to \sigma \to \tau \\ \mathsf{abs}_{\sigma,\tau} & : & (\sigma \to \tau) \to \sigma \to \tau \\ \Pi_{\sigma,\tau} & : & \sigma \to \tau \to \sigma \times \tau \\ \Pi^0_{\sigma,\tau} & : & \sigma \times \tau \to \sigma \\ \Pi^1_{\sigma,\tau} & : & \sigma \times \tau \to \tau \\ \exists^+_\sigma & : & o \to \sigma \to o \times \sigma \\ \exists^-_{\sigma,\tau} & : & o \times \sigma \to (o \to \sigma \to \tau) \to \tau \end{array}$$

We deliberately excluded constants for $\forall$-introduction and -elimination, because universal quantification becomes the same as implication after collapsing the types. Every derivation tree d can be translated as a β-normal form d^* in this signature in the following inductive way (let $\sigma := A^*$ and $\tau := B^*$):

$$\begin{array}{rcl} (u^A)^* & = & u^\sigma \\ (\lambda u^A.d^B)^* & = & (\mathsf{abs}_{\sigma,\tau}\lambda u^\sigma.d^*) \\ (d^{A\to B}e)^* & = & (\mathsf{app}_{\sigma,\tau}d^*e^*) \\ \langle d^A, e^B\rangle^* & = & (\Pi_{\sigma,\tau}d^*e^*\rangle) \\ \pi_0(d^{A\wedge B})^* & = & (\Pi^0_{\sigma,\tau}d^*) \\ \pi_1(d^{A\wedge B})^* & = & (\Pi^1_{\sigma,\tau}d^*) \\ (\lambda x.d^A)^* & = & (\mathsf{abs}_{o,\sigma}\lambda x^o.d^*) \\ (d^{\forall x.A}r)^* & = & (\mathsf{app}_{o,\sigma}d^*r) \\ \exists^+[r;d^A]^* & = & (\exists^+_\sigma r d^*) \\ \exists^-[d;x;u^A;e^B]^* & = & (\exists^-_{\sigma,\tau}d^*(\lambda x^o.\lambda u^\sigma.e^*)) \end{array}$$

The rules of the HRS $\mathcal{H}_\exists$ consist of the following schemas, where $\rho, \sigma, \tau \in \mathbb{T}^\times(\{o\})$:

$$\begin{array}{rcl}
\mathsf{app}_{\sigma,\tau}(\mathsf{abs}_{\sigma,\tau}F)X & \mapsto & (FX) \\
\Pi^0_{\sigma,\tau}(\Pi_{\sigma,\tau}XY) & \mapsto & X \\
\Pi^1_{\sigma,\tau}(\Pi_{\sigma,\tau}XY) & \mapsto & Y \\
\exists^-_{\sigma,\tau}(\exists^+_\sigma(X,Y),F) & \mapsto & (FXY)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{app}_{\sigma,\tau}(\exists^-_{\rho,\sigma\to\tau}XF)Y & \mapsto & (\exists^-_{\rho,\tau}X\ \lambda x^o u^\rho.(\mathsf{app}_{\sigma,\tau}(Fxu)Y)) \\
\Pi^0_{\sigma,\tau}(\exists^-_{\rho,\sigma\times\tau}XF) & \mapsto & (\exists^-_{\rho,\sigma}X\ \lambda x^o u^\rho.(\Pi^0_{\sigma,\tau}(Fxu))) \\
\Pi^1_{\sigma,\tau}(\exists^-_{\rho,\sigma\times\tau}XF) & \mapsto & (\exists^-_{\rho,\tau}X\ \lambda x^o u^\rho.(\Pi^1_{\sigma,\tau}(Fxu))) \\
(\exists^-_{\sigma,\tau}(\exists^-_{\rho,o\times\sigma}XF)G) & \mapsto & (\exists^-_{\rho,\tau}X\ \lambda x^o u^\rho.(\exists^-_{\sigma,\tau}(Fxu)G))
\end{array}$$

Note that in the permutative conversions, Y and G come within the scope of the binders λxu. This leads to an implicit renaming, when x or u occur in Y or G.

It is not difficult to check that if $d \to e$ for derivation terms d and e, then also $d^* \to_{\mathcal{H}_\exists} e^*$. The first rule deals with proper reductions for $\to$ and $\forall$; the second and third with the proper $\wedge$-reductions and the fourth takes care of the proper $\exists$-reduction. The last four rules deal with the permutative conversions. Hence termination of $\mathcal{H}_\exists$ implies termination of $\to$ on derivation trees.

As an illustration, consider the proper $\to$-reduction step $(\lambda u.d)e \to d[u := e]$. The first derivation term translates to $(\mathsf{app}(\mathsf{abs}\,\lambda u.d^*)e^*)$. Now the first rule is applicable. Literal replacement yields $(\lambda u.d^*)e^*$, which has to be rewritten to β-normal form, due to the definition of a rewrite step. This normal form is $d^*[u := e^*]$, which is β-normal because d^* and e^* are, and e^* does not start with a λ. It is easy to prove that $d^*[u := e^*]$ equals $(d[u := e])^*$.

5.5.3 Termination of $\mathcal{H}_\exists$

We apply Theorem 5.1.4 to prove termination of $\mathcal{H}_\exists$. We have to provide a termination model, i.e. an ordered domain with strict interpretations for the constants, such that the rules are decreasing. As ordered domain we choose $(\mathbb{N}, >, 0, +)$. The interpretation of abstraction, application, pair building and projections is chosen in the more or less standard way:

$$\begin{array}{rcl}
\mathcal{J}(\mathsf{abs}_{\sigma,\tau})(f)(x) & = & f(x) \oplus_\tau \mathsf{S}_0^{(\sigma\to\tau)\to\sigma\to\tau}(f,x) \\
\mathcal{J}(\mathsf{app}_{\sigma,\tau})(f)(x) & = & f(x) \oplus_\tau \mathsf{S}_0^{(\sigma\to\tau)\to\sigma\to\tau}(f,x) \oplus \underline{1}_\tau \\
\mathcal{J}(\Pi_{\sigma,\tau})(x)(y) & = & (x,y) \oplus_{\sigma\times\tau} \mathsf{S}_0^{\sigma\to\tau\to\sigma\times\tau}(x)(y) \\
\mathcal{J}(\Pi^0_{\sigma,\tau})(x,y) & = & x \oplus_\sigma \mathsf{S}_0^{\sigma\times\tau\to\sigma}(x,y) \oplus \underline{1}_\sigma \\
\mathcal{J}(\Pi^1_{\sigma,\tau})(x,y) & = & y \oplus_\tau \mathsf{S}_0^{\sigma\times\tau\to\tau}(x,y) \oplus \underline{1}_\tau
\end{array}$$

These interpretations are built as "weakly monotonic" plus "strict", so they are strict by Lemma 4.3.7. The elimination constants have an additional $\oplus\underline{1}$ to make the first three proper reduction rules decreasing. This is not proved here in detail, because it is completely analogous to the situation in Section 5.2.2 and 5.4.

The same trick for $\mathcal{J}(\exists^+)$ and $\mathcal{J}(\exists^-)$ will not work, due to the presence of the permutative conversions. We put

$$\mathcal{J}(\exists^+_\sigma)(x)(y) \quad = \quad (x,y) \oplus_{o\times\sigma} \mathsf{S}_0^{o\to\sigma\to o\times\sigma}(x)(y) \ .$$

This is indeed strict, and $\mathcal{J}(\exists^+)(x)(y) \ {}_{\mathrm{wm}}\!\geq (x,y)$. For $\mathcal{J}(\exists^-)$ we have to find something more complex.

An interpretation for $\exists^-$. To see how to define $\mathcal{J}(\exists^-)$, let us first concentrate on the permutative conversions for eliminating implication (and universal quantification) and conjunction. Note that in these cases, the complexity of the type of the $\exists^-$-symbol strictly decreases. This can be used as follows. We let $\exists^-_{\rho,\sigma\to\tau}$ "pay the price" of the $\exists^-_{\rho,\tau}$ that pops up on the right hand side of the corresponding permutative conversion rule. Similarly, the interpretation of $\exists^-_{\rho,\sigma\times\tau}$ has to take into account both $\exists^-_{\rho,\sigma}$ and $\exists^-_{\rho,\tau}$. Because the types decrease, we can capture this idea in an inductive definition. We now define $\mathsf{A}_\sigma : \mathcal{T}_\sigma \Rightarrow \mathcal{T}_\sigma$, which calculates the "price" as indicated above.

$$\begin{array}{rcl} \mathsf{A}_o(n) & := & n+1, \\ \mathsf{A}_{\sigma\to\tau}(f)(x) & := & \mathsf{A}_\tau(\mathcal{J}(\mathsf{app})(f)(x)) \\ \mathsf{A}_{\sigma\times\tau}(x) & := & (\mathsf{A}_\sigma(\mathcal{J}(\Pi_0)(x)), \mathsf{A}_\tau(\mathcal{J}(\Pi_1)(x))) \end{array}$$

We need the following properties of A:

1. For all $\sigma \in \mathbb{T}^\times$, A_σ is strict.
2. For all $\sigma \in \mathbb{T}^\times$ and $x \in \mathcal{WM}_\sigma$, $\mathsf{A}_\sigma(x) \ {}_{\mathrm{wm}}\!> x$

Both can be proved straightforwardly by induction on σ. The first statement uses that $\mathcal{J}(\mathsf{app})$, $\mathcal{J}(\Pi_0)$ and $\mathcal{J}(\Pi_1)$ are strict. The second statement holds mainly because we added $+1$ in the clause defining A_o.

We will write $\mathsf{A}^n_\sigma(x)$ for the n-fold application of A_σ to x. We are now able to define the interpretation of $\exists^-$.

$$\mathcal{J}(\exists^-_{\sigma,\tau})(d)(e) = \mathsf{A}_\tau^{2^{\mathsf{M}(d)}}(e(\pi_0 d)(\mathsf{S}_0^\sigma \oplus \pi_1 d)).$$

Let us first explain the intuition behind this interpretation. Due to the proper reduction rule for $\exists^-$, we need at least $e(\pi_0 d)(\pi_1 d)$. The $\mathsf{S}_0\oplus$-part is added to achieve strictness in e.

In the permutative conversions, the second argument of the $\exists^-$-symbols grows. This has to be compensated somehow. As mentioned before, in the first three permutative conversions the type of $\exists^-$ goes down, which is used in the inductive definition of A. Here the growing second argument is compensated by the decreasing types.

The most problematic rule is the permutative conversion for $\exists^-$. Here the type doesn't go down. The only thing which goes down is the first argument of the $\exists^-$-symbols involved. So the value of $\exists^-$ has to weigh its first argument rather high, to compensate for the increasing second argument. This explains the $2^{\mathsf{M}(d)}$ in the

previous definition. The exact reason for the power of 2 will become clear in the proof that the last rule is decreasing.

We will now verify that the provided definition works, i.e. that $(\mathbb{N}, >, 0, +, \mathcal{J})$ is indeed a termination model. To this end we still have to prove that $\mathcal{J}(\exists^-)$ is strict and that the rules containing $\exists^-$ are decreasing.

Strictness of $\mathcal{J}(\exists^-)$. We now show that for any $\rho, \sigma \in \mathbb{T}^\times$, $\mathcal{J}(\exists^-_{\rho,\sigma})$ is strict. Weak monotonicity of $\exists^-$ immediately follows from weak monotonicity of A. Next strictness is proved. Let e, f, x, y be weakly monotonic, with $e, f : (o \to \rho \to \sigma) \to \sigma$ and $x, y : o \times \rho$. We will prove strictness in the first, the second, and the next arguments, respectively.

- If $x \,{}_{\mathrm{st}}{>}\, y$, then $x \,{}_{\mathrm{wm}}{\geq}\, y$ and by monotonicity of e, $e(\pi_0 x)(\mathsf{S}_0 \oplus \pi_1 x) \,{}_{\mathrm{wm}}{\geq}\, e(\pi_0(y)(\mathsf{S}_0 \oplus \pi_1 y))$. Furthermore, $2^{\mathsf{M}(x)} > 2^{\mathsf{M}(y)}$. Because for all weakly monotonic z, $\mathsf{A}(z) \,{}_{\mathrm{wm}}{>}\, z$, it follows that $\mathcal{J}(\exists^-)(x)(e) \,{}_{\mathrm{wm}}{>}\, \mathcal{J}(\exists^-)(y)(e)$. This proves strictness in the first argument.

- Next, assume that $e \,{}_{\mathrm{st}}{>}\, f$. Note that both $\pi_0 x$ and $\mathsf{S} \oplus \pi_1 x$ are strict (the first is of base type, the second is strict by Lemma 4.3.7). Hence $e(\pi_0 x)(\mathsf{S}_0 \oplus \pi_1 x) \,{}_{\mathrm{st}}{>}\, f(\pi_0 x)(\mathsf{S}_0 \oplus \pi_1 x)$. Now $\mathcal{J}(\exists^-)(x)(e) \,{}_{\mathrm{wm}}{>}\, \mathcal{J}(\exists^-)(x)(f)$ follows from the strictness of A. This proves strictness in the second argument.

- Strictness in the next arguments directly follows from strictness of A.

Thus $\mathcal{J}(\exists^-)$ is strict in all its arguments. ⊠

Decreasingness of the rules. At this point, it remains to show that the last proper reduction rule and the permutative conversion rules are decreasing. We start with the proper one.

Let α be a weakly monotonic valuation. We write $[\![M]\!]$ for $[\![M]\!]_\alpha$. Note that for all weakly monotonic y and z, we have $\mathsf{A}(z) \,{}_{\mathrm{wm}}{>}\, z$, $\mathsf{S} \oplus z \,{}_{\mathrm{wm}}{\geq}\, z$ and $\mathcal{J}(\exists^+)(y)(z) \,{}_{\mathrm{wm}}{\geq}\, (y, z)$. Hence we have:

$$\begin{array}{lcl}
[\![\exists^-(\exists^+ XY)F]\!] & = & \mathsf{A}^{2^{\mathsf{M}([\![\exists^+ XY]\!])}}([\![F]\!](\pi_0[\![\exists^+ XY]\!])(\mathsf{S}_0 \oplus \pi_1[\![\exists^+ XY]\!])) \\
& \,{}_{\mathrm{wm}}{>}\, & [\![F]\!](\pi_0[\![\exists^+ XY]\!])(\mathsf{S}_0 \oplus \pi_1[\![\exists^+ XY]\!]) \\
& \,{}_{\mathrm{wm}}{\geq}\, & [\![F]\!](\pi_0([\![X]\!], [\![Y]\!]))(\pi_1([\![X]\!], [\![Y]\!])) \\
& = & [\![FXY]\!] \ .
\end{array}$$

We proceed with the first permutative conversion. Let α again be an arbitrary monotonic valuation, and $[\![M]\!]$ the value of M under α. We first introduce as abbreviations $P := [\![\exists^- XF]\!]$ and $Q := [\![F]\!](\pi_0[\![X]\!])(\mathsf{S}_0 \oplus \pi_1[\![X]\!])$. Then we compute the left- and the right hand side of the rule:

$$\begin{array}{lcl}
[\![\mathsf{app}(\exists^-_{\rho,\sigma\to\tau} XF)Y]\!] & = & P([\![Y]\!]) \oplus \mathsf{S}_0(P)([\![Y]\!]) \oplus \underline{1} \\
& \,{}_{\mathrm{wm}}{>}\, & P([\![Y]\!]) \\
& = & \mathsf{A}^{2^{\mathsf{M}([\![X]\!])}}_{\sigma\to\tau}(Q)([\![Y]\!])
\end{array}$$

$$[\![\exists^-_{\rho,\tau} X\, \lambda xu.\mathsf{app}(Fxu)Y]\!] \quad = \quad \mathsf{A}_\tau^{2^{\mathsf{M}([\![X]\!])}}([\![\mathsf{app}]\!](Q)([\![Y]\!]))$$

Hence it suffices to show that for any n, $\mathsf{A}^{n+1}_{\sigma\to\tau}(Q)([\![Y]\!]) \,{}_{\mathrm{wm}}{\geq}\, \mathsf{A}^{n+1}_\tau([\![\mathsf{app}]\!](Q)([\![Y]\!]))$. This is proved by induction on n. If $n = 0$, then both sides are equal, by definition of $\mathsf{A}_{\sigma\to\tau}$. In the successor step, we use that $\mathsf{app}(x) \,{}_{\mathrm{wm}}{>}\, x$ for weakly monotonic x, so we have

$$\begin{array}{lcll}
\mathsf{A}^{n+2}_{\sigma\to\tau}(Q)[\![Y]\!] & = & \mathsf{A}_{\sigma\to\tau}(\mathsf{A}^{n+1}_{\sigma\to\tau}(Q))[\![Y]\!] & \\
 & = & \mathsf{A}_\tau([\![\mathsf{app}]\!](\mathsf{A}^{n+1}_{\sigma\to\tau}(Q))[\![Y]\!]) & \text{by definition of } \mathsf{A}_{\sigma\to\tau} \\
 & {}_{\mathrm{wm}}{>} & \mathsf{A}_\tau(\mathsf{A}^{n+1}_{\sigma\to\tau}(Q)[\![Y]\!]) & \\
 & {}_{\mathrm{wm}}{\geq} & \mathsf{A}_\tau(\mathsf{A}^{n+1}_\tau([\![\mathsf{app}]\!](Q)[\![Y]\!])) & \text{by induction hypothesis} \\
 & = & \mathsf{A}^{n+2}_\tau([\![\mathsf{app}]\!](Q)[\![Y]\!]). &
\end{array}$$

Hence the first permutative conversion is decreasing. The proof that the second and third permutative conversion rules are decreasing is very similar; again the definition of A carries the burden of the proof. We omit the details.

We finally show that also the last permutative conversion is decreasing. Again, let α be a weakly monotonic valuation. $[\![M]\!]$ denotes the meaning of M under α. We put as abbreviations $P := [\![\exists^-_{\rho,o\times\sigma} XF]\!]$ and $Q := [\![F]\!](\pi_0[\![X]\!])(\mathsf{S}_0 \oplus \pi_1[\![X]\!])$. Before doing the main computation, we need some little facts:

1. $\mathsf{M}(P) > \mathsf{M}(Q)$.
2. $\mathsf{M}(P) > \mathsf{M}([\![X]\!]) + 1$.

Ad 1. This follows from $P = \mathsf{A}^{2^{\mathsf{M}([\![X]\!])}}(Q) \,{}_{\mathrm{wm}}{\geq}\, \mathsf{A}(Q) \,{}_{\mathrm{wm}}{>}\, Q$.

Ad 2. Note that for weakly monotonic x, $\mathsf{A}(x) \,{}_{\mathrm{wm}}{>}\, x$ and $\mathcal{J}(\Pi^0)(x) \,{}_{\mathrm{wm}}{>}\, \pi_0(x)$. Hence we have

$$\begin{array}{lcl}
\mathsf{M}(P) & = & \mathsf{M}_o(\pi_0 P) + \mathsf{M}(\pi_1 P) \\
 & \geq & \pi_0 P \\
 & = & \pi_0(\mathsf{A}^{2^{\mathsf{M}([\![X]\!])}}_{o\times\sigma}(Q)) \\
 & \geq & \mathsf{A}^{2^{\mathsf{M}([\![X]\!])}}_o(\mathcal{J}(\Pi^0)(Q)) \\
 & > & \mathsf{A}^{2^{\mathsf{M}([\![X]\!])}}_o(\pi_0(Q)) \\
 & \geq & 2^{\mathsf{M}([\![X]\!])} \\
 & \geq & \mathsf{M}([\![X]\!]) + 1
\end{array}$$

From the two inequalities above, we obtain $2^{\mathsf{M}(P)} > 2^{\mathsf{M}(Q)} + 2^{\mathsf{M}([\![X]\!])}$. At this point we really need the powers of 2.

Now we compute for the last rule:

$$\begin{array}{lcl}
[\![\exists^-_{\sigma,\tau}(\exists^-_{\rho,o\times\sigma} XF)G)]\!] & = & [\![\exists^-]\!](P)(G) \\
 & = & \mathsf{A}^{2^{\mathsf{M}(P)}}(G(\pi_0 P)(\mathsf{S}_0 \oplus \pi_1 P)) \\
 & {}_{\mathrm{wm}}{>} & \mathsf{A}^{2^{\mathsf{M}([\![X]\!])}}(\mathsf{A}^{2^{\mathsf{M}(Q)}}(G(\pi_0 P)(\mathsf{S}_0 \oplus \pi_1 P))) \\
 & {}_{\mathrm{wm}}{\geq} & \mathsf{A}^{2^{\mathsf{M}([\![X]\!])}}(\mathsf{A}^{2^{\mathsf{M}(Q)}}(G(\pi_0 Q)(\mathsf{S}_0 \oplus \pi_1 Q))) \\
 & = & \mathsf{A}^{2^{\mathsf{M}([\![X]\!])}}([\![\exists^-]\!](Q)(G)) \\
 & = & \mathsf{A}^{2^{\mathsf{M}([\![X]\!])}}([\![\exists^-]\!]([\![F]\!](\pi_0[\![X]\!])(\mathsf{S}_0 \oplus \pi_1[\![X]\!]))(G)) \\
 & = & [\![\exists^-_{\rho,\tau} X\, \lambda x^o u^\rho.(\exists^-_{\sigma,\tau}(Fxu)G)]\!]
\end{array}$$

We conclude that all rules are decreasing, so we have found a termination model. By Theorem 5.1.4, $\mathcal{H}_\exists$ is terminating.

5.6 Incompleteness and Possible Extensions

Contrary to Theorem 3.1.3, we have Theorem 5.1.4 only in one direction. The other direction really fails, as illustrated by the following example, which is a simplification of the one occurring in [Kah96]. Hence our proof method is not complete.

EXAMPLE. Consider the HRS $\mathcal{H}$ that has one base type, o, and one function symbol, $c : o \to o$ and the following rule only, where F and X are variables:

$$c(F(FX)) \mapsto (FX)$$

This system terminates, because with any rewrite step the number of c-symbols decreases. This can be seen by considering an arbitrary θ-instance of the rule. Assume w.l.o.g. that $F^\theta = \lambda x.s$ and $X^\theta = t$ with s and t in $\beta\overline{\eta}$-normal form. Let j be the number of occurrences of x in s (possibly 0), k the number of occurrences of c in s and let ℓ be the number of occurrences of c in t. Then $c(F(FX))^\theta \downarrow_{\beta\overline{\eta}}$ contains $1 + k + j \cdot (k + j \cdot \ell)$ occurrences of c, and $(FX)^\theta \downarrow_{\beta\overline{\eta}}$ only $k + j \cdot \ell$.

Assume, towards a contradiction, that $(A, >, 0, +, C)$ is a termination model for $\mathcal{H}$. Note that by assigning $\boldsymbol{\lambda} y \in A.y$ to F and 0 to X (both weakly monotonic), we get by decreasingness of the rewrite rule, $C(0) > 0$. Applying strictness of C we get $C(C(0)) > C(0)$. Now we define a weakly monotonic function G by putting

$$G := \boldsymbol{\lambda} x \in A.\textbf{if } x < C(C(0)) \textbf{ then } 0 \textbf{ else } C(0) \textbf{ fi}$$

G is weakly monotonic (use that $0 < C(0) < C(C(0))$ holds). G is chosen in such a way that $G(C(C(0))) = C(0)$ and $G(C(0)) = 0$. Now define the weakly monotonic valuation α such that $\alpha(F) := G$ and $\alpha(X) := C(C(0)$. By decreasingness we get:

$$C(0) = C(G(G(C(C(0))))) = [\![c(F(F(X)))]\!]_\alpha > [\![FX]\!]_\alpha = G(C(C(0))) = C(0),$$

which is impossible by irreflexivity of $>$. Hence a termination model doesn't exist. ⊠

Discussion. The reason that the previous example could not be handled by Theorem 5.1.4 is that there exist too many weakly monotonic functionals, like G. Hence we have to verify decreasingness for valuations involving G, although G cannot be built from lambda terms. It seems that we can change our setup quite easily, by restricting the class of weakly monotonic functions as follows: A function is modified weakly monotonic if in any argument, it either completely discards that argument, or it strictly preserves the order in that argument. On number theoretic functions, f is weakly monotonic if it is non-decreasing, while f is only modified weakly monotonic if f is either constant, or increasing.

It is possible to prove that any lambda term denotes a modified weakly monotonic functional. Moreover the whole theory of Chapter 4 seems to remain valid if

we replace "weakly monotonic" by "modified weakly monotonic" everywhere. The only difference is that the function $Pred_\sigma$ is not modified weakly monotonic, contradicting the modified version of Lemma 4.4.3.(3). This lemma is used in the proof of Proposition 4.4.4. But this proposition is not used in the correctness proof of the method.

Regarding the example above, we now have a modified termination model $(\mathbb{N}, >, 0, +, Succ)$. For any modified weakly monotonic $f \in \mathbb{N} \Rightarrow \mathbb{N}$, we have $f(f(x)) \geq f(x)$, so we get $Succ(f(f(x)) > f(f(x) \geq f(x)$, so the rewrite rule is indeed decreasing in this model.

Note that we do not have $f(x) \geq x$ for modified weakly monotonic f. This is fortunate, in view of the *non*-terminating rewrite system $c(FX) \mapsto X$. Choosing $\lambda x.0$ for F and $c(0)$ for X yields the rewrite step $c(0) \mapsto c(0)$.

It is not clear whether the modified termination method is stronger than the method we presented. We only have that it is not weaker. Moreover, it is not clear at all that the modified version is a complete termination method. Therefore we do not systematically change to the modified version.

Extensions of the method. The method for proving termination can be extended in various directions. First, the underlying theory can be extended to other type disciplines. Secondly, we can try more complex examples.

We already noted that coproduct types can be treated. In [Gan80, Kah95] this is worked out in the setting of the hereditarily monotonic functionals. We don't see a problem in adapting that work to the setting of our strict functionals, so this should be a routine extension. We note that Loader uses an adaptation of the semantical proof method in order to prove strong normalization of System F [Loa95]. We have not investigated whether this approach can be generalized to arbitrary higher-order rewrite systems with polymorphic types. (Nor is it clear whether such systems would be useful).

As to other examples, it seems possible to treat the extension of Gödel's T to countably branching trees (known as Kleene's O or as Zucker's T_1 trees). These trees have constants 0^T, $Succ^{T \to T}$ and $Lim^{(o \to T) \to T}$ (the limit tree). Recursion over these trees is expressed in three recursion rules,

$$\begin{array}{rcl} RFGH0 & \mapsto & F \\ RFGH(SuccX) & \mapsto & G(RFGHX) \\ RFGH(LimX) & \mapsto & H(\lambda z^o.(RFGH(Xz))) \end{array}$$

We have not worked out the details for this system. Another challenge would be to give a semantical termination proof for the system of Bar Recursion (see [Bez86] for a termination proof using compact sets of terms).

Chapter 6

Computability versus Functionals of Finite Type

In this chapter, we compare the semantic termination proofs, with the traditional strong-normalization proofs, that use strong computability predicates. This chapter is a full version of the paper that appeared as [Pol96].

The computability method is often attributed to Tait [Tai67], who used convertibility predicates to prove a normal form theorem for various systems. Troelstra [Tro73] uses similar predicates (now called strong computability) in strong normalization proofs. Prawitz [Pra71] used a variant, to deal with permutative conversions, arising from natural deduction for first order predicate logic (see the Appendix). Girard [Gir72] introduced a stronger variant, to deal with the impredicative system F. For the moment we are interested in simply-typed lambda calculus and Gödel's T, a system with higher-order primitive recursion; therefore we can stick to Troelstra's variation on Tait's predicates.

We will compare this with the method to prove strong normalization by using functionals of finite type, invented by Gandy [Gan80] and discussed in Section 3.3. In this method, to each typed term a functional of the same type is associated. This functional is measured by a natural number. In order to achieve that a rewrite step gives rise to a decrease of the associated number, the notion *hereditarily monotonic functional* was developed. The number is an upper bound for the length of reduction sequences starting from a certain term. De Vrijer [Vrij87] used a variant to compute the exact length of the longest reduction sequence.

Gandy deals with simply-typed lambda calculus, Gödel's T and with β-reductions in proof theory including disjunction and existential quantification. However, the permutative conversions for these connectives could not be dealt with. In Chapter 5 of this thesis we showed how to generalize the semantical method to higher-order rewrite systems [Pol94] and in Section 5.5, how to prove termination of the permutative conversions with the extended theory [PS95].

In the literature, these two methods are often put in contrast (e.g. [Gan80, § 6.3]

and [GLT89, § 4.4]). Using functionals seems to be more transparent and economizes on proof theoretical complexity; strong computability should generalize to more complex systems. On the other hand, seeing the two proofs one gets the feeling that "somehow, the same thing is going on". Indeed De Vrijer [Vrij87, § 0.1] remarks that a proof using strong computability can be seen as abstracting from concrete information in the functionals that is not strictly needed in a termination proof, but which provides for an estimate of reduction lengths.

In this chapter we will substantiate this feeling. First, the proof à la Tait will be *decorated* with concrete numbers. This is done by introducing binary predicates $\mathrm{SN}(M,k)$, which mean that the term M may perform at most k reduction steps. A formal, constructive proof of $\exists k.\mathrm{SN}(M,k)$ is given for any M. From this proof, we extract a program, via the *modified realizability interpretation*. Remarkably, this program equals (more or less) the functional $[\![\mathsf{L}M^*]\!]_{\alpha_\iota}$, assigned to M in the proof à la Gandy.

The idea of using a realizability interpretation to extract functionals from Tait's SN-proof already occurs in [Ber93]. In that paper, a program to compute the *normal form* of a term is extracted. Our contribution is, that by extracting numerical upper bounds for the length of reduction sequences, a comparison with Gandy's proof can be made. Furthermore, we also deal with Gödel's T, which yields a sharper upper bound than provided by Gandy's proof.

The chapter is organized as follows. In Section 6.1, we decorate Tait's SN-proof for simply-typed lambda calculus. Modified realizability is introduced in Section 6.2. In Section 6.3 the proofs of Section 6.1 are formalized; also the program extraction is carried out there. In Section 6.3.3, the extracted functionals are compared with those used by Gandy. The same project is carried out for Gödel's T in Section 6.4. Other possible extensions are considered in Section 6.5.

Notation. The notation is as in Section 2.4. Recall that $\vec{\sigma} \to \tau$ means $\sigma_1 \to (\sigma_2 \to \cdots \to (\sigma_n \to \tau))$ and $M\vec{N}$ means $(((MN_1)N_2)\cdots N_n)$ and $\lambda\vec{x}.N$ means $(\lambda x_1.(\lambda x_2.\cdots(\lambda x_n.N)))$. We also use the following (less standard) notation for sequences of variables, terms and types:

- $\vec{x}^{\vec{\sigma}} := x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$
- $\vec{\sigma} \to \vec{\tau} := \vec{\sigma} \to \tau_1, \ldots, \vec{\sigma} \to \tau_n$
- $\vec{M}\vec{N} := M_1\vec{N}, \ldots, M_n\vec{N}$
- $\lambda\vec{x}.\vec{M} := \lambda\vec{x}.M_1, \ldots, \lambda\vec{x}.M_n$

Note that by this convention $\vec{\sigma} \to \epsilon = \epsilon$, the empty sequence and $\epsilon \to \sigma = \sigma$. In particular, $\epsilon \to \epsilon = \epsilon$.

6.1 Strong Computability for Termination of $\lambda_\beta^\to$

In this section, we present a well known proof that every reduction sequence of β-steps is finite. In fact we prove something more, because we prove that for any term M, there is an upper bound k such that $\mathrm{SN}(M,k)$ holds.

Tait's method to prove strong normalization starts with defining a "strong computability" predicate which is stronger than "strong normalizability". The proof consists of two parts: One part stating that strongly computable terms are strongly normalizing, and one part stating that any term is strongly computable. The first is proved with induction on the types (simultaneously with the statement that every variable is strongly computable). The second part is proved with induction on the term structure (in fact a slightly stronger statement is proved). We will present a version of this proof that contains information about reduction lengths.

Definition 6.1.1 *The set of strongly computable terms is defined inductively as follows:*

- $\mathrm{SC}_\iota(M)$ *iff there exists a* k *such that* $\mathrm{SN}(M,k)$.
- $\mathrm{SC}_{\sigma\to\tau}(M)$ *iff for all* N *with* $\mathrm{SC}_\sigma(N)$, *we have* $\mathrm{SC}_\tau(MN)$.

Lemma 6.1.2 (SC Lemma)

(a) For all terms M, *if* $\mathrm{SC}(M)$ *then there exists a* k *with* $\mathrm{SN}(M,k)$.

(b) For all terms M *of the form* $x\vec{M}$, *if there exists a* k *with* $\mathrm{SN}(M,k)$, *then* $\mathrm{SC}(M)$.

In (b), $\vec{M}$ may be the empty sequence.

Proof: (Simultaneous induction on the type of M)

(a) Assume $\mathrm{SC}(M)$.

If M is of base type, then $\mathrm{SC}(M)$ just means that there exists a k with $\mathrm{SN}(M,k)$.

If M is of type $\sigma\to\tau$, we take a variable x^σ, which is of the form $x\vec{M}$. Note that x is in normal form, hence $\mathrm{SN}(x,0)$ holds. By IH(b), $\mathrm{SC}(x)$; and by the definition of $\mathrm{SC}(M)$, we obtain $\mathrm{SC}(Mx)$. By IH(a) we have that there exists a k such that $\mathrm{SN}(Mx,k)$. We can take the same k as a bound for M, because any reduction sequence from M gives rise to a sequence from Mx of the same length. Hence $\mathrm{SN}(M,k)$ holds.

(b) Assume that $M\equiv x\vec{M}$ and $\mathrm{SN}(M,k)$ for some k.

If M is of base type, then the previous assumption forms exactly the definition of $\mathrm{SC}(M)$.

If M has type $\sigma\to\tau$, assume $\mathrm{SC}(N)$ for arbitrary N^σ. By IH(a), $\mathrm{SN}(N,\ell)$ for some ℓ. Because reductions in $x\vec{M}N$ can only take place inside $\vec{M}$ or N, we have $\mathrm{SN}(x\vec{M}N,k+\ell)$. IH(b) yields that $\mathrm{SC}(x\vec{M}N)$. This proves $\mathrm{SC}(M)$.

⊠

Lemma 6.1.3 (Abstraction Lemma) *For all terms M, N and $\vec{P}$ and variables x, it holds that if* $\mathrm{SC}(M[x := N]\vec{P})$ *and* $\mathrm{SC}(N)$*, then* $\mathrm{SC}((\lambda x.M)N\vec{P})$*.*

Proof: (Induction on the type of $M\vec{P}$.) Let M, x, N and $\vec{P}$ be given, with $\mathrm{SC}(M[x := N]\vec{P})$ and $\mathrm{SC}(N)$. Let σ be the type of $M\vec{P}$.

If $\sigma = \iota$, then by definition of SC, we have an ℓ such that $\mathrm{SN}(M[x := N]\vec{P}, \ell)$. By Lemma 6.1.2(a) we obtain the existence of k, such that $\mathrm{SN}(N, k)$. We have to show, that there exists a p with $\mathrm{SN}((\lambda x.M)N\vec{P}, p)$. We show that we can put $p := k + \ell + 1$.

Consider an arbitrary reduction sequence of $(\lambda x.M)N\vec{P}$. Without loss of generality, we assume that it consists of first a steps in M (yielding M'), b steps in $\vec{P}$ (yielding $\vec{P}\,'$) and c steps in N (yielding N'). After this the outermost redex is contracted, yielding $M'[x := N']\vec{P}\,'$, and finally d steps occur. Clearly, $c \leq k$. Notice that we also have a reduction sequence $M[x := N]\vec{P} \twoheadrightarrow_\beta M'[x := N']\vec{P}\,'$ of at least $a + b$ steps (we cannot count reductions in N, because we do not know whether x occurs free in M). So surely, $a + b + d \leq \ell$. Summing this up, we have that any reduction sequence from $(\lambda x.M)N\vec{P}$ has length at most $k + \ell + 1$.

Let $\sigma = \rho \to \tau$. Assume $\mathrm{SC}(P)$, for arbitrary P^ρ. Then by definition of $\mathrm{SC}(M[x := N]\vec{P})$, we have $\mathrm{SC}_\tau(M[x := N]\vec{P}P)$, and by IH $\mathrm{SC}((\lambda x.M)N\vec{P}P)$. This proves $\mathrm{SC}((\lambda x.M)N\vec{P})$. ⊠

Lemma 6.1.4 (Main Lemma) *For all terms M and substitutions θ, if* $\mathrm{SC}(x^\theta)$ *for all free variables x of M, then* $\mathrm{SC}(M^\theta)$*.*

Proof: (Induction on the structure of M.) Let M and θ be given, such that $\mathrm{SC}(x^\theta)$ for all $x \in \mathrm{FV}(M)$.

If $M \equiv x$, then the last assumption yields $\mathrm{SC}(M^\theta)$.

If $M \equiv NP$, we have $\mathrm{SC}(N^\theta)$ and $\mathrm{SC}(P^\theta)$ by IH for N and P. Then by definition of $\mathrm{SC}(N^\theta)$, we have $\mathrm{SC}(N^\theta P^\theta)$, hence by equality of $N^\theta P^\theta$ and $(NP)^\theta$, $\mathrm{SC}(M^\theta)$ follows.

If $M \equiv \lambda x.N$, assume that $\mathrm{SC}(P)$ for an arbitrary P. By IH for N, applied on the substitution $\theta[x := P]$, we see that $\mathrm{SC}(N^{\theta[x:=P]})$, hence by equality $\mathrm{SC}((N^\theta)[x := P])$. Now we can apply Lemma 6.1.3, which yields that $\mathrm{SC}((\lambda x.N^\theta)P)$. Again by using equality, we see that $\mathrm{SC}((\lambda x.N)^\theta P)$ holds. This proves $\mathrm{SC}(M^\theta)$. (Note that implicitly renaming of bound variables is required.) ⊠

Theorem 6.1.5 *For any term M there exists a k, such that* $\mathrm{SN}(M, k)$*.*

Proof: Let θ be the identity substitution, with as domain the free variables of M. By Lemma 6.1.2(b), $\mathrm{SC}(x)$ is guaranteed. Now we can apply Lemma 6.1.4, yielding $\mathrm{SC}(M^\theta)$. Because $M^\theta \equiv M$, we obtain $\mathrm{SC}(M)$. Lemma 6.1.2(a) yields the existence of a k with $\mathrm{SN}(M, k)$. ⊠

6.2 A Refinement of Realizability

As mentioned before, we want to extract the computational content from the SN-proof of Section 6.1. To this end we use *modified realizability*, introduced by Kreisel [Kre59]. In [Tro73, § 3.4] modified realizability is presented as a translation of HA^ω into itself. This interpretation eliminates existential quantifiers, at the cost of introducing functions of finite type (functionals), represented by λ-terms.

Following Berger [Ber93], we present modified realizability as an interpretation of a first order fragment (MF) into a higher-order, negative (i.e. $\exists$-free) fragment (NH). We will also take over a refinement by Berger, which treats specific parts of a proof as *computationally irrelevant*.

6.2.1 The Modified Realizability Interpretation

A formula can be seen as the specification of a program. E.g. $\forall x.\exists y.P(x,y)$ specifies a program f of type $o{\to}o$, such that $\forall x.P(x,f(x))$ holds. In general a sequence of programs is specified.

A refinement by Berger enables to express that existentially quantified variables are independent of certain universal variables, by underlining the universal ones. In $\underline{\forall x}.\exists y.P(x,y)$ the underlining means that y is not allowed to depend on x. So a number m is specified, with $\forall x.P(x,m)$. This could of course also be specified by the formula $\exists y.\forall x.P(x,y)$, but in specifications of the form $\underline{\forall x}.P(x) \to \exists y.Q(x,y)$ the underlining cannot be eliminated that easily. This formula specifies a number m, such that $\forall x.P(x) \to Q(x,m)$ holds. The $\underline{\forall x}$ cannot be pushed to the right, nor can the $\exists y$ be pulled to the left, without changing the intuitionistic meaning.

Specifications are expressed in minimal many-sorted first-order logic (MF). This logic is based upon a many-sorted first-order signature. Terms over such a signature are defined as usual ($r, s, t, \ldots$ denote arbitrary terms). The formulae of MF are either atomic ($P\vec{t}$), or of the form $\varphi \to \psi$, $\forall x^\iota.\varphi$, $\underline{\forall x^\iota}.\varphi$ or $\exists x^\iota.\varphi$. Here $\varphi, \psi, \ldots$ denote arbitrary MF formulae. This logic is **M**inimal, because negation is not included, and it quantifies over **F**irst-order objects only.

As programming language, we use the simply-typed lambda calculus. Because programs are higher-order objects, MF cannot talk about them. To express correctness of programs, we introduce **N**egative **H**igher-order logic (NH). The terms of NH are simply typed λ-terms *considered modulo* $\beta\eta$, with the MF sorts as base types, MF function symbols as constants and with the MF predicate symbols. We let $r, s, t, \ldots$ range over equivalence classes of $\beta\eta$-equal terms. The formulae are atomic ($P\vec{s}$), or composed from $\varphi \to \psi$ or $\forall x^\rho.\varphi$. Here $\varphi, \psi, \ldots$ denote arbitrary NH formulae. **N**egative means that there are no existential quantifiers, and **H**igher-order refers to the objects.

Below we define $\tau(\varphi)$, the sequence of types of the programs specified by the MF formula φ. This operation is known as "forgetting dependencies" (of types on terms). Furthermore, if $\vec{s}$ is a sequence of programs of type $\tau(\varphi)$, we define an NH formula $\vec{s}\ \mathbf{mr}\ \varphi$ (**m**odified **r**ealizes). This NH formula expresses correctness of $\vec{s}$ with

respect to the specification φ. Both notions are defined with recursion on the logical complexity of the formula φ.

Definition 6.2.1 *(modified realizability interpretation)*

$$\begin{array}{rcl}
\tau(P\vec{t}) & := & \epsilon \\
\tau(\varphi \to \psi) & := & \tau(\varphi) \to \tau(\psi) \\
\tau(\forall x^\iota.\varphi) & := & \iota \to \tau(\varphi) \\
\tau(\underline{\forall x^\iota}.\varphi) & := & \tau(\varphi) \\
\tau(\exists x^\iota.\varphi) & := & \iota, \tau(\varphi)
\end{array}$$

$$\begin{array}{rcl}
\epsilon\ \mathbf{mr}\ P\vec{t} & := & P\vec{t} \\
\vec{s}\ \mathbf{mr}\ \varphi \to \psi & := & \forall \vec{x}^{\tau(\varphi)}.(\vec{x}\ \mathbf{mr}\ \varphi) \to (\vec{s}\vec{x}\ \mathbf{mr}\ \psi) \\
\vec{s}\ \mathbf{mr}\ \forall x^\iota.\varphi & := & \forall x^\iota.(\vec{s}x\ \mathbf{mr}\ \varphi) \\
\vec{s}\ \mathbf{mr}\ \underline{\forall x^\iota}.\varphi & := & \forall x^\iota.(\vec{s}\ \mathbf{mr}\ \varphi) \\
r, \vec{s}\ \mathbf{mr}\ \exists x^\iota.\varphi(x) & := & \vec{s}\ \mathbf{mr}\ \varphi(r)
\end{array}$$

In the **mr**-clauses, x^ι should not occur in $\vec{s}$ and $\vec{x}$ should be fresh. Note that only existential quantifiers give rise to a longer sequence of types. In particular, if φ has no existential quantifiers, then $\tau(\varphi) = \epsilon$. (We use that $\vec{\sigma} \to \epsilon = \epsilon$). Nested implications give rise to arbitrarily high types. In $\underline{\forall x^\iota}\varphi$, the program specified by φ may not depend on x, so the "$\iota \to$" is discarded in the τ-clause. In the **mr**-clause, the programs $\vec{s}$ do not get x as input, as intended. But to avoid that x becomes free in φ, we changed Berger's definition by adding $\forall x^\iota$.

By induction on the structure of the MF-formula φ one sees that if $\vec{s}$ is of type $\tau(\varphi)$, then $\vec{s}\ \mathbf{mr}\ \varphi$ is a correct formula of NH, so in particular, it will not contain $\exists$- and $\underline{\forall}$-quantifiers (nor of course the symbol **mr**).

Strictly speaking, $\varphi(r)$ in the last clause is not an MF-formula, because r is not a first-order term, but only a lambda term of base type. This can be repaired by enlarging the domain of **mr** to such formulae. In any case, **mr** maps formulae of MF to formulae of NH.

6.2.2 Derivations and Program Extraction

In the previous section we introduced the formulae of MF, the formulae of NH and a translation of the former into the latter. In this section we will introduce proofs for MF and for NH. The whole point will be, that from an MF proof of φ a program can be extracted, together with an NH proof that this program meets its specification φ.

Proofs are formalized by derivation terms, a linear notation for natural deduction. Derivation terms are defined as the least set containing assumption variables $(u^\varphi, v^\psi, \ldots)$ and closed under certain syntactic operations. To express some side conditions, the sets of free assumption variables ($\mathrm{FA}(d)$) and of computational relevant variables ($\mathrm{CV}(d)$) are defined simultaneously. By convention, x and y range over object variables. We let d, e range over derivations.

The difference with Definition 5.5.1 is that we don't have conjunction, but we do have the $\underline{\forall x}$ quantifier and we now have higher-order objects.

The introduction rule for the $\underline{\forall}$-quantifier has an extra proviso: we may only extend a derivation d of φ to one of $\underline{\forall x}.\varphi$, if x is not *computationally relevant* in d. Roughly speaking, all free object variables of d occurring as argument of a $\forall$-elimination or as witness in an $\exists$-introduction are computationally relevant.

Definition 6.2.2 *(derivations, free assumptions, computational relevant variables)*

assumptions :	u^{φ}	
$\rightarrow$ *-introduction* :	$(\lambda u^{\varphi}.d^{\psi})^{\varphi\rightarrow\psi}$	
$\rightarrow$ *-elimination* :	$(d^{\varphi\rightarrow\psi}e^{\varphi})^{\psi}$	
$\forall$*-introduction* :	$(\lambda x^{\sigma}.d^{\varphi})^{\forall x^{\sigma}.\varphi}$	*provided (1)*
$\forall$*-elimination* :	$(d^{\forall x^{\sigma}.\varphi(x)}t^{\sigma})^{\varphi(t)}$	
$\underline{\forall}$*-introduction* :	$(\underline{\lambda x^{\sigma}}.d^{\varphi})^{\underline{\forall x^{\sigma}}.\varphi}$	*provided (2)*
$\underline{\forall}$*-elimination* :	$(d^{\underline{\forall x^{\sigma}}.\varphi(x)}\underline{t^{\sigma}})^{\varphi(t)}$	
$\exists$*-introduction* :	$(\exists^{+}[t^{\sigma};d^{\varphi(t)}])^{\exists x^{\sigma}.\varphi(x)}$	
$\exists$*-elimination* :	$(\exists^{-}[d^{\exists x^{\sigma}.\varphi(x)};y;u^{\varphi(y)};e^{\psi}])^{\psi}$	*provided (3)*

$\mathrm{FA}(u)=\{u\}$	$\mathrm{CV}(u)=\emptyset$
$\mathrm{FA}(\lambda u.d)=\mathrm{FA}(d)\setminus\{u\}$	$\mathrm{CV}(\lambda u.d)=\mathrm{CV}(d)$
$\mathrm{FA}(de)=\mathrm{FA}(d)\cup\mathrm{FA}(e)$	$\mathrm{CV}(de)=\mathrm{CV}(d)\cup\mathrm{CV}(e)$
$\mathrm{FA}(\lambda x.d)=\mathrm{FA}(d)$	$\mathrm{CV}(\lambda x.d)=\mathrm{CV}(d)\setminus\{x\}$
$\mathrm{FA}(dt)=\mathrm{FA}(d)$	$\mathrm{CV}(dt)=\mathrm{CV}(d)\cup\mathrm{FV}(t)$
$\mathrm{FA}(\underline{\lambda x}.d)=\mathrm{FA}(d)$	$\mathrm{CV}(\underline{\lambda x}.d)=\mathrm{CV}(d)$
$\mathrm{FA}(d\underline{t})=\mathrm{FA}(d)$	$\mathrm{CV}(d\underline{t})=\mathrm{CV}(d)$
$\mathrm{FA}(\exists^{+}[t;d])=\mathrm{FA}(d)$	$\mathrm{CV}(\exists^{+}[t;d])=\mathrm{CV}(d)\cup\mathrm{FV}(t)$
$\mathrm{FA}(\exists^{-}[d;y;u;e])$	$\mathrm{CV}(\exists^{-}[d;y;u;e])$
$\quad=\mathrm{FA}(d)\cup(\mathrm{FA}(e)\setminus\{u\})$	$\quad=\mathrm{CV}(d)\cup(\mathrm{CV}(e)\setminus\{y\})$

where the provisos are:

(1) $x\notin\mathrm{FV}(\psi)$ *for any* $u^{\psi}\in\mathrm{FA}(d)$.

(2) $x\notin\mathrm{FV}(\psi)$ *for any* $u^{\psi}\in\mathrm{FA}(d)$ *and moreover,* $x\notin\mathrm{CV}(d)$.

(3) $y\notin\mathrm{FV}(\psi)$ *and* $y\notin\mathrm{FV}(\chi)$ *for all* $v^{\chi}\in\mathrm{FA}(e)\setminus\{u\}$.

An MF*-derivation is a derivation with all quantifier rules restricted to base types. An* NH*-derivation is a derivation without the* $\underline{\forall x}$ *and the* $\exists$*-rules. We will write* $\Phi\vdash_{\mathrm{MF}}\psi$ *if there exists a derivation* d^{ψ}*, with all free assumptions among* Φ*. Likewise for* $\vdash_{\mathrm{NH}}$.

From MF-derivations, we can read off a program and a correctness proof for this program. This is best illustrated by the $\exists^{+}$ rule: If we use this rule to prove $\exists x.\varphi(x)$, then we immediately see the witness t and a proof d of $\varphi(t)$. In general, we can define $\mathbf{ep}(d)$, the sequence of extracted programs from a derivation d. To deal with assumption variables in d, we fix for every assumption variable u^{φ} a sequence of object variables $\vec{x}_u^{\tau(\varphi)}$. The extracted program is defined with respect to this choice.

Definition 6.2.3 *(extracted program from* MF*-derivations)*

$$\begin{array}{rcl@{\qquad}rcl}
\mathbf{ep}(u^\varphi) & := & \vec{x}_u^{\tau(\varphi)} & \mathbf{ep}(d^{\forall x^\iota.\varphi(x)}t^\iota) & := & \mathbf{ep}(d)t \\
\mathbf{ep}(\lambda u^\varphi.d^\psi) & := & \lambda\vec{x}_u^{\tau(\varphi)}.\mathbf{ep}(d) & \mathbf{ep}(\underline{\lambda x^\iota}.d^\varphi) & := & \mathbf{ep}(d) \\
\mathbf{ep}(d^{\varphi\to\psi}e^\varphi) & := & \mathbf{ep}(d)\mathbf{ep}(e) & \mathbf{ep}(d^{\underline{\forall x^\iota}.\varphi(x)}\underline{t^\iota}) & := & \mathbf{ep}(d) \\
\mathbf{ep}(\lambda x^\iota.d^\varphi) & := & \lambda x^\iota.\mathbf{ep}(d) & \mathbf{ep}(\exists^+[t^\iota;d^{\varphi(t)}]) & := & t,\mathbf{ep}(d)
\end{array}$$

$$\mathbf{ep}(\exists^-[d;y;u^{\varphi(y)};e^\psi]) := \mathbf{ep}(e)[y:=s][\vec{x}_u:=\vec{t}], \textit{where } s,\vec{t}=\mathbf{ep}(d^{\exists x^\iota.\varphi(x)})$$

The whole enterprise is justified by the following

Theorem 6.2.4 (Correctness [Ber93]) *If d is an* MF *derivation of φ, then there exists an* NH *derivation $\mu(d)$ of $\mathbf{ep}(d)$ $\mathbf{mr}$ φ. Moreover, the only free assumptions in $\mu(d)$ are of the form $\vec{x}_u$ $\mathbf{mr}$ ψ, for some assumption u^ψ occurring in d already.*

Proof: First the following facts are verified by induction on d:

1. $\mathrm{FV}(\mathbf{ep}(d)) \subseteq \bigcup\{\vec{x}_u | u \in \mathrm{FA}(d)\} \cup \mathrm{CV}(d)$.
2. $\mathbf{ep}(d^\varphi)$ is a sequence of terms of type $\tau(\varphi)$.

We now define $\mu(d)$ by induction on d.

$$\begin{array}{rcl}
\mu(u^\varphi) & := & u^{\vec{x}_u\mathbf{mr}\varphi} \\
\mu(\lambda u^\varphi.d^\psi) & := & \lambda\vec{x}_u^{\tau(\varphi)}.\lambda u^{\vec{x}_u\mathbf{mr}\varphi}.\mu(d) \\
\mu(de) & := & \mu(d)\mathbf{ep}(d)\mu(e) \\
\mu(\lambda x^\iota.d) & := & \lambda x^\iota.\mu(d) \\
\mu(dt) & := & \mu(d)t \\
\mu(\underline{\lambda x^\iota}.d) & := & \lambda x^\iota.(\mu(d)) \\
\mu(d\underline{t}) & := & \mu(d)t \\
\mu(\exists^+[t;d^{\varphi(t)}]) & := & \mu(d) \\
\mu(\exists^-[d^{\exists x.\varphi(x)};y;u^{\varphi(y)};e^\psi]) & := & \mu(e)[y:=s][\vec{x}_u:=\vec{t}][u:=\mu(d)], \\
 & & \text{where } s,\vec{t}=\mathbf{ep}(d)
\end{array}$$

By induction on d one verifies that $\mu(d)$ is a valid proof of the correctness formula, and that its free assumption variables are of the form $u^{\vec{x}_u\mathbf{mr}\varphi}$ for $u^\varphi \in \mathrm{FA}(d)$. We only deal with three cases, see e.g. [Ber93] for the other cases:

$\to^+$: By the induction hypothesis, $\mu(d) : \mathbf{ep}(d)$ $\mathbf{mr}$ ψ. Then $\mu(\lambda u.d)$ proves

$$\begin{array}{cll}
 & \forall\vec{x}_u.(\vec{x}_u\ \mathbf{mr}\ \varphi \to \mathbf{ep}(d)\ \mathbf{mr}\ \psi) & \\
\equiv & \forall\vec{x}_u.(\vec{x}_u\ \mathbf{mr}\ \varphi \to (\lambda\vec{x}_u.\mathbf{ep}(d))\vec{x}_u\ \mathbf{mr}\ \psi) & \text{(identify } \beta\text{-equal terms)} \\
\equiv & \forall\vec{x}_u.(\vec{x}_u\ \mathbf{mr}\ \varphi \to (\mathbf{ep}(\lambda u.d))\vec{x}_u\ \mathbf{mr}\ \psi) & \text{(definition } \mathbf{ep}) \\
\equiv & \mathbf{ep}(\lambda u.d)\ \mathbf{mr}\ (\varphi\to\psi) & \text{(definition } \mathbf{mr}).
\end{array}$$

$\underline{\forall^+}$: By induction hypothesis, we have $\mu(d)$ proves $\mathbf{ep}(d)$ $\mathbf{mr}$ φ. By the proviso of $\underline{\forall^+}$, $x \notin \mathrm{CV}(d)$, hence (by the first fact about $\mathbf{ep}(d)$) $x \notin \mathrm{FV}(\mathbf{ep}(d))$. Furthermore, x doesn't occur in free assumptions of d, hence not in assumptions of $\mu(d)$, so $\lambda x.\mu(d)$ is a correct derivation of $\forall x.(\mathbf{ep}(d)\ \mathbf{mr}\ \varphi)$, which is equivalent (because $x \notin \mathrm{FV}(\mathbf{ep}(d))$) to $\mathbf{ep}(\underline{\lambda x}.d)$ $\mathbf{mr}$ $\underline{\forall x}.\varphi$.

$\exists^-$: Let $s, \vec{t} := \mathbf{ep}(d)$. By induction hypothesis we have proofs $\mu(d)$ of $\mathbf{ep}(d)$ **mr** $\exists x.\varphi(x) \equiv \vec{t}\,\mathbf{mr}\,\varphi(s)$ and $\mu(e)$ of $\mathbf{ep}(e)$ **mr** ψ, possibly with $u^{\vec{x}_u \mathbf{mr} \varphi(y)}$ among its free assumption variables. As neither y nor $\vec{x}_u$ occur in ψ, $\mu(e)[y := s][\vec{x}_u := \vec{t}]$ (possibly with $u^{\vec{t}\mathbf{mr}\varphi(s)}$ among its free assumption variables) is a proof of $(\mathbf{ep}(e)[y := s][\vec{x}_u := \vec{t}])$ **mr** ψ. Hence $\mu(\exists^-[d; y; u; e])$ is a proof of $\mathbf{ep}(\exists^-[d; y; u; e])$ **mr** ψ, with the intended free assumptions.

⊠

6.2.3 Realization of Axioms for Equality, Negation, Induction

In this section we will explore the use of axioms. If we use an axiom $\mathbf{ax}^\varphi$ (as open assumption) in a proof d of MF, then the extracted program $\mathbf{ep}(d)$ contains free variables $\vec{x}_{\mathbf{ax}}^{\tau(\varphi)}$ (as holes), and the correctness proof $\mu(d)$ contains free assumption variables $\mathbf{ax} : \vec{x}_{\mathbf{ax}}\ \mathbf{mr}\ \varphi$ (according to Theorem 6.2.4).

The goal is to complete the program in a correct way. More specifically, we look for potential realizers $\vec{t}_{\mathbf{ax}}$, such that we can find an NH-derivation $d_{\mathbf{ax}}$ of the correctness statement $\vec{t}_{\mathbf{ax}}\ \mathbf{mr}\ \varphi$. The derivation $d_{\mathbf{ax}}$ may contain acceptable assumptions. If such a proof exists, we call $\vec{t}_{\mathbf{ax}}$ the *realizer* of the axiom. This is a flexible notion, because we have not specified which assumptions are acceptable. The extracted program can be completed by taking $\mathbf{ep}(d)[\vec{x}_{\mathbf{ax}} := \vec{t}_{\mathbf{ax}}]$. The correctness proof can be mended by substituting the subproof $d_{\mathbf{ax}}$ for the free assumption variable $\mathbf{ax}$. It is clear that the justification of postulated principles should be given in terms of NH, because in this logic the correctness proofs live.

We will summarize several situations that can arise by adding realizable axioms to MF. The various possibilities are characterized by the realizers and the assumptions needed in the correctness proofs. Moreover, we will briefly mention their typical use. In the subsequent sections the correctness of these axioms is described in more detail. We distinguish:

1. True $\exists$-free axioms: they have a trivial realizer ϵ and the correctness proof contains the same axioms (up to underlinings). These will typically be non-logical $\exists$-free axioms that are true in the intended model, e.g. symmetry of $=$. The computation is not affected by these and the correctness proof relies on true assumptions. This enables us to reduce the amount of proof to be formalized. We will benefit a lot of it in Section 6.3.

2. Purely logical axioms with *absolute* realizers, i.e. realizers that have a correctness proof without any assumptions. These will be purely logical axioms, exploiting the realizability interpretation of the $\underline{\forall}$-quantifier. They give some insight in the meaning of the $\underline{\forall}$-quantifier. Some of them will be used in Section 6.4.3.

3. Axiom schemata with realizers for which the correctness proof contains new instances of the same schema. Typical examples are "ex falso quod libet" and "replacement of equals by equals".

4. Induction axioms. The realizers are operators for simultaneous primitive recursion; the correctness proof is given in an extended framework. Induction is needed to deal with Gödel's T in Section 6.4.

This is well known theory, apart from the axioms under (2), which explore the special nature of the $\underline{\forall}$-quantifier. Axioms as under (1) are exploited in [Ber93]. Case (3) and (4) can be found in [Tro73].

6.2.3.1 $\exists$-free Axioms and Harrop Formulae

Consider a $\exists$-free MF formula φ. We have $\tau(\varphi) = \epsilon$, so the only potential realizer is the empty sequence. Let φ' be the formula obtained from φ by deleting all underlinings. We have ϵ **mr** $\varphi \equiv \varphi'$. So the program obtained from a proof using φ as an axiom will be correct, whenever φ' is true. In this sense we are allowed to axiomatize new predicates and functions by $\exists$-free axioms.

More generally, we can consider the class of Harrop formulae, i.e. φ with $\tau(\varphi) = \epsilon$. Roughly speaking, these formulae don't have existential quantifiers in their conclusion. They have the empty sequence as a potential realizer. However, we lose the property that ϵ **mr** $\varphi \equiv \varphi'$, as the following example shows:

$$\epsilon \textbf{ mr } (\forall x^o.\exists y^o.Q(x,y)) \to P(\vec{t}) \equiv \quad \forall f^{o\to o}.(\forall x^o.Q(x,fx)) \to P(\vec{t})$$

We are tempted to write this last formula as $(\exists f^{o\to o}.\forall x^o.Q(x,fx)) \to P(\vec{t})$, but this is neither a formula of NH nor of MF. HA^ω + AC is needed, to prove $(\epsilon$ **mr** $\varphi) \leftrightarrow \varphi$ for all Harrop formulae φ.

6.2.3.2 Realizable Axioms

Inspection of the derivation rules for MF reveals an asymmetry. Although the introduction rule for $\underline{\forall}$ has a stronger proviso than that of $\forall$, the elimination rules are the same. The result is that there are some principles that are intuitively true, but not provable in MF. One of them is $(\underline{\forall x}.\exists y.\varphi) \to \exists y.\underline{\forall x}.\varphi$: If for all x there exists a y independent of x, then one such y suffices for all x. So the witness for y on the left hand side should also suffice on the right hand side. This suggests to postulate this formula as an axiom, with the identity as realizer.

Admitting axioms like this one, goes a step further than admitting $\exists$-free formulae as axioms. In the case of $\exists$-free formulae, we can remove all underlinings from the proof, and we obtain a correct proof in a well known logic (i.e. usual minimal first-order predicate logic). If we use the axiom above, this is no longer possible, as it becomes false (even classically) after removing the underlining. On the other hand, for the axioms in this section we can postulate realizers that have a closed correctness proof. In this respect they have a firm base. We will propose the following axiom schemata, where H ranges over Harrop Formulae, i.e. $\tau(H) = \epsilon$. **IP** stands for *independence of*

premise and **IU** stands for *independence of underlined quantifier.*

$$\begin{array}{rrcll} \mathbf{IU}: & (\underline{\forall x}.\exists y^\iota.\varphi) & \to & \exists y^\iota.\underline{\forall x}.\varphi & \\ \mathbf{IP}: & (H \to \exists x^\iota.\varphi) & \to & \exists x^\iota.(H \to \varphi) & (x \notin \mathrm{FV}(H)) \\ \mathbf{intro}: & (\forall x.H) & \to & \underline{\forall x}.H & \end{array}$$

The first two have associated type $\iota, \tau(\varphi) \to \iota, \tau(\varphi)$ and they are realized by the identity on sequences of this type. The third is realized by the empty sequence. We compute the correctness formulae, which have trivial proofs in NH:

$$\begin{array}{rcl} \mathbf{IU}: & & (\lambda y^\iota, \vec{z}^{\tau(\varphi)}.y, \vec{z})\ \mathbf{mr}\ ((\underline{\forall x}.\exists y^\iota.\varphi) \to \exists y^\iota.\underline{\forall x}.\varphi) \\ & \equiv & \forall y, \vec{z}.(y, \vec{z}\ \mathbf{mr}\ \underline{\forall x}.\exists y^\iota.\varphi) \to y, \vec{z}\ \mathbf{mr}\ \exists y^\iota.\underline{\forall x}.\varphi \\ & \equiv & \forall y, \vec{z}.(\forall x.(\vec{z}\ \mathbf{mr}\ \varphi)) \to \forall x.(\vec{z}\ \mathbf{mr}\ \varphi) \\ \\ \mathbf{IP}: & & (\lambda x, \vec{z}.x, \vec{z})\ \mathbf{mr}\ ((H \to \exists x^\iota.\varphi) \to \exists x^\iota.(H \to \varphi)) \\ & \equiv & \forall x, \vec{z}.(\epsilon\ \mathbf{mr}\ H \to \vec{z}\ \mathbf{mr}\ \varphi) \to (\epsilon\ \mathbf{mr}\ H \to \vec{z}\ \mathbf{mr}\ \varphi) \\ \\ \mathbf{intro}: & & \epsilon\ \mathbf{mr}\ (\forall x.H \to \underline{\forall x}.H) \\ & \equiv & (\forall x.(\epsilon\ \mathbf{mr}\ H)) \to \forall x.(\epsilon\ \mathbf{mr}\ H) \end{array}$$

This leads to the following

Theorem 6.2.5 Second Correctness Theorem
If $\mathbf{IP}+\mathbf{IU}+\mathbf{intro} \vdash_{\mathrm{MF}} \varphi$ *then there exists a sequence* $\vec{t}$ *such that* $\vdash_{\mathrm{NH}} \vec{t}\ \mathbf{mr}\ \varphi$. *If* d^φ *is the* MF*-derivation,* $\vec{t}$ *can be obtained from* $\mathbf{ep}(d)$ *by replacing all free variables introduced by the axioms* **IP**, **IU** *and* **intro** *by the identity.*

We will not address the question whether the inverse of this correctness result also holds. In [Tro73, § 3.4.8] it is proved that HA^ω+IP+AC, axiomatizes modified realizability. However, AC is neither a formula of MF nor of NH, as it contains both higher-order variables and existential quantifiers, so we cannot use that result here directly. Axiomatizing modified realizability is interesting, because it gives more understanding of the $\underline{\forall}$-quantifier.

6.2.3.3 Axioms for Negation and Equality

We have seen how to realize $\exists$-free axioms. It is not always possible to axiomatize predicates $\exists$-free. We will for example need equality, with the axiom schema $s = t \to \varphi(s) \to \varphi(t)$. Another example is negation, with axioms $\bot \to \varphi$, which can be dealt with in a similar way as equality.

Let $=$ be a binary predicate symbol. The usual axioms of reflexivity, symmetry and transitivity are $\exists$-free and hence harmless. Instances of the replacement schema may contain existential quantifiers. Let **repl** stand for axioms of the form $s = t \to \varphi(s) \to \varphi(t)$. We will provide for realizers such that the correctness formula gets provable in NH enriched with the schema **repl**.

Note that $\tau(s = t \to \varphi(s) \to \varphi(t)) \equiv \tau(\varphi) \to \tau(\varphi)$. The identity can be taken as realizer, as the following calculation shows:

$$\begin{aligned} & \lambda\vec{x}^{\tau(\varphi)}.\vec{x}\ \mathbf{mr}\ s = t \to \varphi(s) \to \varphi(t) \\ \equiv\ & s = t \to \forall\vec{x}.\vec{x}\ \mathbf{mr}\ \varphi(s) \to \vec{x}\ \mathbf{mr}\ \varphi(t), \end{aligned}$$

which can be proved using the **repl** schema on NH-formulae. This means that we can use equality axioms within our proofs. Because they are realized by the empty sequence, or by the identity on sequences, we can discard their use when extracting the program. The correctness proofs contain the same axioms schema, which is regarded as valid.

6.2.3.4 Induction Axioms

It is straightforward to introduce induction in this context. Induction can be postulated by introducing axioms

$$ind_\varphi : \varphi(0) \to (\forall n, \varphi(n) \to \varphi(\mathsf{S}n)) \to \forall n.\varphi(n)\ .$$

In the general case, induction can be realized by simultaneous primitive recursion operators (See [Tro73, § 1.6.16, § 3.4.5]). We will only need the special case that $\tau(\varphi) = \sigma$, so the induction formula is realized by exactly one term. In this case, the usual recursion operator is a potential realizer. However, for the correctness proof we have to extend NH in two directions: We have to add induction axioms to it and we have to consider object terms modulo $\beta\mathsf{R}$-equality. Let $\tau(\varphi) = \sigma$, then we show that in extended NH, R_σ **mr** ind_φ is provable.

$$\begin{aligned} & \mathsf{R}_\sigma\ \mathbf{mr}\ ind_\varphi \\ & \equiv \forall x.(x\ \mathbf{mr}\ \varphi(0)) \to \forall f.(f\ \mathbf{mr}\ \forall n.\varphi(n) \to \varphi(\mathsf{S}n)) \to \forall n.(\mathsf{R}xfn)\ \mathbf{mr}\ \varphi(n)\ . \end{aligned}$$

So assume x **mr** $\varphi(0)$ and f **mr** $\forall n.\varphi(n) \to \varphi(\mathsf{S}n)$. By induction on n we prove $(\mathsf{R}xfn)$ **mr** $\varphi(n)$.

If $n = 0$, we identify $\mathsf{R}xf0$ with x, and the first assumption applies.

If $n = (\mathsf{S}m)$, we may assume that $(\mathsf{R}xfm)$ **mr** $\varphi(m)$ (IH). Our second hypothesis can be rewritten to $\forall n.\forall y.(y\ \mathbf{mr}\ \varphi(n)) \to (fny)\ \mathbf{mr}\ \varphi(\mathsf{S}n)$. This can be applied to m and $(\mathsf{R}xfm)$ and after identification of $\mathsf{R}xf(\mathsf{S}m)$ with $fm(\mathsf{R}xfm)$, it follows that $\mathsf{R}xf(\mathsf{S}m)$ **mr** $\varphi(\mathsf{S}m)$.

6.3 Formal Comparison for β-Reduction

In this section the proof of Section 6.1 will be formalized in first-order predicate logic, as introduced in Section 6.2. This is not unproblematic as the informal proof contains induction on types and terms, which is not a part of the framework. This is solved by defining a series of proofs, by recursion over types or terms. In this way the induction is shifted to the metalevel. There is a price to be paid: instead of a uniform function U, such that $U(M)$ computes the desired upper bound for a term M, we only extract

for any M an expression Upper$[M]$, which computes an upper bound for term M only. So here we lose a kind of uniformity. It is well known that the absence of a uniform first-order proof is essential, because the computability predicate is not arithmetizable [Tro73, § 2.3.11].

Another incompleteness arises, because some combinatorial results will be plugged in as axioms. This second incompleteness is harmless for our purpose, because all these axioms are formulated without using existential quantifiers. Hence they are realized by the empty sequence (and finding formal proofs for these facts would be waste of time).

6.3.1 Fixing Signature and Axioms

As to the language, we surely have to represent λ-terms. To this end, we adopt for each type ρ new sorts $\mathcal{V}_\rho$ and $\mathcal{T}_\rho$, that interpret variables and terms *modulo α-conversion* of type ρ, respectively. Constants of sort $\mathcal{V}_\rho$ are added to represent variables (we write $\ulcorner x \urcorner$ for the formal representation of x). Function symbols for typed application and abstraction are included as well. With $\ulcorner M \urcorner$, we denote the representation of a λ-term M in this first-order language, using the following function symbols:

$Var_\rho : \mathcal{V}_\rho \to \mathcal{T}_\rho$, to inject variables into terms;
$\_ \bullet_{\rho,\sigma} \_ : \mathcal{T}_{\rho\to\sigma} \times \mathcal{T}_\rho \to \mathcal{T}_\sigma$, denoting typed application;
$\lambda\!\!\lambda_{\rho,\sigma} : \mathcal{V}_\rho \times \mathcal{T}_\sigma \to \mathcal{T}_{\rho\to\sigma}$, denoting typed abstraction.

Note that $\ulcorner x \urcorner$ is overloaded: it can be of sort $\mathcal{T}_\rho$ and of sort $\mathcal{V}_\rho$. We use r, s and t as formal variables over $\mathcal{T}_\rho$; x and y are variables of sorts $\mathcal{V}_\rho$; We abbreviate $((s \bullet t_1) \bullet \cdots \bullet t_n)$ by $s \bullet \vec{t}$. Type decoration is often suppressed.

Note that e.g. $\ulcorner \lambda x.x \urcorner \equiv \lambda\!\!\lambda(\ulcorner y \urcorner, Var(\ulcorner y \urcorner))$, for some arbitrary but fixed choice of y. Although the terms in the intended model are taken modulo α-conversion, the first-order terms cannot have this feature. We will also need function symbols to represent simultaneous substitution: for any sequence of types $\sigma, \tau_1, \ldots, \tau_n$, a symbol $\_(\_, \_, \ldots := \_, \_, \ldots)$ of arity $\mathcal{T}_\sigma \times \mathcal{V}_{\tau_1} \times \cdots \times \mathcal{V}_{\tau_n} \times \mathcal{T}_{\tau_1} \times \cdots \times \mathcal{T}_{\tau_n} \to \mathcal{T}_\sigma$. The intended meaning of $s(\vec{x} := \vec{t})$ is the simultaneous substitution in s of x_i by t_i. If for some i and j, x_i and x_j happen to be the same, the first occurrence from left to right takes precedence (so the other substitution is simply discarded).

In order to represent upper bounds for reduction sequences, we introduce a sort nat, denoting the natural numbers, with constants 0^{nat}, 1^{nat} and $\_ + \_$ of arity $\mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$, with their usual meaning. We use m and n for formal variables over sort nat.

Finally, we add binary predicate symbols $\_ =_\rho \_$ for equality on sort $\mathcal{T}$ and SN_σ of arity $\mathcal{T}_\sigma \times \mathsf{nat}$, representing the binary SN-relation of Definition 2.2.1.

We can now express the axioms that will be used in the formal proof. We will use the axiom schema **repl** : $s = t \to \varphi(s) \to \varphi(t)$ to replace equals by equals. Furthermore, we use all well typed instances of the following axiom schemata.

1. $\underline{\forall x}.\ \mathrm{SN}_\rho(Var(x), 0)$

2. $\underline{\forall x, \vec{t}, s, m, n.}\ \mathrm{SN}_{\rho\to\sigma}(\mathit{Var}(x) \bullet \vec{t}, m) \to \mathrm{SN}_\rho(s, n) \to \mathrm{SN}_\sigma(\mathit{Var}(x) \bullet \langle \vec{t}, s\rangle, m + n)$

3. $\underline{\forall s, x, m.}\ \mathrm{SN}_\sigma(s \bullet \mathit{Var}(x), m) \to \mathrm{SN}_{\rho\to\sigma}(s, m)$

4. $\underline{\forall r, y, \vec{x}, s, \vec{t}, \vec{r}, m, n.}\ \mathrm{SN}_\iota(r(y, \vec{x} := s, \vec{t}) \bullet \vec{r}, m) \to \mathrm{SN}_\rho(s, n) \to \mathrm{SN}_\iota(\mathbb{\lambda}(y, r)(\vec{x} := \vec{t}) \bullet \langle s, \vec{r}\rangle, m + n + 1)$

5. $\underline{\forall \vec{t}.}\ t_i = \mathit{Var}(\ulcorner x_i \urcorner)(\ulcorner \vec{x} \urcorner := \vec{t}),$ provided i is the first occurrence of $\ulcorner x_i \urcorner$ in $\ulcorner \vec{x} \urcorner$.

6. $\underline{\forall r, s, \vec{x}, \vec{t}.}\ r(\vec{x} := \vec{t}) \bullet s(\vec{x} := \vec{t}) = (r \bullet s)(\vec{x} := \vec{t})$

7. $\underline{\forall s, \vec{x}.}\ s(\vec{x} := \mathit{Var}(\vec{x})) = s,$ where $\mathit{Var}(\vec{x})$ stands for $\mathit{Var}(x_1), \ldots, \mathit{Var}(x_m)$

In the formal proofs, we will refer to these axioms by number (e.g. ax_5). Axioms 1–3 express simple combinatorial facts about SN. The equations 5–7 axiomatize substitution. Axiom 4 is a mix, integrating a basic fact about reduction and an equation for substitution. The reason for this mixture is that we thus avoid variable name clashes. This is the only axiom that needs some elaboration.

In the intended model, $(\lambda x.r)[\vec{x} := \vec{t}]$ equals $\lambda x.(r[\vec{x} := \vec{t}])$, because we can perform an α-conversion, renaming x. However, we cannot postulate the similar equation

$$\forall x, \vec{x}, \vec{t}, r.\ \mathbb{\lambda}(x, r)(\vec{x} := \vec{t}) = \mathbb{\lambda}(x, r(\vec{x} := \vec{t}))$$

as an axiom, because we cannot avoid that e.g. t_1 gets instantiated by a term containing the free variable x, such that the same x would occur both bound and free. Now in the proof of Lemma 6.1.3 it is shown how the reduction length of $(\lambda y.t)s\vec{r}$ can be estimated from the reduction lengths of s and $t[y := s]\vec{r}$. After substituting $r[\vec{x} := \vec{t}]$ for t, and using the abovementioned equation (thus avoiding that variables in $\vec{t}$ become bound), we get Axiom 4.

6.3.2 Proof Terms and Extracted Programs

As in the informal proof, we define formulae $\mathrm{SC}_\rho(t)$ by induction on the type ρ. These will occur as abbreviations in the formal derivations.

$$\begin{cases} \mathrm{SC}_\iota(t) & := & \exists n^{\mathsf{nat}}.\mathrm{SN}_\iota(t, n) \\ \mathrm{SC}_{\rho\to\sigma}(t) & := & \underline{\forall s^{\mathcal{T}_\rho}}.\mathrm{SC}_\rho(s) \to \mathrm{SC}_\sigma(t \bullet s) \end{cases}$$

Due to the underlined quantifier, $\tau(\mathrm{SC}_\sigma(s)) = \sigma'$, where σ' is obtained from σ by renaming base types ι to nat. In the sequel, the prime ($'$) will be suppressed. The underlined quantifier takes care that numerical upper bounds only use numerical information about subterms: the existential quantifier hidden in $\mathrm{SC}(t \bullet s)$ can only use the existential quantifier in $\mathrm{SC}(s)$; not s itself. In fact, this is the reason for introducing the underlined quantifier.

6.3.2.1 Formalizing the SC Lemma.

We proceed by formalizing Lemma 6.1.2. We will define proofs

$$\Phi_\rho : \underline{\forall t.}\, \mathrm{SC}_\rho(t) \to \exists n.\mathrm{SN}_\rho(t,n) \quad \text{and}$$

$$\Psi_\rho : \underline{\forall x, \vec{t}.}\, (\exists m.\mathrm{SN}_\rho(Var(x) \bullet \vec{t}, m)) \to \mathrm{SC}_\rho(Var(x) \bullet \vec{t})$$

with simultaneous induction on ρ:

$$\begin{array}{l}
\Phi_\iota := \underline{\lambda t.}\lambda u^{\mathrm{SC}(t)}.u \\
\Phi_{\rho\to\sigma} := \underline{\lambda t}\lambda u^{\mathrm{SC}(t)}. \\
\qquad \exists^-[\Phi_\sigma \underline{(t \bullet Var(x))} \Big(u \underline{Var(x)} (\Psi_\rho \underline{x} \exists^+[0; (ax_1 \underline{x})])\Big); \\
\qquad\quad m; v^{\mathrm{SN}(t\bullet Var(x),m)}; \\
\qquad\quad \exists^+[m; (ax_3 \underline{txmv})]]
\end{array}$$

$$\begin{array}{l}
\Psi_\iota := \underline{\lambda x \vec{t}.}\lambda u^{\exists m.\mathrm{SN}(Var(x)\bullet\vec{t},m)}.u \\
\Psi_{\rho\to\sigma} := \underline{\lambda x, \vec{t}.}\ \lambda u^{\exists m.\mathrm{SN}(Var(x)\bullet\vec{t},m)}.\underline{\lambda s.}\lambda v^{\mathrm{SC}(s)}. \\
\qquad \exists^-[u; m; u_0^{\mathrm{SN}(Var(x)\bullet\vec{t},m)}; \\
\qquad\quad \exists^-[(\Phi_\rho \underline{s} v); n; v_0^{\mathrm{SN}(s,n)}; \\
\qquad\qquad \Psi_\sigma \underline{x\vec{t}s}\ \exists^+[(m+n); (ax_2 \underline{x\vec{t}smn} u_0 v_0)]]]
\end{array}$$

Having the concrete derivations, we can extract the computational content, using the definition of **ep**. Note that the underlined parts are discarded, and that an $\exists$-elimination gives rise to a substitution. The resulting functionals are $\mathbf{ep}(\Phi_\rho) : \rho \to \mathsf{nat}$ and $\mathbf{ep}(\Psi_\rho) : \mathsf{nat} \to \rho$,

$$\begin{array}{rcl}
\mathbf{ep}(\Phi_\iota) & = & \lambda x_u.x_u \\
\mathbf{ep}(\Phi_{\rho\to\sigma}) & = & \lambda x_u.m[m := \mathbf{ep}(\Phi_\sigma)(x_u(\mathbf{ep}(\Psi_\rho)0))] \\
 & = & \lambda x_u.\mathbf{ep}(\Phi_\sigma)(x_u(\mathbf{ep}(\Psi_\rho)0)) \\
\mathbf{ep}(\Psi_\iota) & = & \lambda x_u.x_u \\
\mathbf{ep}(\Psi_{\rho\to\sigma}) & = & \lambda x_u.\lambda x_v.\mathbf{ep}(\Psi_\sigma)(m+n)[n := \mathbf{ep}(\Phi_\rho)x_v][m := x_u] \\
 & = & \lambda x_u.\lambda x_v.\mathbf{ep}(\Psi_\sigma)(x_u + (\mathbf{ep}(\Phi_\rho)x_v))
\end{array}$$

6.3.2.2 Formalizing the Abstraction Lemma.

We proceed by formalizing Lemma 6.1.3, which deals with abstractions. Let r have sort $\mathcal{T}_{\vec{\rho}\to\rho}$, and each r_i sort $\mathcal{T}_{\rho_i}$ (so $r \bullet \vec{r}$ has sort $\mathcal{T}_\rho$). Let s have sort $\mathcal{T}_\sigma$, y sort $\mathcal{V}_\sigma$, each t_i sort $\mathcal{T}_{\tau_i}$ and each x_i sort $\mathcal{V}_{\tau_i}$. We construct proofs

$$\begin{array}{l}
\Lambda_{\rho,\sigma,\vec{\rho},\vec{r}} : \underline{\forall r, y, \vec{x}, s, \vec{t}, \vec{r}.}\, \mathrm{SC}_\rho(r(y, \vec{x} := s, \vec{t}) \bullet \vec{r}) \to \mathrm{SC}_\sigma(s) \to \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{SC}_\rho(\lambda\!\!\lambda(y, r)(\vec{x} := \vec{t}) \bullet \langle s, \vec{r}\rangle)
\end{array}$$

by induction on ρ. This corresponds to the induction on ρ in the informal proof. The base case uses Axiom 4. Only the first two subscripts will be written in the sequel.

$$\begin{array}{l}\Lambda_{\iota,\sigma} = \underline{\lambda r,y,\vec{x},s,\vec{t},\vec{r}.}\ \lambda u^{\mathrm{SC}_\iota(r(y,\vec{x}:=s,\vec{t})\bullet\vec{r})}.\lambda v^{\mathrm{SC}(s)}.\\ \qquad\qquad \exists^-[u;m;u_0^{\mathrm{SN}(r(y,\vec{x}:=s,\vec{t})\bullet\vec{r},m)};\\ \qquad\qquad\quad \exists^-[(\Phi_\sigma \underline{s} v);n;v_0^{\mathrm{SN}(s,n)};\\ \qquad\qquad\qquad \exists^+[m+n+1;(ax_4\underline{ry\vec{x}s\vec{t}\vec{r}mnu_0v_0})]]]\end{array}$$

$$\begin{array}{l}\Lambda_{\rho\to\tau,\sigma} = \underline{\lambda r,y,\vec{x},s,\vec{t},\vec{r}.}\ \lambda u^{\mathrm{SC}(r(y,\vec{x}:=s,\vec{t})\bullet\vec{r})}.\lambda v^{\mathrm{SC}(s)}.\\ \qquad\qquad \underline{\lambda r'}.\lambda w^{\mathrm{SC}_\rho(r')}.(\Lambda_{\tau,\sigma}\underline{ry\vec{x}s\vec{t}\vec{r}r'}(u\underline{r'}w)v)\end{array}$$

Having these proofs, we can extract their programs, using the definition of **ep**. In this way we get $\mathbf{ep}(\Lambda_{\rho,\sigma}) : \rho\to\sigma\to\rho$,

$$\begin{array}{rcl}\mathbf{ep}(\Lambda_{\iota,\sigma}) &=& \lambda x_u.\lambda x_v.(m+n+1)[n:=\mathbf{ep}(\Phi_\sigma)x_v][m:=x_u]\\ &=& \lambda x_u.\lambda x_v.(x_u+(\mathbf{ep}(\Phi_\sigma)x_v)+1)\\ \mathbf{ep}(\Lambda_{\rho\to\tau,\sigma}) &=& \lambda x_u.\lambda x_v.\lambda x_w.(\mathbf{ep}(\Lambda_{\tau,\sigma})(x_u x_w)x_v)\end{array}$$

6.3.2.3 Formalizing the Main Lemma.

The main lemma (6.1.4) states that every term M is strongly computable, even after substituting strongly computable terms for variables. The informal proof of Lemma 6.1.4 is by induction on M. Therefore, we can only give a formal proof for each M separately. Given a term M with all free variables among $\vec{x}$, we construct by induction on the term structure of M a proof

$$\Pi_{M,\vec{x}} : \underline{\forall t_1,\dots,t_n.}\ \mathrm{SC}(t_1)\to\cdots\to\mathrm{SC}(t_n)\to\mathrm{SC}(\ulcorner M\urcorner(\ulcorner\vec{x}\urcorner:=\vec{t})).$$

$\Pi_{x_i,\vec{x}} := \underline{\lambda\vec{t}}.\lambda\vec{u}.(\mathbf{repl}\ (ax_5\,\underline{\vec{t}})\ u_i)$
$\Pi_{MN,\vec{x}} := \underline{\lambda\vec{t}}.\lambda\vec{u}.(\mathbf{repl}\ (ax_6\underline{\ulcorner M\urcorner\ulcorner N\urcorner\ulcorner\vec{x}\urcorner\vec{t}})\ (\Pi_{M,\vec{x}}\underline{\vec{t}\vec{u}}\ \underline{\ulcorner N\urcorner(\ulcorner\vec{x}\urcorner:=\vec{t})}\ (\Pi_{N,\vec{x}}\underline{\vec{t}\vec{u}})))$
$\Pi_{\lambda x.M,\vec{x}} := \underline{\lambda\vec{t}}.\lambda\vec{u}.\underline{\lambda s}.\lambda v^{\mathrm{SC}(s)}.(\Lambda_{\rho,\sigma}\underline{\ulcorner M'\urcorner\ulcorner y\urcorner\ulcorner\vec{x}\urcorner s\vec{t}}\ (\Pi_{M',y,\vec{x}}\underline{s\vec{t}}v\vec{u})v)$,

where in the last equation, we assume that $\ulcorner\lambda x.M\urcorner = \mathbb{\lambda}(\ulcorner y\urcorner,\ulcorner M'\urcorner)$, with $x:\sigma$ and $M:\rho$.

Again we extract the programs from these formal proofs. Because the realizer of **repl** is the identity, we can safely drop it from the extracted program. For terms M^σ with free variables among $\vec{x}$, each $x_i:\tau_i$, we get $\mathbf{ep}(\Pi_{M,\vec{x}}) : \vec{\tau}\to\sigma$,

$$\begin{array}{rcl}\mathbf{ep}(\Pi_{x_i,\vec{x}}) &=& \lambda\vec{x}_u.x_{u,i}\\ \mathbf{ep}(\Pi_{MN,\vec{x}}) &=& \lambda\vec{x}_u.(\mathbf{ep}(\Pi_{M,\vec{x}})\vec{x}_u(\mathbf{ep}(\Pi_{N,\vec{x}})\vec{x}_u))\\ \mathbf{ep}(\Pi_{\lambda x.M,\vec{x}}) &=& \lambda\vec{x}_u.\lambda x_v.(\mathbf{ep}(\Lambda_{\rho,\sigma})(\mathbf{ep}(\Pi_{M',y,\vec{x}})x_v\vec{x}_u)x_v),\end{array}$$

where again it is assumed that $\ulcorner\lambda x.M\urcorner = \mathbb{\lambda}(\ulcorner y\urcorner,\ulcorner M'\urcorner)$, $x:\sigma$ and $M:\rho$.

6.3.2.4 Formalization of the Theorem.

Now we are able to give a formal proof of $\exists n.\mathrm{SN}(\ulcorner M\urcorner,n)$, for any term M. Extracting the computational content of this proof, we get an upper bound for the length of reduction sequences starting from M. We will define formal proofs $\Omega_M : \exists n.\mathrm{SN}(\ulcorner M\urcorner,n)$

for each term M ($\ulcorner M \urcorner$ denotes the representation of M). Let $\vec{x}$ be the sequence of free variables in $M : \sigma$, each $x_i : \tau_i$.

$$\Omega_M := (\Phi_\sigma \underline{\ulcorner M \urcorner}(\mathbf{repl}\ (ax_\tau \underline{\ulcorner M \urcorner \ulcorner \vec{x} \urcorner})\ (\Pi_{M,\vec{x}} \underline{Var(\ulcorner \vec{x} \urcorner)} \Psi_1 \cdots \Psi_n))),$$

where $\Psi_i := (\Psi_{\tau_i} \underline{\ulcorner x_i \urcorner} \exists^+[0; (ax_1 \underline{\ulcorner x_i \urcorner})])$ is a proof of $\mathrm{SC}(Var(\ulcorner x_i \urcorner))$ (Ψ is defined in Section 6.3.2.1) and $Var(\ulcorner \vec{x} \urcorner)$ stands for $Var(\ulcorner x_1 \urcorner), \ldots, Var(\ulcorner x_n \urcorner)$. As extracted program, we get $\mathbf{ep}(\Omega_M) : \mathsf{nat}$,

$$\mathbf{ep}(\Omega_M) \quad = \quad \mathbf{ep}(\Phi_\sigma)(\mathbf{ep}(\Pi_{M,\vec{x}})(\mathbf{ep}(\Psi_{\tau_1})0) \cdots (\mathbf{ep}(\Psi_{\tau_n})0))$$

6.3.3 Comparison with Gandy's Proof

In order to compare the extracted programs from the formalized proofs with the strictly monotonic functionals used by Gandy [Gan80], we recapitulate these programs and introduce a readable notation for them.

$$\begin{array}{rcll}
\mathsf{M}_\sigma & : & \sigma \to \mathsf{nat} & := \quad \mathbf{ep}(\Phi_\sigma) \\
\mathsf{S}_0^\sigma & : & \sigma & := \quad \mathbf{ep}(\Psi_\sigma)0 \\
\mathsf{A}_{\rho,\sigma} & : & \rho \to \sigma \to \rho & := \quad \mathbf{ep}(\Lambda_{\rho,\sigma}) \\
[\![M^\sigma]\!]_{\vec{x}\mapsto\vec{t}} & : & \sigma & := \quad \mathbf{ep}(\Pi_{M,\vec{x}})\vec{t} \\
\mathrm{Upper[M]} & : & \mathsf{nat} & := \quad \mathbf{ep}(\Omega_M).
\end{array}$$

Function application is written more conventionally as $f(x)$ and some recursive definitions are unfolded. Assuming that $\sigma = \sigma_1 \to \cdots \to \sigma_n \to \mathsf{nat}$, these functionals obey the following equations:

$$\begin{array}{rcl}
\mathsf{M}_\sigma(f) & = & f(\mathsf{S}_0^{\sigma_1}, \ldots, \mathsf{S}_0^{\sigma_n}) \\
\mathsf{S}_0^\sigma(\vec{x}) & = & \mathsf{M}_{\sigma_1}(x_1) + \cdots + \mathsf{M}_{\sigma_n}(x_n) \\
\mathsf{A}_{\sigma,\tau}(f, y, \vec{x}) & = & f(\vec{x}) + \mathsf{M}_\tau(y) + 1 \\
[\![x_i]\!]_{\vec{x}\mapsto\vec{t}} & = & t_i \\
[\![MN]\!]_{\vec{x}\mapsto\vec{t}} & = & [\![M]\!]_{\vec{x}\mapsto\vec{t}}([\![N]\!]_{\vec{x}\mapsto\vec{t}}) \\
[\![\lambda x^\sigma.M^\rho]\!]_{\vec{x}\mapsto\vec{t}}(y) & = & \mathsf{A}_{\rho,\sigma}([\![M]\!]_{x,\vec{x}\mapsto y,\vec{t}}, y) \\
\mathrm{Upper}[M^\tau] & = & \mathsf{M}_\tau([\![M]\!]_{\vec{x}\mapsto\vec{\mathsf{S}}_0}).
\end{array}$$

The Correctness Theorem 6.2.4 guarantees that $\mathrm{SN}(\ulcorner M \urcorner, \mathrm{Upper}[M])$ is provable in NH, so $\mathrm{Upper}[M]$ puts an upper bound on the length of reduction sequences from M. This expression can be compared with the functionals in the proof of Gandy.

First of all, the ingredients are the same. In [Gan80] a functional (L, see Section 3.3.2) is defined, that plays the rôle of both S_0 and M (and indeed, $\mathsf{S}_0^{\sigma\to\mathsf{nat}} = \mathsf{M}_\sigma$). S is a special strictly monotonic functional and M serves as a measure on functionals. Then Gandy gives a translation M^* of a term M, by assigning the special strict functional to the free variables, and interpreting λ-abstraction by a λI term, so that reductions in the argument will not be forgotten. This corresponds to our $[\![M]\!]_{\vec{x}\mapsto\vec{\mathsf{S}}_0}$, where in the λ-case the argument is remembered by $\mathsf{A}_{\rho,\sigma}$ and eventually added to the result. Finally, Gandy shows that in each reduction step the measure of the assigned

functionals decreases. So the measure of the non-standard interpretation serves as an upper bound.

Looking into the details, there is one slight difference. The bound Upper$[M]$ is sharper than the upper bound given by Gandy. The reason is that Gandy's special functional (resembling S and M by us) is inefficient. It obeys the equation (with $\sigma = \sigma_1 \to \cdots \to \sigma_n \to \mathsf{nat}$)

$$\mathsf{L}_\sigma(x_1, \ldots, x_n) := \mathsf{L}_{\sigma_1 \to \mathsf{nat}}(x_1) + 2^0 \mathsf{L}_{\sigma_2 \to \mathsf{nat}}(x_2) + \cdots + 2^{n-2} \mathsf{L}_{\sigma_n \to \mathsf{nat}}(x_n).$$

Gandy defines L_σ with a $+$ functional on all types and a peculiar induction. By program extraction, we found functionals defined by simultaneous induction, using an extra argument as accumulator (see the definition of $\mathbf{ep}(\Phi)$ and $\mathbf{ep}(\Psi)$), thus avoiding the $+$ functional and the implicit powers of 2.

We conclude this section by stating that program extraction provides a useful tool to compare the two SN-proofs in the case of simply-typed lambda calculus. The program extracted from the decorated proof à la Tait is very similar to the upper bound expression that Gandy used.

6.4 Extension to Gödel's T

Gödel's T extends simply-typed lambda calculus with higher-order primitive recursion. The set of base types includes o, the type of natural numbers. Constants 0^o and $\mathsf{S}^{o \to o}$ are added. For each type σ, we add a constant $\mathsf{R}_\sigma : \sigma \to (o \to \sigma \to \sigma) \to o \to \sigma$. The following rules express higher-order primitive recursion:

$$\mathsf{R}_\sigma MN0 \mapsto M \qquad \text{and} \qquad \mathsf{R}_\sigma MN(\mathsf{S}P) \mapsto NP(\mathsf{R}_\sigma MNP) \ .$$

With $\to_{\beta\mathsf{R}}$ we denote the compatible closure of the β rule and the two recursion rules. It is a well known fact that $\to_{\beta\mathsf{R}}$ is a terminating rewrite relation.

The proof à la Tait of this fact (see e.g. [Tro73, 2.2.31]) extends the case of the β-rule, by proving that the new constants are strongly computable. We will present a version with concrete upper bounds. It turns out to be rather cumbersome to give a concrete number. Some effort has been put in identifying and proving the right "axioms" (Lemma 6.4.2–6.4.5) from which the decorated proof can be constructed (Lemma 6.4.6, 6.4.7). The extracted upper bounds are compared with the functionals used by Gandy (Section 6.4.4).

6.4.1 Changing the Interpretation of $\mathrm{SN}(M, n)$

Consider the following consequence of $\mathrm{SC}_{o \to o}(r)$ for fixed r. This formula is equivalent to $\underline{\forall p}.\forall m.\mathrm{SN}(p, m) \to \exists n.\mathrm{SN}(rp, n)$. So we can bound the reduction length of rp uniformly in the upper bound for p. More precisely, if $\mathrm{SN}(p, m)$ then $\mathrm{SN}(rp, [\![r]\!](m))$. A stronger uniformity principle appears in [Vrij87, § 2.3.4]).

The uniformity principle does not hold if we substitute $\mathsf{R}_o MN$ for r: Although $\mathrm{SN}(\mathsf{S}^k 0, 0)$ holds for each k, $\mathsf{R}MN(\mathsf{S}^k 0)$ can perform k reduction steps. So $\mathrm{SC}(\mathsf{R}MN)$

cannot hold. This shows that it is impossible to prove SC(R) with SC as in Definition 6.1.1. Somehow, the numerical value (k) has to be taken into account too.

To proceed, we have to change the interpretation of the predicate $\mathrm{SN}(M,n)$. We have to be a bit careful here, because speaking about *the* numerical value of a term M would mean that we assume the existence of a unique normal form. The following definition avoids this assumption:

Definition 6.4.1

1. *Second interpretation of* SN*:* $\mathrm{SN}(M,n)$ *holds if and only if for all reduction sequences of the form* $M \equiv N_0 \to_{\beta\mathsf{R}} N_1 \to_{\beta\mathsf{R}} \cdots \to_{\beta\mathsf{R}} N_m \equiv \mathsf{S}^k(P)$*, we have* $m+k \leq n$*. Note that* k *can only be non-zero for terms of type* o*.*

2. *A finite reduction sequence* $M_0 \to_{\beta\mathsf{R}} \cdots \to_{\beta\mathsf{R}} M_n$ *is* maximal *if* M_n *is normal (i.e. there is no term* N *with* $M_n \to_{\beta\mathsf{R}} N$*). An infinite reduction sequence is always maximal.*

So $\mathrm{SN}(M,n)$ means that for any reduction sequence from M to some P, n is at least the length of this sequence plus the number of leading S-symbols in P. Note that $\mathrm{SN}(M,n)$ already holds, if n bounds the length plus value of all *maximal* reduction sequences from M.

We settle the important question to what extent the proofs of Section 6.3 remain valid. Because these are formal proofs, with SN just as a predicate symbol, the derivation terms remain correct. These derivation terms contain axioms, the validity of which was shown in the intended model. But we have changed the interpretation of the predicate symbol SN. So what we have to do, is to verify that the axioms of Section 6.3.1 remain correct in the new interpretation.

The axiom schema **repl**, ax_5, ax_6 and ax_7 are independent of the interpretation of SN. Axioms 1, 2 and 3 remain true, because the terms in their conclusion have no leading S-symbols (note that 1 and 2 have a leading variable; 3 is of arrow type). Axiom 4 is proved by a slight modification of the proof of Lemma 6.1.3. The following observation is used: If $(\lambda x.M)N\vec{P} \twoheadrightarrow_{\beta\mathsf{R}} \mathsf{S}^\ell(Q)$ for some Q, then at some point we contract the outermost β-redex, say $(\lambda x.M')N'\vec{P'} \to_\beta M'[x := N']\vec{P'}$. The latter is also a reduct of $M[x := N]\vec{P}$, so ℓ is already bounded by the upper bound for the numerical value of this term.

6.4.2 Informal Decorated Proof

The goal of this section is to prove that the constants (especially R) are strongly computable. To this end, we first need some basic facts, about the length of the reduction sequences.

From now on we use r, s, t for metavariables over lambda terms as well as for formal variables ranging over lambda terms. We reserve p, q for metavariables or formal variables ranging over lambda terms of type o. $a, b, c, d, e, i, j, k, \ell, m, n$ are metavariables or formal variables over sort nat.

6.4.2.1 The Basic Facts

To prove SC(0), SC(S) and SC(R), we need some axioms, expressing basic truths about SN. In this section, $\to$ is written for $\to_{\beta\mathsf{R}}$. For 0 and S we have:

Lemma 6.4.2

1. $\mathrm{SN}(0^o, 0^{\mathsf{nat}})$
2. $\forall p.\forall m.\mathrm{SN}_o(p,m) \to \mathrm{SN}_o((\mathsf{S}p), m+1)$.

Proof: 0 is normal and has no leading S-symbols. If $(\mathsf{S}p) \to^n \mathsf{S}^k(r)$ for some n, k and r, then $p \to^n \mathsf{S}^{k-1}(r)$. From $\mathrm{SN}(p,m)$ we obtain $k+n \leq m+1$. This holds for every reduction sequence, so $\mathrm{SN}((\mathsf{S}p), m+1)$ holds. ⊠

It is less clear which facts we need for the recursion operator. To prove $\mathrm{SC}(\mathsf{R}_\sigma)$ (see Lemma 6.4.7), we need to prove $\mathrm{SC}_\sigma(\mathsf{R}stp)$ for strongly computable s, t and p. If p is strongly computable, then $\mathrm{SN}(p,m)$ holds for some m. By induction on m, we will prove $\forall p.(\mathrm{SN}(p,m) \to \mathrm{SC}_\sigma(\mathsf{R}stp))$. We need two axioms to establish the base case and the step case of this induction. For the base case, we need (schematic in the type σ):

Lemma 6.4.3
$\forall s,t,\vec{r},p,\ell,n.\ \mathrm{SN}_\iota(s\vec{r},\ell) \to \mathrm{SN}_{o\to\sigma\to\sigma}(t,n) \to \mathrm{SN}_o(p,0) \to \mathrm{SN}_\iota(\mathsf{R}_\sigma stp\vec{r}, \ell+n+1)$.

Proof: Assume $\mathrm{SN}(s\vec{r},\ell)$, $\mathrm{SN}(t,n)$ and $\mathrm{SN}(p,0)$. The latter assumption tells that p is normal and cannot be a successor. If $p \not\equiv 0$, then reductions in $\mathsf{R}stp\vec{r}$ can only occur inside s, t and $\vec{r}$, and these are bounded by $\ell+n$. Also if $p \equiv 0$, the assumptions exclude that there exists an infinite internal reduction. Hence a maximal reduction of $\mathsf{R}stp\vec{r}$ will consist of first finitely many steps within s, t and $\vec{r}$ (of respectively a, b and c steps, say) followed by an application of the first recursion rule, and finally d more steps (we show below that also d is finite). This gives a reduction of the form:

$$\mathsf{R}st0\vec{r} \to^{a+b+c} \mathsf{R}s't'0\vec{r'} \to s'\vec{r'} \to^d \mathsf{S}^i(r)$$

We can construct a reduction sequence from $s\vec{r}$ via $s'\vec{r'}$ to $\mathsf{S}^i(r)$ of length $a+c+d$. By the first assumption, $a+c+d+i \leq \ell$, by the second assumption $b \leq n$, so $a+b+c+1+d+i \leq \ell+n+1$. As this upper bound holds for an arbitrary maximal reduction sequence, it holds for all reduction sequences, so we get $\mathrm{SN}(\mathsf{R}stp\vec{r}, \ell+n+1)$. ⊠

The next lemma is needed for the step case. Note that if $\mathrm{SN}(p, m+1)$ holds, then p may reduce to either 0 (in at most $m+1$ steps) or to $(\mathsf{S}p')$ (in at most m steps). This explains the first two hypotheses of the following lemma.

Lemma 6.4.4

$$\begin{aligned}\forall s,t,\vec{r},\ell,m,n.\quad & \mathrm{SN}_\iota(s\vec{r},\ell) \to \\ & \Big(\forall q.\mathrm{SN}_o(q,m) \to \mathrm{SN}_\iota(tq(\mathsf{R}stq)\vec{r},n)\Big) \to \\ & \Big(\forall p.\mathrm{SN}_o(p,m+1) \to \mathrm{SN}_\iota(\mathsf{R}stp\vec{r},\ell+m+n+1)\Big)\end{aligned}$$

Proof: Assume $\mathrm{SN}_\iota(s\vec{r},\ell)$, $\forall q.\mathrm{SN}(q,m) \to \mathrm{SN}_\iota(tq(\mathsf{R}stq)\vec{r},n)$ and $\mathrm{SN}(p,m+1)$, for arbitrary $s,t,\vec{r},\ell,m,n$ and p. Consider an arbitrary maximal reduction sequence from $\mathsf{R}stp\vec{r}$. The assumptions exclude that there exists an infinite internal reduction. So the maximal reduction consists of finitely many reduction steps inside s, t, p and $\vec{r}$ (of a, b, c and d steps to the terms s', t', p' and $\vec{r'}$, respectively), followed by an application of a recursion rule when applicable, and concluded by some more steps. We make a case distinction to the shape of the reduct p' after the initial internal steps:

Case A: $p' \equiv 0$ Then the maximal reduction has the following shape:

$$\mathsf{R}stp\vec{r} \to^{a+b+c+d} \mathsf{R}s't'0\vec{r'} \to s'\vec{r'} \to^{e} \mathsf{S}^i(r)$$

We can construct a reduction from $s\vec{r}$ to $\mathsf{S}^i(r)$ of $a+d+e$ steps, hence, by the first assumption, $a+d+e+i \leq \ell$. From the third assumption, we get $c \leq m+1$. To bound b, we can only use the second hypothesis. Note that $\mathsf{S}^m(0)$ is normal, hence $\mathrm{SN}(\mathsf{S}^m(0),m)$ holds. Furthermore, $Rst(\mathsf{S}^m0)$ can perform at least $m+1$ reduction steps. Now we apply the second assumption to S^m0, which yields $\mathrm{SN}(t(\mathsf{S}^m0)(Rst(\mathsf{S}^m0)),n)$. This term can perform at least $b+m+1$ steps, so $b+m+1 \leq n$. Now the reduction sequence can be bounded, viz. $a+b+c+d+1+e+i \leq \ell+b+m+2 \leq \ell+n+1$.

Case B: $p' \equiv (\mathsf{S}q)$ Then the maximal reduction has the following shape:

$$\mathsf{R}stp\vec{r} \to^{a+b+c+d} \mathsf{R}s't'(\mathsf{S}q)\vec{r'} \to t'q(\mathsf{R}s't'q)\vec{r'} \to^{e} \mathsf{S}^i(r)$$

First, $\mathrm{SN}(q,m)$ holds, because if $q \to^j \mathsf{S}^k(q')$, then $p \to^{c+j} \mathsf{S}^{k+1}(q')$, so $c+j+k+1 \leq m+1$, hence $j+k \leq m$. Next note, that there is a reduction from $tq(\mathsf{R}stq)\vec{r}$ to $\mathsf{S}^i(r)$ of $a+2b+d+e$ steps. Now the second assumption can be applied, which yields that $a+2b+d+e+i \leq n$. Finally, $c \leq m$. Adding up all information, we get $a+b+c+d+1+e+i \leq m+n+1$.

Case C: If cases A and B do not apply, then p' is normal (because a maximal reduction sequence is considered), and no recursion rule applies. The reduction sequence has length $a+b+c+d$ and the result has no leading S-symbols. Now $c \leq m+1$, $a+d \leq \ell$ and $b+m+1 \leq n$ can be obtained as in Case A. Clearly $a+b+c+d \leq \ell+n$.

In all cases, the length of the maximal reduction plus the number of leading S-symbols is bounded by $\ell+m+n+1$, so indeed $\mathrm{SN}(\mathsf{R}stp\vec{r},\ell+m+n+1)$ holds. (Note that in fact we have the even sharper upper bound $n+1+\max(\ell,m)$.) ⊠

The nice point is that this lemma is $\exists$-free, so it hides no computational content. Unfortunately, it is not strong enough to enable the induction step in the proof of $\mathrm{SC}(\mathsf{R})$. We have $\forall q.\mathrm{SN}(q,m) \to \mathrm{SC}(\mathsf{R}stq)$ as induction hypothesis, and we may assume $\mathrm{SN}(p,m+1)$. In order to apply Lemma 6.4.4, we are obliged to give an n, such that $\forall q.\mathrm{SN}(q,m) \to \mathrm{SN}(tq(\mathsf{R}stq)\vec{r},n)$ holds, but using the induction hypothesis we can only find an n for each q separately.

We give two solutions of this problem. Both solutions rely on the fact that the upper bound n above doesn't really depend on q. In the formalism of Section 6.2, this is expressed by the $\underline{\forall q}$-quantifier.

The first solution uses the axioms **IP** and **IU**, which gives us the following inference:

$$\big(\underline{\forall q}.\mathrm{SN}(q,m) \to \exists n.\mathrm{SN}(tq(\mathsf{R}stq)\vec{r},n)\big) \to \big(\exists n.\underline{\forall q}.\mathrm{SN}(q,m) \to \mathrm{SN}(tq(\mathsf{R}stq)\vec{r},n)\big)$$

The advantage is that we use a general method. The disadvantage is that after removing the underlinings in the obtained derivation, the proof is no longer valid, because **IU** is clearly false after removing the underlinings. However, formally speaking this is not a problem, because the correctness proof doesn't rely on **IU** (Theorem 6.2.5).

The other solution changes Lemma 6.4.4, by relaxing the second hypothesis of it and by weakening its conclusion consequently. We then get an axiom which can be used in the proof of SC(R), but it contains existential quantifiers and consequently we have to plug in a realizer by hand. This leads to:

Lemma 6.4.5

$$\begin{aligned}\underline{\forall s,t,\vec{r},\ell,m.}\quad & \mathrm{SN}_\iota(s\vec{r},\ell) \to \\ & \Big(\underline{\forall q}.\mathrm{SN}_o(q,m) \to \exists n.\mathrm{SN}_\iota(tq(\mathsf{R}stq)\vec{r},n)\Big) \to \\ & \Big(\underline{\forall p}.\mathrm{SN}_o(p,m+1) \to \exists n.\mathrm{SN}_\iota(\mathsf{R}stp\vec{r},\ell+m+n+1)\Big)\end{aligned}$$

The justification of this lemma has to be given in terms of NH, as pointed out in Section 6.2.3. Lemma 6.4.5 contains existential quantifiers, so we have to insert a realizer. Of course we take as realizer $\lambda n.n$. Now it can be verified that

$\lambda n.n$ **mr** (Lemma 6.4.5) $\equiv$ (Lemma 6.4.4) .

Strictly speaking, we don't need a proof of this lemma, because it is realizable (by the identity). This statement corresponds to Lemma 6.4.4, so the correctness proof has already been given. But we motivated this lemma by the wish to obtain a valid proof after removing the underlining, and in that case it is important that the axioms are true (not only realizable). Therefore we prove the previous lemma, with the underlinings omitted.

Proof: Assume $\mathrm{SN}_\iota(s\vec{r},\ell)$, $\forall q.\mathrm{SN}(q,m) \to \exists n.\mathrm{SN}_\iota(tq(\mathsf{R}stq)\vec{r},n)$ and $\mathrm{SN}(p,m+1)$. Consider a reduction sequence from $\mathsf{R}stp\vec{r}$ of i steps to a term $\mathsf{S}^j(r)$, such that $i+j$ is maximal. For this sequence one of the cases A, B or C in the proof of Lemma 6.4.4 applies. In all cases we find an appropriate q with $\mathrm{SN}(q,m)$, that we can use to instantiate the second assumption. This yields an n, for which $\mathrm{SN}_\iota(tq(\mathsf{R}stq)\vec{r},n)$ holds. Now $i+j$ can be bounded by $\ell+m+n+1$, just as in the applicable case of Lemma 6.4.4. ⊠

6.4.2.2 The Constants are Strongly Computable

Eventually, we can prove that the new constants are strongly computable. The Numeral Lemma is a direct consequence of Lemma 6.4.2. The Recursor Lemma uses Lemmas 6.4.3, 6.4.5 and 6.1.2. The SC-formula is an abbreviation introduced in Section 6.3.2. The proofs below are in MF, so the underlining is important.

Lemma 6.4.6 (Numeral Lemma) $\mathrm{SC}(0)$ *and* $\mathrm{SC}(\mathsf{S})$.

Lemma 6.4.7 (Recursor Lemma) *For all* σ, $\mathrm{SC}(\mathsf{R}_\sigma)$ *is strongly computable.*

Proof: Note that R_σ has type $\sigma \to (o \to \sigma \to \sigma) \to o \to \sigma$. We assume $\mathrm{SC}(s)$, $\mathrm{SC}(t)$ and $\mathrm{SC}(p)$ for arbitrary terms s, t and p. We have to show $\mathrm{SC}_\sigma(\mathsf{R}_\sigma stp)$. From the definition of $\mathrm{SC}_o(p)$ we obtain $\exists m.\mathrm{SN}(p,m)$. Now $\forall m.\underline{\forall p}.\mathrm{SN}(p,m) \to \mathrm{SC}(\mathsf{R}stp)$ is proved by induction on m, which finishes the proof.

Case 0**:** Let $\mathrm{SN}(p,0)$. Let arbitrary, strongly computable $\vec{r}$ be given. We have to prove $\exists k.\mathrm{SN}(\mathsf{R}stp\vec{r},k)$. From $\mathrm{SC}(s)$ and $\mathrm{SC}(\vec{r})$ we get $\mathrm{SC}(s\vec{r})$, hence $\mathrm{SN}(s\vec{r},\ell)$ for some ℓ (using the definition of SC repeatedly). Lemma 6.1.2 and the assumption $\mathrm{SC}(t)$ imply $\mathrm{SN}(t,n)$ for some n. Now Lemma 6.4.3 applies, yielding $\mathrm{SN}(\mathsf{R}stp\vec{r},\ell+n+1)$. So we put $k := \ell+n+1$.

Case $m+1$**:** Assume $\underline{\forall q}.\mathrm{SN}(q,m) \to \mathrm{SC}(\mathsf{R}stq)$ (IH) and $\mathrm{SN}(p,m+1)$. Let arbitrary, strongly computable $\vec{r}$ be given. We have to prove $\exists k.\mathrm{SN}(\mathsf{R}stp\vec{r},k)$. As in Case 0, we obtain $\mathrm{SN}(s\vec{r},\ell)$ for some ℓ. In order to apply Lemma 6.4.5, we additionally have to prove $\underline{\forall q}.\mathrm{SN}(q,m) \to \exists n.\mathrm{SN}(tq(\mathsf{R}stq)\vec{r},n)$.

So assume $\mathrm{SN}(q,m)$ for arbitrary q. This implies $\mathrm{SC}(q)$ and, by IH, $\mathrm{SC}(\mathsf{R}stq)$. Now by definition of $\mathrm{SC}(t)$, we have $\mathrm{SC}(tq(\mathsf{R}stq)\vec{r})$, i.e. $\mathrm{SN}(tq(\mathsf{R}stq)\vec{r},n)$ for some n. Now Lemma 6.4.5 applies, yielding $\mathrm{SN}(\mathsf{R}stp\vec{r},\ell+m+n'+1)$ for some n'. We put $k := \ell+m+n'+1$. ⊠

The alternative proof uses 6.4.4, **IP** and **IU** instead of Lemma 6.4.5.

6.4.3 Formalized Proof

In order to formalize the proof of Section 6.4.2, we extend the language of Section 6.3.1 with constants $0^{\mathcal{T}_o}$ and infinitely many R_σ of sort $\mathcal{T}_{\sigma\to(o\to\sigma\to\sigma)\to o\to\sigma}$. Note the difference between $0^{\mathcal{T}_o}$ and 0^{nat}. Only on sort nat induction axioms will be postulated.

6.4.3.1 List of Additional Axioms

In the formalized proof, we use instances of induction (for formulae with a single realizer) and the axioms below, which are underlined versions of Lemma 6.4.2–6.4.5. Axioms 10, 11a and 11b are schematic in σ. To enhance readability, we don't write the infix application symbol $\bullet$, so e.g. $s\vec{r}$ denotes $s \bullet \vec{r}$.

8. $\mathrm{SN}(0^{\mathcal{T}_o}, 0^{\mathsf{nat}})$
9. $\underline{\forall p,m.}\ \mathrm{SN}(p,m) \to \mathrm{SN}(\mathsf{S}p, m+1)$
10. $\underline{\forall s,t,\vec{r},p,\ell,n.}\ \mathrm{SN}_\iota(s\vec{r},\ell) \to \mathrm{SN}_{o\to\sigma\to\sigma}(t,n) \to \mathrm{SN}_o(p,0) \to \mathrm{SN}_\iota(\mathsf{R}_\sigma stp\vec{r}, \ell+n+1)$

11a. $\underline{\forall s,t,\vec{r},\ell,m,n.}\quad \mathrm{SN}_\iota(s\vec{r},\ell) \to \Big(\underline{\forall q}.\mathrm{SN}_o(q,m) \to \mathrm{SN}_\iota(tq(\mathsf{R}stq)\vec{r},n)\Big) \to \Big(\underline{\forall p}.\mathrm{SN}_o(p,m+1) \to \mathrm{SN}_\iota(\mathsf{R}_\sigma stp\vec{r}, \ell+m+n+1)\Big)$

$$\text{11b.}\quad \underline{\forall s,t,\vec{r},\ell,m.}\quad \mathrm{SN}_\iota(s\vec{r},\ell)\to \Big(\underline{\forall q}.\mathrm{SN}_o(q,m)\to\exists n.\mathrm{SN}_\iota(tq(\mathsf{R}stq)\vec{r},n)\Big)\to \Big(\underline{\forall p}.\mathrm{SN}_o(p,m+1)\to\exists n.\mathrm{SN}_\iota(\mathsf{R}_\sigma stp\vec{r},\ell+m+n+1)\Big)$$

6.4.3.2 Realization of Axiom 11

By the underlining, it becomes clear that in Axiom 11b, the existentially quantified n, doesn't depend on q and p. We show that Axiom 11a is equivalent in NH to the correctness statement "the identity realizes Axiom 11b":

$$\begin{array}{lll}
\lambda n^{\mathsf{nat}}.n\ \mathbf{mr}\ \underline{\forall s,t,\vec{r},\ell,m.} & \mathrm{SN}_\iota(s\vec{r},\ell)\to & \\
& \big(\underline{\forall q}.\mathrm{SN}(q,m)\to\exists n.\mathrm{SN}_\iota(tq(\mathsf{R}stq)\vec{r},n)\big)\to & \\
& \big(\underline{\forall p}.\mathrm{SN}(p,m+1)\to\exists n.\mathrm{SN}_\iota(\mathsf{R}stp\vec{r},\ell+m+n+1)\big) & \\
\equiv\ \forall s,t,\vec{r},\ell,m. & \mathrm{SN}_\iota(s\vec{r},\ell)\to & \\
& \forall n.\Big(\ n\ \mathbf{mr}\ \big(\underline{\forall q}.\mathrm{SN}(q,m)\to\exists n.\mathrm{SN}_\iota(tq(\mathsf{R}stq)\vec{r},n)\big)\to & \\
& \quad n\ \mathbf{mr}\ \big(\underline{\forall p}.\mathrm{SN}(p,m+1)\to\exists n.\mathrm{SN}_\iota(\mathsf{R}stp\vec{r},\ell+m+n+1)\big)\Big) & \\
\equiv\ \forall s,t,\vec{r},\ell,m. & \mathrm{SN}_\iota(s\vec{r},\ell)\to & \\
& \forall n.\Big(\ \big(\forall q.\mathrm{SN}(q,m)\to\mathrm{SN}_\iota(tq(\mathsf{R}stq)\vec{r},n)\big)\to & \\
& \quad \big(\forall p.\mathrm{SN}(p,m+1)\to\mathrm{SN}_\iota(\mathsf{R}stp\vec{r},\ell+m+n+1)\big)\Big) &
\end{array}$$

6.4.3.3 Formalization of the Numeral and Recursor Lemma

We first give the formal proof Σ_0 of the formula $\mathrm{SC}(0)$, which abbreviates $\exists n.\mathrm{SN}(0,n)$:

$$\begin{array}{rcl}\Sigma_0 & := & \exists^+[0,ax_8]\\ \mathbf{ep}(\Sigma_0) & \equiv & 0\end{array}$$

Now the formal proof Σ_{S} of the formula $\mathrm{SC}(\mathsf{S})$, which after unfolding the definition of SC equals $\underline{\forall p}.(\exists m.\mathrm{SN}(p,m))\to\exists n.\mathrm{SN}(\mathsf{S}p,n)$ follows:

$$\begin{array}{rcl}\Sigma_{\mathsf{S}} & := & \underline{\lambda p}.\lambda u^{\mathrm{SC}(p)}.\exists^-[u;m;u_1^{\mathrm{SN}(p,m)};\exists^+[m+1;(ax_9\underline{pm}u_1)]]\\ \mathbf{ep}(\Sigma_{\mathsf{S}}) & \equiv & \lambda x_u.x_u+1\end{array}$$

Finally, we define formal proofs Σ_{R} of $\mathrm{SC}(\mathsf{R}_\rho)$, schematic in ρ. As mentioned in Section 6.4.2, we give two alternatives. Both use Axiom 10 in the base case of the induction. In the induction step, the first uses Axiom 11a, **IP** and **IU** and the other uses Axiom 11b. The latter one is closest to the informal proof of Section 6.4.2. To enhance readability, we will write $\underline{\lambda\vec{r}}.\lambda\vec{w}$ instead of the more correct $\underline{\lambda r_1}.\lambda w_1.\underline{\lambda r_2}.\lambda w_2.\cdots$.

$$\Sigma_{\mathsf{R}} := \underline{\lambda s}.\lambda u^{\mathrm{SC}(s)}.\underline{\lambda t}.\lambda v^{\mathrm{SC}(t)}.\underline{\lambda p}.\lambda w^{\mathrm{SC}(p)}.\exists^-[w;m;w_1^{\mathrm{SN}(p,m)};\ (ind\ Base\ Step\ m\ \underline{p}\ w_1)],$$

where ind is induction w.r.t. $\forall m.\underline{\forall p}.\mathrm{SN}(p,m)\to\mathrm{SC}_\rho(\mathsf{R}stp)$.

$$\begin{array}{rl}Base := \underline{\lambda p}.\lambda a^{\mathrm{SN}(p,0)}.\underline{\lambda\vec{r}}.\lambda\vec{w}^{\mathrm{SC}(\vec{r})}. & \\ \qquad \exists^-[(u\underline{\vec{r}}\vec{w});\ell;u_1^{\mathrm{SN}(s\vec{r},\ell)}; & \end{array}$$

$$\exists^-[(\Phi_{o\to\rho\to\rho}\underline{t}v);n;v_1^{\mathrm{SN}(t,n)};$$
$$\exists^+[\ell+n+1;(ax_{10}\underline{st\vec{r}p\ell n}u_1v_1a)]]]$$

and

$$\begin{aligned}Step :=\ &\lambda m.\lambda IH^{\underline{\forall q}.\mathrm{SN}(q,m)\to\mathrm{SC}(\mathsf{R}stq)}\\ &\underline{\lambda p}.\lambda a^{\mathrm{SN}(p,m+1)}.\underline{\lambda\vec{r}}.\lambda\vec{w}^{\mathrm{SC}(\vec{r})}.\\ &\quad\exists^-[(u\underline{\vec{r}}\vec{w});\ell;u_1^{\mathrm{SN}(s\vec{r},\ell)};\\ &\quad\exists^-[(ax_{11b}\underline{st\vec{r}\ell m}u_1\\ &\qquad\Big(\underline{\lambda q}.\lambda b^{\mathrm{SN}(q,m)}.(v\underline{q}(\exists^+[m,b])\underline{(\mathsf{R}stq)}(IH\underline{q}b)\underline{\vec{r}}\vec{w})\Big)\\ &\qquad\underline{p}a);\\ &\quad\ n;c^{\mathrm{SN}(\mathsf{R}stp\vec{r},\ell+m+n+1)};\\ &\quad\exists^+[\ell+m+n+1,c]]].\end{aligned}$$

In the alternative proof, only the induction step differs. We can exchange $Step$ for:

$$\begin{aligned}Step' :=\ &\lambda m.\lambda IH^{\underline{\forall q}.\mathrm{SN}(q,m)\to\mathrm{SC}(\mathsf{R}stq)}\\ &\underline{\lambda p}.\lambda a^{\mathrm{SN}(p,m+1)}.\underline{\lambda\vec{r}}.\lambda\vec{w}^{\mathrm{SC}(\vec{r})}.\\ &\quad\exists^-[(u\underline{\vec{r}}\vec{w});\ell;u_1^{\mathrm{SN}(s\vec{r},\ell)};\\ &\quad\exists^-[(\mathbf{IU}\ (\mathbf{IP}\ \underline{\lambda q}.\lambda b^{\mathrm{SN}(q,m)}.\Big(v\underline{q}(\exists^+[m,b])\underline{(\mathsf{R}stq)}(IH\underline{q}b)\underline{\vec{r}}\vec{w}\Big)));\\ &\qquad n;c^{\underline{\forall q}.\mathrm{SN}(q,m)\to\mathrm{SN}(tq(\mathsf{R}stq)\vec{r},n)};\\ &\quad\exists^+[\ell+m+n+1;(ax_{11a}\underline{st\vec{r}\ell mn}u_1\underline{cp}a)]]]\end{aligned}$$

The proof uses induction, so the extracted program will use recursion. The structure of the induction formula reveals that $\mathbf{ep}(ind)=\mathsf{R}_\rho$. The extracted program of the Recursor Lemma is:

$$\mathbf{ep}(\Sigma_{\mathsf{R}})\equiv\lambda x_u.\lambda x_v.\lambda x_w.(\mathsf{R}_\rho\ \mathbf{ep}(Base)\ \mathbf{ep}(Step)\ x_w),$$

where

$$\mathbf{ep}(Base)\equiv\lambda\vec{x_w}.(x_u\vec{x_w})+(\mathbf{ep}(\Phi_{o\to\rho\to\rho})x_v)+1$$

$$\mathbf{ep}(Step)=_\beta\mathbf{ep}(Step')=_\beta\lambda m.\lambda x_{IH}.\lambda\vec{x_w}.(x_u\vec{x_w})+m+(x_vmx_{IH}\vec{x_w})+1$$

The extracted program of $Step$ contains $\mathbf{ep}(ax_{11b})$, that of $Step'$ contains $\mathbf{ep}(\mathbf{IU})$ and $\mathbf{ep}(\mathbf{IP})$. All these axioms are realized by the identity, which we left out.

Remark: In [Tro73, § 2.2.18] König's Lemma (or intuitionistically the Fan Theorem) is used to prove that in the reduction tree of a strongly normalizing term, the maximal value is bounded. To avoid this, one can either prove uniqueness of normal forms, or strengthen SC by stating properties of reduction trees, which is rather cumbersome. In our proof König's Lemma is avoided by having a binary SN-predicate, which gives an upper bound on the numerical value.

6.4.4 Comparison with Gandy's Functionals

We compare the results of the program extraction, with the functionals given in [Gan80, PS95] and Section 5.3. First, we present the extracted programs in a more readable fashion. Note that the programs contain the primitive recursor R_σ, because Lemma 6.4.7 contains induction on a formula φ with $\tau(\varphi) = \sigma$. Using the notation of Section 6.3.3, the extracted functionals read:

$$\begin{array}{rcl} [\![0]\!] & = & 0 \\ [\![\mathsf{S}]\!](m) & = & m+1 \\ [\![\mathsf{R}_\sigma]\!](x,f,0,\vec{z}) & = & x(\vec{z}) + \mathsf{M}_{o\to\sigma\to\sigma}(f) + 1 \\ [\![\mathsf{R}_\sigma]\!](x,f,m+1,\vec{z}) & = & x(\vec{z}) + m + f(m,[\![\mathsf{R}_\sigma]\!](x,f,m),\vec{z}) + 1 \end{array}$$

These clauses can be added to the definition of $[\![\_]\!]$ (Section 6.3.3), which now assigns a functional to each term of Gödel's T. This also extends Upper$[_]$, which now computes the upper bound for reduction lengths of terms in Gödel's T. But, due to the changed interpretation of the SN-predicate, we know even more. In fact, Upper$[M]$ puts an upper bound on the length *plus the numerical value* of each reduction sequence. More precisely, if $M \to^i \mathsf{S}^j(N)$ then $i + j \leq \text{Upper}[M]$.

Gandy's SN-proof can be extended by giving a strictly monotonic interpretation R^* of R, such that the recursion rules are decreasing. The functional used by Gandy resembles the one above, but gives larger upper bounds. It obeys the following equations:

$$\begin{array}{rcl} \mathsf{R}^*(x,f,0,\vec{z}) & = & x(\vec{z}) + \mathsf{L}(f) + 1 \\ \mathsf{R}^*(x,f,m+1,\vec{z}) & = & f(m,\mathsf{R}^*(x,f,m),\vec{z}) + \mathsf{R}^*(x,f,m,\vec{z}) + 1. \end{array}$$

Here L is Gandy's version of the functional M (see Section 6.3.3). Clearly, the successor step of R^* uses the previous result twice, whereas $[\![\mathsf{R}]\!]$ uses it only once. Both are variants of the usual recursor. In the base case, the step function f is remembered by both. This is necessary, because the first recursor rule drops its second argument, while reductions in this argument may not be discarded. In step $m+1$ the two versions are really different; R^* adds the results of the steps $0,\ldots,m$, while $[\![\mathsf{R}]\!]$ only adds the result of step 0 and the numerical argument m. The addition of the result of step 0 is necessary to achieve monotonicity of $[\![\mathsf{R}]\!]$ in its third argument. But note that in the case that x, f and m are constantly zero, $[\![R]\!](x,f,0,\vec{z}) = [\![R]\!](x,f,1,\vec{z})$. Hence $[\![R]\!]$ is not strictly monotonic in its third argument. To amend this, we had to modify the 1 into a 2 in the proof using strict functionals (Section 5.3).

We conclude by stating that also for Gödel's T, program extraction reveals a similarity between the SN proof à la Tait and the SN proof of Gandy. However, the extracted functional from Tait's proof gives a sharper upper bound than the functional given by Gandy. Moreover, because we changed our interpretation of $\text{SN}(M,n)$ in order to verify the axioms, we know that this sharper upper bound holds for the sum of the length and the numerical value of each reduction sequence. Both results however, could have been easily obtained when using functionals directly.

6.5 Conclusion

With two case studies we showed, that modified realizability is a useful tool to reveal the similarity between SN-proofs using strong computability and SN-proofs using strictly monotonic functionals. The extra effort for Gödel's T has paid off, because we found sharper upper bounds than in [Gan80, PS95]. Moreover, the new upper bound puts a bound on the sum of the length and the numerical value of each reduction sequence. This information helps to improve the proof that uses strictly monotonic functionals (Section 5.3).

We think that our method can be applied more often. In a typical computability proof SC-predicates are defined by induction on types. It is then proved by induction on terms, that any term satisfies SC. By induction on the types, SN follows. After decorating such a proof with an administration for reduction lengths, the appropriate modified realizability interpretation maps SC-predicates to functionals of the original type and SN-predicates to numbers. The extracted program follows the induction on terms to obtain a non-standard interpretation of the term. This object is mapped to an upper bound by the proof that SC implies SN.

The realizability interpretation follows the type system closely. To deal with Gödel's T, induction was added. In the same way, conjunction and disjunction can be added to deal with products and coproducts (see also [Gan80]). Recently, Loader [Loa95] extended Gandy's proof to System F. As he points out, Girard's SN proof for System F (using reducibility candidates, see e.g. [GLT89]) can be decorated, after which modified realizability yields the same upper bound expressions. Another extension could deal with recursion over infinitely branching trees (known as Kleene's O or Zucker's T_1-trees).

A problem arises with the permutative conversions for existential quantifiers in first order logic. The semantical proof given in Section 5.5 is based on strict functionals.

Prawitz [Pra71] gives an SN-proof using strong validity (SV), see the Appendix. But the SV-predicate is defined using a general inductive definition, hence the computational contents of Prawitz's proof is not clear. Consequently, the two SN-proofs cannot be related by our method.

Appendix A

Strong Validity for the Permutative Conversions

In [Pra71] Prawitz proves strong normalization for a rewrite relation on proof trees of full classical logic. The rewrite rules include not only the usual β-rules, but also the so called permutative reductions. In this appendix we reproduce this proof, with some minor modifications. Instead of proof trees, derivation terms are used. The definition of an end segment is formalized (Definition 5). In the definition of strong validity a little deviation of the definition in [Pra71] can be found. Lemma 6, 7 and 11 are not proved in [Pra71]. Note that in Section 5.5 we give a semantical termination proof of this system without disjunction.

0. Variable conventions

u, v	: assumption variables.
d, e, f, g	: derivation terms.
x, y	: individual variables.
s, t	: individual terms.
$\vec{\alpha}$, $\vec{\beta}$	: finite sequences of variables and terms.
φ, ψ, χ	: formulae of predicate logic, without $\neg$.
$\square$	: $\{\supset, \&, \vee, \forall, \exists\}$.
i	: $\{0, 1\}$.
k, l, ℓ, m, n	: natural numbers.

1. Derivation terms and free assumptions

- u^{φ}, $\mathrm{FA}(u) = \{u\}$.
- $\supset^{+} \langle u^{\varphi}, d^{\psi} \rangle^{\varphi \to \psi}$, $\mathrm{FA}(\supset^{+} \langle u, d \rangle) = \mathrm{FA}(d) \setminus \{u\}$.
- $\supset^{-} \langle d^{\varphi \to \psi}, e^{\varphi} \rangle^{\psi}$, $\mathrm{FA}(\supset^{-} \langle d, e \rangle) = \mathrm{FA}(d) \cup \mathrm{FA}(e)$.

- $\&^+\langle d^\varphi, e^\psi\rangle^{\varphi\wedge\psi}$, $\mathrm{FA}(\&^+\langle d,e\rangle) = \mathrm{FA}(d)\cup\mathrm{FA}(e)$.
- $\&_i^-\langle d^{\varphi_0\wedge\varphi_1}\rangle^{\varphi_i}$, $\mathrm{FA}(\&_i^-\langle d\rangle) = \mathrm{FA}(d)$.
- $\vee_i^+\langle d^{\varphi_i}\rangle^{\varphi_0\vee\varphi_1}$, $\mathrm{FA}(\vee_i^+\langle d\rangle) = \mathrm{FA}(d)$.
- $\vee^-\langle d^{\varphi\vee\psi}, u^\varphi, e^\chi, v^\psi, f^\chi\rangle^\chi$,
  $\mathrm{FA}(\vee^-\langle d,u,e,v,f\rangle) = \mathrm{FA}(d)\cup(\mathrm{FA}(e)\setminus\{u\})\cup(\mathrm{FA}(f)\setminus\{v\})$.
- $\forall^+\langle x, d^\varphi\rangle^{\forall x\varphi}$, $\mathrm{FA}(\forall^+\langle x,d\rangle) = \mathrm{FA}(d)$,
  (provided $x\notin\mathrm{FV}(\psi)$, for any $u^\psi\in\mathrm{FA}(d)$).
- $\forall^-\langle d^{\forall x\varphi}, t\rangle^{\varphi[x:=t]}$, $\mathrm{FA}(\forall^-\langle d,t\rangle) = \mathrm{FA}(d)$.
- $\exists^+\langle t, d^{\varphi[x:=t]}\rangle^{\exists x\varphi}$, $\mathrm{FA}(\exists^+\langle t,d\rangle) = \mathrm{FA}(d)$.
- $\exists^-\langle d^{\exists x\varphi}, x, u^\varphi, e^\psi\rangle^\psi$, $\mathrm{FA}(\exists^-\langle d,x,u,e\rangle) = \mathrm{FA}(d)\cup(\mathrm{FA}(e)\setminus\{u\})$,
  (provided $x\notin\mathrm{FV}(\psi)$ and $x\notin\mathrm{FV}(\chi)$, for any $v^\chi\in\mathrm{FA}(e)\setminus\{u\}$).

On each line of this definition, d, e and f (if present) are immediate subderivations of the defined term, as opposed to x, u and v, which are not. In the lines defining a $\Box^-$ term, d is called the major premise of the derivation term being defined.

2. Reduction rules We want to study termination of the rewrite relation generated by the following rules.

1. $\supset^-\langle\supset^+\langle u,d\rangle, e\rangle \mapsto d[u := e]$.
2. $\&_i^-\langle\&^+\langle d_0,d_1\rangle\rangle \mapsto d_i$.
3. $\vee^-\langle\vee_i^+\langle d\rangle, u_0, e_0, u_1, e_1\rangle \mapsto e_i[u_i := d]$.
4. $\forall^-\langle\forall^+\langle x,d\rangle, t\rangle \mapsto d[x := t]$.
5. $\exists^-\langle\exists^+\langle t,d\rangle, x, u, e\rangle \mapsto e[x,u := t,d]$.
6. $\Box^-\langle\exists^-\langle d,x,u,e\rangle,\vec\alpha\rangle \mapsto \exists^-\langle d,x,u,\Box^-\langle e,\vec\alpha\rangle\rangle$.
7. $\Box^-\langle\vee^-\langle d,u,e,v,f\rangle,\vec\alpha\rangle \mapsto \vee^-\langle d,u,\Box^-\langle e,\vec\alpha\rangle, v, \Box^-\langle f,\vec\alpha\rangle\rangle$.

The right hand side of each rule is called an *immediate reduct* of the corresponding left hand side. In case this rule is among the first five, we call this reduct a *proper* reduct. In the last two cases we call it a *permutative* reduct. We write $d\to e$, if e can be obtained by replacing a subterm of d by an immediate reduct of it.

3. Lemma If $\Box^+\langle\alpha_1,\ldots,\alpha_n\rangle \to e$, then (for some $\vec\beta$ and $1\le k\le n$) $e = \Box^+\langle\beta_1,\ldots,\beta_n\rangle$, α_k is an immediate subderivation, $\alpha_k\to\beta_k$ and for all $1\le\ell\le n$ except k, $\alpha_\ell = \beta_\ell$.

4. Lemma If $d \to d'$, then $d[x, u := t, e] \to d'[x, u := t, e]$.

5. Segment relation To formalize the sentence in Prawitz: "e occurs in an end segment of d", we define an inductive binary relation ES on derivation terms: $\mathrm{ES}(d, e)$ if

1. $d = e$, or
2. $d = \exists^-\langle f, x, u, d'\rangle$ and $\mathrm{ES}(d', e)$, for some f, x, u, or
3. $d = \vee^-\langle f, u, d_0, v, d_1\rangle$ and $\mathrm{ES}(d_i, e)$, for some i, f, u, v, d_0, d_1.

6. Lemma

Let $\mathrm{ES}(d_1, \exists^+\langle t, e\rangle)$. Then for some d_2, $\exists^-\langle d_1, x, u, f\rangle \to^+ d_2$, and $\mathrm{ES}(d_2, f[x, u := t, e])$.

Proof: Induction over the definition of $\mathrm{ES}(d_1)$.

1. If $d_1 = \exists^+\langle t, e\rangle$, we can put $d_2 := f[x, u := t, e]$, for this is a proper reduct of $\exists^-\langle d_1, x, u, f\rangle$.
2. If $d_1 = \exists^-\langle g, y, v, d'\rangle$, with $\mathrm{ES}(d', \exists^+\langle t, e\rangle)$, then we have an existential permutative reduction $\exists^-\langle d_1, x, u, f\rangle \to \exists^-\langle g, y, v, \exists^-\langle d', x, u, f\rangle\rangle$. Now by the induction hypothesis we have a reduction of the last term to $\exists^-\langle g, y, v, d_2'\rangle$, with $\mathrm{ES}(d_2', f[x, u := t, e])$. This is our d_2.
3. If $d_1 = \vee^-\langle g_1, v_1, e_1, v_2, e_2\rangle$, with $\mathrm{ES}(e_1, \exists^+\langle t, e\rangle)$, then we have a permutative reduction $\exists^-\langle d_1, x, u, f\rangle \to \vee^-\langle g_1, v_1, \exists^-\langle e_1, x, u, f\rangle, v_2, \exists^-\langle e_2, x, u, f\rangle\rangle$. Now the induction hypothesis yields a reduction to $\vee^-\langle g_1, v_1, d', v_2, \exists^-\langle e_2, x, u, f\rangle\rangle$, with $\mathrm{ES}(d', f[x, u := t, e])$. This term is taken as d_2. The case $\mathrm{ES}(e_2, \exists^+\langle t, e\rangle)$ is similar. ⊠

7. Lemma If $\mathrm{ES}(d_1, \vee_i^+\langle e\rangle)$, then for some d_2, we have $\vee^-\langle d_1, v_0, e_0, v_1, e_1\rangle \to^+ d_2$, and $\mathrm{ES}(d_2, e_i[v_i := e])$.

Proof: Similar to the proof of Lemma 6. ⊠

8. Strong validity The predicate *Strongly Valid* (SV) is defined on derivations terms by the following clauses. It proceeds by induction on the formula the terms prove, and for a fixed formula it is an inductive definition.

1. $\mathrm{SV}(\&^+\langle d, e\rangle)$ if $\mathrm{SV}(d)$ and $\mathrm{SV}(e)$.
2. $\mathrm{SV}(\vee_i^+\langle d\rangle)$ if $\mathrm{SV}(d)$.
3. $\mathrm{SV}(\exists^+\langle t, d\rangle)$ if $\mathrm{SV}(d)$.

4. $\mathrm{SV}(\supset^+\langle u, d\rangle)$ if for any e with $\mathrm{SV}(e)$, $\mathrm{SV}(d[u := e])$.
5. $\mathrm{SV}(\forall^+\langle x, d\rangle)$ if for any t, $\mathrm{SV}(d[x := t])$.
6. If d is not an introduction, then $\mathrm{SV}(d)$ if
   (a) for all d' with $d \to d'$, $\mathrm{SV}(d')$;
   (b) and if $d = \vee^-\langle d_1, u_0, e_0, u_1, e_1\rangle$, then
      (i) $\mathrm{SV}(e_0)$ and $\mathrm{SV}(e_1)$, and
      (ii) if $d_1 \twoheadrightarrow d'$ and $\mathrm{ES}(d', \vee_i^+\langle e\rangle)$, then $\mathrm{SV}(e_i[u_i := e])$;
   (c) and if $d = \exists^-\langle d_1, x, u, e\rangle$, then
      (i) $\mathrm{SV}(e)$, and
      (ii) if $d_1 \twoheadrightarrow d'$ and $\mathrm{ES}(d', \exists^+\langle t, f\rangle)$, then $\mathrm{SV}(e[x, u := t, f])$.

Remark. In fact, 6.b.(ii) and 6.c.(ii) are superfluous as shown by [Joa95, p. 99]. The reason is that these parts can be proved after Lemma 11 has been proved. This not only simplifies the definition, but also the proof of Lemma 12.

9. Lemma Let $d_1 \to d_2$ and $\mathrm{SV}(d_1)$. Then $\mathrm{SV}(d_2)$.

Proof: Induction over the definition of $\mathrm{SV}(d_1)$.

1. Let $d_1 = \&^+\langle d, e\rangle$, with $\mathrm{SV}(d)$ and $\mathrm{SV}(e)$. By Lemma 3, we know that $d_2 = \&^+\langle d', e'\rangle$, with $d \to d'$ and $e = e'$, or $e \to e'$ and $d = d'$. Then by induction hypothesis, $\mathrm{SV}(d')$ and $\mathrm{SV}(e')$, so $\mathrm{SV}(\&^+\langle d', e'\rangle)$.

2,3. The cases $d_1 = \vee_i^+\langle d\rangle$ and $d_1 = \exists^+\langle t, d\rangle$ can be proved in the same way.

4. Let $d_1 = \supset^+\langle u, d\rangle$ and $\mathrm{SV}(d[u := e])$, whenever $\mathrm{SV}(e)$. By Lemma 3, $d_2 = \supset^+\langle u, d'\rangle$ and $d \to d'$. Let e be given with $\mathrm{SV}(e)$. By Lemma 4, $d[u := e] \to d'[u := e]$ and by induction hypothesis, $\mathrm{SV}(d'[u := e])$. Therefore $\mathrm{SV}(d_2)$.

5. The case $d_1 = \forall^+\langle x, d\rangle$ can be proved similarly.

6. If d_1 is not an introduction, we get $\mathrm{SV}(d_2)$ from Definition 8.6.a. ⊠

10. Theorem If $\mathrm{SV}(d)$ then $\mathrm{SN}(d)$.

Proof: Induction over the definition of $\mathrm{SV}(d)$.

1. If $d = \&^+\langle e, e'\rangle$ with $\mathrm{SV}(e)$ and $\mathrm{SV}(e')$, then by induction hypothesis, $\mathrm{SN}(e)$ and $\mathrm{SN}(e')$. An infinite rewrite sequence of d would lead to an infinite rewrite sequence in e, or in e' (by Lemma 3 and the pigeon hole principle). It follows that $\mathrm{SN}(d)$.

2,3. These cases go similarly.

4. Let $d = \supset^+ \langle u, e \rangle$. Note that SV($u$). (It is not an introduction, and it trivially satisfies 8.6.a–c). By Definition 8.4, SV(e), so SN(e) by induction hypothesis. By Lemma 3, an infinite rewrite sequence from d leads to an infinite rewrite sequence in e. It follows that SN(d).

5. Case $d = \forall^+ \langle x, e \rangle$ follows similarly.

6. Assume that d is not an introduction and SV(e), for any e with $d \to e$. By induction hypothesis, SN(e) whenever $d \to e$. But then also SN(d). ⊠

11. Lemma

1. If ES(d, e) and SV(d) then SV(e).

2. If ES(d, e) and $e \to e'$, then there exists a d', such that ES(d', e') and $d \to d'$.

3. If ES(d, e) and SN(d) then SN(e).

Proof:

1. Let ES(d, e) and SV(d). We proceed with induction over the definition of ES(d, e).

   - If $d = e$, then clearly SV(e).
   - If $d = \exists^- \langle d_1, x, u, d_2 \rangle$ and ES(d_2, e), then by Definition 8.6.c.(i) for d, SV(d_2). So the induction hypothesis applies, yielding SV(e).
   - If $d = \vee^- \langle d_1, u_0, e_0, u_1, e_1 \rangle$, and ES($e_i, e$), then by Definition 8.6.b.(i) for d, SV(e_i), and by induction hypothesis SV(e).

2. Let ES(d, e) and $e \to e'$. Again the proof is by induction on ES(d, e).

   - If $d = e$, then we can simply choose $d' := e'$.
   - If $d = \exists^- \langle d_1, x, u, d_2 \rangle$, and ES($d_2, e$), we have by induction hypothesis a term d_2', with $d_2 \to d_2'$ and ES(d_2', e). We put $d' := \exists^- \langle d_1, x, u, d_2' \rangle$.
   - Case $d = \vee^- \langle d_1, u_0, e_0, u_1, e_1 \rangle$ can be proved similarly.

3. This is a direct consequence of (2). ⊠

12. Lemma If the following conditions are satisfied, then SV($\Box^- \langle \vec{\alpha} \rangle$)

1. SN(α_ℓ), for any ℓ such that α_ℓ is an immediate subderivation.

2. If $\Box \in \{\&, \supset, \forall\}$, then SV($\alpha_\ell$), for any ℓ such that α_ℓ is an immediate subderivation.

3. If $\Box \in \{\vee, \exists\}$, then clauses 8.6.b and 8.6.c of the definition of strongly valid are satisfied.

Proof: To any derivation tree $d = \Box^-\langle\vec{\alpha}\rangle$, we assign an *induction value*, which is a triple (k, l, m). The induction values are ordered lexicographically. The components of this value are:

k = the length of the longest reduction from the major premise of d.

l = the depth of the major premise of d.

m = the sum of the lengths of the longest reductions from the immediate subderivations of d.

Let (1), (2) and (3) be satisfied for some $d = \Box^-\langle\vec{\alpha}\rangle$, with induction value (k, l, m). Note that k and m are finite, by assumption (1). We have to show 6.a–c of Definition 8, but (b) and (c) are assumed under (3). So let $d \to d'$, and our task is to prove $\mathrm{SV}(d')$. We distinguish three cases.

I. d' is obtained by reducing a proper subterm of d. Then $d' = \Box^-\langle\vec{\beta}\rangle$, with $\alpha_\ell \to \beta_\ell$ for some immediate subderivation α_ℓ. The other immediate subterms are not changed. Let (k', l', m') be the induction value of d'. If α_ℓ was the major premise of d, then $k > k'$. Otherwise, $k = k'$, $l = l'$, and $m > m'$. So the induction value is lowered, and we can use the induction hypothesis, which says that it is enough to prove (1), (2) and (3) for d'.

1. follows from (1) for d.
2. follows from (2) for d, using Lemma 9 for β_ℓ.
3. Let $d = \exists^-\langle d_1, x, u, e_1\rangle$, then $d' = \exists^-\langle d_2, x, u, e_2\rangle$, with $d_1 \to d_2$ and $e_1 = e_2$, or $d_1 = d_2$ and $e_1 \to e_2$. We have to prove 8.6.c.(i),(ii) for d'.
   (i) follows from 8.6.c.(i) for d, using Lemma 9 in case $e_1 \to e_2$.
   (ii) Let $d_2 \twoheadrightarrow d_3$, with $\mathrm{ES}(d_3, \exists^+\langle t, f\rangle)$. Then also $d_1 \twoheadrightarrow d_3$. With 8.6.c.(ii) for d, it follows that $\mathrm{SV}(e_1[x, u := t, f])$. Then also $\mathrm{SV}(e_2[x, u := t, g])$. (If $e_1 \to e_2$, we use Lemma 4 and 9).

   The case $d = \vee^-\langle d_1, u_1, e_1, u_2, e_2\rangle$ can be proved similarly.

II. d' is a proper reduct of d. In this case, $d = \Box^-\langle\Box^+\langle\vec{\beta}_1\rangle, \vec{\beta}_2\rangle$. We consider the possibilities:

1. $d = \&_i^-\langle\&^+\langle d_1, d_2\rangle\rangle$. Then we have to prove $\mathrm{SV}(d_i)$. From (2) for d, we know $\mathrm{SV}(\&^+\langle d_1, d_2\rangle)$, with Definition 8.1 it follows that $\mathrm{SV}(d_i)$.
2. $d = \supset^-\langle\supset^+\langle u, d_1\rangle, d_2\rangle$. We have to prove $\mathrm{SV}(d_1[u := d_2])$. From (2) for d, we know that $\mathrm{SV}(\supset^+\langle u, d_1\rangle)$, and $\mathrm{SV}(d_2)$. By Definition 8.4 $\mathrm{SV}(d_1[u := d_2])$ follows.
3. $d = \forall^-\langle\forall^+\langle x, d_1\rangle, t\rangle$ can be proved similarly.
4. $d = \exists^-\langle\exists^+\langle t, d_1\rangle, x, u, d_2\rangle$. We have to prove $\mathrm{SV}(d_2[x, u := t, d_1])$. From 8.6.c.(ii) for d, this follows immediately.
5. $d = \vee^-\langle\vee_i^+\langle d_1\rangle, u_1, e_1, u_2, e_2\rangle$ goes similarly.

III. d' is a permutative reduct of d. We only treat existential permutative reductions. Then $d = \Box^-\langle \exists^-\langle d_1, x, u, d_2\rangle, \vec{\beta}\rangle$, and $d' = \exists^-\langle d_1, x, u, \Box^-\langle d_2, \vec{\beta}\rangle\rangle$, with induction value (k', l', m'). Both k' and m' are finite, as will be shown soon under (1). We have to prove $\mathrm{SV}(d')$. Note that the major premise of d' is an immediate subderivation of the major premise of d. Therefore $k' \leq k$, and $l' < l$, so we can apply the induction hypothesis, which says that it suffices to prove (1), (2) and (3) for d'.

1. $\mathrm{SN}(d_1)$ follows from (1) for d. $\mathrm{SN}(\Box^-\langle d_2, \vec{\beta}\rangle)$ follows by Theorem 10 from $\mathrm{SV}(\Box^-\langle d_2, \vec{\beta}\rangle)$, which we are going to prove in (3).
2. is satisfied trivially.
3. We have to prove 8.6.c for d', so (i) $\mathrm{SV}(\Box^-\langle d_2, \vec{\beta}\rangle)$ and (ii) If $d_1 \twoheadrightarrow d_1'$, with $\mathrm{ES}(d_1', \exists^+\langle t, f\rangle)$, then $\mathrm{SV}(\Box^-\langle d_2[x, u := t, f], \vec{\beta}\rangle)$. ($\vec{\beta}$ doesn't contain x and u, because d is a correct derivation.) We first deduce some facts:

   F1. $\mathrm{SN}(\vec{\beta})$, $\mathrm{SN}(\exists^-\langle d_1, x, u, d_2\rangle)$ and $\mathrm{SN}(d_2)$. (follows from (1) for d.)

   F2. If $\Box \in \{\&, \supset, \forall\}$, then $\mathrm{SV}(\vec{\beta})$, $\mathrm{SV}(\exists^-\langle d_1, x, u, d_2\rangle)$ and $\mathrm{SV}(d_2)$. The first and second follow from (2) for d; the third follows from the second by Definition 8.6.c.(i).

   Now we are going to prove (i) and (ii).

   (i) Let (k_1, l_1, m_1) be the induction value of $\Box^-\langle d_2, \vec{\beta}\rangle$. Then $k_1 \leq k$, and $l_1 < l$, so we can apply the induction hypothesis, which grants us to prove (1), (2) and (3) for $\Box^-\langle d_2, \vec{\beta}\rangle$. (1) follows from F1 and (2) follows from F2. For (3), let $\Box \in \{\vee, \exists\}$. We only treat case $\exists$, case $\vee$ going similarly. Then $d = \exists^-\langle \exists^-\langle d_1, x, u, d_2\rangle, y, v, d_3\rangle$, and $d' = \exists^-\langle d_1, x, u, \exists^-\langle d_2, y, v, d_3\rangle\rangle$. We have to prove 8.6.c.(i) and 8.6.c.(ii) for $\exists^-\langle d_2, y, v, d_3\rangle$, or more precisely, $\mathrm{SV}(d_3)$ and if $d_2 \twoheadrightarrow d_2'$, with $\mathrm{ES}(d_2', \exists^+\langle s, g\rangle)$, then $\mathrm{SV}(d_3[y, v := s, g])$. The first follows from 8.6.c.(i) for d. The other from 8.6.c.(ii) for d, as $\exists^-\langle d_1, x, u, d_2\rangle \twoheadrightarrow \exists^-\langle d_1, x, u, d_2'\rangle$, and by Definition 5, $\mathrm{ES}(\exists^-\langle d_1, x, u, d_2'\rangle, \exists^+\langle s, g\rangle)$.

   (ii) Let $d_1 \twoheadrightarrow d_1'$, and $\mathrm{ES}(d_1', \exists^+\langle t, f\rangle)$. Let (k_2, l_2, m_2) be the induction value assigned to $\Box^-\langle d_2[x, u := t, f], \vec{\beta}\rangle$. We have the following fact, as a direct consequence of Lemma 6:

   F3. For some e, $\exists^-\langle d_1, x, u, d_2\rangle \rightarrow^+ e$ and $\mathrm{ES}(e, d_2[x, u := t, f])$.

   From F3 and Lemma 11.2, we have $k_2 < k$, so the induction hypothesis applies, and we only need to show (1), (2) and (3) for $\Box^-\langle d_2[x, u := t, f], \vec{\beta}\rangle$.

   1. $\mathrm{SN}(\vec{\beta})$ by F1, $\mathrm{SN}(d_2[x, u := t, f])$, by F3 and F1 and Lemma 11.3.
   2. Let $\Box \in \{\&, \supset, \forall\}$. Then $\mathrm{SV}(\vec{\beta})$ by F2, and $\mathrm{SV}(d_2[x, u := t, f])$, by 8.6.c.(ii) for $\exists^-\langle d_1, x, u, d_2\rangle$, which is strongly valid by F2.
   3. Let $\Box \in \{\vee, \exists\}$. Again we only do case $\exists$, as the other case goes in the same way. Then $d = \exists^-\langle \exists^-\langle d_1, x, u, d_2\rangle, y, v, d_3\rangle$,

and $d' = \exists^-\langle d_1, x, u, \exists^-\langle d_2, y, v, d_3\rangle\rangle$. We have to prove 8.6.c for $\exists^-\langle d_2[x, u := t, f], y, v, d_3\rangle$. More precisely, we have to prove (i) $\mathrm{SV}(d_3)$, and (ii) If $d_2[x, u := t, f] \twoheadrightarrow d_2'$, with $\mathrm{ES}(d_2', \exists^+\langle s, g\rangle)$, then $\mathrm{SV}(d_3[y, v := s, g])$. (i) follows from 8.6.c.(i) for d, and (ii) follows from 8.6.c.(ii) for d, if we can find an e', such that $\exists^-\langle d_1, x, u, d_2\rangle \twoheadrightarrow e'$ and $\mathrm{ES}(e', \exists^+\langle s, g\rangle)$. Take e as in F3, then from Lemma 11.2, we get e', such that $e \twoheadrightarrow e'$, and $\mathrm{ES}(e', d_2')$. By transitivity of both $\twoheadrightarrow$ and ES this e' satisfies. ⊠

13. Strong validity under substitution A *substitution* (σ) is a correctness preserving mapping which maps free term variables to terms and free assumption variables to derivation terms. A substitution is *strongly valid* ($\mathrm{SV}(\sigma)$) if it maps assumption variables to strongly valid derivations terms. A derivation term d is called *strongly valid under substitution* ($\mathrm{SV}^*(d)$) if for any strongly valid substitution σ, $\mathrm{SV}(d^\sigma)$. Clearly $\mathrm{SV}^*(d)$ implies $\mathrm{SV}(d)$.

14. Theorem $\mathrm{SV}^*(d)$.

Proof: Induction on the structure of d.

- Let $d = u$, let $\mathrm{SV}(\sigma)$. Then $d^\sigma = u$, or $d^\sigma = \sigma(u)$. The first is strongly valid, because it is normal (Definition 8.6.a); the second is strongly valid, because σ is.
- If $d = \&^+\langle d_1, d_2\rangle$, then by induction hypothesis, $\mathrm{SV}^*(d_1)$ and $\mathrm{SV}^*(d_2)$. Let $\mathrm{SV}(\sigma)$, then $\mathrm{SV}(d_1^\sigma)$ and $\mathrm{SV}(d_2^\sigma)$. By Definition 8.1, $\mathrm{SV}(\&^+\langle d_1, d_2\rangle^\sigma)$.
- $d = \exists^+\langle t, d_1\rangle$ and $d = \vee_i^+\langle d_1\rangle$ go similarly.
- If $d = \supset^+\langle u, d_1\rangle$, then by induction hypothesis, $\mathrm{SV}^*(d_1)$. So, if $\mathrm{SV}(e)$ and $\mathrm{SV}(\sigma)$, then $\mathrm{SV}(d_1^{\sigma[u:=e]})$. By Definition 8.4, $\mathrm{SV}(\supset^+\langle u, d_1\rangle^\sigma)$.
- $d = \forall^+\langle x, d_1\rangle$ goes in the same way.
- Let $d = \Box^-\langle\vec{\alpha}\rangle$. Let $\mathrm{SV}(\sigma)$. We have to prove (1), (2) and (3) of Lemma 12 for d^σ. By induction hypothesis, $\mathrm{SV}(\alpha_\ell^\sigma)$, for immediate subderivations α_ℓ. By Theorem 10, also $\mathrm{SN}(\alpha_\ell^\sigma)$. This proves (1) and (2). For (3), we have to prove 8.6.b and 8.6.c for d.

  Let $d = \exists^-\langle d_1, x, u, d_2\rangle$. By induction hypothesis, $\mathrm{SV}^*(d_2)$ and $\mathrm{SV}^*(d_1)$, so $\mathrm{SV}(d_2^\sigma)$ and $\mathrm{SV}(d_1^\sigma)$. Furthermore, if $d_1^\sigma \twoheadrightarrow d'$, with $\mathrm{ES}(d', \exists^+\langle t, e\rangle)$, then by Lemma 9, $\mathrm{SV}(d')$, by Lemma 11.1, $\mathrm{SV}(\exists^+\langle t, e\rangle)$, and by Definition 8.3, $\mathrm{SV}(e)$. We conclude that $\mathrm{SV}(d_2^\sigma[x, u := t, e])$. This proves 8.6.c.

  The case that $d = \vee^-\langle d_1, u_0, e_0, u_1, e_1\rangle$ (8.6.b) goes similarly. ⊠

15. Main theorem $\mathrm{SN}(d)$.

Proof: Theorem 10 and Theorem 14. ⊠

Bibliography

[AB91] G.J. Akkerman and J.C.M. Baeten. Term rewriting analysis in process algebra. Technical Report CS-R9130, CWI, June 1991.

[Acz78] P. Aczel. A general Church-Rosser theorem. Technical report, University of Manchester, July 1978.

[AGM92] S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors. *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[AGM94] D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors. *Proceedings of the International Workshop on Semantics of Specification Languages,* Utrecht, The Netherlands, 1993, Workshops in Computing. Springer-Verlag, 1994.

[Aka93] Y. Akama. On Mints' reduction for ccc-calculus. In Bezem and Groote [BG93], pages 1–12.

[Bak92] S. van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102(1):135–163, August 1992.

[Bar84] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, Amsterdam, second, revised edition, 1984.

[Bar92] H.P. Barendregt. Lambda calculi with types. In Abramsky et al. [AGM92], pages 117–310.

[Ber93] U. Berger. Program extraction from normalization proofs. In Bezem and Groote [BG93], pages 91–106.

[Bez86] M.A. Bezem. *Bar Recursion and Functionals of Finite Type.* PhD thesis, Utrecht University, October 1986.

[BFG94] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization and confluence in the algebraic-λ-cube. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science,* Paris, France, pages 406–415, juli 1994. To appear in the *Journal of Functional Programming.*

[BG90] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83:3–28, 1990.

[BG93] M. Bezem and J.F. Groote, editors. *Proceedings of the First International Conference on Typed Lambda Calculi and Applications,* Utrecht, The Netherlands, volume 664 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[BK84] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In *Proceedings of the* 11^{th} *ICALP,* Antwerpen, volume 172 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 1984.

[Bre88] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science,* Edinburgh, Scotland, pages 82–90, July 1988.

[Bru72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. *Indagationes Math.*, 34:381–392, 1972.

[CFC58] H.B. Curry, R. Feys, and W. Craig. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958. Second printing 1968.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CK94] R. Di Cosmo and D. Kesner. Simulating expansions without expansions. *Mathematical Structures in Computer Science*, 4:315–362, 1994.

[CR36] A. Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–116, 1987. Corrigendum: 4 (3): 409–410.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier, 1990.

[Dou92] D.J. Dougherty. Adding algebraic rewriting to the untyped lambda calculus. *Information and Computation*, 101:251–267, 1992.

[Dou93] D.J. Dougherty. Some lambda calculi with categorical sums and products. In Kirchner [Kir93], pages 137–151.

[Fer95] M.C.F. Ferreira. *Termination of Term Rewriting.* PhD thesis, Universiteit Utrecht, November 1995.

[Gan80] R.O. Gandy. Proofs of strong normalization. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477. Academic Press, London, 1980.

[Ges94] A. Geser. An improved general path order. Technical Report MIP-9407, Fakultät für Mathematik und Informatik, Universität Passau, June 1994.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[Gir87] J.-Y. Girard. *Proof theory and Logical Complexity, volume I.* Studies in Proof Theory. Bibliopolis, Napoli, 1987.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge tracts in theoretical computer science.* Cambridge University Press, Cambridge, 1989.

[GP90] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, 1990.

[GP94] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In Andrews et al. [AGM94], pages 232–251.

[HL78] G. Huet and D.S. Lankford. On the uniform halting problem for term rewriting systems. Technical Report Rapport Laboria 283, INRIA, 1978.

[HMMN94] J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors. *Proceedings of the First International Workshop on Higher-Order Algebra, Logic and Term Rewriting,* Amsterdam, The Netherlands, HOA '93, volume 816 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[HO80] G. Huet and D. Oppen. Equations and rewrite rules – a survey. In *Formal Language Theory – Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.

[Hsi95] J. Hsiang, editor. *Proceedings of the Sixth International Conference on Rewriting Techniques and Applications,* Kaiserslautern, Germany, volume 914 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995.

[JO91] J.-P. Jouannaud and M. Okada. Executable higher–order algebraic specification languages. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science,* Amsterdam, The Netherlands, pages 350–361, 1991.

[Joa95] Felix Joachimski. Kontrolloperatoren und klassische Logik. Master's thesis, Mathematisches Institut der Universität München, 1995.

[JR96] J.-P. Jouannaud and A. Rubio. A recursive path ordering for higher-order terms in η-long β-normal form. In *Proceedings of the Seventh International Conference on Rewriting Techniques and Applications,* New Brunswick, NJ, USA, pages 108–122, 1996.

[Kah95] S. Kahrs. Towards a domain theory for termination proofs. In Hsiang [Hsi95], pages 241–255.

[Kah96] S. Kahrs. Termination proofs in an abstract setting. Obtainable via `http://www.dcs.ed.ac.uk/generated/home-links/smk/`, 1996.

[Kir93] C. Kirchner, editor. *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications,* Montreal, Canada, volume 690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[KL80] S. Kamin and J.J. Lévy. Two generalizations of the recursive path ordering. Technical report, University of Illinois, 1980.

[Klo80] J.W. Klop. *Combinatory Reduction Systems.* PhD thesis, Rijksuniversiteit Utrecht, Amsterdam, 1980.

[Klo92] J.W. Klop. Term rewriting systems. In Abramsky et al. [AGM92], pages 1–116.

[KOR93] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theoretical Computer Science*, 121(1–2):279–308, December 1993.

[Kre59] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland, 1959.

[Loa95] R. Loader. Normalisation by translation. Note distributed on "types" mailing list. Obtainable via `http://sable.ox.ac.uk/~loader/`, April 1995.

[Lor94] C.A. Loria-Saenz. *A Theoretical Framework for Reasoning about Program Construction based on Extensions of Rewrite Systems.* PhD thesis, Fachbereich Informatik der Universität Kaiserslautern, 1994.

[LP95] O. Lysne and J. Piris. A termination ordering for higher order rewrite systems. In Hsiang [Hsi95], pages 26–40.

[LS93] C.A. Loria-Saenz and Joachim Steinbach. Termination of combined (rewrite and λ-calculus) systems. In Rusinowitch and Rémy [RR93], pages 143–147.

[Mid89] A. Middeldorp. A sufficient condition for the termination of the direct sum of term rewriting systems. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science ,* Pacific Grove, pages 396–401, 1989.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, 1989*, volume 475 of *Lecture Notes in Computer Science*, pages 253–281. Springer-Verlag, 1991.

[MOZ96] A. Middeldorp, H. Ohsaki, and H. Zantema. Transforming termination by self-labelling. Technical Report UU-CS-1996-15, Utrecht University, 1996. To appear in CADE-13.

[Ned73] R.P. Nederpelt. *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types.* PhD thesis, Eindhoven Technological University, The Netherlands, 1973.

[New42] M.H.A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[Nip91] T. Nipkow. Higher-order critical pairs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science,* Amsterdam, The Netherlands, pages 342–349, 1991.

[Nip93] T. Nipkow. Orthogonal higher-order rewrite systems are confluent. In Bezem and Groote [BG93], pages 306–317.

[Oos94] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting.* PhD thesis, Vrije Universiteit, Amsterdam, 1994.

[OR94] V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In Heering et al. [HMMN94], pages 276–304.

[Pol94] J.C. van de Pol. Termination proofs for higher-order rewrite systems. In Heering et al. [HMMN94], pages 305–325.

[Pol96] J.C. van de Pol. Two *different* strong normalization proofs? Computability versus functionals of finite type. In G. Dowek, J. Heering, K. Meinke, and B. Möller, editors, *Proceedings of the Second International Workshop on Higher-Order Algebra, Logic and Term Rewriting,* Paderborn, Germany, HOA '95, volume 1074 of *Lecture Notes in Computer Science*, pages 201–220. Springer-Verlag, 1996.

[Pra71] D. Prawitz. Ideas and results in proof theory. In Jens Erik Fenstad, editor, *Proc. of the Second Scandinavian Logic Symposium*, pages 235–307, Amsterdam, 1971. North–Holland.

[PS95] J.C. van de Pol and H. Schwichtenberg. Strict functionals for termination proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications,* Edinburgh, Scotland, volume 902 of *Lecture Notes in Computer Science*, pages 350–364. Springer-Verlag, 1995.

[Raa96] F. van Raamsdonk. *Confluence and Normalisation for Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1996.

[RR93] M. Rusinowitch and J.-L. Rémy, editors. *Proceedings of the Third International Workshop on Conditional Term Rewriting Systems,* Pont-à-Mousson, France, CTRS '92, volume 656 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[Rus87] M. Rusinowitch. On termination of the direct sum of term rewriting systems. *Information Processing Letters*, 26:65–70, 1987.

[Sel94] M.P.A. Sellink. Verifying process algebra proofs in type theory. In Andrews et al. [AGM94], pages 315–339.

[Sør96] M.H. Sørensen. Strong normalization from weak normalization in typed lambda-calculi. Technical report, University of Copenhagen, Denmark, 1996. Submitted for publication; obtainable via `http://www.diku.dk/research-groups/topps/personal/rambo.html`.

[Tai67] W.W. Tait. Intensional interpretation of functionals of finite types I. *Journal of Symbolic Logic*, 32:198–212, 1967.

[Toy87] Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25:141–143, 1987.

[Tro73] A.S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Number 344 in LNM. Springer Verlag, Berlin, 1973. Second corrected edition appeared as report ILLC X-93-05, University of Amsterdam, 1993.

[Vrij87] R. de Vrijer. Exactly estimating functionals and strong normalization. *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, 90(4):479–493, December 1987.

[Wol93] D.A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, Cambridge, 1993.

[Zan94] H. Zantema. Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17:23–50, 1994.

[Zan95] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

Index

Samenvatting

Herschrijven en Terminatie

Het woord *herschrijven* suggereert een berekeningsproces. De objecten die berekend worden zijn uitdrukkingen in een bepaalde formele taal. Een berekening bestaat uit een aantal stappen. In elke stap wordt een deel van de uitdrukking vervangen door een andere, al naar gelang de regels dit toestaan. Deze regels variëren van systeem tot systeem. Met een herschrijfsysteem bedoelen we een collectie van regels. Op deze wijze wordt een herschrijfrij $s_0 \to s_1 \to s_2 \to \cdots$ gevonden.

In principe kan een herschrijfrij oneindig lang worden. De berekening geeft dan geen resultaat. De berekening kan ook termineren. Dat gebeurt wanneer we een s_n vinden waarop geen enkele regel meer van toepassing is. In dat geval is de herschrijfrij eindig. Zo'n s_n noemen we een *normaalvorm*. De gevonden normaalvorm is het resultaat van de berekening.

Een interessante vraag die we bij een herschrijfsysteem kunnen stellen, is of de regels een oneindige herschrijfrij toestaan, of niet. We zeggen dat het systeem *terminerend* is, als alle herschrijfrijen eindig zijn. In een terminerend herschrijfsysteem kan iedere uitdrukking berekend worden. We lopen daarbij niet het gevaar in een oneindige berekening te belanden. Hoe vervelend dat laatste is, weet iedereen die zelf geprogrammeerd heeft. Stel namelijk dat een bepaalde berekening al erg lang duurt. Het is dan moeilijk uit te maken of we nog wat langer moeten wachten, of dat er werkelijk een oneindige lus in het programma zit. Dit voorbeeld toont ook aan dat het in het algemeen moeilijk is om terminatie te bewijzen, want een test geeft geen uitsluitsel.

Twee andere interessante eigenschappen die een herschrijfsysteem kan hebben, zijn confluentie en zwakke terminatie. In het vervolg schrijven we $s \twoheadrightarrow t$ wanneer s in nul, één of meer stappen naar t herschreven kan worden. *Zwakke terminatie* geldt, als er voor iedere uitdrukking s een normaalvorm t is, zodanig dat $s \twoheadrightarrow t$. Dus ook bij zwakke terminatie kan iedere uitdrukking berekend worden. Echter, naast deze ene succesvolle berekening kan er ook heel goed een oneindige berekening zijn die in s begint. Om dit uit te sluiten is (echte) terminatie nodig. Terminatie impliceert zwakke terminatie.

Een systeem heet *confluent*, als er voor iedere r, s, t waarvoor $r \twoheadrightarrow s$ en $r \twoheadrightarrow t$ geldt, een u bestaat met de eigenschap $s \twoheadrightarrow u$ en $t \twoheadrightarrow u$. In woorden betekent dit dat als

er twee verschillende berekeningen in r starten, we die altijd zo kunnen voortzetten dat ze weer bij elkaar komen. Deze eigenschap is belangrijk, omdat hij garandeert dat normaalvormen uniek zijn. (Als s en t hierboven normaalvormen zijn, dan moet u in 0 stappen uit s volgen, dus u en s zijn gelijk. Evenzo zijn u en t aan elkaar gelijk, dus s en t zijn dezelfde normaalvorm). Zwakke terminatie garandeert alleen dat iedere uitdrukking minstens één normaalvorm heeft; confluentie garandeert dat iedere uitdrukking hoogstens één normaalvorm heeft.

Er is ook een begrip *zwak confluent*. Een herschrijfsysteem is zwak confluent als er voor iedere r, s, t waarvoor $r \to s$ en $r \to t$ geldt, een u bestaat met de eigenschap $s \twoheadrightarrow u$ en $t \twoheadrightarrow u$. De gemeenschappelijke opvolger wordt nu dus alleen gegarandeerd na een éénstapsberekening. Zwakke confluentie impliceert niet dat de normaalvormen uniek zijn. Het is wel gemakkelijker aan te tonen dan confluentie, omdat we geen divergente herschrijfrijen hoeven te beschouwen, maar alleen divergente herschrijfstappen.

De vraag rijst nu, waarom we ons met terminatie bezighouden. Immers, in een *zwak* terminerend en confluent herschrijfsysteem heeft iedere uitdrukking een unieke normaalvorm. Er zijn twee belangrijke redenen. Een praktische reden is, dat een zwak terminerend systeem ook oneindige herschrijfrijen toelaat. Daarom is er nog een strategie nodig om de juiste berekening te vinden. Bij een terminerend systeem kunnen we onbekommerd herschrijfstappen toepassen; dit proces stopt altijd. De theoretische reden om terminatie te beschouwen is een oud resultaat (1942) dat terminatie en zwakke confluentie samen confluentie impliceren. Dus voor een terminerend systeem is zwakke confluentie voldoende om het bestaan van unieke normaalvormen te garanderen.

Hogere orde herschrijfsystemen

Herschrijfsystemen kunnen worden ingedeeld naar de formele taal waaruit de uitdrukkingen genomen worden. In *termherschrijfsystemen* zijn de uitdrukkingen eerste orde termen. Die worden gebouwd uit functiesymbolen (met een vast aantal argumenten) en variabelen. Als voorbeeld nemen we een paar operaties op getallen. Om met eindig veel functiesymbolen alle natuurlijke getallen te kunnen representeren, voeren we de symbolen 0 (geen argumenten) en s (successor, één argument) in. Het getal 3 is de derde successor van 0, en wordt dus gerepresenteerd als $s(s(s(0)))$. Verder gebruiken we a voor optellen (twee argumenten). We kunnen nu optellen definiëren door de volgende regels:

$$\begin{cases} a(X,0) & \mapsto & X \\ a(X,s(Y)) & \mapsto & s(a(X,Y)) \end{cases}$$

Hierin zijn X en Y variabelen waar willekeurige termen voor ingevuld mogen worden. Een herschrijfstap ontstaat door een deel van een expressie waar de linkerkant van een regel op past, te vervangen door de corresponderende rechterkant. Voor de variabelen in de regel mogen daarbij termen ingevuld worden. Als voorbeeld controleren we of $2 + 1 \twoheadrightarrow 3$:

$$a(s(s(0)), s(0)) \to s(a(s(s(0)), 0)) \to s(s(s(0)))$$

Een ander formalisme is de *lambdacalculus*. Dit is een formalisme voor het manipuleren van functies. De twee manieren om lambdatermen samen te stellen komen overeen met de volgende operaties op functies: Een functie toepassen op een argument (applicatie) en een functie definiëren door middel van een voorschrift (abstractie). De notatie is als volgt: MN betekent de toepassing van M op N; $\lambda x.M$ betekent de functie die x naar M stuurt; hierbij mag de variabele x in M voorkomen. Het verband tussen applicatie en abstractie wordt gelegd door de β-regel:

$$(\lambda x.M)N \quad \mapsto \quad M[x := N]$$

In woorden: het resultaat van de functie die x op M afbeeldt, toegepast op N is M, waarin alle voorkomens van x vervangen zijn door N. Een voorbeeld van de toepassing van de β-regel is: $(\lambda x.x^2 + 8 \cdot x)3 \to 3^2 + 8 \cdot 3$. We werken alleen met getypeerde lambdatermen. Een van de redenen is dat de β-regel alleen termineert, wanneer we de verzameling lambdatermen beperken tot getypeerde termen. De types leggen het domein en bereik van de functies vast.

Hogere orde herschrijfsystemen combineren termherschrijfsystemen met lambdacalculus. Ze vormen op twee manieren een uitbreiding van termherschrijfsystemen. In de eerste plaats kunnen voor variabelen nu ook functies ingevuld worden; niet alleen termen. Verder kunnen in deze systemen gebonden variabelen voorkomen. Door deze verhoogde uitdrukkingskracht kunnen in hogere orde herschrijfsystemen ook transformaties op programma's (met locale variabelen), formules (met gekwantificeerde variabelen) en bewijzen (met variabelen voor abstracties) beschreven worden. Dit vergroot het toepassingsgebied van herschrijfsystemen aanzienlijk.

Een voorbeeld van een functie die gebruik maakt van functievariabelen is d(ubbel):

$$d(F, X) \quad \mapsto \quad F(F(X))$$

Een voorbeeld van een functie die gebonden variabelen gebruikt is:

$$\begin{cases} \sum_{i \le 0} E & \mapsto & E[i := 0] \\ \sum_{i \le s(n)} E & \mapsto & a(\sum_{i \le n} E, E[i := s(n)]) \end{cases}$$

De eerste functie past zijn eerste argument tweemaal toe op het tweede argument. De andere functie sommeert de waarden van de expressie $E[i]$ voor $0 \le i \le n$. In hogere orde herschrijven wordt $E[i]$ opgevat als functie, namelijk $\lambda i.E$. De substitutie $E[i := 0]$ kunnen we dan eenvoudig opvatten als de applicatie $(\lambda i.E)0$. De β-regel wordt gebruikt om de substitutie echt uit te voeren.

Terminatie via de semantische methode

Een interpretatie van een herschrijfsysteem is een verzameling A, met voor ieder functiesymbool f een functie over A met het juiste aantal argumenten. Elke term kan nu geïnterpreteerd worden als een element in A (gegeven een waarde voor de variabelen). We schrijven $[\![t]\!]$ voor de interpretatie van t in A. Zo'n interpretatie is een *model* als iedere herschrijfregel na interpretatie een ware gelijkheid is.

Het (bestaande) idee van een semantisch terminatiebewijs is nu om een interpretatie te zoeken, waarin de linkerkant van elke regel groter is dan de rechterkant (in plaats van gelijk aan). Extra eisen zijn dat de ordening geen oneindig dalende rij mag bevatten, en dat de gebruikte functies strikt monotoon moeten zijn. Dat laatste wil zeggen, dat als $x > y$ dan $f(\cdots x \cdots) > f(\cdots y \cdots)$. Zo'n interpretatie noemen we een terminatiemodel. Dit bestaat dus uit de volgende ingrediënten:

1. Een welgefundeerde partiële ordening $(A, >)$.

2. Voor ieder functiesymbool f een strikt monotone functie over A met het juiste aantal argumenten.

3. Voor iedere regel $l \mapsto r$ geldt dat $[\![l]\!] > [\![r]\!]$, dit laatste onder de interpretatie gegeven onder 2 en voor iedere mogelijke invulling van de variabelen.

We bewijzen nu dat een termherschrijfsysteem dat een terminatiemodel heeft terminerend moet zijn. Wegens (2) en (3) gaat bij iedere herschrijfstap de bijbehorende interpretatie omlaag. Hierbij is (2) vereist, omdat een herschrijfstap ook een deel van een term kan vervangen. De context waarin dit gebeurt moet de ordening respecteren. Zodoende correspondeert iedere herschrijfrij met een evenlange dalende keten in A. Wegens (1) is deze keten eindig, dus de oorspronkelijke herschrijfrij is ook eindig.

We verduidelijken de methode door te bewijzen dat de regels voor optelling een terminerend systeem vormen. Beschouw als ordening de natuurlijke getallen met de gebruikelijke groter-dan relatie. Voor $0, s, a$ kiezen we de volgende interpretatie:

$$\begin{aligned} [\![0]\!] &= 1 \\ [\![s]\!](x) &= x + 1 \\ [\![a]\!](x, y) &= x + 2y \end{aligned}$$

Deze functies zijn strikt monotoon, en $>$ is welgefundeerd. We hoeven dus alleen nog maar (3) te controleren. Dit kan gemakkelijk gedaan worden:

$$[\![a(x, 0)]\!] = x + 2 > x = [\![x]\!]$$

$$[\![a(x, s(y))]\!] = x + 2(y + 1) > x + 2y + 1 = [\![s(a(x, y))]\!]$$

Dus het termherschrijfsysteem dat optelling definiëert is terminerend.

Omdat hogere orde herschrijfsystemen op lambdatermen werken, moeten de terminatiemodellen voor dergelijke systemen op functies gebaseerd zijn. Dit proefschrift bevat een geschikte uitbreiding van het begrip strikt monotoon tot functies van hogere types. Ook wordt een willekeurige partiële ordening op A uitgebreid tot een partiële ordening op functies over A. Zo wordt het begrip terminatiemodel ook toepasbaar op hogere orde herschrijfsystemen. We bewijzen dat een hogere orde herschrijfsysteem termineert als het een terminatiemodel heeft.

Resultaten van dit Proefschrift

1. We introduceren het begrip terminatiemodel en bewijzen dat een hogere orde herschrijfsysteem termineert als het een terminatiemodel heeft.

2. Hieruit kan gemakkelijk een methode gedestilleerd worden om terminatie van hogere orde herschrijfsystemen te bewijzen.

3. Deze methode wordt ondersteund door een scala aan rekenregels op functies van hoger type.

4. De methode wordt toegepast op verschillende niet triviale voorbeelden. De belangrijkste zijn Gödel's T en een systeem dat natuurlijke deducties normalizeert; dit laatste systeem bevat ook de gecompliceerde permutatieve conversies.

5. De hoofdstelling levert een nieuw bewijs van de stelling, dat een terminerend termherschrijfsysteem uitgebreid met β-reductie in de getypeerde lambdacalculus weer termineert.

6. De semantische methode wordt uitgebreid vergeleken met een bestaande methode, die gebaseerd is op het begrip *sterke berekenbaarheid*.

Curriculum Vitae

- Geboren op 6 april 1969 te Barneveld.
- 1981–1987: Ongedeeld VWO aan de Van Lodenstein Scholengemeenschap te Amersfoort.
- 1987–1992: Studie Informatica aan de Rijksuniversiteit Utrecht. Cum Laude afgestudeerd op het onderwerp "Modularity in many-sorted term rewriting systems". Tweejarig studentassistentschap "begeleiding van buitenlandse studenten".
- 1992–1996: AIO bij de Faculteit der Wijsbegeerte aan de Universiteit Utrecht.
- 1994: Wetenschappelijk onderzoeker aan het Mathematisches Institut van de Ludwig–Maximilians–Universität te München.
- 1996 – : Postdoc bij de Technische Universiteit Eindhoven, Faculteit Wiskunde en Informatica, sectie Technische Toepassingen. Onderzoek naar formele methoden voor de specificatie van systeemeisen aan Command & Control systemen.

Jaco van de Pol