T U M

I N S T I T U T F Ü R I N F O R M A T I K

Solving Higher-Order Equations: From Logic to Programming

Christian Prehofer

TUM-I9508
März 1995

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-03-1995-I9508-350/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

Solving Higher-Order Equations: From Logic to Programming

Christian Prehofer

Fakultät für Informatik
der Technischen Universität München

Solving Higher-Order Equations: From Logic to Programming

Christian Prehofer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Wilfried Brauer

Prüfer der Dissertation:

1. Univ.-Prof. Tobias Nipkow, Ph.D.
2. Univ.-Prof. Dr. Harald Ganzinger

Die Dissertation wurde am 15. November 1994 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 6. Februar 1995 angenommen.

Abstract

Higher-order constructs provide the necessary level of abstraction for concise and natural formulations in many areas of computer science. We present constructive methods for higher-order equational reasoning with applications ranging from theorem proving to novel programming concepts. A major problem of higher-order programming is the undecidability of higher-order unification. In the first part, we develop several classes with decidable second-order unification. As the main result, we show that the unification of a linear higher-order pattern s with an arbitrary second-order term that shares no variables with s is decidable and finitely solvable. This is the unification needed for second-order functional-logic programming.

The second main contribution is a framework for solving higher-order equational problems by narrowing. In the first-order case, narrowing is the underlying computation rule for the integration of logic programming and functional programming. We argue that there are some principal problems with lifting the standard notion of first-order narrowing to the higher-order case. In contrast, the alternative approach, lazy narrowing, solves goals in a top-down manner and can be adapted to the higher-order case. Several refinements that utilize the deterministic evaluation of functional programs, such as normalization, are developed for this approach. We further introduce a restricted class of equational goals that suffices for programming applications. This class, called Simple Systems, enjoys decidable unification in the second-order case, using the results of the first part. It facilitates several other optimizations, e.g. recognizing solved system is simple. Integrating these refinements leads to a new narrowing strategy where intermediate goals can safely be delayed and are only solved when needed.

This work forms a new basis for truly higher-order functional-logic programming that is oriented more towards higher-order functional programs than to horn clauses as in logic programming. We argue that many techniques of first-order (functional-)logic programming can be modeled more directly in our higher-order functional approach.

Acknowledgments

I wish to take the opportunity to sincerely thank Tobias Nipkow for his continuous support. His criticism has always been a source of motivation and inspiration.

I am grateful to many friends and colleagues for comments and discussions on the subject. They include Olaf Müller, Michael Kohlhase, Joachim Niehren, Jaco van de Pol, Heinrich Hußmann, Oscar Slotosch, Cornel Klein, Konrad Slind, Max Moser, Andreas Werner, Mario Rodríguez-Artalejo, Michael Hanus, Gilles Dowek, Vincent van Oostrom, and Gérard Huet. Furthermore, I wish to thank Robert Furtner for his efforts on implementing parts of this thesis and for valuable feedback.

I am indebted to Manfred Broy and Tobias Nipkow for providing the fruitful environment that made this thesis possible. This includes many others of the Munich research group as well.

The contributions of several others are less direct. Alan Frisch made me enjoy the art of (scientific) writing and Nachum Dershowitz introduced me to term rewriting.

Finally, I want to thank my relatives and friends, particularly Andrea, for enduring me on this adventure.

To Andrea

The Emperor counsels simplicity.
First principles.
Of each particular thing, ask:
What is it in itself,
in its own constitution?
What is its causal nature?

Dr. Hannibal Lecter, in *The Silence of the Lambs*,
Thomas Harris

Contents

List of Figures

Chapter 1

Main Goals and Results

Higher-order constructs provide the necessary level of abstraction for concise and natural formulations in many areas of computer science. Examples are functional programming [MTH90, HJW92, Pau91, Ste90] and specification [Bro88, Möl86], program transformation and synthesis [HL78, Hag91b], machine learning [Har90, DW88], and theorem proving systems, e.g. [AINP90, Pau94, CAB+86, DFH+93]. The goal of this thesis is to develop tractable refinements for higher-order equational reasoning that are suitable for programming. The major application we aim for is declarative programming, in particular the integration of logic and functional programming on a higher-order basis. Figure 1.1 gives an overview of existing programming paradigms. Narrowing provides a nice generalization of both logic and functional programming. This approach has been examined extensively for the first-order case and has led to several implementations (for a survey see [Han94b]).

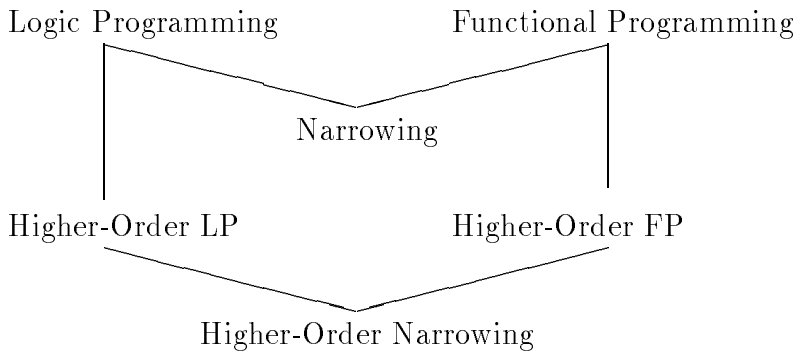

Figure 1.1: Declarative Programming Paradigms

Whereas higher-order programming is standard in functional programming, logic programming is in large parts still tied to the first-order world. Only a few languages, most notably λ-Prolog, are fully higher-order. The language λ-Prolog pioneered in the use of higher-order unification for logic programming and has shown its practical utility despite its undecidability.

This gap has been recognized and many higher-order extensions of functional-logic languages [BG86, CKW89, GMHGRA92, Loc93, She90] have been developed. To our knowledge, however, all of these are limited to first-order unification and are not com-

plete in a higher-order sense. Several works [Smo86, Loc93, SJ92] on functional-logic programming explicitly state that second-order unification cannot be used due to its undecidability. We prove this to be wrong for the context of functional-logic programming.

This work develops the foundations of truly higher-order functional-logic programming. The main steps towards this goal are as follows:

- Decidable second-order unification problems that can be applied to functional-logic programming.
- A framework for higher-order narrowing with first completeness results.
- Several refinements of narrowing, e.g. deterministic simplification, conditional narrowing.
- A class of equational goals that suffices for higher-order functional-logic programming and which enjoys several optimizations.

This work forms a new basis for truly higher-order functional-logic programming that is more oriented towards higher-order functional programs than to first-order horn clauses. This programming paradigm not only supports applicative higher-order programming, but in addition new functional objects can be computed by unification with logic programming techniques. Its practical use is shown by several examples. These include high-level programming, automating mathematics, and program transformation. A similar language is described nicely in [Llo94] by many examples.

Compared to higher-order logic programming, where higher-order λ-terms only serve as data structures, our functional setting supports higher-order programming as in functional languages directly. It furthermore enjoys two sources of optimizations:

- Left-linearity of rewrite rules can be exploited and for instance leads to decidable unification in the second-order case and to a new strategy for needed narrowing, where intermediate goals are only solved when needed.
- Convergent systems allow the restriction to normalized solutions, facilitating deterministic operations.

More detailed overviews of the results can be found at the beginning of each chapter. The structure of the work is as follows. The next chapter gives an informal outline of this work. Chapter 3 presents simply typed λ-calculus and other basic preliminaries. An introduction to higher-order unification and term rewriting follows in Chapter 4. Chapter 5 develops decidable classes of second-order unification. Higher-order narrowing is the subject of Chapter 6 and can be read, with a few exceptions, independently of Chapter 5. This is followed by examples for higher-order narrowing in Chapter 7 and concluding remarks in Chapter 8.

Chapter 2

Introduction and Overview

In the following, we informally introduce the main topics and outline the main results of this work. We proceed from first-order term rewriting and narrowing to higher-order unification and higher-order narrowing.

2.1 Term Rewriting

Term rewriting is a model of computation. Rewriting is based on the idea of "replacing equals by equals". Following this idea, equations between terms are oriented into rewrite rules. For instance, the equations $0 + Z = Z$ and $X + succ(Y) = succ(X + Y)$ form a specification of the function $+$, assuming the term constructors 0 and $succ$. Let us orient the equations into the following two rules

$$R = \left\{ \begin{array}{rcl} 0 + Z & \rightarrow & Z \\ X + succ(Y) & \rightarrow & succ(X + Y) \end{array} \right\}$$

With orientation, we gain an operational model: reduction. We can reduce a term with the rules of R, e.g.:

$$a + succ(0 + b) \longrightarrow a + succ(b) \longrightarrow succ(a + b)$$

where a and b are some constants. These two reduction steps reduce $a + succ(0 + b)$ to its normal form $succ(a + b)$ wrt. R. Notice that this is an abstract model of computation; it is not directly used as a programming language since reduction is in general not deterministic. Programming languages usually restrict rewrite systems to be confluent, which implies that normal forms are unique, e.g. $succ(a + b)$ above. Then it suffices to perform only a particular reduction strategy.

Term rewriting as an abstract model is, for instance, useful for symbolic reasoning with equations and for the analysis of programming languages. In particular, term rewriting is an abstract model of first-order functional programming languages. Current languages such as LISP variants [Ste90], Haskell [HJW92] or SML [MTH90] are higher-order and originate from the λ-calculus. In such languages, reduction to normal form is called evaluation.

The simplicity of the concept of term rewriting has attracted much research, concerning properties of rewrite systems such as termination or confluence. For surveys we refer to [DJ90, Klo92]. Apart from programming, well developed applications are theorem

proving, both automatic systems [Hsi85] and interactive systems [Gor88, Pau94], program synthesis via completion [Bac91] and algebraic specifications [GTW89, EM85, FH91].

2.2 Narrowing

Starting from an equational specification, it is often not only desirable to evaluate terms, but also to solve equations. For instance, with the rules of R, a simple goal is to ask for what values of X the equation

$$succ(X + a) =^{?}_{R} succ(a)$$

holds. *Narrowing* is a general mechanism for solving such goals in a systematic way. The idea is to find values for X by unification. Whereas term rewriting searches for matches of a rule, narrowing uses *unification* to find an instance of a term such that a rewrite step applies.

For instance, unifying the left-hand side of the first rule of R, $0 + Z$, with $X + a$ yields a solution by the substitution

$$\theta = \{X \mapsto 0, Z \mapsto a\}$$

Then we have the narrowing step

$$succ(X + a) \leadsto^{0+Z \to Z}_{\theta} succ(a) \tag{2.1}$$

The gist of narrowing is that it need not be applied to variable subterms. For narrowing the restriction to R-normalized solutions, which map variables to terms in R-normal form, implies that this is not needed.

Compared to paramodulation [RW69, Bra75], an early precursor for equational reasoning, narrowing [Sla74] assumes rewrite rules instead of undirected equations. Research on first-order narrowing was initiated by the papers of Fay [Fay79] and Hullot [Hul80]. Hullot first showed correctness and completeness of narrowing, which reads roughly as:

> Assume a rewrite system R, two terms s and t, and an R-normalized substitution θ. If $s \to^{?} t$ has solution θ, i.e. $\theta s \xrightarrow{*}{}^{R} t$, then there exists a sequence of narrowing steps $s \overset{*}{\leadsto}{}^{R}_{\sigma} t'$ such that σ and t' are more general than θ and t.

This result only deals with matching, but unification is easy to encode (see Section 6.1). Hence narrowing serves as a complete method for unification modulo a theory given by a convergent term rewriting system: as narrowing is complete wrt. normalized substitutions and since for every substitution there exists an equivalent normalized one.

Narrowing forms the underlying computation rule for programming languages [Red85, DO90]. For instance, logic programming can be viewed as narrowing [BGM88] and the work on integrating logic and functional programming is usually based on narrowing. Many of the early proposals for functional-logic programming can be found in [DL86]. When performing narrowing as a programming language, reduction is viewed as evaluation.

As the search space of naive narrowing is very large, there exists an abundance of refinements that remove redundant narrowing derivations (see [Han94b, MH94] for

overviews). For convergent systems, there is a strategy that is optimal in the sense that no solution is computed twice [BKW93]. For a restricted class of term rewriting systems, which suffices for simple programming languages, there exists a simple strategy [AEH94] that computes reductions of minimal length.

Apart from the notion of narrowing explained above, there exists another notion of narrowing, called *lazy narrowing.* To avoid confusion, we call the first notion *plain narrowing.* Plain narrowing searches for an instance such that some subterm can be rewritten. In contrast, lazy narrowing integrates the rules of unification into narrowing. The idea is to simplify terms by unification until only rewrite steps at the outermost position have to be considered in a "lazy" fashion.

For instance, to model the (plain) narrowing step in (2.1) by lazy narrowing, we start with a goal

$$succ(X + a) \rightarrow^? succ(a)$$

and look for a solution θ such that $succ(\theta X + a)$ rewrites to $succ(a)$. We first apply a decomposition step on $succ$, yielding the subgoal

$$X + a \rightarrow^? a$$

Then a lazy narrowing step applies at the function symbol $+$ with the rule $0 + Z \rightarrow Z$. The unification of the subterms of the rewrite rule with the goal is delayed by posting two new goals for the unification of $X + a$ with $0 + Z$:

$$X \rightarrow^? 0, a \rightarrow^? Z, Z \rightarrow^? a$$

Lazy narrowing employs such steps only at the root position of a term. In general, the newly added subgoals must again be solved modulo R. In this example, it suffices to take the direct syntactic solution, i.e. $\{X \mapsto 0, Z \mapsto a\}$.

Most papers on narrowing and functional-logic languages employ variations of these two notions of narrowing. Plain narrowing is mostly used for terminating rewrite systems with equational semantics [Han91]. Alternatively, narrowing is also used with denotational semantics [Red85], which are based on a strict equality: two terms are equal if they evaluate to the same constructor or data term. For this semantics, there exist completeness results for narrowing with non-terminating rules, see for instance [MNRA92, GMHGRA92].

2.2.1 Narrowing and Logic Programming

The relationship between logic programming [CM84, Llo87] and narrowing is well examined. Most approaches to functional-logic programming are based on narrowing and aim at extending logic programming by functions [Han94b]. In such languages, narrowing replaces resolution as the basic mechanism of inference. The idea is simple: view predicates as functions and horn clauses as rules with conditions. That is, a clause

$$P :- Q_1, Q_2, \ldots, Q_n$$

is written equivalently as

$$P \rightarrow true \Leftarrow Q_1 \rightarrow true, Q_2 \rightarrow true, \ldots, Q_n \rightarrow true.$$

It has been shown that narrowing, with conditional or unconditional equations, can simulate logic programming and vice versa [BGM88, Huß93]. There exist however more advanced refinements for narrowing that utilize the determinism of functional programs to a large extent. For instance, functional-logic programming with normalization has shown to be more efficient than pure logic programming, see e.g. [CF91, Han92]. These refinements use the deterministic evaluation possible for convergent rewrite rules or functional programs.

In pure logic programming, functions are often encoded in predicates. The functional version is often more concise, as functions can be nested in contrast to predicates. For instance, consider the clause

$$\mathit{fib}_P(s(s(X)), YZ) :- \mathit{fib}_P(s(X), Y), \mathit{fib}_P(X, Z), \mathit{plus}_P(Y, Z, YZ),$$

where the predicate $\mathit{plus}_P(Y, Z, YZ)$ holds if $YZ = Y + Z$. This becomes

$$\mathit{fib}(s(s(X)) = \mathit{fib}(s(X)) + \mathit{fib}(X)$$

in functional-logic programming. Notice that logic programming needs additional local variables.

2.3 Higher-Order Term Rewriting

Higher-order term rewriting is the natural extension of first-order rewriting to reasoning with higher-order equations. Starting with the work of Klop [Klo80], there exist several notions of higher-order term rewriting [Nip91a, Oos94, vR93]. This interest in higher-order rewriting follows the progress in its applications, for instance functional languages and theorem provers, where higher-order concepts are of growing interest. In this work, we follow the approach in [Nip91a]: we consider λ-terms in β-normal form and view the reductions of λ-calculus as implicit operations, e.g. $(\lambda x.f(x))a =_\beta f(a)$ by β-reduction. Furthermore, we compute modulo α-conversion, i.e. renaming of bound variables. For instance $\lambda x.f(x) =_\alpha \lambda y.f(y)$.

For example, the expressiveness of higher-order term rewriting easily deals with scoping, here pushing quantifiers inside:

$$\forall x.P \wedge Q(x) \to P \wedge \forall x.Q(x)$$

In this example the quantifier $\forall$ is a constant of type $(\mathit{term} \to \mathit{bool}) \to \mathit{bool}$, where $\forall(\lambda x.P)$ is written as $\forall x.P$ for brevity. Notice that the variable conventions of λ-calculus allow for a concise statement of the first rule: the variable P in $\lambda x.P \wedge Q(x)$ represents a term not containing the bound variable x.

As another example for the utility of higher-order programming, consider symbolic differentiation. The function $\mathit{diff}(F, X)$, as defined below, computes the differential of a function F at a point X.

$$\begin{aligned}
\mathit{diff}(\lambda y.F, X) &\to 0 \\
\mathit{diff}(\lambda y.y, X) &\to 1 \\
\mathit{diff}(\lambda y.\mathit{sin}(F(y)), X) &\to \mathit{cos}(F(X)) * \mathit{diff}(\lambda y.F(y), X)
\end{aligned}$$

With these rules, we can for instance compute:

$$\begin{array}{ll} diff(\lambda y.sin(sin(y)), X) & \longrightarrow \\ cos(sin(X)) * diff(\lambda y.sin(y), X) & \longrightarrow \\ cos(sin(X)) * cos(X) * diff(\lambda y.y, X) & \longrightarrow \\ cos(sin(X)) * cos(X) * 1 & \end{array}$$

In contrast, first-order term rewriting only permits a limited, first-order version of *diff*, as e.g. in [Bac91, SS86]. For instance, the first rule cannot be expressed directly. Furthermore, nested functions, e.g. $diff(\lambda x.sin(F'(x)))$, where F' is a function, are hard to describe in the first-order case [SS86]. In Section 7 we develop this example further.

Apart from such high-level computations, an important application of higher-order rewriting is to model the basic mechanisms of current, higher-order functional programming languages such as SML or Haskell.

In recent years many results for first-order term rewriting have been lifted to the higher-order case. Among the results obtained for higher-order rewriting are a critical pair lemma for higher-order term rewriting systems (HRS) [Nip91a], confluence of orthogonal HRS [Nip93b, vR93, Oos94], and termination criteria [Pol94].

2.4 Higher-Order Unification

For the step from first-order to higher-order narrowing, we examine another important ingredient: higher-order unification. Higher-order unification is a powerful method for solving equations between higher-order λ-terms modulo the conversions of λ-calculus. For instance, bound variables must be treated correctly: the unification problem

$$\lambda x.sin(F(x)) =^? \lambda x.sin(cos(x))$$

has solution $\{F \mapsto \lambda y.cos(y)\}$, whereas

$$\lambda x.F =^? \lambda x.sin(cos(x))$$

is unsolvable.

Higher-order unification is currently used in theorem provers like Isabelle [Pau90], TPS [AINP90], Nuprl[1] [CAB+86] and for higher-order logic programming in λ-Prolog [NM88]. Other applications of higher-order unification include program synthesis [Hag91b] and machine learning [Har90, DW88, Hag91a].

The first complete set of rules for higher-order unification was presented by Jensen and Pietrzykowski [Pie73, JP76]. The undecidability of higher-order unification was first shown by Huet [Hue73] and Lucchesi [Luc72]. It took several years until the undecidability was shown for the second-order case by Goldfarb [Gol81]. Farmer [Far91] refined this result by showing that only one symbol of arity two is needed and by giving a bound on the number of variables needed to express an undecidable problem by second-order unification.

Figure 2.1 presents an overview of known decidability results for higher-order unification. The column labeled Monadic refers to the unification of terms with unary function

[1] Nuprl uses only second-order pattern matching.

Order	Unification Problem			
	Unification	Patterns	Monadic	Matching
1	decidable			
2	undecidable Goldfarb '81 Farmer '91	⋮	decidable Farmer '88	decidable Huet '73
3	undecidable Huet '73 Lucchesi '72	⋮	undecidable Narendran '90	decidable G. Dowek '92
∞	⋮	decidable D. Miller '91	⋮	?

Figure 2.1: Decidability of Higher-Order Unification

symbols only. Monadic second-order unification is decidable [Far88]. This problem can in fact be related to unification modulo associativity, which was shown to be decidable by Makanin [Mak77]. Again, the third-order monadic case is undecidable [Nar89].

Huet [Hue75] already conjectured that higher-order matching, i.e. unification with a term containing no free variables, is decidable, but the problem is still open. Some progress has been made by Dowek [Dow92], who showed the decidability of third-order matching. Furthermore, fourth-order matching is claimed to be decidable by Vincent Padovani [Pad94]. Wolfram [Wol93] presented a terminating algorithm for higher-order matching, but was not yet able to show its completeness.

Dale Miller, as indicated in the column labeled Patterns, discovered a class of λ-terms, called higher-order patterns, with decidable and even unitary unification, i.e. if some unifier exists then there exists a most general unifier. A term is a higher-order pattern if each free variable has distinct bound variables as arguments. Patterns behave like first-order terms in many respects, e.g. unification is not only unitary but also of linear complexity [Qia93].

Full higher-order unification is highly intractable: there do not exist maximally general unifiers. In other words, there are infinite chains of unifiers, one more general than the other. This is called nullary unification. As noted by Huet [Hue75], this was first observed by Gould [Gou66]. The idea of pre-unification by Huet [Hue75] was a major step towards practically usable systems: pre-unification delays a particular class of equations that is known to be solvable and permits the enumeration of a complete set of unifiers without any redundancy. This is important for any practical application. Pre-unification is still infinitary, i.e. there may be an infinite set of unifiers for two terms.

2.4.1 Decidable Higher-Order Unification Problems

Since higher-order unification is undecidable in general, we are interested in classes where higher-order unification is decidable. An overview of the results can be found in Figure 2.2. Notice that the results only hold for all conditions in the path to the node.

The main restriction we impose is linearity, i.e. we require that some variables may not occur repeatedly. We show that the unification of a linear higher-order pattern s with an arbitrary second-order term that shares no variables with s is decidable and finitary.

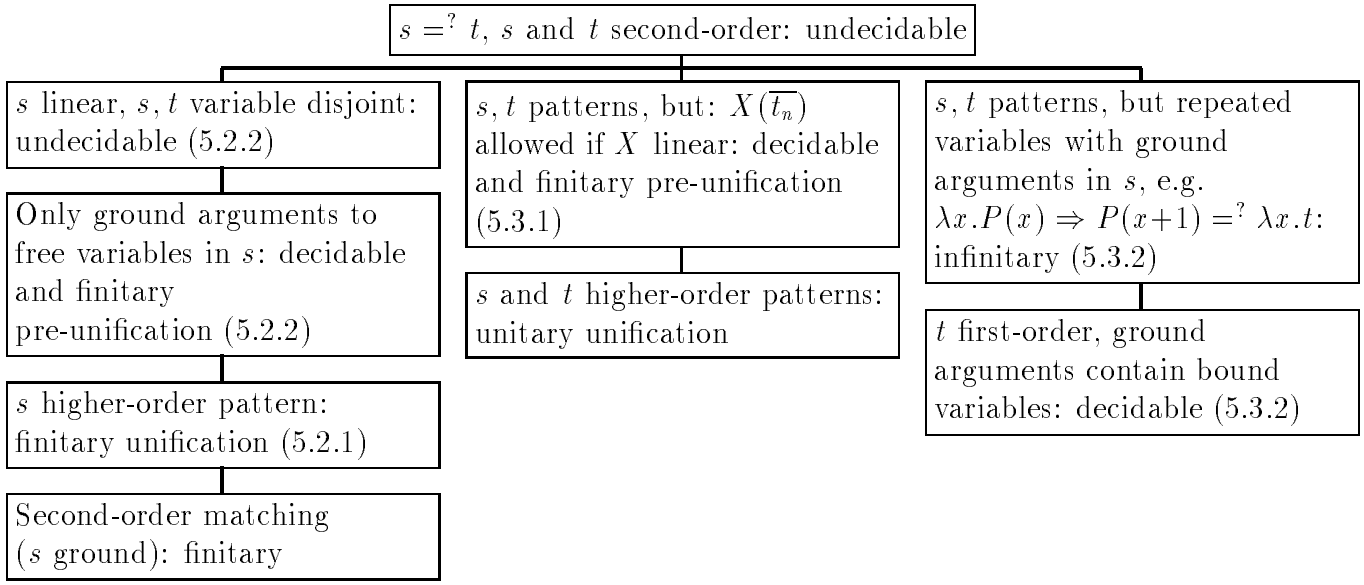

Figure 2.2: Results on Second-Order Unification

In particular, we do not have to resort to pre-unification, as equations with variables as outermost symbols on both sides (flex-flex pairs) can be finitely solved in this case. A few extensions of this unification problem are still decidable; only one of them is included in Figure 2.2. For instance, unifying two second-order terms, where one term is linear, is undecidable if the terms contain bound variables but decidable if they do not.

The main application of this result is the unification of linear left-hand sides of rewrite rules with second-order terms, as employed in higher-order narrowing. For instance, a standard example for functional programs, the function

$$map(F, [X \mid Y]) = [F(X) | map(F, Y)]$$

has a linear left-hand side. Furthermore, it has the non-pattern $F(X)$ on the right-hand side. Hence rewriting with this rule may yield non-pattern terms. Thus higher-order unification is needed for the unification with a left-hand side of a rewrite rule. So far, most functional logic languages even with higher-order terms only use first-order unification, e.g. [GMHGRA92, Loc93].

Furthermore, we present an extension of higher-order patterns with decidable unification and another result that is tailored for the unification of induction schemes with first-order terms. It is shown that the unification of restricted second-order terms with first-order terms is decidable, where the restriction is such that typical inductions schemes can be expressed. An example is the formula $\forall x.P(x) \Rightarrow P(x+1)$ in the inductive axiom

$$P(0), \forall x.P(x) \Rightarrow P(x+1) \vdash \forall x.P(x)$$

With these results only few classes remain where decidability of second-order unification is unknown.

2.5 Narrowing: The Higher-Order Case

The second main contribution of this work is to lift several ideas of first-order narrowing to the higher-order case. We introduce a first framework for higher-order narrowing modulo an higher-order equational theory and give first completeness results.

In first-order narrowing, values for logic or free variables are computed via unification. The variables range over objects of the domains of interest. In a higher-order setting, unification can compute values even for functional objects. For instance, a solution for the free variable F in the goal

$$\lambda x.diff(\lambda x.sin(F(x)), x) \rightarrow^{?} \lambda x.cos(x)$$

can be computed by narrowing. Examples from this and other areas can be found in Section 7.

An overview of the different approaches to higher-order narrowing can be found in Figure 2.3. For plain narrowing, which attempts to lift rewrite steps somewhere inside a term, we show that there are some principal problems in the full higher-order case. In contrast, lazy narrowing can be lifted to the higher-order case.

We develop several optimizations and refinement for lazy narrowing. Particularly important is the restriction to R-normalized solutions in order to limit narrowing steps.

Figure 2.3: A Framework for Higher-Order Narrowing

This also permits deterministic simplification on goals. For instance, the above goal can be simplified by rewriting to obtain the new goal

$$\lambda x.cos(F(x)) * diff(\lambda x.F(x), x) \rightarrow^? \lambda x.cos(x).$$

Another optimization is deterministic variable elimination, which may be incomplete in the general case. Variable elimination for an equation $X =^? t$ simply means binding X to the term t. In our case, we only consider directed goals, e.g.

$$X \rightarrow^? t \text{ and } t \rightarrow^? X,$$

where on the first goal variable elimination is safe and no other rules must be considered.

Another important source of optimization is using left-linear rewrite rules, i.e. where free variables do not occur repeatedly on the left side of a rule. This is a common restriction for programming applications, e.g. in functional(-logic) languages. We show that in such a setting a particular class of goals, called Simple Systems, suffices and has several nice properties. For instance, a variable cannot occur on both sides of a goal, e.g. $X \rightarrow^? f(X)$ is impossible and thus the occurs check is immaterial. Furthermore, solved forms are easy to detect. For instance, a Simple System of the form

$$t_1 \rightarrow^? X_1, \ldots, t_n \rightarrow^? X_n,$$

is guaranteed to have a solution. It follows from the invariant of Simple Systems that all $X_1, \ldots, X_n$ are distinct. Another important property is that unification of second-order Simple Systems is decidable. Thus, as in the first-order case, divergence only results from the main computation paradigm, narrowing.

Integrating Simple Systems with normalized solutions yields a strategy for variable elimination: with normalized solutions, only one case is deterministic, but in Simple Systems the remaining case is undesirable and such goals can safely be delayed. This leads to a new strategy, called Needed Lazy Narrowing, which computes values only if needed and also avoids copying. It thus resembles call-by-need or lazy evaluation with sharing of identical subterms.

The chapter on narrowing concludes with an alternative approach to higher-order narrowing in Section 6.9, which combines plain narrowing and lazy narrowing. The basic idea is to put the truly higher-order terms into constraints where lazy narrowing is used and work on the main goal similar to the first-order case.

2.6 Conditional Narrowing and Higher-Order Programming

A promising application of higher-order narrowing is truly higher-order functional-logic programming. Our approach to higher-order programming via narrowing is more oriented towards functional languages than most other approaches to functional-logic languages. Recall that the core of modern functional languages such as SML [MTH90] and Haskell [HJW92] can be seen as higher-order rewrite rules. Our contribution is to develop a new basis for *functional* logic programming that works with higher-order conditional equations. We consider *normal conditional rules* of the form

$$l \rightarrow r \Leftarrow l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n,$$

where $\overline{l_n \to r_n}$ denote conditions for the application of the rule and $\overline{r_n}$ are ground terms in R-normal form. Although some first-order approaches are less restrictive, we argue that such extensions are not needed in a higher-order setting. Furthermore, this restriction has a significant advantage: for proving conditions of rules, as well as for queries, we consider *oriented* goals of the form $s \to^? t$ with solutions $\theta s \xrightarrow{*} t$. Thus, this restriction permits for a simpler operational model and is powerful enough for encoding functional and logic programs: the core of modern functional languages can be seen as higher-order (unconditional) rewrite rules. Furthermore encoding logic programs is possible, as shown in Section 2.2.1, since the right-hand sides in the conditions is simply the constant *true*.

The restriction to ground right-hand sides is too strong for first-order functional-logic languages, as variables on the right in conditions serve as local variables. Consider for instance the function *unzip*, cutting a list of pairs into a pair of lists. In a functional language *unzip* can be written as

$$\begin{array}{lcl} unzip([pair(X,Y)|R]) & \to & \mathsf{let}\ pair(xs,ys) = unzip(R) \\ & & \mathsf{in}\ pair([X|xs],[Y|ys]) \\ unzip([]) & \to & [] \end{array}$$

where $pair(a,b)$ denotes a pair. The let-construct for pairs, written in infix notation as common in functional languages, can be defined by higher-order rewrite rules (see Section 7.3). In first-order functional-logic programming this function may be written as

$$\begin{array}{lcl} unzip([pair(X,Y)|R]) & \to & pair([X|Xs],[Y|Ys]) \Leftarrow unzip(R) \to pair(Xs,Ys) \\ unzip([]) & \to & [] \end{array}$$

The first of the above conditional rewrite rules has extra variables on the right, which are used to model the let-construct.

Notice that we permit new variables in the left sides of the conditions, which are used as "existential" variables, to be computed by unification as in logic programming. Consider for instance the following example modeling family relations, where a new variable Z is used in the definition of *grand_mother*. For brevity, we write p for a rule $p \to true$ or a goal $p \to^? true$.

$$\begin{array}{l} mother(jane,mary) \\ mother(susan,mary) \\ mother(mary,judy) \\ wife(john,jane) \end{array}$$

$$grand\_mother(X,Y) \Leftarrow mother(X,Z), mother(Z,Y)$$

In the higher-order case, the concept of family relations can be generalized, similar to [Llo94, Nad87]:

$$\begin{array}{lcl} family\_rel(wife) & & \\ family\_rel(mother) & & \\ family\_rel(comp(R_1,R_2)) & \Leftarrow & family\_rel(R_1), family\_rel(R_2) \\ comp(R_1,R_2)(X,Y) & \Leftarrow & R_1(X,Z), R_2(Z,Y) \end{array}$$

In the last rules, *comp* is intended to compose two relations. Thus a query

$$family\_rel(R), R(jane,judy)$$

should be answered by $R \mapsto comp(mother, mother)$. Notice how partial application of *comp* is used in third rule.

We argue that many programming concepts are not only simpler expressed by higher-order functional programming, but also the technical treatment can be simpler. Handling extra variables for narrowing is both difficult, error-prone and gave rise to many works, for an overview see [MH94]. Furthermore, there are several works [BG89, LS93, ALS94b] on confluence and termination of logic programs that correspond to such function constructs (sometimes called well-moded programs). For the termination of logic programs, such local variables are one of the main problems [SD94]. Also, functional programming provides more directionality than logic programs, which is another major problem for proving termination [SD94].

Chapter 3

Preliminaries

Basic definitions and results for higher-order equational reasoning are introduced in this chapter. The first sections contain general background material on reductions and orderings, followed by a brief introduction to λ-calculus. For a comprehensive treatment we refer to [HS86, Bar84].

3.1 Abstract Reductions and Termination Orderings

An abstract reduction is a relation on some set A. The following properties of reductions will be used mostly for term rewriting, which is a reduction on terms.

Definition 3.1.1 For some abstract reduction $\longrightarrow$, let $\longrightarrow^+$ denote its transitive closure, $\stackrel{*}{\longrightarrow}$ its reflexive transitive closure, and $\longleftarrow$ its inverse. Furthermore, define $\longleftrightarrow = \longrightarrow \cup \longleftarrow$. We write $\longrightarrow^n$ for some reduction of length n, i.e. $s_0 \longrightarrow^n s_n$ stands for a sequence $s_0 \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s_n$.

A relation is an **equivalence relation** if it is reflexive, transitive and symmetric. A **partial ordering** is a reflexive, transitive and anti-symmetric relation. A **strict partial ordering** is a transitive and irreflexive relation.

A partial ordering $\leq$ is a **total ordering** if $a \leq b$ or $b \leq a$ holds for all a and b. A partial or total ordering $\leq$ is **compatible** with another partial ordering $\leq'$ if $\leq' \subseteq \leq$.

Definition 3.1.2 An abstract reduction is called **terminating** if no infinite reduction exists. An element a is called in **normal form** if no reduction from a exists.

Two terms s and t are **joinable** by a reduction $\longrightarrow^R$, written as $s{\downarrow_R}t$, if there exists u with $s \stackrel{*}{\longrightarrow}^R u$ and $t \stackrel{*}{\longrightarrow}^R u$. A reduction is called **locally confluent**, if any two reductions from a term t are joinable, i.e. if $t \longrightarrow u$ and $t \longrightarrow v$ then $u{\downarrow}v$. It is called **confluent**, if $\stackrel{*}{\longrightarrow}$ is locally confluent, i.e. if $t \stackrel{*}{\longrightarrow} u$ and $t \stackrel{*}{\longrightarrow} v$ then $u{\downarrow}v$.

Definition 3.1.3 The **lexicographic combination** of two reductions $\longrightarrow_1$ and $\longrightarrow_2$ on sets A and B, written as $R = (\longrightarrow_1, \longrightarrow_2)_{lex}$, is a reduction on $A \times B$, with $(a, b)\ R\ (a', b')$ if

- $a \longrightarrow_1 a'$ or

- $a = a'$ and $b \longrightarrow_2 b'$.

The important property of the lexicographic combination is the following:

Lemma 3.1.4 *The lexicographic combination of terminating reductions is terminating.*

The lexicographic combination of n abstract reductions $R^n = \longrightarrow_n, \ldots, \longrightarrow_1$ is defined recursively as $R^n_{lex} = (\longrightarrow_1, R^{n-1}_{lex})_{lex}$.

A **multiset** M over a set A is a mapping from A to $\{0, 1, 2, \ldots\}$. A multiset M can be viewed as a set where repeated elements are allowed, i.e. M maps an element $a \in A$ to its number of occurrences. A multiset M is finite if $M(x) > 0$ holds only for finitely many $x \in A$.

Removing an element from a multiset reduces the number of occurrences by one, if it occurs at all. Formally, removing an element a from a multiset M gives a new mapping $M' = M - a$ with $M'(x) = M(x)$ if $x \neq a$ and

$$M'(a) \quad = \quad \begin{cases} M(a) - 1 & \text{if } M(a) > 0 \\ 0 & \text{if } M(a) = 0 \end{cases}$$

Removing a multiset M from M', written as $M' - M$, is defined as the result of removing each element of M from M'. Adding an element to a multiset, written as $a + M$, and the union $M \cup M'$ of two multisets are defined correspondingly.

An important method for termination proofs is to extend an ordering $\ll$ on a set A to multisets of A. A **multiset** N **is smaller** than M, written as $N \ll_{multi} M$, if it can be obtained by removing one element from M plus adding finitely many smaller elements. Formally we have:

$$N \ll_{multi} M \quad \Leftrightarrow \quad \exists x \in A. M - x = N \cup N',$$

where N' is a finite multiset with $n \ll x, \quad \forall n \in N'$.

The following result allows to extend termination orderings to multisets:

Theorem 3.1.5 ([DM79]) *The multiset extension of a terminating ordering is terminating.*

Besides (multi-)sets, we often used **lists,** which are denoted by square brackets, i.e. appending a list R to an element t is written as $[t|R]$. The application of a function f to a list, written as $f[\overline{t_n}]$, is defined as $[\overline{f(t_n)}]$.

3.2 Higher-Order Types and Terms

This section introduces our term language: simply typed λ-terms. The set of types $\mathcal{T}$ for the simply typed λ-terms is generated by a set $\mathcal{T}_0$ of **base types** (e.g. int, bool) and the **function type constructor** $\rightarrow$. Notice that $\rightarrow$ is right associative, i.e. $\alpha \rightarrow \beta \rightarrow \gamma = \alpha \rightarrow (\beta \rightarrow \gamma)$. We assume a set of variables V_τ, and a set of constants C_τ for all types $\tau \in \mathcal{T}$, where $V_\tau \cap V_{\tau'} = C_\tau \cap C_{\tau'} = \{\}$. The set of all variables is $V = \bigcup_{\tau \in \mathcal{T}} V_\tau$, which is disjoint from the set of all constants, $C = \bigcup_{\tau \in \mathcal{T}} C_\tau$. The following **naming conventions** are used in the sequel:

- F, G, H, P, X, Y free variables,

- a, b, c, f, g (function) constants,
- x, y, z bound variables,
- α, β, τ type variables.

Further, we often use s and t for terms and u, v, w for constants or bound variables. The following grammar defines the syntax for **untyped λ-terms**

$$t \quad = \quad F \mid x \mid c \mid \lambda x.t \mid (t_1\ t_2),$$

where $(t_1\ t_2)$ denotes the **application** of two terms. The term $\lambda x.t$ denotes an **abstraction** over x and thus creates a new functional object. An occurrence of a variable x in a term t is **bound**, if it occurs below a binder for x, i.e. the occurrence of x is in a subterm $\lambda x.t'$. Otherwise it is **free**. Free and bound variables of a term t will be denoted as $\mathcal{FV}(t)$ and $\mathcal{BV}(t)$, respectively.

Notice that there can be many such binders, e.g. $\lambda x.\lambda x.x$, but only the innermost one is associated with x. To avoid such cases, we will adopt assumptions (see below) on the naming of bound variables for simplicity.

A list of syntactic objects $s_1, \ldots, s_n$ where $n \geq 0$ is abbreviated by $\overline{s_n}$. We will use n-fold abstraction and application, written as $\lambda\overline{x_n}.s = \lambda x_1 \ldots \lambda x_n.s$ and $a(\overline{s_n}) = ((\cdots(a\ s_1)\cdots)\ s_n)$, respectively. For instance

$$\lambda\overline{x_m}.f(\overline{s_n}) = \lambda x_1 \ldots \lambda x_m.((\cdots(f\ s_1)\cdots)\ s_n)$$

A **type judgment** stating that t is of type τ is written as $t : \tau$. The following inference rules inductively define the set of **simply typed λ-terms**.

$$\frac{x \in V_\tau}{x : \tau} \qquad \frac{c \in C_\tau}{c : \tau}$$

$$\frac{s : \tau \to \tau' \quad t : \tau}{(s\ t) : \tau'} \qquad \frac{x : \tau \quad s : \tau'}{(\lambda x.s) : \tau \to \tau'}$$

The **order of a type** $\varphi = \alpha_1 \to \ldots \to \alpha_n \to \beta, \quad \beta \in \mathcal{T}_0$ is defined as

$$Ord(\varphi) = \begin{cases} 1 & \text{if } n = 0, \text{ i.e. } \varphi = \beta \in \mathcal{T}_0 \\ 1 + k & \text{otherwise, where} \\ & k = max(Ord(\alpha_1), \ldots, Ord(\alpha_n)) \end{cases}$$

We say a symbol is of order n if it has a type of order n. A **term of order** n is restricted to

- function constants of order $\leq n + 1$ and
- variables of order $\leq n$.

For instance, if a term $F(\overline{t_n})$ is second-order, then all subterms t_i must be of base type. We say a term t is **weakly second-order** if it is second-order, but with the exception that bound variables of arbitrary type may occur as arguments to free variables. For instance, $F(\lambda z.x(z), a)$ is weakly second-order, but not second-order.

Let $\{x \mapsto y\}t$ denote the result of replacing every free occurrence of x in t by y. The **conversions in λ-calculus** are defined as:

- α-conversion: $\lambda x.t \succ_\alpha \lambda y.(\{x \mapsto y\}t)$
- β-conversion: $(\lambda x.s)t \succ_\beta \{x \mapsto t\}s$
- η-conversion: if $x \notin \mathcal{FV}(t)$, then $\lambda x.(t\ x) \succ_\eta t$

The first of the above, α-conversion, serves for renaming bound variables. β-conversion replaces the formal parameter of a function $\lambda x.s$ by the argument t. A **β-redex** is a term of the form $(\lambda x.s)t$ where β-reduction applies, and similarly for the other reductions. For $\phi \in \{\alpha, \beta, \gamma\}$ we write $s \to_\phi t$, called **ϕ-reduction**, if t is obtained from s by ϕ-conversion on some subterm of s. Let $\to_{\beta,\eta}$ be defined as $\to_\beta \cup \to_\eta$, and similarly for other combinations. The reflexive, symmetric and transitive closure of some Φ-reduction induces an equivalence relation on terms, written as $s =_\Phi t$, where $\Phi \subseteq \{\alpha, \beta, \eta\}$. Application of the conversion rules in the other direction is called **expansion**. Reduction in the simply typed λ-calculus is confluent and terminating w.r.t. β-reduction (and w.r.t. η-reduction), see e.g. [Bar84].

The **β-normal form** (**η-normal form**) of a term t is denoted by $t{\downarrow_\beta}$ ($t{\downarrow_\eta}$). Let t be in β-normal form. Then t is of the form $\lambda\overline{x_n}.v(\overline{u_m})$, where v is called the **head** of t, and written as $Head(t)$. The **η-expanded form** of a term $t = \lambda\overline{x_n}.v(\overline{u_m})$ is defined by

$$t{\uparrow_\eta} = \lambda\overline{x_{n+k}}.v(\overline{u_m{\uparrow_\eta}}, x_{n+1}{\uparrow_\eta}, \ldots, x_{n+k}{\uparrow_\eta})$$

where $t : \overline{\tau_{n+k}} \to \tau$ and $x_{n+1}, \ldots, x_{n+k} \notin \mathcal{FV}(\overline{u_m})$. We call $t{\downarrow_\beta}{\uparrow_\eta}$ the **long $\beta\eta$-normal form** of a term t, also written as $t{\updownarrow_\beta^\eta}$. A term t is in long $\beta\eta$-normal form if $t = t{\updownarrow_\beta^\eta}$. It is well known [HS86] that $s =_{\alpha\beta\eta} t$ iff $s{\updownarrow_\beta^\eta} =_\alpha t{\updownarrow_\beta^\eta}$.

The **size** $|t|$ of a term t in long $\beta\eta$-normal form is defined as the number of symbols occurring in t, not counting binders λx:

$$\begin{array}{lcl} |\lambda x.t| & = & |t|, \\ |s\ t| & = & |s| + |t|, \\ |v| & = & 1, \quad v \in V \cup C \end{array}$$

A variable is **isolated** if it occurs only once (in a term or in a system of equations). A term is **linear** if no free variable occurs repeatedly. A term $\lambda\overline{x_k}.v(\overline{t_n})$ is called **flexible** if v is a free variable and **rigid** otherwise.

Assumptions. We will in general assume that terms are in long $\beta\eta$-normal form. For brevity, we write variables in η-normal form, e.g. X instead of $\lambda\overline{x_n}.X(\overline{x_n})$. We assume that the transformation into long $\beta\eta$-normal form is an implicit operation.

We work in the following completely modulo α-conversion, that is α-equivalent terms are identified. A representation for λ-terms that achieves this on a syntactical basis is possible with de Bruijn indices [dB72]: bound variables are represented as natural numbers, indicating the corresponding binder. The main result is that two α-equivalent terms have the same de Bruijn representation.

We follow the **variable convention** that free and bound variables are kept disjoint (see also [Bar84]). We cannot enforce this convention completely. For instance, in the congruence rule used in Section 4.3.1, $s = t \Rightarrow \lambda x.s = \lambda x.t$, x occurs both free and bound. More seriously, this distinction permits so-called **loose bound** variables, i.e. "bound" variables without a binder. Such variables are typically created when a subterm of a term is considered or manipulated. For instance, $f(x)$ is a subterm of $\lambda x.g(f(x))$

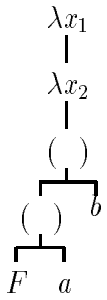

Figure 3.1: $\lambda x_1, x_2.F(a,b)$ as Binary Tree

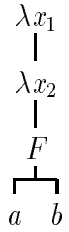

Figure 3.2: $\lambda x_1, x_2.F(a,b)$ as n-nary Tree

with a loose bound variable. In such cases, these variables can be viewed as bound variables where the binder is (implicit) in the context. In general, loose bound variables may create inconsistencies. Although sometimes convenient, we will avoid loose bound variables whenever possible.

For simplicity, we assume that bound variables with different binders have different names. As a consequence of our conventions, it suffices to write $s = t$ instead of $s =_{\alpha,\beta,\eta} t$, as we assume long $\beta\eta$-normal form and work modulo α-conversion. These conventions for instance permit the following definition.

We say a bound variable y in a term $\lambda\overline{x_n}.t$ in long $\beta\eta$-normal form is **outside bound** if $y = x_i$ for some i. The set of all outside bound variables of a term $\lambda\overline{x_n}.t$ is written as $\mathcal{OBV}(\lambda\overline{x_n}.t) = \mathcal{BV}(\lambda\overline{x_n}.t) \cap \{\overline{x_n}\}$.

3.3 Positions in λ-Terms

We describe positions in λ-terms by sequences over natural numbers, as we have adopted n-ary application. Such a sequence describes the **path** to a subterm of a term. Positions in λ-terms are often written as sequences over 1 and 2. It is easy to translate one representation into the other, as in the following example for the term $\lambda x_1, x_2.F(a,b)$ in Figures 3.1 and 3.2. Notice that our representation of terms as trees is a generalization of usual first-order terms and positions.

Let ϵ denote the **empty sequence**, let $i.p$ denote the sequence p appended to an element i, and let $p + p'$ concatenate two sequences. A sequence p is a **prefix** of p', if $\exists q.p + q = p'$, and similarly p is a **postfix** if $\exists q.q + p = p'$.

Definition 3.3.1 The **subterm** of s at **position** p, written as $s|_p$, is defined as

- $s|_\epsilon = s$
- $v(\overline{t_m})|_{i.p} = t_i|_p$ if $i \leq m$
- $\lambda\overline{x_m}.t|_{1.p} = (\lambda x_2, \ldots, x_m.t)|_p$
- undefined otherwise

The following notion of subterm extends the definition of subterms to account for a binding environment. A term $s = \lambda\overline{x_n}.s_0$ is a **subterm modulo binders** of $t = \lambda\overline{x_n}.t_0$, written as $s <_{sub} t$, if s_0 is a (true) subterm of t_0.

A term t with the subterm at position p replaced by s is written as $t[s]_p$. Two positions p and q are **independent** if none is a prefix of the other. For a term s of the form $\lambda\overline{x_k}.v(\overline{t_n})$, the position of v is called the **root position**. A (sub-)term $t|_p$ is called **ground** if no free variables of t occur in $t|_p$.

If p is a position in s then let $\mathcal{BV}(s,p)$ be the set of all λ-abstracted variables on the path to p in s. Such a path is called **rigid** if it contains no free variables.

3.4 Substitutions

Substitutions are finite mappings from variables to terms, denoted by $\{\overline{X_n \mapsto t_n}\}$, and extend homomorphically from variables to terms. In general, substitutions map only the free variables of a term. If $s = \theta t$ for some substitution θ, then s is called an **instance** of t.

Define $\mathcal{D}om(\theta) = \{X \mid \theta X \neq X\}$, $\mathcal{I}m(\theta) = \bigcup_{X \in \mathcal{D}om(\theta)} \theta X$ and $\mathcal{R}ng(\theta) = \mathcal{FV}(\mathcal{I}m(\theta))$. The **free variables of a substitution** θ are defined as $\mathcal{FV}(\theta) = \mathcal{D}om(\theta) \cup \mathcal{R}ng(\theta)$. For a list of syntactical objects $\overline{C_n}$ we write $\mathcal{FV}(\overline{C_n})$ instead of $\mathcal{FV}(C_1) \cup \ldots \cup \mathcal{FV}(C_n)$. Two **substitutions are equal** on a set of variables W, written as $\theta =_W \theta'$, if $\theta X = \theta' X$ for all $X \in W$. The **restriction of a substitution** to a set of variables W is defined by $\theta|_W X = \theta X$ if $X \in W$ and $\theta|_W X = X$ otherwise. The **composition** $\delta\theta$ of two substitutions is defined as $(\delta\theta)(s) = \delta(\theta(s))$.

Definition 3.4.1 A substitution θ is **more general** than θ' over a set of variables W, written as $\theta' \leq_W \theta$, if $\theta' =_W \sigma\theta$ for some substitution σ.

For brevity, we will often leave the set of variables W implicit and write $\theta' \leq \theta$ or $\theta = \theta'$. A substitution θ is **idempotent** iff $\theta = \theta\theta$. We will in general assume that substitutions are idempotent. This is justified by the following two basic lemmata [SG89].

Lemma 3.4.2 *A substitution θ is idempotent if $\mathcal{R}ng(\theta) \cup \mathcal{D}om(\theta) = \{\}$.*

In the higher-order order-case, this condition for idempotence is only sufficient but not necessary, as noted in [SG89].

Lemma 3.4.3 *For any substitution θ and set of variables W with $\mathcal{D}om(\theta) \subseteq W$, there exists an idempotent substitution θ' such that $\mathcal{D}om(\theta) = \mathcal{D}om(\theta')$, $\theta' \leq \theta$ and $\theta \leq_W \theta'$.*

As we syntactically distinguish between bound and free variables, we can speak of **well-formed** substitutions: a substitution is well-formed, if it does not contain loose bound variables, i.e. bound variables without binder. With a few exceptions, we will in general assume well-formed substitutions. Thus, for instance, $\theta\lambda\overline{x_k}.t = \lambda\overline{x_k}.\theta t$ by convention.

Properties of terms extend to substitutions in the component-wise way, i.e. to the terms in the image. For instance, a substitution θ is ground (in long $\beta\eta$-normal form) if all terms in the image of θ are ground (in long $\beta\eta$-normal form).

3.5 Unification and Unification Theory

Unification of two terms s and t aims at finding a substitution θ such that $\theta s = \theta t$, where θ is called a **unifier** of s and t. Unification problems are written as $s =^? t$. There exist several surveys on the subject [BS94, JK91].

An **equational theory** is an equivalence relation on terms that is stable under substitutions, i.e. $s =_E t$ implies $\theta s =_E \theta t$ for any substitution θ. Usually, equational theories are generated by a set of equations, as discussed for the higher-order case in Section 4.3.1. The conversions of λ-calculus, i.e. the $\alpha\beta\eta$-rules, are equations with meta-level conditions. These can be considered immaterial when working modulo α-conversion.

A substitution τ is **more general** than σ, modulo E over a set of variables W, written as $\sigma \leq_{E,W} \tau$, if $\exists\delta.\sigma =_{E,W} \delta\tau$. Accordingly, $\sigma =_{E,W} \tau$ if $\sigma X =_E \tau X$ for all $X \in W$. For simplicity, we often leave the parameter W implicit.

Unification modulo an equational theory $=_E$, or **E-unification,** aims at finding a substitution with $\theta s =_E \theta t$. Then θ is called an E-unifier of s and t. As there can be many solutions to a unification problem $s =^? t$, it is desirable to find minimal sets of solutions, as defined next:

Definition 3.5.1 A set of substitutions S is a **minimal, complete set of unifiers (MCSU)** of a unification problem $s =^? t$ for some equational theory E, iff

- Each element of S is an E-unifier of $s =^? t$.
- For every E-unifier of $s =^? t$ there exists a more general E-unifier in S.
- The elements of S are incomparable.

It can be shown that such a set of incomparable common instances is uniquely defined except for variants of its elements [FH86]. There exist several classes of unification problems, depending on the existence of a MCSU. An E-unification problem is called

unitary if a MCSU is either empty or a singleton,

finitary if a finite MCSU exists,

infinitary if a possibly infinite MCSU exists,

nullary if no MCSU may exist.

This classification extends to an equational theory E, if for all E-unification problems the property (e.g. unitary) holds.

Another distinction of unification problems is sometimes considered in the first-order case [BS94]: are only constant symbols of a fixed signature allowed in the terms to be unified or arbitrary constants? In the first-order case, the above classification may depend on this distinction. This distinction is immaterial in a higher-order context, as we must deal with local "constants", i.e. bound variables.

3.6 Higher-Order Patterns

The following subclass of λ-terms was introduced originally by Dale Miller [Mil91a] and is often called higher-order patterns in the literature.

Definition 3.6.1 A simply typed λ-term s in β-normal form is a **relaxed higher-order pattern**, if all free variables in s only have bound variables as arguments, i.e. if $X(\overline{t_n})$ is a subterm of s, then all $t_i{\downarrow_\eta}$ are bound variables.

Examples of relaxed higher-order patterns are $\lambda x, y.F(x, x, y)$ and $\lambda x.f(G(\lambda z.x(z)))$, where the latter is at least third-order. Non-patterns are $\lambda x, y.F(a, y)$, $\lambda x.G(H(x))$.

In most of the existing literature [Mil91b, Nip91a], patterns are required to have distinct bound variables as arguments to a free variable. This restriction is necessary for unitary unification, but for some of the results on decidability of higher-order unification in Chapter 5 this is not relevant.

Definition 3.6.2 A (**higher-order**) **pattern** is a relaxed pattern where the arguments to free variables are distinct bound variables.

For instance, $\lambda x, y.F(x, x, y)$ is not a higher-order pattern, but $\lambda x, y.G(x, \lambda z.y(z))$ is. Unification of patterns is decidable and a most general unifier exists if they are unifiable [Mil91a, Nip91a], as shown in Section 4.2. Furthermore, a most general unifier can be computed in linear time [Qia93]. This shows that unification with patterns behaves similar to the first-order case.

Several important properties of patterns with respect to term rewriting are examined later and are based on the important fact that β-reduction on patterns only renames bound variables. For this reason β-reduction on patterns is also called β_0-reduction in [Mil91a].

Chapter 4

Higher-Order Equational Reasoning

This chapter introduces higher-order unification and term rewriting. It assumes some knowledge of first-order unification. Section 4.1 reviews a set of transformation rules for full higher-order (pre-)unification. This is followed by an important special case, higher-order patterns, where unification proceeds almost as in the first-order case.

4.1 Higher-Order Unification by Transformations

We present in the following a version of the transformation system PT for higher-order unification of Snyder and Gallier [SG89]. More precisely, we adapt the primed transformations for pre-unification of Section 5 in [SG89].

Consider solving an equation $\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m})$ where v is not a free variable. Such equations are called **flex-rigid**. Clearly, for any solution θ to F the term $\theta F(\overline{t_n})$ must have (after β-reduction) the symbol v as its head. There are two possibilities:

- In the first case, v already occurs in (the solution to) some t_i. For instance, consider the equation $F(a) =^? a$, where $\{F \mapsto \lambda x.x\}$ is a solution based on a **projection**. In general, a projection binding for F is of the from $\{F \mapsto \lambda\overline{x_n}.x_i(\ldots)\}$. As some argument, here a, is carried to the head of the term, such a binding is called projection. This name was introduced in [JP76].

- The second case is that the head of the solution to F is just the desired symbol v. For instance, in the last example, an alternative solution is $\{F \mapsto \lambda x.a\}$. This is called **imitation.** Notice that imitation is not possible if v is a bound variable.

To solve a flex-rigid pair, the strategy is to guess an appropriate imitation or projection binding only for one rigid symbol, here a, and thus approximate the solution to F. Unification proceeds by iterating this process which focuses only on the outermost symbol. Roughly speaking, the rest of the solution for F is left open by introducing new variables, as shown formally in the next definition of these bindings.

Definition 4.1.1 Assume an equation $\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m})$, where all terms are in long $\beta\eta$-normal form. An **imitation binding** for F is of the form

$$F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})$$

Deletion

$$\{t =^? t\} \cup S \quad \Rightarrow \quad S$$

Decomposition

$$\{\lambda\overline{x_k}.v(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_n})\} \cup S \quad \Rightarrow \quad \{\overline{\lambda\overline{x_k}.t_n =^? \lambda\overline{x_k}.t'_n}\} \cup S$$

Elimination

$$\{F =^? t\} \cup S \quad \Rightarrow^{\theta} \quad \theta S \quad \text{if } F \notin \mathcal{FV}(t) \text{ and where } \theta = \{F \mapsto t\}$$

Imitation

$$\{\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.f(\overline{t'_m})\} \cup S \quad \Rightarrow^{\theta} \quad \{\overline{\lambda\overline{x_k}.H_m(\overline{\theta\, t_n}) =^? \lambda\overline{x_k}.\theta t'_m}\} \cup \theta S$$
where $\theta = \{F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$
is an appropriate imitation binding

Projection

$$\{\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m})\} \cup S \quad \Rightarrow^{\theta} \quad \{\lambda\overline{x_k}.\theta\, t_i(\overline{H_j(\overline{t_n})}) =^? \lambda\overline{x_k}.v(\overline{\theta t'_m})\} \cup \theta S$$
where $\theta = \{F \mapsto \lambda\overline{x_n}.x_i(\overline{H_j(\overline{x_n})})\}$,
is an appropriate projection binding

Figure 4.1: System PT for Higher-Order Unification

where $\overline{H_m}$ are new variables of appropriate type. A **projection binding** for F is of the form

$$F \mapsto \lambda\overline{x_n}.x_i(\overline{H_p(\overline{x_n})})$$

where $\overline{H_p}$ are new variables with $\overline{H_p : \tau_p}$ and $x_i : \overline{\tau_p} \to \tau$. A **partial binding** is an imitation or a projection binding.

Notice that in the above definition, the bindings are not written in long $\beta\eta$-normal form. The long $\beta\eta$-normal form of an imitation or projection binding can be written as

$$F \mapsto \lambda\overline{x_n}.v(\overline{\lambda\overline{z_{j_p}}.H_p(\overline{x_n}, \overline{z_{j_p}})}).$$

A full exhibition of the the types involved can be found in [SG89].

The transformation rules PT for higher-order unification in Figure 4.1 consist of the basic rules for unification, such as Deletion, Elimination and Decomposition plus the two rules explained above: Imitation and Projection. The rules work on sets of pairs of terms to be unified, written as $\{u =^? v, \ldots\}$. We abbreviate a sequence of transformations

$$G_0 \Rightarrow_{PT}^{\delta_1} G_1 \Rightarrow_{PT}^{\delta_2} \ldots \Rightarrow_{PT}^{\delta_{n-1}} G_{n-1} \Rightarrow_{PT}^{\delta_n} G_n$$

by $\stackrel{*}{\Rightarrow}_{PT}^{\delta}$, where $\delta = \delta_n \ldots \delta_1$.

Notice that the rules in Figure 4.1 only perform so-called pre-unification. Pre-unification differs from unification by the handling of so-called **flex-flex pairs.** These are equations of the form $\lambda\overline{x_k}.P(\ldots) =^? \lambda\overline{x_k}.P'(\ldots)$. Huet [Hue76] showed that such pairs of order three may not have a MCSU: there may exist an infinite chain of unifiers, one more general than the other, without any most general one. The important idea to remedy this situation is that flex-flex pairs are guaranteed to have at least one unifier, e.g. $\{P \mapsto \lambda\overline{x_m}.a, P' \mapsto \lambda\overline{x_n}.a\}$. The idea of pre-unification is to handle flex-flex pairs as constraints and not to attempt to solve them explicitly.

A substitution θ is a **pre-unifier** of s and t if the equation $\theta s =^? \theta t$ can be simplified by Deletion, Decomposition and Elimination to a set of Flex-Flex pairs. In other words, $\theta s =^? \theta t$ only differ at subterms that have variable heads. The notions of MCSU and unification classes in Section 3.5 extend straightforwardly to pre-unification.

In this work, we will often use the restriction to second-order terms. The only place where the restriction to second-order terms simplifies the system is the last rule, projection, where x_i must be of base type. Hence the binding to F in this case is of the simpler form $F = \lambda\overline{x_n}.x_i$, which will be important for our results. As we will often encounter this case, we give an explicit simplified rule:

Second-Order Projection

$$\{\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m})\} \cup S \quad \Rightarrow^{\theta} \quad \{\lambda\overline{x_k}.\theta t_i =^? \lambda\overline{x_k}.v(\overline{\theta t'_m})\} \cup \theta S$$
$$\text{where } \theta = \{F \mapsto \lambda\overline{x_n}.x_i\}$$

The following soundness lemma is easy to show:

Theorem 4.1.2 *System PT is a sound transformation system for higher-order pre-unification.*

When applying the rules of system PT to a set of equations, there are two sources of non-determinism:

1. Which rule to apply
2. to which equation.

It was shown in the early work by Huet [Hue76] that completeness does not depend on how the equations are selected. This is implicit in the proof in [SG89], and is also explained at the end of this section. Furthermore, the only branching occurs when both Imitation and Projection apply to some equation. In other words, application of the first three rules is deterministic [SG89].

Another optimization is stripping off a binder λx if x does not occur. For instance, assume the equation $\lambda x, y.P(x) =^? \lambda x, y.f(a)$, for which the Elimination rule does not apply. Yet clearly, the binder λy is superfluous here and can be removed. Then the elimination rule applies directly.

Example 4.1.3 Consider the unification problem at the root of the search tree in Figure 4.2, which is obtained by the transformations PT in Figure 4.1. Notice that in this example all projection bindings are of the form $\lambda x.x$. The failure cases are caused by a clash of distinct symbols and are abbreviated. The partial bindings of the successful path yield the only solution $\{F \mapsto \lambda x.g(x,x), G_1 \mapsto G, G_2 \mapsto G\}$.

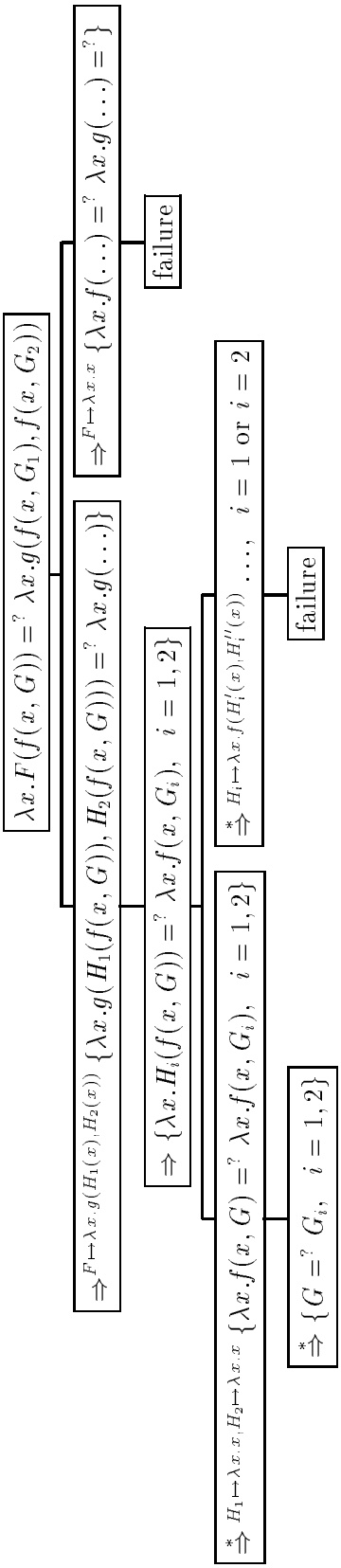

Figure 4.2: Search Tree with System PT

The following general result on higher-order unification will be important for results in the further sections and can e.g. be found in [Sny91].

Lemma 4.1.4 *If θ is a maximally general pre-unifier of $s =^? t$, then $\mathcal{D}om(\theta) \subseteq \mathcal{FV}(s,t)$.*

It should be mentioned that the Elimination rule is not needed for completeness: any equation of the form $P =^? \lambda\overline{x_k}.t$ where Elimination applies can be solved by repeated imitation and projection, until only flex-flex pairs remain. The only difference is that more Flex-Flex pairs may remain, as the Elimination rule also applies to such pairs. We sometimes use the restriction that Elimination is not applied to Flex-Flex pairs, which is sufficient for decidability results. The same is done in the algorithm presented by Snyder et al. [SG89].

This leads to another interesting observation: in contrast to Elimination, Imitation and Projection only compute substitutions that map terms to higher-order patterns. Composing pattern substitutions again yields pattern substitutions.

Fact 4.1.5 *For higher-order pre-unification it is sufficient to consider pattern substitutions.*

Intuitively, this can be explained as for pre-unification terms only have to agree at non-variable positions.

Completeness of Higher-Order Unification

We sketch in the following the completeness result for PT along the lines of [SG89], where the full treatment can be found. We say that a substitution δ **approximates** a substitution θ for a variable F if there exists a substitution θ' with

- $\mathcal{D}om(\theta') = \mathcal{D}om(\theta) - \{F\} \cup \mathcal{R}ng(\delta)$
- $\theta F = \theta'\delta F$
- $\theta =_W \theta'$, where $W = \mathcal{D}om(\theta) - \{F\}$.

The following results are adapted from [SG89]. The next lemma shows that partial bindings approximate solutions.

Lemma 4.1.6 *For any flex-rigid equation $\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m})$ with solution θ there exists a partial binding δ for F such that δ approximates θ.*

Theorem 4.1.7 (Completeness of PT) *If $s =^? t$ has solution θ, i.e. $\theta s = \theta t$, then $\{s =^? t\} \stackrel{*}{\Rightarrow}^{\delta}_{PT} F$ such that δ is more general than θ and F is a set of flex-flex goals.*

Proof The proof proceeds by induction on the following lexicographic termination ordering on $(\overline{E_n}, \theta)$, where for $\overline{E_n}$ is a system of equations with solution θ.

- A: compare the multiset of sizes of the bindings in θ, if equal
- B: compare the multiset of sizes of the equations $\overline{E_n}$.

Notice that a transformation not only changes $\overline{G_n}$, but also the associated solution has to be updated. That is, in case of a binding $F \mapsto t$, the variable F is removed from θ, and, if it is a partial binding, solutions for the new variables in t are added.

If E is in solved form, nothing remains to show. Otherwise, select some non flex-flex equation from $\overline{E_n}$. It is trivial to see that at least one transformation must apply. For each case we show that the ordering is reduced and that the solution is approximated. If the equation is a trivial pair $s =^? s$, ordering B is reduced. In case of the Decomposition rule B is reduced.

When eliminating an equation $F =^? t$, the binding $\{F \mapsto t\}$ clearly approximates θ as $\theta F = \theta t$. Since the new solution contains fewer bindings, A is reduced.

For the Imitation and Projection rule consider an equation $\lambda\overline{x_k}.F(\overline{t_n}) =^? t$. In this case, Lemma 4.1.6 shows that there exists a partial binding δ that approximates θ with θ', i.e. $\theta = \theta'\delta$. Furthermore, all new bindings in θ' for the new variables in δF are smaller than the binding for F in θ, thus reducing A. □

It is easy to see from the recursive structure of the last proof that the completeness does not depend on the selection of goals. Each subgoal is solvable independently, or it is a flex-flex equation. In contrast to the first-order case, the selection is more limited, as flex-flex goals are delayed. Notice that flex-flex goals can become non flex-flex pairs by instantiation.

4.2 Unification of Higher-Order Patterns

Unification of higher-order patterns is a special case of higher-order unification that proceeds similar to first-order unification. The main advantage is that most general unifiers exist for patterns. Compared to higher-order unification, there is no choice between Projection and Imitation. Only the flex-flex cases are more involved than the first-order case. Using efficient data structures, Qian [Qia93] showed that a linear-time implementation of pattern unification is possible.

The following set of rules for unification of higher-order patterns is slightly adapted from [Nip93a]. The exposition there includes a rule that strips off binders, i.e.

$$\lambda x.s =^? \lambda x.t \Rightarrow s =^? t$$

This assumes that bound variables are distinguished syntactically and is in fact closer to an implementation, as working with full binders is rather tedious. Notice that the η-extended form is often not practical, in particular for variables, free or bound, of higher type. Then η-expansion has to be performed during unification.

The transformations in Figure 4.3 work on lists, as the order of application is important for termination [Nip93a]. The problem is that the algorithm introduces new variables on the way and repeating this eagerly may lead to non-termination. For instance, consider $\{c(X) = Y, Y =^? X\} \Rightarrow_{PU} \{c(X) =^? c(Y_1), c(Y_1) =^? X\}$. Here the occurs check applies only after a decomposition and an elimination whereas repeated imitation diverges.

A different method for solving equations of the form $\lambda\overline{x_k}.P(\overline{y_n}) =^? t$ is presented in a more general context in Section 5.1. This method does not introduce temporary variables and is in fact closer to an implementation (e.g. [Nip93a]).

Deletion

$$[t =^? t \mid S] \quad \Rightarrow \quad S$$

Decomposition

$$[\lambda\overline{x_k}.f(\overline{t_n}) =^? \lambda\overline{x_k}.f(\overline{t'_n}) \mid S] \quad \Rightarrow \quad [\overline{\lambda\overline{x_k}.t_n =^? \lambda\overline{x_k}.t'_n} \mid S]$$

Elimination

$$[F =^? t \mid S] \quad \Rightarrow^\theta \quad \theta S \quad \text{if } F \notin \mathcal{FV}(t) \text{ and where } \theta = \{F \mapsto t\}$$

Imitation/Projection

$$[\lambda\overline{x_k}.F(\overline{y_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m}) \mid S] \quad \Rightarrow^\theta \quad [\overline{\lambda\overline{x_k}.H_m(\overline{y_n}) =^? \lambda\overline{x_k}.\theta t'_m} \mid \theta S]$$
where $\theta = \{F \mapsto \lambda\overline{y_n}.v(\overline{H_m(\overline{y_n})})\}$,
v is a constant or $v \in \{\overline{y_n}\}$, and
$F \notin \mathcal{FV}(\lambda\overline{x_k}.v(\overline{t'_m}))$

Flex-Flex Same

$$[\lambda\overline{x_k}.F(\overline{y_n}) =^? \lambda\overline{x_k}.F(\overline{y'_n}) \mid S] \quad \Rightarrow^\theta \quad \theta S \quad \text{where } \theta = \{F \mapsto \lambda\overline{x_n}.F'(\overline{z_p})\}$$
and $\{\overline{z_p}\} = \{y_i \mid y_i = y'_i\}$

Flex-Flex Diff

$$[\lambda\overline{x_k}.F(\overline{y_n}) =^? \lambda\overline{x_k}.F'(\overline{y'_m}) \mid S] \quad \Rightarrow^\theta \quad \theta S \quad \text{where}$$
$\theta = \{F \mapsto \lambda\overline{y_n}.H(\overline{z_p}), F' \mapsto \lambda\overline{y'_m}.H(\overline{z_p})\}$
and $\{\overline{z_p}\} = \{\overline{y_n}\} \cap \{\overline{y'_m}\}$

Figure 4.3: System PU for Pattern Unification

The algorithm coincides with standard first-order unification algorithms, e.g. [JK91], for first-order terms. Notice that in the Flex-Flex rules any permutation of the bound variables $\overline{z_p}$ is sufficient for computing a most general unifier.

Theorem 4.2.1 ([Mil91a, Nip93a]) *System PU computes a most general unifier for two higher-order patterns if a unifier exists.*

Although this algorithm introduces new variables, in contrast to its first-order companion, it has the following important property:

Lemma 4.2.2 *If θ is a most general unifier of two pattern s and t, then $|\mathcal{FV}(\theta s)| \leq |\mathcal{FV}(s,t)|$.*

Proof by induction on the length of PU reductions. □

This lemma and the following property give some insight on the variables introduced by PU and will be important for some termination proofs in Chapter 5.

A substitution θ is **size increasing**, if $|X(\overline{y_n})| < |\theta X(\overline{y_n})|$ for a pattern $X(\overline{y_n})$ in long $\beta\eta$-normal form. In the first-order case, a most general unifier is either empty or decreases the number of variables. For pattern unification, we also have the Flex-Flex Same case with substitutions of the form $\{H \mapsto \lambda\overline{x_n}.H'(\overline{y_m})\}$, where $\{\overline{y_m}\} \subseteq \{\overline{x_n}\}$. Notice that such substitutions do not increase the size. In Section 5.1 we will show the following result, which is difficult to obtain with the rules of System PU: if θ is a most general unifier of two patterns s and t, then either $|\mathcal{FV}(\theta s)| < |\mathcal{FV}(s,t)|$, or θ is not size-increasing.

Patterns have other important properties. A λ-term can be flattened to a pattern plus constraints as follows. For instance,

$$\lambda x.h(\lambda y.f(H(y, G(a))), G(X))$$

can be flattened to

$$\lambda x.h(\lambda y.f(X_1(x, y)), X_2)$$

with constraints

$$X_1 = \lambda x, y.H(y, G(a)), X_2 = G(X).$$

Formally, **flattening** a term t at position p yields $t[X(\overline{y_n})]_p \wedge X = \lambda\overline{y_n}.t\big|_p$ with $\overline{y_n} = \mathcal{BV}(t,p)$ for some new variable X of appropriate type. Intuitively, the pattern part represents the rigid part of a term.

Proposition 4.2.3 *Assume p and q can be flattened to patterns p' and q' with the constraints C. If p' and q' do not unify then p and q do not unify either.*

4.3 Higher-Order Term Rewriting

We will in general follow the notation of first-order term rewriting, see e.g. [DJ90]. Our definitions for higher-order rewrite systems in this section are inspired from [MN94]. We will often, but not in general require that the left-hand side is a higher-order pattern, as done in [Nip91a, Nip93b]. An important restriction is to use rules of base type only, as it simplifies the definition of the rewrite relation: it is close to the first-order case. For alternatives see [Pol94, Wol93] and for an overview we refer to [Oos94].

Definition 4.3.1 A **rewrite rule** is a pair $l \to r$ such that l is not η-equivalent to a free variable, l and r are long $\beta\eta$-normal forms of the same base type, and $\mathcal{FV}(l) \supseteq \mathcal{FV}(r)$. A **General Higher-Order Rewrite System (GHRS)** is a set of rewrite rules.

Definition 4.3.2 Assuming a rule $(l \to r) \in R$ and a position p in a term s in long $\beta\eta$-normal form, a **rewrite step** from s to t is defined as

$$s \longrightarrow_{p,\theta}^{l\to r} t \quad \Leftrightarrow \quad s\big|_p = \theta l \ \wedge \ t = s[\theta r]_p.$$

We often omit some of the parameters $l \to r, p$ and σ of a rewrite step $\longrightarrow_{p,\theta}^{l\to r}$ and for a rewrite step with some rule from a GHRS R we write $s \longrightarrow^R t$.

Recall that we work with terms in long $\beta\eta$-normal form only, and consider this normalization as implicit, e.g. $l\theta = l\theta\downarrow_\beta^\eta$. Notice that the subterm $s\big|_p$ may contain free variables which used to be bound in s. For instance $(\lambda x.f(g(x)))\big|_{1.1} = g(x)$. The following definition will be used to get a formal handle on these variables.

Definition 4.3.3 An $\overline{x_k}$**-lifter** of a term t **away from** W is a substitution $\sigma = \{F \mapsto (\rho F)(\overline{x_k}) \mid F \in \mathcal{FV}(t)\}$ where ρ is a renaming such that $\mathcal{D}om(\rho) = \mathcal{FV}(t)$, $\mathcal{R}ng(\rho) \cap W = \{\}$ and $\rho F : \tau_1 \to \cdots \to \tau_k \to \tau$ if $x_1 : \tau_1, \ldots, x_k : \tau_k$ and $F : \tau$.

For example, $\{G \mapsto G'(x)\}$ is an x-lifter of $g(G)$ away from any set of variables W not containing G'. For simplicity, we often assume that W contains all variables used so far and leave W implicit. A term t is $\overline{x_k}$**-lifted** if an $\overline{x_k}$-lifter has been applied to t. Similarly, a rewrite rule $l \to r$ is $\overline{x_k}$-lifted, if l and r are $\overline{x_k}$-lifted.

Now we can give an alternative definition for rewriting (see also [Fel92]). We have $s \longrightarrow^{l \to r}_{p,\theta} t$ if $\lambda \overline{x_k}.s|_p = \overline{x_k}.\theta l$ and $t = s[\theta r]_p$, where $\{\overline{x_k}\} = \mathcal{BV}(s,p)$ and $l \to r$ is $\overline{x_k}$-lifted away from $V \supseteq \mathcal{FV}(s)$.

For instance, consider the rewrite step $\lambda x.f(x) \longrightarrow^{f(Y) \to g(Y)}_{\{Y \mapsto x\}} \lambda x.g(x)$. With the latter notion of rewriting, we first apply the lifter $\sigma = \{Y \mapsto Y'(x)\}$ to $f(Y) \to g(Y)$. Then $\lambda x.f(x) = \theta \lambda x.f(Y'(x))$ with $\theta = \{Y' \mapsto \lambda x.x\}$. For rewriting, lifting seems unnecessary, since only matching l with $s|_p$ is performed. However for narrowing, as developed later, unification is needed instead of matching and hence lifting is essential.

In contrast to the first-order notion of term rewriting, $\longrightarrow$ is not stable under substitution: reducibility of s does not imply reducibility of θs. Its transitive reflexive closure is however stable:

Lemma 4.3.4 *Assume an GHRS R. If $s \xrightarrow{*}{}^R t$, then $\theta s \xrightarrow{*}{}^R \theta t$.*

The proof of this seemingly simple lemma is rather involved and can be found in [MN94]; a similar result is shown in [LS93] for conditional rules.

A GHRS where all rules have patterns on the left-hand side is called **HRS**. This corresponds to the original definition in [Nip91a]. We call a rule $l \to r$ **pattern rule,** if both l and r are patterns. Furthermore, an HRS with pattern rules only is called a **pattern HRS**. A rule $l \to r$ is **left-linear**, if l is linear. An HRS is called left-linear, if it consists of left-linear rules.

For programming languages, the set of constants is often divided into **constructors** and defined symbols. A symbol f is called a **defined symbol**, if a rule $f(\ldots) \longrightarrow t$ exists. It is assumed that constructors are injective, i.e. $c(\overline{t}) = c(\overline{t'})$ iff $\overline{t} = \overline{t'}$, and that different constructors build different terms, i.e. $c(\overline{t}) \neq c'(\overline{t'})$ if $c \neq c'$. Constructor symbols are denoted by c and d. A term is a constructor term if no defined symbols occur.

We often identify an HRS R with its associated rewrite relation. For instance, we say an HRS R is terminating, if $\longrightarrow^R$ is terminating. A term is in R**-normal form** if no rule from R applies and a substitution θ is R**-normalized** if θX is in R-normal form for all $X \in \mathcal{D}om(\theta)$. For a term t we denote the R-normal form by $t\downarrow_R$, if uniquely defined, and similarly for substitutions.

A rewrite step $s \longrightarrow^{l \to r}_p t$ is **innermost** wrt. some GHRS R, if s is not R-reducible at a position below p. A sequence of reductions $s \xrightarrow{*}{}^R t$ is innermost, if each step in the sequence is innermost. An **outermost** rewrite step is defined correspondingly as a step where no rewrite step applies above. In programming applications, innermost reduction corresponds to eager evaluation and outermost to lazy evaluation. By abuse of notation, we write $s \longrightarrow^{l \to r}_{\neq \epsilon} t$ for a rewrite step that occurs below the root position of s.

Since β-reduction on patterns only renames bound variables, we obtain the following result on reducibility of substitutions. It generalizes the first-order case and is crucial for narrowing, as developed in Chapter 6.

Fact 4.3.5 *Assume an HRS R and a substitution θ. Then $\theta F(\overline{x_n})$ is R-reducible, iff θF is R-reducible.*

This result will be often used for higher-order patterns, where free variables occur only in the form as in the result above.

4.3.1 Equational Logic

A rewrite system R induces an equivalence on terms. This equational theory $=_R$ is defined by the inference rules in Figure 4.4. It is shown in [Wol93] that the equivalence relation $=_R$ coincides with model theoretic semantics for higher-order equational logic.

Rule	$\dfrac{}{l =_R r}$	$l \to r \in R$
Reflexivity	$\dfrac{}{t =_R t}$	
Symmetry	$\dfrac{s =_R t}{t =_R s}$	
Transitivity	$\dfrac{s =_R t \quad t =_R u}{s =_R u}$	
Abstraction	$\dfrac{s =_R t}{\lambda x.s =_R \lambda x.t}$	
Application	$\dfrac{s =_R s' \quad t =_R t'}{(s\ t) =_R (s'\ t')}$	
Conversion	$\dfrac{s =_{\alpha\beta\gamma} t}{s =_R t}$	

Figure 4.4: Equational Theory of an GHRS R

Notice that in the higher-order case the application rule implies the usual congruence rule of the form

$$\frac{t_1 =_R t'_1, \ldots, t_n =_R t'_n}{f(\overline{t_n}) =_R f(\overline{t'_n})}$$

Also, in the higher-order case, the standard substitution rule

$$\frac{s =_R t}{\theta s =_R \theta t}$$

can be inferred from the above by repeated abstractions and applications. For instance, assume $\theta = \{x \mapsto u\}$, then

$$\frac{\dfrac{s =_R t}{\lambda x.s =_R \lambda x.t} \qquad \dfrac{}{u =_R u}}{(\lambda x.s)u =_R (\lambda x.t)u}$$

For higher-order equational theories, the following equivalence of the equational theory and term rewriting has first been shown for HRS in [Nip91a] and has been extended to GHRS in [MN94].

Theorem 4.3.6 *For any GHRS R the following are equivalent:*

$$s =_R t \quad \Leftrightarrow \quad s\downarrow_\beta^\eta \overset{*}{\leftrightarrow}_R t\downarrow_\beta^\eta$$

The proof in [Wol93], which gives a similar result without restrictions on the left-hand sides only holds for terms in β-normal form, as observed by Nipkow [MN94]. A similar result for conditional equations can be found in [LS93].

4.3.2 Confluence

Some of the important confluence criteria for first-order rewriting (see e.g. [Klo92, Hue80]) have been lifted to the higher-order case. As in the first-order case, most confluence criteria are based on an analysis of overlaps:

A rule $l \to r$ of some HRS **overlaps** with a pattern t, if $\theta t \longrightarrow_p^{l\to r} s$ for some substitution θ at a non-variable position p in t. Since l and t are patterns, we assume that θ is the most general unifier of $t|_p$ and l (modulo lifting). Two rules $l_0 \to r_0$ and $l_1 \to r_1$ have an **overlap**, if $l_1 \to r_1$ overlaps with l_0 or vice versa.

An HRS is called **orthogonal**, if it is left-linear and there are no overlaps. For orthogonal HRS, confluence is shown in [MN94, Nip93b]. For an overview with results in a general setting see [Oos94] (also in [OR94]). Orthogonal HRS cover an important class of rewrite rules: (higher-order) functional programs are left-linear and either allow no overlaps, or only weak overlaps [Oos94], for which confluence holds as shown in [Oos94].

If there exist overlaps, they give rise to so-called critical pairs. A pair (u, v) is called a **critical pair** of $l_0 \to r_0$ and $l_1 \to r_1$ if the rules overlap at position p with substitution θ and $\theta l_i \longrightarrow_p^{l_j\to r_j} u$ and $v = \theta r_i$, where $i \in \{0, 1\}$ and $j = 1 - i$.

The well-known (first-order) critical pair lemma [KB70] has been lifted to HRS:

Theorem 4.3.7 ([MN94, Nip91a]) *An HRS R is locally confluent if all critical pairs (u, v) are joinable, i.e. $u\downarrow_R v$.*

This yields the important result that confluence of terminating HRS is decidable, as local confluence implies confluence for terminating HRS.

For first-order rewriting, there is a difference between confluence and ground confluence [Höl89]. An HRS is **ground confluent** if it is confluent on ground terms of a fixed signature. For higher-order term rewriting ground confluence and confluence coincide, as ground terms may contain local "constants" in the form of bound variables. Then, as in the first-order case without the restriction to a certain signature (see [Höl89] for a detailed discussion), both are equivalent.

4.3.3 Termination

Termination of rewriting is undecidable, but there exist many results for terminating classes of rewrite systems or semi-decision procedures (see e.g. [DJ90]). An ordering $<$ is called a **termination ordering** of some HRS R if $\longrightarrow^R \subseteq >$ and $<$ terminates. Usually, to show termination for an HRS $R = \overline{l_n \to r_n}$, one has to find an ordering $<$ with $\overline{l_n > r_n}$

that extends to $\longrightarrow^R$. For the first-order case, there exist large classes of orderings that are known to extend to $\longrightarrow^R$.

In the higher-order case, such orderings are more difficult, as $>$ must be preserved by higher-order substitutions. The approach in [Pol94] is based on strictly monotonic interpretations of terms in monotonic domains. That is, (higher-order) symbols are interpreted by monotonic functions. An example can be found in Section 7.1. It is shown that an HRS terminates if the interpretation of the right hand-side is smaller than the left-hand side for each rule.

A different approach that extends lexicographic orderings on first-order terms [DJ90] to higher-order terms is shown in [LS93, ALS94a].

A confluent and terminating HRS R is called **convergent**. It follows from Theorem 4.3.6 for convergent R that $s =_R t$ can be decided by comparing $s{\downarrow_R}$ and $t{\downarrow_R}$. For a terminating GHRS R, we define $\longrightarrow^R_{sub}$ as

$$\longrightarrow^R_{sub} = \longrightarrow^R \cup >_{sub} .$$

For the first-order case, termination of this reduction was shown in [JK86]. The proof in the latter can be generalized to the higher-order case as follows. We first need the following trivial lemmata.

Lemma 4.3.8 *If $s \longrightarrow^R t$ for a GHRS R and s is a subterm of s', i.e. $s'|_p = s$, then $s' \longrightarrow^R s'[t]_p$.*

Lemma 4.3.9 *Assume an GHRS R. If*

$$s >^*_{sub} t \longrightarrow^R u$$

then there exists $t' >_{sub} t$ such that

$$s \longrightarrow^R t' >^*_{sub} u$$

Theorem 4.3.10 *The reduction $\longrightarrow^R_{sub} = \longrightarrow^R \cup >_{sub}$ is terminating for a GHRS R if $\longrightarrow^R$ is terminating.*

Proof by contradiction. Assume an infinite sequence of $\longrightarrow^R_{sub}$ reductions. If the reduction does not contain some $>_{sub}$-step or only $>_{sub}$-steps, we clearly have a contradiction. Otherwise, assume the first $>_{sub}$-step occurs after a sequence of n $\longrightarrow^R$-steps. Then by Lemma 4.3.9, we can construct a sequence of length $n + 1$. Repeating this yields a contradiction. □

Chapter 5

Decidability of Higher-Order Unification

In many works concerning higher-order unification [Wol93, BS94, Nad87, Pau94], it is observed that non-termination of higher-order (pre-)unification occurs very rarely in practice. As the known decidability results (see Section 2.4) do not cover many practical cases, we examine decidability of higher-order unification more closely, mostly considering the second-order case. For an overview, we refer again to Figure 2.2.

We show in Section 5.2.1 that unification of a linear higher-order pattern with an arbitrary second-order term is decidable and finitary, if the two terms share no variables. In particular, we do not have to resort to pre-unification, as equations with variables as outermost symbols on both sides (flex-flex) pairs can be finitely solved in this case. Further extensions are discussed in Section 5.2.2. For instance, unifying two second-order terms, where one term is linear, is shown to be undecidable if the terms contain bound variables and decidable otherwise.

Then we develop an extension of higher-order patterns with decidable unification in Section 5.3, where second-order linear variables are permitted. The case with repeated variables is discussed in Section 5.3.2. The main result here is that unification of "induction schemes", e.g. $\forall x.P(x) \Rightarrow P(x+1)$, with first-order terms is decidable.

5.1 Elimination Problems

In this section we consider a particular class of unification problems, called **elimination problems**, of the form

$$\lambda\overline{x_n}.P(\overline{y_m}) =^? \lambda\overline{x_n}.t,$$

where $P \notin \mathcal{FV}(t)$. In the first-order case such equations are trivially solvable, here such an equation may not have a solution due to bound variables. For instance, the unification problem $\lambda x, y.P(x) =^? \lambda x, y.f(y)$ has no solution. Among the applications of elimination problems are certain flex-flex pairs. This will allow later to use unification instead of pre-unification in some cases.

We call this class elimination problems, as they generalize first-order elimination. Secondly, the strategy to solve such goals is to eliminate the bound variables $\{\overline{x_n}\} - \{\overline{y_m}\}$ in t by appropriate substitutions. For instance, the equation

$$\lambda x, y.P(x) =^? \lambda x, y.f(x, X(y))$$

has the most general solution $\{X \mapsto \lambda z.X', P \mapsto \lambda x, y.f(x, X')\}$. This example actually falls into the class of patterns and is thus solvable by System PU. The main difference is that System PU introduces many temporary variables for partial bindings for P. Intuitively, all we need for solving $\lambda\overline{x_n}.P(\overline{y_m}) =^? \lambda\overline{x_n}.t$, where t is a pattern, is the following:

- Let $W = \{\overline{x_n}\} - \{\overline{y_m}\}$.
- If some $x \in W$ occurs on a rigid path in t then fail, otherwise,
- for each occurrence of free a variable $X(\overline{z_n})$ in t, bind X to $\lambda\overline{z_n}.X'(\{\overline{z_n}\} \cap W)$, where X' is a new variable of appropriate type.

Hereby the last expression assumes an arbitrary conversion of the set of arguments to X' to a list. The reason why we explain this special case into such detail is that this strategy is actually used in implementations of PU, see e.g. [Nip93a].

In addition, this strategy shows that for solving elimination problems, no "real" new variables have to be introduced, only the variables in t are mapped to new variables with fewer arguments. As in addition P is bound to some term, the total number of variables decreases, which will be important for some results in Section 5.3.

The main focus of this section is on elimination problems where t is an arbitrary second-order term. For this case, there can be many different solutions to an elimination problem, as the next example shows:

Example 5.1.1 Consider the pair

$$\lambda x, y.F(x) =^? \lambda x, y.F'(F''(x), F''(y)).$$

There are two ways to eliminate y on the rhs, i.e. $\theta_1 = \{F' \mapsto \lambda z_1, z_2.F'_1(z_1)\}$ and $\theta_2 = \{F'' \mapsto \lambda z_1.F''_1\}$, where F'_1 and F''_1 are new variables.

We first need some notation to formalize these ideas. For a variable F of type $\overline{\alpha_n} \to \alpha_0$ we define the **i-th parameter eliminating substitution** $\tau_{F,i}$ as

$$\tau_{F,i} = \{F \mapsto \lambda\overline{x_n}.F'(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)\},$$

where F' is a new variable of appropriate type.

The transformation rules $\Rightarrow_{EL}$ in Figure 5.1 transform triples of the form (θ, l, W), where θ is the computed substitution, l is the list of remaining terms, and W is the set of bound variables to be eliminated. We say system EL succeeds if it reduces a triple to $(\theta, [], W)$. For the flex-flex pair in Example 5.1.1 system EL works as follows, starting with the triple

$$(\{\}, [\lambda x, y.F'(F''(x), F''(y))], \{y\}).$$

Then EL can either eliminate the second argument of F' or it can proceed until the triple $(\{\}, [\lambda x, y.F''(y)], \{y\})$ is reached and then eliminate y. In these two cases, EL succeeds with θ_1 and θ_2, respectively, as in Example 5.1.1. All other cases fail.

Observe that system EL is not optimal, as it can produce the same solution twice. For instance, consider the pair $\lambda x.F =^? \lambda x.F'(F'(x))$. There are two different transformation sequences that yield the unifier $\{F' \mapsto \lambda s.F'', \ldots\}$. More precisely, this happens only if a bound variable occurs below nested occurrences of a variable at subtrees with the same index.

We first show the correctness of EL.

Eliminate

$$(\theta, [\lambda\overline{x_k}.P(\overline{t_n})|R], W) \Rightarrow_{EL} (\tau_{P,i}\theta, \tau_{P,i}[\lambda\overline{x_k}.P(\overline{t_n})|R], W) \quad \text{if } \exists x \in W \cap \mathcal{BV}(\lambda\overline{x_k}.t_i)$$

Proceed

$$(\theta, [\lambda\overline{x_k}.v(\overline{t_n})|R], W) \Rightarrow_{EL} (\theta, [\overline{\lambda\overline{x_k}.t_n}|R], W) \quad \text{unless } v \text{ is a bound variable in } W$$

Figure 5.1: System EL for Eliminating Bound Variables

Lemma 5.1.2 (Correctness of EL) *Let $\lambda\overline{x_k}.P(\overline{y_m}) =^? \lambda\overline{x_k}.t$ be a pair where P does not occur in t. Assume further $W = \{\overline{x_k}\} - \{\overline{y_m}\}$. If $(\{\}, [t], W) \stackrel{*}{\Rightarrow}_{EL} (\theta, [], W)$ then $\theta \cup \{P \mapsto \theta\lambda\overline{y_m}.t\}$ is a unifier of $\lambda\overline{x_k}.P(\overline{y_m}) =^? \lambda\overline{x_k}.t$.*

Proof We show that $\{P \mapsto \theta\lambda\overline{y_m}.t\}$ is a well-formed substitution, i.e. all bound variables in θt are locally bound or are in $\overline{y_m}$. As any successful sequence of EL reductions must traverse the whole term $\lambda\overline{x_k}.t$ to succeed, only bound variables in $\{\overline{y_m}\}$ can remain; occurrences of $\{\overline{x_k}\} - \{\overline{y_m}\}$ are either eliminated by some substitution $\tau_{P,i}$ in rule Eliminate, or the algorithm fails as the rule Proceed does not permit these bound variables.

□

The next lemma states that if θ eliminates all occurrences of variables in W from $\overline{t_n}$, then there is a sequence of EL reductions that approximates θ.

Lemma 5.1.3 *If $\tau[\overline{t_n}] = [\overline{t_n}]$, $\mathcal{BV}(\overline{\theta t_n}) \cap W = \emptyset$, $\theta = \delta\tau$ for some substitution δ, and $\overline{t_n}$ are weakly second-order terms, then there exist a reduction $(\tau, [\overline{t_n}], W) \stackrel{*}{\Rightarrow}_{EL} (\theta', [], W)$ and a substitution δ' such that $\theta = \delta'\theta'$.*

Proof by induction on the sum of the sizes of the terms in $[\overline{t_n}]$. Clearly, each $\Rightarrow_{EL}$ reduction reduces this sum. The base case, where $n = 0$, is trivial. We show that for each such problem some EL step applies and that the induction hypothesis can be applied. Depending on the form of t_1 and the conditions of the rules of EL, we apply different rules. Assume t_1 is of the form $\lambda\overline{x_k}.P(\overline{u_m})$ and $\theta P = \lambda\overline{y_m}.t$. By our variable conventions, we can assume that $W \cap \mathcal{BV}(\theta P) = \emptyset$. As $\lambda\overline{x_k}.P(\overline{u_m})$ is a weakly second-order term, some bound variable from W appears in $\theta\lambda\overline{x_k}.P(\overline{u_m})$ if and only if it appears in some $\theta\lambda\overline{x_k}.u_i$ where $y_i \in \mathcal{BV}(\lambda\overline{y_m}.t) = \mathcal{BV}(\theta P)$: if some u_k is a bound variable, then only renaming takes place, otherwise, u_k must be first-order and hence y_k must occur at a leaf in t. Then let

$$i = Min\{j \mid \exists x \in \mathcal{BV}(\theta\lambda\overline{x_k}.u_j) \cap W\}.$$

The above set describes the indices of bound variables that may not occur in $\theta P = \lambda\overline{y_m}.t$ by assumption on θ, e.g. $y_i \notin \mathcal{BV}(\lambda\overline{y_m}.t)$. If the above set is empty and no j exists, we apply the second rule and can then safely apply the induction hypothesis.

In case the minimum i exists, we have $\mathcal{BV}(\theta\lambda\overline{x_k}.u_i) \cap W \subseteq \mathcal{BV}(\lambda\overline{x_k}.u_i) \cap W$. Hence the Eliminate rule applies with $\tau_{P,i} = \{P \mapsto \lambda\overline{x_m}.P_0(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_m)\}$. Then we can apply the induction hypothesis to $(\tau_{P,i}\tau, \tau_{P,i}[\overline{t_n}], W)$: define δ' such that $\delta' X = \delta X$

if $X \neq P$ and $\delta' P_0 = \lambda y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_m.t$. Notice that δ' is well-formed, as $y_i \notin \mathcal{BV}(\lambda\overline{y_m}.t)$. Clearly, the premises for the induction hypothesis are fulfilled, as $\theta = \delta'\tau_{P,i}\tau$ follows from $\tau P = P$. Then the induction hypothesis assures that both EL succeeds with a substitution θ' and that a substitution δ'' exists such that $\theta = \delta''\theta'$.

The remaining cases of t_1 are trivial as the Proceed rule does not compute substitutions. □

Now we can show that EL captures all unifiers. We use EL to solve elimination problems of the form $\lambda\overline{x_k}.P(\overline{y_m}) =^? t$, where t is not η-equivalent to a free variable. In the latter case the solution considered in the next lemma introduces more new variables than the trivial solution $t \mapsto \lambda\overline{x_k}.P(\overline{y_m})$.

Lemma 5.1.4 (Completeness of EL) *Assume θ is a unifier of a pair of the form $\lambda\overline{x_k}.P(\overline{y_m}) =^? \lambda\overline{x_k}.t$, where $\lambda\overline{x_k}.t$ is not η-equivalent to a free variable and $\lambda\overline{x_k}.P(\overline{y_m})$ is a pattern. Assume further $\lambda\overline{x_k}.t$ is weakly second-order and does not contain P. Let $W = \{\overline{x_k}\} - \{\overline{y_m}\}$. Then there exist a substitution $\theta'' = \theta' \cup \{P \mapsto \theta'\lambda\overline{y_m}.t\}$ and a reduction $(\{\}, [\lambda\overline{x_k}.t], W) \stackrel{*}{\Rightarrow}_{EL} (\theta', [], W)$ such that θ'' is more general than θ.*

Proof It is clear that any unifier must eliminate all bound variables from W on the right-hand side. Then the proof follows easily from Lemma 5.1.3. □

It can be shown that EL computes at most a quadratic number of different substitutions. Let n be the number of occurrences of variables to be eliminated and let m be the maximal number of nested free variables. Then there can be at most m distinct ways to eliminate some particular variable. As m and n are both linear in the size, the maximal number of solutions, i.e. mn, is quadratic.

Observe that EL is not complete for the third-order case. Here, if a free variable has two arguments, one can be a function. If in some solution this function is applied to the other argument, then this function could eliminate, in the above sense, the other argument. For instance, consider the third-order pair $\lambda x, y.F(x) =^? \lambda x, y.F'(\lambda z.F''(z), y)$. Here EL would not uncover the solution

$$\{F' \mapsto \lambda y, z.F'_0(y(z)), F'' \mapsto \lambda x.a, F \mapsto \lambda y, z.F'_0(a)\}.$$

With System EL we can show the following result on pattern unification much easier than with System PU, as EL introduces fewer variables.

Lemma 5.1.5 *Assume θ is a most general unifier of two patterns s and t, then either $|\mathcal{FV}(\theta s)| < |\mathcal{FV}(s,t)|$, or θ is not size-increasing.*

Proof Assume a reduction $[s =^? t] \stackrel{*}{\Rightarrow}^{\theta}_{PU} \overline{G_n}$. If no Elimination, Imitation or Projection is applied, then the substitution is not size increasing; this is trivial for Deletion and Decomposition and simple for the Flex-Flex rules. Otherwise, we apply System PU, but use EL instead for all equations of the form $\lambda\overline{x_n}.F(\overline{y_m}) =^? \lambda\overline{x_n}.t$. For this case, we show that a solution to such an equation reduces the number of variables. It is evident that the number of free variables remains unchanged under the parameter eliminating substitution δ computed by EL. As F is bound to some term $\lambda\overline{x_n}.\delta t$, the number of variables reduces. □

5.1.1 Repeated Bound Variables

In the last section, we did not allow repeated bound variables on the left-hand side. In the next lemma we extend this result to relaxed patterns, which causes some technical overhead. Repeated variables may cause an additional number of distinct unifiers in each case, as there can be different permutations if a repeated variable occurs in the common instance. Consider for example the pair $\lambda x.F(x,x) =^? \lambda x.c(x)$. There are the two solutions $\{F \mapsto \lambda y, z.c(y)\}$ and $\{F \mapsto \lambda y, z.c(z)\}$.

As evident from this example, there can be an exponential number of incomparable unifiers in the general case. Consider for instance $\lambda x.F(x,x) =^? \lambda x.v$, where x occurs in v exactly n times. Then there are 2^n different solutions. Although this may seem very impractical, we conjecture that large numbers of unifiers are rare.

In the following result we do not formalize these possible permutations explicitly. For simplicity, we only specify the properties of the correct permutations. As the number of permutations is clearly finite, this is sufficient, but does not yield an effective algorithm for computing these.

Lemma 5.1.6 *A unification problem $\lambda\overline{x_k}.P(\overline{y_m}) =^? t$ where $\lambda\overline{x_k}.P(\overline{y_m})$ is a relaxed pattern and t is weakly second-order and does not contain P, is finitely solvable.*

Proof Consider a pair $\lambda\overline{x_k}.P(\overline{y_m}) =^? \lambda\overline{x_k}.t$ and assume some bound variables occur several times in $P(\overline{y_m})$. Assume EL succeeds with $(\theta, [], \{\overline{x_k} - \overline{y_m}\})$. Let $p(i,j)$ be the position of the j-th occurrence of x_i in θt. For this solution of EL, all solutions for P are of the form $\{P \mapsto \lambda\overline{z_m}.t'\}$, where $Head(t'|_{p(i,j)}) = z_i$ and $y_i = x_i$ for all positions $p(i,j)$ of some x_i in θt and $Head(t'|_q) = Head(\theta t|_q)$ otherwise. Here the last equations allow for many permutations, as some x_j may occur repeatedly in $\overline{y_m}$. All these permutations are clearly independent from the remaining parts of the computed unifier, as P does not occur elsewhere, and can easily be computed. □

It would be interesting to develop deterministic and efficient implementations of EL that compute the set of all unifiers. For instance, if a variable from W occurs on a path where no free variable occurs, then this branch can safely fail. Furthermore, an effective version should also detect when it produces the same solution twice.

5.2 Unification of a Second-Order with a Linear Term

As second-order unification is undecidable, we are interested in identifying decidable subclasses. The restriction discussed here is that one term of the unification problem is linear, i.e. has no repeated variables. We present in the following several results on the decidability of such unification problems, which range from finitary unification over finitary pre-unification to pure decidability. A major application of the results is narrowing with left-linear rules, as discussed in Section 6.7. In Section 6.7.1 we will extend the results in this section to sets of equational goals.

5.2.1 Unifying Linear Patterns with Second-Order Terms

In this section we show that unification of second-order λ-terms with linear patterns is decidable and finitary. Let us first use system PT to solve the pre-unification problem.

We use in the following weakly second-order terms, since this is needed in the next Chapter.[1]

Lemma 5.2.1 *System PT terminates for a unification problem with two variable-disjoint terms $s =^? t$ if s is a linear pattern and t is weakly second-order. Furthermore, PT terminates with a set of flex-flex pairs of the form $\lambda\overline{x_k}.P(\overline{y_i}) =^? \lambda\overline{x_k}.P'(\overline{u_i})$ where all y_i are bound variables and P is isolated.*

Proof We show that system PT terminates for this unification problem. We start with the goal $s =^? t$ and apply the transformations modulo commutativity of $=^?$ in Figure 4.1. By this we achieve that after any sequence of transformations, all free variables on the left-hand sides (lhs) are isolated in the system of equations, as all newly introduced variables on the lhs are linear also. The latter can easily be seen by examining the cases for Imitation and Projection, the other rules are trivial. Another important invariant is that the left-hand sides remain patterns, which is easy to verify.

Since the first three transformations preserve the set of solutions, as shown in [SG89], we assume that Decomposition is applied after applying Projection to a lhs. We do not apply Elimination to flex-flex pairs, which could increase the size of some rhs if a bound variable occurs repeatedly on the lhs, e.g. $\lambda x.c(x,x) =^? G$.

We use the following lexicographic termination ordering on the multiset of equations:

A: Compare the number of constant symbols on all lhs's, if equal

B: compare the number of occurrences of bound variables on all lhs's that are not below a free variable, if equal

C: compare the multiset of the sizes of the right-hand sides (rhs).

Now we show that the transformations reduce the above ordering:

Deletion trivial

Decomposition A or B is reduced.

Elimination Although this transformation eliminates one equation, it is not trivial that it also reduces the above ordering. Consider the possible equations Elimination is applied to:

- $F =^? \lambda\overline{x_k}.t$: as the free variable F is isolated, A and B remain constant and C is reduced.
- $\lambda\overline{x_k}.a(\ldots) =^? F$: the elimination of an equation with a constant a reduces A.
- $\lambda\overline{x_k}.x_i(\ldots) =^? F$: here B is reduced (and possibly A).

Imitation We have two cases:

[1] This extends the earlier results in [Pre94a].

- $\lambda\overline{x_k}.F(\overline{y_n}) =^? \lambda\overline{x_k}.f(\overline{t_m})$: the imitation binding for F is of the form $F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})$. Now, we replace the above equation by a set of equations of the form $\lambda\overline{x_j}.H_i(\overline{y_n}) =^? \lambda\overline{x_j}.t_i$, where $i = 1, \ldots, m$. Notice that the number of constants on the lhs (A) does not increase, as all y_m are bound variables. Also, B remains unchanged. As F is isolated and hence does not occur on any right-hand side, C decreases.
- $\lambda\overline{x_k}.f(\overline{t_n}) =^? \lambda\overline{x_k}.F(\overline{u_m})$: we obtain an imitation binding as above. Then the number of constant symbols on the lhs's decreases, since F may not occur on the lhs's.

Projection We again have two cases:

- $\lambda\overline{x_k}.F(\overline{y_n}) =^? \lambda\overline{x_k}.y_i(\overline{t_m})$: as $\overline{y_n}$ are bound variables, this rule applies only if the head of the rhs is a bound variable as well, say y_i. Then the case is similar to the Imitation case above, as after Projection, the Decomposition rule applies.
- $\lambda\overline{x_k}.v(\overline{t_n}) =^? \lambda\overline{x_k}.F(\overline{u_m})$: as we have weakly second-order variables on the rhs, we again have two cases. If v is a bound variable, Decomposition applies after Projection and we proceed as in the Imitation case. In the remaining cases, projection bindings are of the form $F \mapsto \lambda\overline{x_m}.x_i$, where x_i is first-order. Hence the lhs's (i.e. A and B) are unchanged, whereas C decreases, as we assume terms in long $\beta\eta$-normal form.

□

So far, we have shown that pre-unification is decidable. To solve the remaining flex-flex pairs, notice that all of these are elimination problems of the form

$$\lambda\overline{x_k}.P(\overline{y_m}) =^? \lambda\overline{x_k}.P'(\overline{u_n}),$$

where P is isolated and $\{\overline{y_m}\}$ are bound variables.

Theorem 5.2.2 *Assume t is a weakly second-order λ-term and s is a linear pattern such that s shares no variables with t. Then the unification problem $s =^? t$ is decidable and finitary.*

Proof From Lemma 5.2.1 we know that PT terminates with a set of flex-flex pairs, where the lhs is a pattern. Then by Lemma 5.1.6 we can use EL to compute a complete and finite set of unifiers for some flex-flex pair, as EL terminates and is finitely branching. This unifier is applied to the remaining equations. Repeat this for all flex-flex pairs. This procedure terminates and works correctly as all lhs's are patterns and only have isolated variables. Notice that a flex-flex pair remains flex-flex when applying a unifier computed by EL. □

We have shown in Section 5.1 that EL may compute an exponential number of solutions when repeated variables are permitted. Clearly, the most concise representation of all unifiers is still a flex-flex pair. Which representation is best clearly depends on the application. For instance, flex-flex pairs may not be satisfactory for programming languages where explicit solutions are desired. For automated theorem proving, flex-flex pairs are a more compact representation and may reduce the search space.

It should also be noted that the unification problem in Lemma 5.2.1 allows for some nice optimizations for implementors. For instance, no occurs check is needed: the proof of Lemma 5.2.1 uses the invariant that all variables on the left-hand sides are isolated. Hence no variable can occur on a left-hand side and at the same time on some right-hand side.

5.2.2 Extensions

In the following sections, we will examine extensions of the above decidability result. First, notice that the linearity restriction is essential; otherwise full second-order unification can easily be embedded. But even with one linear term, this embedding still works:

Example 5.2.3 Consider the unification problem

$$\lambda x.F(f(x, G)) =^? \lambda x.g(f(x, t_1), f(x, t_2)),$$

where t_1 and t_2 are arbitrary second-order terms. By applying the transformations PT it is easy to see (compare to Example 4.1.3) that in all solutions of the above problem $F \mapsto \lambda x.g(x, x)$ and $t_1 =^? t_2$ must be solved, which is clearly undecidable.

Notice that this example requires a function symbol of arity two whereas second-order unification with monadic function symbols is decidable.

Motivated by this example, we consider the following two extensions. First, we assume that arguments of free variables are second-order ground terms. Secondly, we consider the case where an argument of a free variable contains no bound variables. These two cases can be combined in a straightforward way, as shown towards the end of this section. Thus arguments of free variables may either be ground second-order terms or terms with no bound variables. The generalization where only one term is linear follows easily from Example 5.2.3:

Corollary 5.2.4 *It is undecidable to determine if two second-order terms unify, even if one is linear.*

Pre-unification of two linear second-order terms without bound variables is however decidable and finitary, as shown by Dowek [Dow93]. This result is generalized in Section 5.3 to higher-order patterns with linear second-order variables.

Ground Second-Order Arguments to Free Variables

We now loosen the restriction that one term must be a linear pattern. As long as all arguments of free variables are either bound variables or ground second-order terms, we can still solve the pre-unification problem. In particular, for the second-order case, this can be rephrased as disallowing nested free variables. However, we only solve the pre-unification problem, as the resulting flex-flex pairs are more intricate than in the last section.

Similar to the above, we present a termination ordering for a particular strategy of the PT transformations. We will see that in essence only one new case results from these ground second-order terms. This case can be handled separately by second-order

matching, which is decidable and finitary. (It is also an instance of Theorem 5.2.2.) That is, whenever such a matching problem occurs, this is solved immediately (considering all its solutions). Hence we first need a lemma about matching.

Lemma 5.2.5 *Solving a second-order matching problem with system PT yields only solutions that are ground substitutions.*

Proof by induction on the length of the transformation sequence. The base case, length zero, is trivial. The induction step has the following cases:

Deletion,Decomposition trivial

Elimination Consider the equation to which Elimination is applied:

$$\lambda\overline{x_k}.t =^? F$$

The claim is trivial as $\lambda\overline{x_k}.t$ is ground.

Imitation

$$\lambda\overline{x_k}.a(\overline{t_n}) =^? \lambda\overline{x_k}.F(\overline{u_m})$$

The imitation binding for F is of the form $F \mapsto \lambda\overline{y_m}.a(\overline{H_n(\overline{y_m})}$. Now, we replace the above equation by a set of equations of the form

$$\lambda\overline{x_k}.t_i =^? \lambda\overline{x_k}.H_i(\overline{u_m})$$

Clearly, for any matcher θ, $H_i \in \mathcal{D}om(\theta)$, and by induction hypothesis θH_i is ground. Hence in the solution to $\lambda\overline{x_k}.a(\overline{t_n}) =^? \lambda\overline{x_k}.F(\overline{u_m})$, F is mapped to a ground term.

Projection As we have second-order variables, we only have projection bindings of the form $F \mapsto \lambda\overline{y_m}.y_i$, which are trivially ground.

□

This result does not hold for the higher-order case, as noted by Dowek [Dow93]: e.g. $\{F \mapsto \lambda x.x(Y)\}$ is a solution to $F(\lambda x.a) =^? a$, but no complete set of ground matchers exists. Now we can show the desired theorem:

Theorem 5.2.6 *Assume s, t are λ-terms such that t is second-order, s is linear and s shares no variables with t. Furthermore, all arguments of free variables in s are either*

- *bound variables of arbitrary type or*
- *second-order ground terms of base type.*

Then the pre-unification problem $s =^? t$ is decidable and finitary.

Proof We give a termination ordering for system PT with the same additional assumptions as in the proof of Lemma 5.2.1. In addition, we consider solving a second-order matching problem an atomic operation, with possibly many solutions. In particular, after a projection on a lhs, this step eliminates one equation and applies a (ground) substitution to the rhs. It is easy to see that the two premises, only isolated variables and no nested free variables on the lhs's, are invariant under the transformations.

We use the following (lexicographic) termination ordering on the multiset of equations:

A: Compare the number of occurrences of constant symbols and of bound variables that are not below a free variable on a lhs, if equal

B: compare the number of free variables in all rhs's, if equal

C: compare the multiset of the sizes of the rhs's.

Now we show that the transformations reduce the above ordering:

Deletion trivial

Decomposition A is reduced.

Elimination Although one equation is eliminated, it is not trivial that it also reduces the above ordering. Consider the equations this rule is applied to:

- $F =^? \lambda\overline{x_k}.t$: as the free variable F is isolated, A and B remain constant and C is reduced.
- $\lambda\overline{x_k}.v(\ldots) =^? F$: the elimination of an equation with a constant or bound variable v reduces A, as F does not occur on any rhs.

Imitation We have two cases, where a is a constant:

- $\lambda\overline{x_k}.F(\overline{u_m}) =^? \lambda\overline{x_k}.a(\overline{t_n})$: the imitation binding for F is of the form $F \mapsto \lambda\overline{y_m}.a(\overline{H_n(\overline{y_m})})$. Now we replace the above equation by a set of equations of the form $\lambda\overline{x_j}.H_i(\overline{u_m}) =^? \lambda\overline{x_j}.t_i$. Notice that the number of constants and bound variables not below a free variable on the lhs's (A) does not increase. As F is an isolated variable and does not occur on any right-hand side, B remains unchanged and C decreases.
- $\lambda\overline{x_k}.a(\overline{t_n}) =^? \lambda\overline{x_k}.F(\overline{u_m})$: we obtain an imitation binding as above, and the number of constant symbols on the lhs's (i.e. A) decreases, since F may not occur on the lhs.

Projection We again have two cases:

- $\lambda\overline{x_k}.F(\overline{t_m}) =^? \lambda\overline{x_k}.v(\overline{u_k})$: since F is an isolated variable, we obtain a single matching problem $\lambda\overline{x_k}.t_i =^? \lambda\overline{x_k}.v(\overline{u_k})$ or, if the i-th argument is a bound variable, the proof works as the case above (similar to the proof of Theorem 5.2.2). In the former case, any solution to this is a ground substitution by Lemma 5.2.5. Hence either B is reduced or, if the substitution is empty, B remains unchanged and C decreases.
- $\lambda\overline{x_k}.v(\overline{t_n}) =^? \lambda\overline{x_k}.F(\overline{u_m})$: as we have second-order variables on the rhs, we only have projection bindings of the form $F \mapsto \lambda\overline{y_m}.y_i$. Then the lhs's (i.e. A) are unchanged and both B and C decrease.

□

It might seem tempting to apply the same technique to arguments that are third-order ground terms, as third-order matching is known to be decidable. However, there can be an infinite number of matchers and without a concise representation for these the extension of the above method seems difficult.

No Bound Variables in an Argument of a Free Variable

We show that the remaining case, where an argument of a free variable contains no (outside-)bound variables, can be reduced to a simpler case. This method checks unifiability, but does not give a complete set of unifiers.

Theorem 5.2.7 *Assume* $s =^? u[H(t_1, \ldots, t_i, \ldots, t_n)]_p$ *and* t *are variable disjoint* λ*-terms such that* s *is linear. Assume further* $\mathcal{OBV}(\lambda\overline{y_m}.t_i) = \emptyset$*, where* $\overline{y_m} = \mathcal{BV}(s, p)$*. Then the unification problem* $s =^? t$ *has a solution, iff* $\lambda x_0.u[H(t_1, \ldots, x_0, \ldots)] =^? \lambda x_0.t$*, where* x_0 *does not occur elsewhere, is solvable.*

Proof Consider the unification problem

$$u[H(t_1, \ldots, t_i, \ldots, t_n)]_p =^? t$$

where H occurs only once in $u[H(t_1, \ldots, t_i, \ldots)]_p$ and t_i does not contain bound variables. Assume $\{X_1, \ldots, X_m\} = \mathcal{FV}(t_i)$. Let a solution to this problem be of the form $\{H \mapsto \lambda\overline{x_n}.t_0\} \cup \{\overline{X_o \mapsto u_o}\} \cup S$. As H does not occur elsewhere, we can construct a substitution $\theta = \{H \mapsto \lambda\overline{x_n}.\{x_i \mapsto t_i'\}t_0\} \cup S$, where $t_i' = \{\overline{X_o \mapsto u_o}\}t_i$, which is a solution to

$$\lambda x_0.u[H(t_1, \ldots, x_0, \ldots)]_p =^? \lambda x_0.t$$

Notice that θ is well-formed, as $\lambda\overline{y_m}.t_i$ does not contain (outside) bound variables. The other direction is simple, since x_0 does not occur elsewhere, i.e. not in an instance of $\lambda x_0.t$. □

Notice that the above procedure only helps deciding unification problems but does not imply that pre-unification or even unification is finitary.

Putting It All Together

Now we can combine the previous results. Recall that the remaining case is undecidable in general.

Theorem 5.2.8 *Assume* s, t *are* λ*-terms such that* t *is second-order,* s *is linear and* s *shares no variables with* t*. Furthermore, if* $s|_p = F(\overline{t_n})$*, then all* $\overline{t_n}$ *are either*

- *bound variables of arbitrary type or*
- *second-order ground terms of base type or*
- *second-order terms of base type without bound variables form* $\mathcal{BV}(s, p)$*.*

Then the unification problem $s =^? t$ *is decidable.*

Proof First apply Theorem 5.2.7 to the unification problem until s has no nested free variables. This argument can be applied repeatedly, as the lhs is linear and hence the substitutions of multiple applications do not overlap. Then Theorem 5.2.6 can be applied to decide this problem. □

A special case often considered (e.g. [Gol81]) is terms with second-order variables, but no bound variables. Then we get the following stronger result as an instance of Theorem 5.2.8:

Proposition 5.2.9 *Assume* s, t *are second-order* λ*-terms such that* s *is linear and shares no variables with* t*. Furthermore,* s *contains no bound variables. Then the unification problem* $s =^? t$ *is decidable.*

5.3 Relaxing the Linearity Restrictions

In this section we discuss unification problems with shared and repeated variables which were disallowed in the last section. The first result is an extension of higher-order patterns. The only known extension of higher-order patterns with unitary unification is due to Dale Miller [Mil91a]. Miller permits arguments to free variables that are patterns, but must have a bound variable as the outermost symbol. For instance, $\lambda x, y.P(x, y(f(x)))$ is permitted. The decidability result in the next section below allows second-order variables with patterns as arguments, as long as these variables occur only once.

The results in Section 5.3.2 show that unitary unification is easily lost when going beyond higher-order patterns. A further class with decidable unification is considered that does not subsume higher-order patterns but is interesting for some applications. For instance, unification of first-order terms with a term $\forall x.P(x) \Rightarrow P(x+1)$ is shown to be decidable.

5.3.1 Extending Patterns by Linear Second-Order Terms

We consider in the following an extension of higher-order patterns where subterms of the form $X(\overline{t_n})$ are permitted for some patterns $\overline{t_n}$ as long as X is second-order and does not occur elsewhere. This generalizes a result by Dowek [Dow93] which covers second-order terms with linear second-order variables, but without bound variables. Hence it does not subsume higher-order patterns.

We first need the following notation. A **linear second-order system** of equations is of the form

$$\overline{\lambda\overline{x_k}.X_n(\overline{t_{n_m}}) =^? \lambda\overline{x_k}.t_n},$$

where all $\overline{X_n}$ are distinct and do not occur elsewhere and furthermore all $\overline{\lambda\overline{x_k}.t_{n_m}}$ and $\overline{\lambda\overline{x_k}.t_n}$ are higher-order patterns. By abuse of notation, we write $\overline{t_{n_m}}$, avoiding nested bars.

For the next result recall from Section 4.1 that the elimination rule in System PT is not needed for completeness.

Theorem 5.3.1 *Unification of linear second-order systems is decidable.*

Proof We show that System PT for higher-order pre-unification terminates for linear second-order systems if the elimination rule is not used. We use the following lexicographic termination ordering for a system $S = \{\overline{\lambda\overline{x_k}.X_n(\overline{t_{n_m}}) =^? t_n}\}$:

A: $|\mathcal{FV}(\overline{t_n}) \cup \mathcal{FV}(\overline{\lambda\overline{x_k}.t_{n_m}})|$

B: the multiset of sizes of $\overline{t_n}$

Let us show that the transformations of PT reduce this ordering. After a projection on the left (it may not occur on the right), an equation between two patterns is created. We consider solving this as an atomic operation (possibly reducing A). We maintain the invariant that the system remains a linear second-order system, which is easy to show. Hence we only have to consider the imitation and projection cases:

Imitation: in this case, the number of isolated variables on the left increases, but A remains constant and B is reduced after decomposition.

Projection reduces one equation to an equation between two patterns. Applying a solution of this equation (if it exists) to the remaining goals yields two cases as in Lemma 5.1.5: either A is reduced, or, if A remains the same, the substitution must not increase the size and thus B is reduced as one equation is removed.

□

Now we can show the desired result, where we represent the non-pattern terms in the unification problem by a substitution. This in fact yields a more general result, as the permitted non-pattern subterms may occur repeatedly. For instance, $f(X(a), X(a)) =^? p$ falls into this class, but $f(X(a), X(b)) =^? p$ does not, where p is a pattern.

Theorem 5.3.2 *Assume a substitution* $\theta = \{\overline{X_n \mapsto \lambda\overline{x_k}.X'_n(\overline{t_{n_m}})}\}$, *where* $\overline{t_{n_m}}$ *are patterns, and two patterns* s *and* t. *If all* $\overline{X'_n}$ *are distinct, second-order and further do not occur elsewhere, then the unification problem* $\theta s =^? \theta t$, *is decidable.*

Proof It is sufficient to solve the pattern unification problem $s =^? t$ first, yielding a pattern substitution δ in a successful case. Then

$$\overline{\lambda\overline{x_k}.X'_n(\delta\overline{t_{n_m}}) =^? \lambda\overline{x_k}.\delta X_n}$$

is a second-order linear system and is decidable by Theorem 5.3.1. □

A typical application of Theorem 5.3.2 are contexts, which are often used to describe positions in terms. These are sometimes viewed as "terms with holes" and these holes are written as boxes. For instance, $f(\Box, a)$ and $C(\Box)$ can be viewed as contexts. It is clearly much more precise to express contexts by second-order terms. In particular, if a term has several different "holes". For instance, we would write $\lambda\Box.f(\Box, a)$ and $\lambda\Box.C(\Box)$ instead of the above and would let β-reduction perform the substitution for concrete values for "holes". Thus contexts can be modeled by linear second-order variables. With the above result, we have a method to determine if a first-order term t unifies with a (linear) context filled with some term. For instance, in order to find overlaps of two rules $l_i \to r_i, \quad i = 0, 1$ in an abstract fashion, the equations $(\lambda\Box.C(\Box))l_i =^? l_{1-i}, \quad i = 0, 1$ are to be solved. Notice that the last unification problems permit trivial solutions, e.g. $\{C \mapsto l_{i-1}\}$, which are not of interest here.

As another example, we can model term rewriting with contexts. Assume a rule $l \to r$. Checking if s is reducible by this rule is done by matching $(\lambda\Box.C(\Box))l$ with s. Similarly, for narrowing, as we see in the next chapter, unification of $(\lambda\Box.C(\Box))l$ with s is needed.

5.3.2 Repeated Second-Order Variables

We show in this section another decidability result for second-order unification that is tailored for a particular application. As we aim at relaxing the linearity conditions in results of the last section, we need several technical restrictions. Notice that this extension easily leads to infinitary unification problems. For instance, if ground terms are permitted as arguments to free variables, the following example shows that there exist infinitely many unifiers: the problem

$$F(f(a)) =^? f(F(a))$$

has the solutions $\{F \mapsto \lambda x.f^n(a)\},\ n \geq 0$ where $f^0(X) = X$ and $f^{n+1}(X) = f(f^n(X))$.

Apart from the above example, equations of the form $F(t) =^? t'$, where F occurs in t', are unsolvable in most cases. We conjecture that the solvable cases are based on some symmetries. For instance, consider the equation

$$F(f(a,a)) =^? f(F(a),F(a)).$$

The solutions are of the form

$$\{F \mapsto \lambda x.x\}, \{F \mapsto \lambda x.f(x,x)\}, \{F \mapsto \lambda x.f(f(x,x),f(x,x))\}, \ldots$$

We conjecture that all solutions to equations $F(t) =^? s$ with $F \in \mathcal{FV}(s)$ are of such a form and can possibly be described by finite automata or grammars [Tho90].

Some interesting examples fall into this class. Consider unification with a typical induction scheme:

$$P(0), \forall x.P(x) \Rightarrow P(x+1) \vdash \forall x.P(x)$$

Typically in such formulas, some arguments to free variables are not bound variables, but ground (constructor) terms, here $P(x+1)$. Recall that the quantifier $\forall$ can be viewed as a second-order constant and that $\forall x.P(x)$ is nicer syntax for $\forall(\lambda x.P(x))$.

Unification of a term with a repeated free variable with some higher-order pattern permits infinitely many solutions: consider for instance

$$\forall x.f(P(x)) \Rightarrow P(f(x)) =^? \forall y.X(y) \rightarrow X(y)$$

which is similar to the above unification problem $F(f(a)) =^? f(F(a))$.

The main result of this section is that unification of such terms with (quasi) first-order terms is decidable. A term $\lambda\overline{x_k}.t$ is **quasi first-order** if t is first-order. For instance, $\lambda x.F(x)$, $f(\lambda x.x)$ are not quasi first-order, but $\lambda x.f(x,P)$ is quasi first-order.[2] A simple property of quasi first-order terms we will use is the following: if t is quasi first-order, p is a pattern, and $\theta p = t$ then θ is quasi first-order on $\mathcal{FV}(p)$.

A **pattern with ground arguments** is a pattern with the exception that arguments to free variables are ground terms which contain at least one outside bound variable but no local binders. An example is $\lambda x.P(x+1,x)$, but $\lambda x.P(f(\lambda y.y))$ is not.

Lemma 5.3.3 *Assume $\lambda\overline{x_k}.t$ is a quasi first-order term, $\lambda\overline{x_k}.P(\overline{t_n})$ is a second-order pattern with ground arguments. Then the unification problem $\lambda\overline{x_k}.P(\overline{t_n}) =^? \lambda\overline{x_k}.t$ is decidable and, furthermore, if θ is a maximally general solution then θP is quasi first-order.*

Proof Decidability follows from Theorem 5.3.1 as $\lambda\overline{x_k}.P(\overline{t_n}) =^? \lambda\overline{x_k}.t$ is a linear system and $P \notin \mathcal{FV}(\lambda\overline{x_k}.t)$ since t is first-order. We apply Imitation and Projection of System PT, except on elimination problems. This terminates by Theorem 5.3.1 for second-order linear systems. In case of a Projection, a matching problem of the form $\lambda\overline{x_k}.t_i =^? \lambda\overline{x_k}.t'$, where t' is quasi first-order, is created. This only has quasi first-order solutions, since t_i has no local binders.

Imitation may create elimination problems of the form $\lambda\overline{x_k}.P'(\overline{t_n}) =^? \lambda\overline{x_k}.X$, which can be solved by System EL. This yields the solution $\{P' \mapsto \lambda\overline{x_k}.X\}$, as all $\overline{t_n}$ contain bound

[2] This is more restrictive than the definition of quasi-first-order in [LS93, ALS94a].

variables. As P' cannot occur elsewhere in a linear system, the remaining unification problems do not change and the system remains linear. Thus decidability of the original unification follows and maximally general unifiers do exist. Furthermore, any solution for some X on the right is quasi first-order. Hence for any solution θ computed, $\lambda\overline{x_k}.\theta t$ is quasi first-order. This entails that θP must be quasi first-order as well, as $\theta P(\overline{t_n}) = \theta t$: if $\lambda\overline{x_k}.\theta P$ is not quasi first-order, then $\lambda\overline{x_k}.\theta P(\overline{t_n})$ cannot be quasi first-order, as all $\overline{t_n}$ are ground and of base type. □

Lemma 5.3.4 *Assume $\lambda\overline{x_k}.p$ is a higher-order pattern where no abstractions occur in p, and $\lambda\overline{x_k}.t$ is a quasi first-order term. Then maximally general solutions of the unification problem $\lambda\overline{x_k}.p =^? \lambda\overline{x_k}.t$ are quasi first-order.*

Proof We first construct a solution θ to the problem $\lambda\overline{x_k}.p =^? \lambda\overline{x_k}.t$. Then we show that θ cannot map some free variable in $\lambda\overline{x_k}.t$ to a term containing bound variables.

We apply the rules of system PU except on elimination problems. Since p has no locally bound variables and t is quasi first-order, we can assure the invariant that there are no bound variables except $\overline{x_k}$. Elimination problems are either of the form $\lambda\overline{x_k}.t_1 =^? \lambda\overline{x_k}.X$ or of the form $\lambda\overline{x_k}.P(\overline{y_m}) =^? \lambda\overline{x_k}.t_2$, and are solved by System EL. If an elimination problem is of the first form, it is clear that any solution for X must be first-order, as all bound variables in t_1 must be eliminated and since there are no locally bound variables on some lhs. For the second form of elimination problems, if solvable, the obvious solution $P \mapsto \lambda\overline{y_m}.t_2$ is quasi first-order. Thus any solution θ computed is quasi first-order.

This entails that $\theta\lambda\overline{x_k}.p = \theta\lambda\overline{x_k}.t$ is quasi first-order and hence θ must be quasi first-order for the free variables in p as well. □

Now we are ready for the main result of this section.

Theorem 5.3.5 *Assume $\lambda\overline{x_k}.p$ is a higher-order pattern where no abstractions occur in p, $\overline{\lambda\overline{x_k}.P(\overline{t_{n_m}})}$ are second-order patterns with ground arguments, $\overline{\lambda\overline{x_k}.P_n(\overline{y_{n_o}})}$ are patterns, and $\lambda\overline{x_k}.t$ is quasi first-order. Then the unification problem*

$$\lambda\overline{x_k}.p =^? \lambda\overline{x_k}.t, \overline{\lambda\overline{x_k}.P(\overline{t_{n_m}}) =^? \lambda\overline{x_k}.P_n(\overline{y_{n_o}})}$$

where $P \notin \mathcal{FV}(\lambda\overline{x_k}.p, \lambda\overline{x_k}.t)$ is decidable.

Proof The structure of the proof is as follows. We solve the equation $\lambda\overline{x_k}.p =^? \lambda\overline{x_k}.t$ as in Theorem 5.3.4, yielding a quasi first-order substitution θ. Then we show that the remaining equations can be solved under this substitution. Wlog. we assume $P \notin \mathcal{FV}(\theta)$.

After solving $\lambda\overline{x_k}.p =^? \lambda\overline{x_k}.t$, there remain the following cases for the equations

$$\overline{\lambda\overline{x_k}.P(\overline{t_{n_m}}) =^? \lambda\overline{x_k}.\theta P_n(\overline{y_{n_o}})} \tag{5.1}$$

- All equations in (5.1) are flex-flex and thus have a solution. Otherwise
- some $P_i \in \mathcal{D}om(\theta)$. We solve the i-th equation $\lambda\overline{x_k}.P(\overline{t_{i_m}}) =^? \lambda\overline{x_k}.\theta P_i(\overline{y_{i_m}})$ as in Lemma 5.3.3, as $\lambda\overline{x_k}.\theta P_i(\overline{y_{i_m}})$ is quasi first-order. This yields a quasi first-order substitution θ' for P. Applying this to the remaining equations yields a set of equations with higher-order patterns only. Thus solving the remaining goals is decidable.

□

The last result applies directly to first-order theorem proving with additional induction schemes written as second-order formulas. For instance, consider a data structure for binary trees with the destructors $left\_tree, right\_tree$. Then a premise of an induction scheme for binary trees may read as

$$\forall x.P(left\_tree(x)) \wedge P(right\_tree(x)) \Rightarrow P(x)$$

Encoding a unification problem of a term $\forall x.t$ with the above scheme into in the form required for the last result yields:

$$\begin{aligned} \forall x.P_1(x) \wedge P_2(x) \Rightarrow P_3(x) &=^? \forall x.t \\ \forall x.P(left\_tree(x)) &=^? \forall x.P_1(x) \\ \forall x.P(right\_tree(x)) &=^? \forall x.P_2(x) \\ \forall x.P(x) &=^? \forall x.P_3(x) \end{aligned}$$

Although we have found another decidable class of unification problems, the result also shows that it is increasingly complicated to describe these classes.

5.4 Applications and Open Problems

As mentioned in the introduction, higher-order unification is currently used in several theorem provers, programming languages, and logical frameworks. With the above results we can now develop simplified and somewhat restricted versions of the above applications that enjoy decidable unification. It should be mentioned that several systems such as Elf [Pfe91] and Isabelle[3] have already resorted to higher-order patterns, where unification behaves much like the first-order case.

The main restriction we use to achieve decidability is linearity. There is an interesting variety of applications where linearity is a common and sometimes also useful restriction. The main application and also the original motivation for this work is higher-order narrowing, which will be developed in the next chapter.

Recall for instance the rule

$$map(F, cons(X, Y)) \rightarrow cons(F(X), map(F, Y))$$

which has a linear pattern as the left-hand side. Interestingly, when coding functions such as map into predicates, as for instance done in higher-order logic programming [NM88], the head of the literal, e.g.

$$map_P(F, cons(X, Y), cons(F(X), L)) :- map_P(F, Y, L),$$

is not linear. However, when this rule is used only on goals of the form $map_P(t, t', Z)$, where Z is a fresh variable,[4] then the unification problem is decidable as it is equivalent to a unification with a linear term. Thus our results also explain to some extent why unification in higher-order logic programming rarely diverges.

Another application area is type inference, which is mostly based on unification, whereby decidable static type inference for programming languages is desired. In many

[3] Isabelle still uses full higher-order pre-unification, if the terms are not patterns.

[4] Such variables are also called "output-variables" in [Red86].

advanced type systems such as Girard's system F [GLT89] variables may range over functions from types to types, i.e. second-order type variables. In particular, Pfenning [Pfe88] relates type inference in the n-th-order polymorphic λ-calculus with n-th-order unification. As another example, for SML [MTH90] some restrictions avoid second-order unification problems in the module system. Thus progress in higher-order unification may help finding classes where type inference is decidable. However, non-unitary unification often means no principal (i.e. most general) type.

Other applications are described in the following.

Theorem Proving

Higher-order theorem provers often work with some form of a sequent calculus, where most rules have linear premises and conclusions e.g.

$$\frac{? \vdash A \quad ? \vdash B}{? \vdash A \wedge B}$$

Furthermore, non-linear unification problems occur mostly with rewriting, e.g. with rules such as $P \wedge P \longrightarrow P$. For rewriting, however, only matching is required.

Another interesting result for theorem proving was discussed in Section 5.3: unification of first-order terms with induction schemes of the form $\forall x.P(x) \Rightarrow P(x+1)$.

Associative Unification

Unification modulo the law of associativity was an open problem for a long time, until Makanin [Mak77] showed its decidability.

It is known that associative unification can be embedded into higher-order unification, see e.g. [Pau94]. With Theorem 5.2.6, we can show the decidability of a class of problems that extends associative matching (which is rather trivial). The idea for the encoding is that function application is associative, for instance

$$\lambda x.(\lambda y.f(g(y)))h(y) = \lambda x.f(g(h(x))).$$

Thus associative lists are coded by functions, e.g. the list $[a,b]$ is coded by the term $\lambda x.cons(a, cons(b,x))$. It may seem that this representation for lists is clumsy, but it has the advantage that appending a list to another can be done by a single β-reduction. This representation has been discussed in [Hug86] and corresponds to the idea of difference lists in logic programming, as discussed in Section 7.3.

For instance, the matching problem

$$\lambda x.F(G(cons(c,x))) =^? \lambda x.cons(a, cons(b, cons(c,x)))$$

has the three solutions

$$\{F \mapsto \lambda y.y,\ G \mapsto \lambda y.cons(a, cons(b,y))\},$$
$$\{F \mapsto \lambda y.cons(a,y),\ G \mapsto \lambda y.cons(b,y)\},$$
$$\{F \mapsto \lambda y.cons(a, cons(b,y)),\ G \mapsto \lambda y.y\}.$$

As the left-hand side of the last matching problem is a second-order term, we can use Theorem 5.2.6 to decide associative unification problems where one side only has linear variables with ground arguments, e.g. the problem

$$\lambda x.F(G(cons(c,x))) =^? \lambda x.cons(Y, X(x))$$

is decidable. In this example, $\lambda x.F(G(cons(c,x)))$ represents all lists ending in c and $\lambda x.cons(Y,X(x))$ stands for a non-empty list.

5.4.1 Open Problems

We briefly mention some open problems for future examination. Is the unification of a pattern with a linear second-order term decidable? This might be equivalent to unification of two linear second-order terms. Another question is whether second-order flex-flex pairs can be solved finitely. This may be possible by extending EL. The counterexample in [Hue76] only gives a third-order pair with nullary unification.

The above unification problems are at least NP-hard, as they subsume second-order matching, which is NP-complete [Bax76]. Are they also NP-complete?

Is there any way to extend the results to the third-order case? Not an obvious one, since this would subsume third-order matching which may have infinitely many incomparable solutions. Another question is whether the particular strategy in Theorem 5.2.6 is really necessary for termination.

An interesting idea would be to combine the nice properties of higher-order patterns with the above decidability results. Assume we want to unify an arbitrary pattern with a second-order term. Then there are two overlapping decidable subclasses of this problem, i.e. pattern unification and Theorem 5.2.8. Apart from selecting the appropriate algorithm depending on the occasion, there is a more interesting way. The main problem for combining these is the linearity restriction in the above results. The idea is to linearize one of the terms to be unified. More precisely, if we unify an arbitrary pattern p with a second-order term t, we can first make p linear and add some equality constraints. Then we solve the unification problem and apply the solutions to the equality constraints.

For instance, if $p = \lambda y.f(X(y),X)$, we replace the unification problem $p =^? t$ by

$$\lambda y.f(X(y),Y) =^? t, X =^? Y.$$

Now if the first problem is solvable by θ and if t is a pattern as well, the resulting problem $\theta X =^? \theta Y$ is a pattern unification problem. With this construction, we can integrate both results. In fact, we can even decide further problems, although it seems difficult to describe this class. For instance, the unification problem

$$\lambda x.g(f(x,G_1),f(x,G_1)) =^? \lambda x.F(f(x,G)),$$

is similar to Example 4.1.3 but does not fall into any decidable class. We transform this into

$$\lambda x.g(f(x,G_1),f(x,G_2)) =^? \lambda x.F(f(x,G)), G_1 =^? G_2.$$

Solving first the linear version is equivalent to Example 4.1.3 and yields the solution $\{F \mapsto \lambda x.g(x,x), G_1 \mapsto G, G_2 \mapsto G\}$. It remains to solve the trivial equation $G =^? G$. Interestingly, repeating this linearization procedure may yield an algorithm for general second-order unification.

Chapter 6

Higher-Order Narrowing

This chapter discusses several approaches for solving higher-order equations by narrowing. Inspired by the different notions of first-order narrowing, we develop a framework for higher-order narrowing. For an overview, we refer again to Figure 2.3. The results in Section 5.2 on the decidability of unification of two second-order λ-terms, where one term is linear, are one of the main motivations for this work: the unification problem needed for second-order narrowing is decidable if the left-hand sides of the rewrite rules are linear higher-order patterns.

The structure of this chapter is as follows. We first discuss some general aspects of narrowing in Section 6.1. The first approach we consider is the general notion of (plain) narrowing, for which many refinements exist, e.g. basic narrowing [Hul80]. The idea of this approach is to find an instance of a term such that a rewrite step somewhere in the term becomes possible. For this, Section 6.2 presents an abstract view of higher-order narrowing, where a problem with locally bound variables in the solutions becomes apparent. We show in Section 6.3 that the first-order notion of plain narrowing can be lifted to higher-order patterns and argue that it is problematic when going beyond higher-order patterns. In the general approach in Section 6.2 most real problems are hidden in the unification. We discuss some of these in Section 6.4.

As the approach to lift plain narrowing is not satisfactory, we consider an alternative approach to first-order narrowing in the higher-order setting: lazy narrowing. The idea here is to integrate narrowing into unification and to permit only narrowing at root positions. This is discussed in Section 6.5, where we show that many of the problems encountered in the first approach can be avoided.

The restriction to normalized substitutions is standard for first-order plain narrowing. For lazy narrowing we consider both normalized and reducible solutions. As in the first-order case, normalized solutions usually require a terminating HRS, but allow to restrict the most unconstrained case of narrowing: narrowing at variable positions. This refinement is examined in Section 6.6.1. Following this line, including normalization into narrowing, as shown in Section 6.6.4, is desirable, as normalization is a deterministic operation. The restriction allows for a further optimization: deterministic eager variable elimination, as examined in Section 6.6.2. In general, it is an open question if eager variable elimination is a complete strategy. In our setting, we can differentiate two cases of variable elimination, where elimination is deterministic in one case.

Equational programming is a special case of general equation solving, e.g. the restriction to left-linear rules is common. We show that for this class of problems a certain

class of equational goals suffices. These are called Simple Systems (Section 6.7.1) and enjoy several nice properties. For instance, with the results on second-order unification of Section 5.2.1, we show that the syntactic solvability of second-order Simple Systems remains decidable, as it is in the first-order case. Furthermore, solved forms are much easier to detect than in the general case.

Combining the results for normalized substitutions with the properties of Simple Systems in Section 6.6.2 leads to an effective narrowing strategy, Needed Lazy Narrowing. The basis for this strategy is a classification of the variables occurring in Simple Systems in Section 6.7.2. This allows to recognize and to delay intermediate goals, which are only solved when needed.

As some of these refinements for lazy narrowing build upon others, we show these dependencies in Figure 6.1. Notice that all refinements can be combined in a straightforward way.

Conditional rules are a common extension of term rewriting, useful in many applications. We consider the general case of arbitrary conditions for lazy narrowing in Section 6.8. In Section 6.8.2 we argue that a restricted class of rules where no extra variables are allowed on the right sides of conditions, called normal conditional rules, are sufficiently expressive for higher-order functional-logic programming. We show that the refinements developed for unconditional lazy narrowing can be extended to normal conditional narrowing.

Another approach of higher-order narrowing is discussed in the last section of this chapter. The main problems of plain narrowing in the higher-order case come from the fact that narrowing at variable positions is needed. Section 6.9 shows that we can factor out this complicated case by flattening the terms to patterns plus adding some constraints. Then narrowing on the pattern part proceeds almost as in the first-order case and it remains to solve the constraints, which can be done by lazy narrowing. In that way we have a modular structure, and higher-order lazy narrowing is used only where needed.

6.1 Scope and Completeness of Narrowing

In the following, we explain the assumptions of this approach to narrowing and discuss their implications.

In our approach, we only show completeness of narrowing wrt. solutions, sometimes only normalized substitutions. That is, for a goal $s \to^? t$, we consider solutions θ with $\theta s \xrightarrow{*} t$. We view this as the most general and basic concept of narrowing, as most of the common notions of completeness are easy to derive. For instance, for a convergent HRS R, this yields a complete algorithm for matching modulo the equational theory of R (for unification see below). In convergent theories, for any solution there exists an equivalent normalized one, thus our results suffice for complete R-unification or R-matching. For some results we explicitly require a convergent HRS and also give results tailored towards convergent HRS.

An alternative notion of completeness has been developed for programming language applications. Taking denotational semantics as the basis, two terms are equal, roughly speaking, if they can be evaluated to the same constructor term. This is called strict or continuous equality. For this notion of completeness wrt. denotational semantics [Red85],

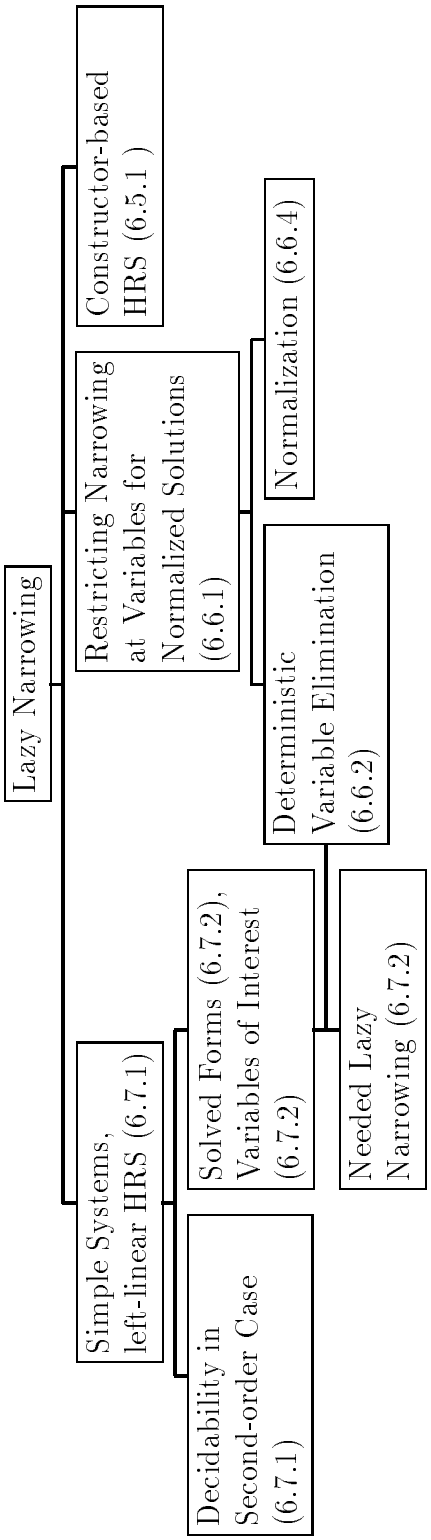

Figure 6.1: Dependencies of Lazy Narrowing Refinements

it suffices to consider only constructor based solutions, which are clearly normalized. This approach has led to implementations of functional-logic programming, e.g. [MNRA92]. It has also been extended to non-confluent HRS, which are used for non-deterministic programming, as developed in [Hußß93].

Strict equality permits non-terminating rewrite rules, which has been claimed as an advantage of this approach. In contrast, we argue in Section 7.3.1 that non-terminating rules, as used in lazy functional languages, are not needed in logic programming.

Strict equality can be encoded with left-linear rules in our setting. We simply define a function *s_equal* that forces the evaluation to a constructor term. For instance, for natural numbers the rules

$$\begin{aligned} s\_equal(s(X), s(Y)) &\rightarrow s\_equal(X, Y) \\ s\_equal(0, 0) &\rightarrow true \end{aligned}$$

suffice, assuming the constructors s and 0. This encoding works in a straightforward manner for first-order data types. It is however unclear how to extend strict equality to the higher-order case.

For the higher-order case, an alternative approach for non-convergent HRS is to embed the calculi we develop into higher-order logic programming [Nad87], for with model-theoretic semantics exist [Wol94].

In the first-order case, our notion of plain narrowing (with some additional control strategy) is also called lazy narrowing (see e.g. [Han94a, LLFRA93]). Furthermore, lazy narrowing as defined here is called lazy unification in the first-order case [Han94c, MRM89]. Our naming conventions are based on some earlier works [Sny90, Höl88, Höl89].

6.1.1 Oriented Goals

We consider in this work only *oriented* (or directed) goals $s \rightarrow^? t$ with solutions θ such that $\theta s \xrightarrow{*} t$. Systems of such goals are used directly for lazy narrowing. For plain narrowing, it suffices to consider narrowing derivations starting from one term, here s.

In other works, solutions with reduction in both directions, i.e. $\theta s \downarrow \theta t$, are considered. Directed goals simplify the technical treatment in many respects and are essential for some refinements. For instance, we show in Section 6.7.1 that strong invariants for sets of directed goals are possible for functional-logic programming and permit deterministic variable elimination. Directed goals are also more appropriate for programming language applications, as they are operationally more perspicuous. The expressiveness lost by this assumption can easily be recovered by the following technique: add an equality predicate $=$ and the rule $X = X \rightarrow true$ to a rewrite system R. Then the R-unification problem of two terms s and t can be stated as $s = t \rightarrow^? true$ and solved by narrowing. This yields a semi-decision procedure for unification modulo a convergent R, as narrowing is complete wrt. normalized substitutions. It is important to observe that this added rule $X = X \rightarrow true$ does not destroy convergence. Notice that this rule is not left-linear, which is essential for some refinements regarding programming. In essence, this shows that for left-linear rules, there is a difference between matching and unification. For instance, there are cases where matching is decidable but unification is not [DMS92, Pre94c].

6.2 A General Notion of Higher-Order Narrowing

The idea of first-order plain narrowing is, roughly speaking, to find an instance of a term such that some subterm can be rewritten. Repeating this yields a complete method for matching modulo a theory given by a convergent rewrite system R.

Since λ-calculus can express a notion of subterm, we can model narrowing in a very abstract way. Already in this very general setting we will identify a problem with locally bound variables in solutions. To handle bound variables correctly within λ-calculus, it will be necessary to guess these variables beforehand, which is clearly unsatisfactory.

We simulate a context where reduction takes place by an appropriate higher-order variable C, i.e. instead of $s \longrightarrow^{l \to r} t$ we can write $s = \theta C(l) \longrightarrow \theta C(r) = t$ for an appropriate substitution θ. For instance, to rewrite $c(f(X))$ with $f(X) \to g(X)$, it suffices to take $C \mapsto \lambda x.c(x)$. This yields the following generalization of first-order narrowing, where most of the real problems are hidden in the unification.

Definition 6.2.1 A λ-term s **narrows** to t with a rule $l \to r$ and with a substitution θ, written as $s \leadsto_\theta^{l \to r} t$, if

- τ is a $\overline{y_k}$-lifter of l,
- θ is a unifier of $s =^? C(\lambda\overline{y_k}.\tau l)$, where C is a new variable of appropriate type, and
- $t = \theta\, C(\lambda\overline{y_k}.\tau r)$.

A few comments are in order:

- The $\overline{y_k}$-lifter employed is completely arbitrary, any $k \geq 0$ is possible. This causes infinite branching.
- Even for restricted left-hand sides the relation may not be decidable.
- The equation $s =^? C(l)$ may be a flex-flex pair; such pairs are usually not solved, as only higher-order pre-unification is used in applications. Furthermore, for such equations, minimal complete set of unifiers may not exist.
- Instead of explicitly replacing a subterm at position p, we use β-reduction for this purpose. It is possible to make the subterm explicit where the replacement takes place, but this considerably complicates the completeness proof.
- Note that l may occur repeatedly or not at all in $\theta C(l)$, i.e. $\theta s = \theta t$ is possible.

Lemma 6.2.2 (One Step Lifting) *Let R be a GHRS and let $l \to r \in R$. Suppose we have two terms s and t with $\theta s = t$ for a substitution θ and a set of variables V such that $\mathcal{FV}(s) \cup \mathcal{D}om(\theta) \subseteq V$. If $t \longrightarrow_p^{l \to r} t'$, then there exist a term s' and substitutions δ and σ such that*

- $s \leadsto_\sigma^{l \to r} s'$,
- $\delta s' = t'$,
- $\delta\sigma =_V \theta$,

- $\mathcal{FV}(s') \cup \mathcal{D}om(\delta) \subseteq V - \mathcal{D}om(\sigma) \cup \mathcal{R}ng(\sigma)$.

Proof Assume $\overline{y_k} = \mathcal{BV}(t,p)$ and $t \longrightarrow^{l \to r}_{\tau,p} t'$, where $l \to r \in R$ is a $\overline{y_k}$-lifted rule, away from V. Let $\delta' = \theta \cup \{C \mapsto \lambda x.t[x(\overline{y_k})]_p\} \cup \tau$. Then δ' is a unifier of $s =^? C(\lambda\overline{y_k}.l)$. Let σ be a more general unifier σ such that $\delta' =_V \delta\sigma$ for some δ. Assume wlog. $\mathcal{D}om(\delta) \subseteq \mathcal{FV}(\sigma s)$. As $\delta' =_V \theta$, we have $\theta =_V \delta\sigma$. Then, by definition, $s \leadsto^{l \to r}_{\sigma} s'$ and $\delta s' = t'$ follows from

$$\delta s' = \delta\sigma\, C(\lambda\overline{y_k}.r) = \delta' C(\lambda\overline{y_k}.r) = t[(\lambda\overline{y_k}.\delta' r)\overline{y_k}]_p = t'.$$

Using $\mathcal{FV}(r) \subseteq \mathcal{FV}(l)$ we obtain

$$\mathcal{FV}(s') = \mathcal{FV}(\sigma\, C(\lambda\overline{y_k}.r)) \subseteq \mathcal{FV}(\sigma\, C(\lambda\overline{y_k}.l)) = \mathcal{FV}(\sigma s) \subseteq V - \mathcal{D}om(\sigma) \cup \mathcal{R}ng(\sigma).$$

Hence we have

$$\mathcal{FV}(s') \cup \mathcal{D}om(\delta) \subseteq V - \mathcal{D}om(\sigma) \cup \mathcal{R}ng(\sigma)$$

as $\mathcal{D}om(\delta) \subseteq \mathcal{FV}(\sigma s)$. □

With Lemma 6.2.2, completeness of narrowing can be shown easily, as for instance in the next section. Notice that the above proof uses some unifier that is more general than the substitutions of the reduction considered, although it would be sufficient to use the solution θ as the unifier of $s =^? C(\lambda\overline{y_k}.\tau l)$. It would be desirable to use a maximally general unifier instead, but these may not exist for higher-order unification.

For the proof of the above lemma it is important that the rewrite rule $l \to r$ has been lifted over the right number of bound variables. Let us see by an example that the number of variables over which a rule has to be lifted cannot be determined beforehand. The problem occurs when a solution θ for a variable X contains a local λy and a rewrite step in a subterm below where y occurs has to be lifted. When narrowing the replaced subterm is made explicit in $\sigma C(l) \longrightarrow \sigma C(r)$, but y is not visible yet. With the lifting of $l \to r$ it is possible to rename bound variables in r later. A somewhat similar problem with higher-order matching was reported in [Pau86] and [PE88].

Example 6.2.3 Assume $R = \{h(P,a) \to g(P,a)\}$ and consider the matching problem $H(a) \to^? u(\lambda y.g(y,a))$ with the solution $\{H \mapsto \lambda x.u(\lambda y.h(y,x))\}$. When narrowing without lifting, we obtain $H(a) \leadsto^R H''(g(P',a))$, which matches $u(\lambda y.g(y,a))$, but does not subsume the above solution, as $g(P',a)$ cannot be instantiated to $g(y,a)$.

The solution is obtained here by lifting the rule over one parameter. First, the solution to the unification problem $H(a) =^? C(\lambda y.h(P(y),a))$, which is needed for the narrowing step, is

$$\{H \mapsto \lambda x.H'(\lambda y.h(P(y),x)), C \mapsto \lambda x.H'(\lambda y.x(y))\}.$$

Then we have $H(a) \leadsto^R H'(\lambda y.g(P(y),a))$ and the matching problem can be solved with the substitution $\{H' \mapsto \lambda x.u(x), P \mapsto \lambda x.x\}$. In the general case, the solution to H may contain an arbitrary number of locally bound variables, such as y here, but the need to lift over these variables is not visible when looking at $H(a)$. To obtain completeness for this definition of narrowing, we thus have to guess locally bound variables, at least in our framework.

Alternatively, it would be possible to ignore the binding rules of λ-calculus while computing a solution and then to check if no bound variable is captured once a solution is found. This has the disadvantage that failures are detected very late. Thus this approach seems unsatisfactory, both from a practical and from a logical point of view.

The above notion of narrowing is not of great computational interest. For instance, there is little hope to find cases where even the application of narrowing is decidable. We show in the following section that the first-order notion of narrowing can be lifted to higher-order patterns. Then we discuss the problems of extending this approach to the higher-order case.

6.3 Narrowing on Patterns with Pattern Rules

In this section we show that the first-order notion of (plain) narrowing can be adapted to a restricted set of λ-terms, higher-order patterns. Then, as in the first-order case, narrowing at variable positions implies that the used substitution is reducible, thus this step is redundant.

Assumption. We assume in this section that all terms, including the rewrite rules, are patterns.

Although pattern rules are not sufficient for expressing higher-order functional programs (see e.g. Section 8.1), there are examples from other areas, where bound variables are involved. For instance, scoping rules for quantifiers (as in [Nip91a]), e.g.

$$P \wedge \forall x.Q = \forall x.(P \wedge Q),$$

can be expressed by patterns.

Definition 6.3.1 A **pattern narrowing** step from a pattern s to t with a pattern rule $l \to r$ at a non-variable position q with substitution θ is defined as $s \underset{p}{\leadsto} {}^{l \to r}_{q,\theta} t$, where

- τ is a $\overline{y_k}$-lifter of l, where $\overline{y_k} = \mathcal{BV}(s,q)$ and
- θ is a most general unifier of $\lambda\overline{y_k}.s|_q$ and $\lambda\overline{y_k}.\tau l$, and
- $t = \theta(s[\tau r]_q)$.

This notion of narrowing coincides with the standard definition of first-order narrowing on first-order terms. Here, in contrast to the notion of narrowing in Section 6.2, we only have to lift the rule $l \to r$ into the context at position q. The problem in Section 6.2 with locally bound variables occurs only when narrowing at variable positions, which is not needed here. When working with first-order equations, as done by Qian [Qia94] and by Snyder [Sny90], this lifting is not strictly needed, as the bound variables in $s|_q$ can be treated as new constants and/or ignored. This enables Qian to lift completeness of first-order narrowing strategies to patterns for first-order equations. We conjecture that most first-order narrowing strategies can also be lifted to our setting, yet not as in [Qia94].

For a sequence

$$s_0 \underset{p}{\leadsto} {}^{R}_{\theta_1} \; s_1 \underset{p}{\leadsto} {}^{R}_{\theta_2} \; \ldots \underset{p}{\leadsto} {}^{R}_{\theta_n} \; s_n$$

we write $s_0 \overset{*}{\underset{p}{\leadsto}} {}^{R}_{\theta} \; s_n$, where $\theta = \theta_n \ldots \theta_1$. We first lift one rewrite step in a solution to one narrowing step. The lemma and its proof resemble closely their first-order counterparts, as e.g. in [MH94]. This result has been developed independently by the author in [Pre94b] and in [ALS94a, LS93] for conditional rules (see Section 8.1 for more details).

Lemma 6.3.2 (One Step Lifting) *Let R be a pattern HRS and let $l \to r \in R$. Suppose we have two patterns s and t with $t = \theta s$ for an R-normalized substitution θ, and a set of variables V such that $\mathcal{FV}(s) \cup \mathcal{D}om(\theta) \subseteq V$. If $t \longrightarrow_{\theta}^{l \to r} t'$, then there exist a term s' and substitutions δ, σ such that*

- $s \underset{p}{\overset{l \to r}{\underset{\sigma}{\rightsquigarrow}}} s'$
- $\delta s' = t'$
- $\delta\sigma =_V \theta$
- *δ is R-normalized*
- $\mathcal{FV}(s') \cup \mathcal{D}om(\delta) \subseteq V - \mathcal{D}om(\sigma) \cup \mathcal{R}ng(\sigma)$

Proof Assume $\theta s \longrightarrow_{p,\varphi}^{i \to r} t'$ and $l' \to r'$ is a rule lifted over $\overline{y_k}$ from $l \to r$ away from V. As l is of base type, $\theta s|_p$ cannot be an abstraction. We have $(\theta s)|_p = \varphi l'$. Since θ is R-normalized, p is a non-variable position in s and $\theta s|_p = \theta(s|_p)$.

Let σ be a most general unifier of $\lambda\overline{y_k}.s|_p$ and $\lambda\overline{y_k}.l'$ such that there exists δ with $\delta\sigma =_V \theta$. Assume that δ is minimal, i.e. $\mathcal{D}om(\delta) \subseteq V - \mathcal{D}om(\sigma) \cup \mathcal{R}ng(\sigma)$ holds. Since σ is a pattern substitution, δ is R-normalized, as θ is.

Then, by definition, $s \underset{p}{\overset{l \to r}{\underset{\sigma,p}{\rightsquigarrow}}} s' = \sigma s[r']_p$. To see that this step lifts the rewrite step on θs, it remains to show

$$\delta s' = \delta\sigma s[r']_p = \theta(s[r']_p) = t'.$$

The second equation follows from $\delta\sigma =_V \theta$. Then from $\mathcal{FV}(r) \subseteq \mathcal{FV}(l)$ and from $\mathcal{FV}(\sigma s|_p) \subseteq \mathcal{FV}(\sigma l)$

$$\mathcal{FV}(s') \subseteq \mathcal{FV}(\sigma s, \sigma r) \subseteq \mathcal{FV}(\sigma s, \sigma l) \subseteq \mathcal{FV}(\sigma s) \subseteq V - \mathcal{D}om(\sigma) \cup \mathcal{R}ng(\sigma)$$

follows. Hence we have $\mathcal{FV}(s') \cup \mathcal{D}om(\delta) \subseteq V - \mathcal{D}om(\sigma) \cup \mathcal{R}ng(\sigma)$ as δ is minimal, which concludes the proof. □

The following lemma holds for patterns as for first-order terms [MH94].

Lemma 6.3.3 *Let σ, θ, θ' be pattern substitutions and V, V' be sets of variables such that $(V' - \mathcal{D}om(\sigma)) \cup \mathcal{R}ng(\sigma) \subseteq V$. If $\theta =_V \theta'$ then $\theta\sigma =_{V'} \theta'\sigma$.*

Completeness of narrowing follows as in the first-order case:

Theorem 6.3.4 (Completeness of Pattern Narrowing) *Let $\mathcal{R}$ be a pattern HRS. Suppose we have terms s and $t = \theta s$ for a substitution θ and a set of variables V such that $\mathcal{FV}(s) \cup \mathcal{D}om(\theta) \subseteq V$. If $t \longrightarrow^{*R} t'$, then there exist a term s' and substitutions δ, σ such that*

- $s \underset{p}{\overset{*}{\underset{\sigma}{\rightsquigarrow}}}^{R} s'$
- $\delta s' = t'$
- $\delta\sigma =_V \theta$

- $\mathcal{FV}(s') \cup \mathcal{D}om(\delta) \subseteq V - \mathcal{D}om(\sigma) \cup \mathcal{R}ng(\sigma)$

Proof by induction on the length of the reduction from t to t'. Assume $t \longrightarrow_{\varphi}^{l \to r} t_1$. By Lemma 6.3.2 there exist a term s_1 and substitutions δ_1, σ_1 such that

- $s \overset{l \to r}{\underset{p}{\underset{\sigma_1}{\rightsquigarrow}}} s'$
- $\delta_1 s' = t_1$
- $\delta_1 \sigma_1 =_V \theta$
- δ_1 is R-normalized
- $\mathcal{FV}(s') \cup \mathcal{D}om(\delta_1) \subseteq V - \mathcal{D}om(\sigma_1) \cup \mathcal{R}ng(\sigma_1)$

Let $V_1 = V - \mathcal{D}om(\sigma_1) \cup \mathcal{R}ng(\sigma_1)$. Then the induction hypothesis yields

- $s_1 \overset{*}{\underset{p}{\underset{\sigma_2}{\rightsquigarrow}}} s'$
- $\delta s_1' = t'$
- $\delta \sigma_2 =_{V_1} \delta_1$
- δ is R-normalized
- $\mathcal{FV}(s') \cup \mathcal{D}om(\delta) \subseteq V_1 - \mathcal{D}om(\sigma_2) \cup \mathcal{R}ng(\sigma_2)$

Let $\sigma = \sigma_2 \sigma_1$. Then $\delta\sigma =_V \theta$ follows as $\delta\sigma_2 =_{V_1} \delta_1$ and $\delta\sigma_2\sigma_1 =_V \delta_1\sigma_1$ yields $\delta\sigma_2\sigma_1 = \delta\sigma$ with Lemma 6.3.3 and hence $\delta\sigma = \theta$. □

6.4 Narrowing Beyond Patterns

We discuss in the following the problems when extending the first-order notion of plain narrowing for patterns to full λ-terms, both in the rules as in the goals. For this purpose, we use in this section the relation $\rightsquigarrow$ in a more informal way to exemplify the problems involved. In Example 6.2.3 we have identified a problem with locally bound variables. This and several other problems stem from the fact that narrowing at variable positions is required, since the rewrite step we lift might have been at a redex created by β-reduction. We discuss this with the following example.

Example 6.4.1 Assuming the rewrite system

$$R_0 = \{f(f(X)) \to g(X)\},$$

narrowing at a variable position is required to find the solution $\{H \mapsto \lambda x.f(x)\}$ to the problem $\lambda x.H(f(x)) \to^? \lambda x.g(x)$:

$$\lambda x.H(f(x)) \rightsquigarrow^{R_0}_{H \mapsto \lambda x.f(x)} \lambda x.g(x)$$

Now the problem is how to define narrowing at variable positions. For instance, consider the solution $\theta = \{H \mapsto \lambda x.h(f(x), x)\}$ to the equational problem

$$\lambda x.H(f(x)) \to^? \lambda x.h(g(x), f(x)),$$

wrt. the R_0-reduction

$$\lambda x.h(f(f(x)), f(x)) \longrightarrow^{R_0} \lambda x.h(g(x), f(x)).$$

The naive approach, to instantiate H as little as possible, as in

$$\lambda x.H(f(x)) \leadsto^{R_0}_{H \mapsto \lambda x.H'(f(x))} \lambda x.H'(g(x)),$$

fails. The problem is that the subterm $f(x)$ is duplicated by θ and the reduction does not occur inside $f(x)$. An idea is to create a "local context" at this variable. Hence, we instantiate H first with $\{H \mapsto \lambda x.H''(H'(x), x)\}$. Then, after β-reduction, the subterm $H'(f(x))$ can be unified by $\{H' \mapsto \lambda x.f(x)\}$ with the left-hand side $f(f(x))$ and can be rewritten. Thus we have

$$\lambda x.H(f(x)) \leadsto^{R_0}_{H \mapsto \lambda x.H'(f(x),x)} \lambda x, y.H'(g(x), x)$$

and the solution, here $\{H' \mapsto \lambda x, y.h(x, y)\}$, is then obtained by unification.

Intuitively, we approximate the desired solution θH in the first argument of H''. A further problem occurs when narrowing on an argument of a free variable. For instance, assume the narrowing step

$$H(f(X)) \leadsto^{R_0}_{X \mapsto f(Y)} H(g(Y)).$$

Then some solution to H may copy the argument of H, thus this narrowing step corresponds to several rewrite steps. As a consequence, the solution $\{X \mapsto f(Y), H \mapsto \lambda x.h(x, x)\}$ with the reduction

$$H(f(f(Y))) \to^? h(g(Y), f(f(Y))$$

to the above matching problem will not be found with the narrowing step above. The redex is copied in the solution, but for narrowing, only one copy is visible.

Due to all these problems, we do not develop the notion of plain narrowing further and instead focus on the alternative, lazy narrowing, in the next section. In addition, we develop another approach that extends pattern narrowing by additional higher-order constraints in Section 6.9.

6.5 Lazy Narrowing

A more goal-directed method to solve equational problems in a top-down manner is lazy narrowing. The main idea is to integrate narrowing into unification. That is, when R-matching s with t, we start with a goal $s \to^? t$ that may be simplified to smaller goals. Then narrowing steps are performed at the root only, where the unification of the left-hand side of the rule with s again has to be done modulo R.

For instance, to solve a goal $h(t_1, t_2) \to^? v(X, Y)$, we either simplify the goal to the goals $t_1 \to^? X$ and $t_1 \to^? Y$ if $h = v$, or apply a narrowing step at the root in a lazy fashion. That is, assuming a rule $h(a, Z) \to g(b)$, we transform the above goal to

$$\{t_1 \to^? a, t_2 \to^? Z, g(b) \to^? v(X, Y)\}.$$

In contrast to plain narrowing, not the first rewrite step in a solution $\theta h(t_1, t_2) \to^? \theta v(X, Y)$ is modeled, but the first outermost one. Assume this is the rewrite step to t' in

$$\theta h(t_1, t_2) \xrightarrow{*} \theta h(a, Z) \longrightarrow_{\epsilon}^{l \to r} t' \xrightarrow{*} \theta t.$$

Now the purpose of the goals $t_1 \to^? a, t_2 \to^? Z$ is easy to see: the rewrite steps in $\theta h(t_1, t_2) \xrightarrow{*} \theta h(a, Z)$ are modeled by these two goals.

In the last example, Z does not occur on the right-hand side of the rule $h(a, Z) \to g(b)$. Speaking in programming terminology, it is not necessary to "evaluate" the term t_2, here to Z. This corresponds to lazy evaluation, as t_2 can be reducible. The reason for this is that lazy narrowing, in its simple form, is also complete for reducible solutions, which makes it possible to model lazy evaluation. Notice that the solution for the intermediate variable Z may not be normalized. In our context, so-called infinite data-structures in lazy languages correspond to reducible terms whose normalization diverges. In contrast, the theory of plain narrowing often considers normalized substitutions with innermost reductions, which corresponds to eager evaluation in a programming language.

It should be noted that the notion of laziness in Lazy Narrowing not only serves for lazy evaluation as in lazy or non-strict languages, but also to lazy instantiation of free variables. Intuitively, this means that instantiations are only performed when needed. This distinction will become clear later, e.g. in Section 7.3.

Let $s \overset{?}{\leftrightarrow} t$ stand for one of $s \to^? t$ and $t \to^? s$. For a sequence $\Rightarrow^{\theta_1} \ldots \Rightarrow^{\theta_n}$ of LN steps, we write $\overset{*}{\Rightarrow}^{\theta}$, where $\theta = \theta_n \ldots \theta_1$.

The full set of rules for lazy higher-order narrowing, called System LN, is shown in Figure 6.2. System LN essentially consists of the rules for higher-order unification [SG89] plus the Lazy Narrowing rule. Observe that the first five rules in Figure 6.2 apply symmetrically as well, in contrast to the narrowing rule.

The subscripts (d) and d on goals only serve for a particular optimization and are not needed for soundness or completeness. The idea is to use **marked goals** $s \to_d^? t$. These are created only in the Lazy Narrowing rule, in order to avoid repeated application of Lazy Narrowing on these goals. The remaining rules work on both marked goals and unmarked goals, indicated by $\to_{(d)}^?$. For both $\overset{?}{\leftrightarrow}$ and $\to_{(d)}^?$ the rules are intended to preserve the orientation for $\overset{?}{\leftrightarrow}$ and marking for $\to_{(d)}^?$. Only the Decomposition rule and the Imitation rule, which includes decomposition, transform marked goals to unmarked goals. In other words, on marked goals Lazy Narrowing may only be applied after some decomposition took place.

For instance, reconsider from Example 6.4.1 the R_0-matching problem

$$\lambda x.H(f(x)) \to^? \lambda x.h(g(x), f(x)),$$

where LN yields

$$\{\lambda x.H_1(f(x)) \to^? \lambda x.g(x), \lambda x.H_2(f(x)) \to^? \lambda x.f(x)\}$$

by the imitation $\{H \mapsto \lambda y.h(H_1(y), H_2(y))\}$. Then the second goal can be solved by Projection, and the first by Lazy Narrowing to

$$\{\lambda x.H_1(f(x)) \to_d^? \lambda x.f(f(X(x))), \lambda x.g(X(x)) \to^? \lambda x.g(x)\}.$$

Notice that the first goal is marked, thus Lazy Narrowing does not re-apply. This is an important restriction, since otherwise infinite reductions occur, as in this case, very often. The two goals can be solved by several higher-order unification steps, which yield the solution

$$\{H_1 \mapsto \lambda y.f(y), X \mapsto \lambda x.x\}.$$

Deletion

$$\{t \to_{(d)}^? t\} \cup S \quad \Rightarrow \quad S$$

Decomposition

$$\{\lambda\overline{x_k}.f(\overline{t_n}) \to_{(d)}^? \lambda\overline{x_k}.f(\overline{t'_n})\} \cup S \quad \Rightarrow \quad \{\overline{\lambda\overline{x_k}.t_n \to^? \lambda\overline{x_k}.t'_n}\} \cup S$$

Elimination

$$\{F \overset{?}{\leftrightarrow}_{(d)} \lambda\overline{x_k}.t\} \cup S \quad \Rightarrow^{\theta} \quad \theta S \text{ if } F \notin \mathcal{FV}(\lambda\overline{x_k}.t) \text{ and where } \theta = \{F \mapsto \lambda\overline{x_k}.t\}$$

Imitation

$$\{\lambda\overline{x_k}.F(\overline{t_n}) \overset{?}{\leftrightarrow}_{(d)} \lambda\overline{x_k}.f(\overline{t'_m})\} \cup S \quad \Rightarrow^{\theta} \quad \{\overline{\lambda\overline{x_k}.H_m(\overline{\theta t_n}) \overset{?}{\leftrightarrow} \lambda\overline{x_k}.\theta t'_m}\} \cup \theta S$$
where $\theta = \{F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$
and $\overline{H_m}$ are new variables

Projection

$$\{\lambda\overline{x_k}.F(\overline{t_n}) \overset{?}{\leftrightarrow}_{(d)} \lambda\overline{x_k}.v(\overline{t'_m})\} \cup S \quad \Rightarrow^{\theta} \quad \{\lambda\overline{x_k}.\theta t_i(\overline{H_j(\overline{t_n})}) \overset{?}{\leftrightarrow}_{(d)} \lambda\overline{x_k}.v(\overline{\theta t'_m})\} \cup \theta S$$
where $\theta = \{F \mapsto \lambda\overline{x_n}.x_i(\overline{H_j(\overline{x_n})})\}$,
$t_i : \overline{\tau_j} \to \tau_0$ and $\overline{H_j}$ are new variables

Lazy Narrowing

$$\{\lambda\overline{x_k}.s \to^? \lambda\overline{x_k}.t\} \cup S \quad \Rightarrow \quad \{\lambda\overline{x_k}.s \to_d^? \lambda\overline{x_k}.l, \lambda\overline{x_k}.r \to^? \lambda\overline{x_k}.t\} \cup S$$
where $l \to r$ is an $\overline{x_k}$-lifted rule

Figure 6.2: System LN for Lazy Narrowing

At first glance, the Lazy Narrowing rule of System LN looks rather simple. The hidden restrictions by the marking of goals can be made more explicit by splitting Lazy

Lazy Narrowing with Decomposition

$$\{\lambda\overline{x_k}.f(\overline{t_n}) \rightarrow^? \lambda\overline{x_k}.t\} \cup S \;\Rightarrow\; \{\overline{\lambda\overline{x_k}.t_n \rightarrow^? \lambda\overline{x_k}.l_n}\} \cup \{\lambda\overline{x_k}.r \rightarrow^? \lambda\overline{x_k}.t\} \cup S$$
where $f(\overline{l_n}) \rightarrow r$ is an $\overline{x_k}$-lifted rule

Lazy Narrowing at Variable

$$\{\lambda\overline{x_k}.H(\overline{t_n}) \rightarrow^? \lambda\overline{x_k}.t\} \cup S \;\Rightarrow\; \{\lambda\overline{x_k}.H(\overline{t_n}) \rightarrow^?_d \lambda\overline{x_k}.l\} \cup \{\lambda\overline{x_k}.r \rightarrow^? \lambda\overline{x_k}.t\} \cup S$$
where $l \rightarrow r$ is an $\overline{x_k}$-lifted rule

Figure 6.3: The Two Cases of the Lazy Narrowing Rule of System LN

Narrowing into two rules, depending on the head of the left-hand side. This is shown in Figure 6.3. The first rule is easily inferred from LN as in this case the Lazy Narrowing rule yields a marked goal $\lambda\overline{x_k}.f(\ldots) \rightarrow^?_d \lambda\overline{x_k}.f(\ldots)$, where only decomposition applies. Observe that the two rules do not permit narrowing steps on goals of the form $\lambda\overline{x_k}.x_i(\ldots) \rightarrow^? \lambda\overline{x_k}.t$. As the two rules are more intuitive and clearly equivalent to the Lazy Narrowing rule of System LN, we often use the two rules above instead whenever convenient.

The completeness proof of system LN is built upon the completeness proof of higher-order unification in a modular way: the termination ordering is a lexicographic extension of the one in Theorem 4.1.7.

Theorem 6.5.1 (Completeness of LN) *If $s \rightarrow^? t$ has solution θ, i.e. $\theta s \xrightarrow{*}{}^R \theta t$ for some GHRS R, then $\{s \rightarrow^? t\} \stackrel{*}{\Rightarrow}^{\delta}_{LN} F$ such that δ is more general modulo the newly added variables than θ and F is a set of flex-flex goals.*

Proof The proof proceeds by induction on the following lexicographic termination ordering on $(\overline{G_n}, \theta)$, where for $\overline{G_n} = \overline{s_n \rightarrow^?_{(d)} t_n}$ is a system of goals with solution θ, i.e. $\overline{\theta s_n \xrightarrow{*} \theta t_n}$. Notice that a transformation not only changes $\overline{G_n}$, but also the associated solution has to be updated as in Theorem 4.1.7. The ordering assumes an arbitrary, but fixed reduction $\overline{\theta s_n \xrightarrow{*} \theta t_n}$.

- A: compare the multiset of sizes of the number of R-reductions in each goal θG_i, if equal
- B: compare the multiset of sizes of the bindings in θ, if equal
- C: compare the multiset of sizes of the goals $\overline{G_n}$.

We maintain the following invariant for marked goals: if $s \rightarrow^?_d t$, then $Head(\theta s) = Head(\theta t)$ is not a free variable and furthermore, no rewrite step at root position occurs in $\theta s \xrightarrow{*}{}^R \theta t$. Then the Lazy Narrowing rule does not need to be applied to marked goals as shown below. Notice that Decomposition and Imitation on marked goals decompose the outermost symbol and yield unmarked goals. Thus these rules preserve the invariant.

First consider the case that all goals are flex-flex pairs. Then the goals are considered solved. If not, we show that for any non flex-flex goal some rule applies that reduces the ordering.

Select some non flex-flex goal $s \to^?_{(d)} t$ from $\overline{G_n}$. In the base case for criteria A, that is $\theta s = \theta t$, some higher-order unification rule applies, as in Theorem 4.1.7. It is clear that this does not increase A and also approximates θ.

Otherwise, there must be a rewrite step in $\theta s \xrightarrow{*} \theta t$. In the first case, assume there is no rewrite step at the root position in $\theta s \xrightarrow{*} \theta t$. Hence all terms in this sequence have the same root symbol. Then one of the unification rules must apply, similar to the last case.

Now assume there are rewrite steps in $\theta s \xrightarrow{*} \theta t$ at root position. Then we consider the first of these, which we assume to be $\theta s \xrightarrow{*} \lambda\overline{x_k}.s_1 \longrightarrow^{l \to r}_{\epsilon} \lambda\overline{x_k}.t_1$. Hence $s_1 \to t_1$ must be an instance of $l \to r$, and there exists δ such that $\delta l \to \delta r = s_1 \to t_1$. Assume l is of the form $f(\overline{l_m})$. Lazy Narrowing yields the new goals $s \to^?_d \lambda\overline{x_k}.l$ and $\lambda\overline{x_k}.r \to^? t$. We can extend θ for the newly added variables: define $\theta' = \theta \cup \delta$. This is well defined, as we assume that $l \to r$ is renamed by an appropriate lifter. Thus θ' is a solution of $s \to^?_d \lambda\overline{x_k}.l$ and $\lambda\overline{x_k}.r \to^? t$, that coincides with θ on $\mathcal{FV}(s,t)$. The two new goals have solutions with a smaller number of steps, thus reducing the termination ordering. Since we consider the first rewrite step a root position, the new marked goal $s \to^?_d \lambda\overline{x_k}.l$ fulfills the invariant, as $Head(\theta s) = Head(\theta l)$ and no rewrite step can occur at root position. □

Compared to the approach in Section 6.2, many problems are now taken care of by higher-order unification. For instance, locally bound variables in a solution are computed in an outside-in manner before the inner Lazy Narrowing step needs to lift over these. Furthermore, flex-flex pairs can express a possibly infinite number of solutions. This is already very useful for higher-order unification, but even more for higher-order equational unification. It must however be noted that the Imitation and Projection rules copy subterms several times, which implicitly solves many of the problem encountered in Section 6.4.

With plain narrowing, narrowing at variable positions is needed. The corresponding goals in lazy narrowing can often be delayed as flex-flex pairs. For instance, consider the goal $\lambda x.c(F(f(x))) \to^? \lambda x.c(G(x))$ wrt. R_0, where lazy narrowing stops after one decomposition step, whereas plain narrowing may blindly narrow at $F(\ldots)$.

As discussed for System PT in Section 4.1, there are two sources of non-determinism for such systems of transformations: which rules to apply and how to select the equations. As in Theorem 4.1.7, completeness does not depend on the goal selection, as each subgoal is independently solvable. Compared to pure higher-order unification, there is an important difference as the Elimination and Decomposition rules are not deterministic any more.

It is interesting to compare LN with recent work on first-order lazy unification in [Han94c]. Restricting our system to the first-order case almost yields Hanus's system (with the difference that we consider oriented goals). For instance, the transformations in [Han94c] yield so-called quasi-solved systems, which correspond to systems of (first-order) flex-flex pairs. Notice that the Imitation rule coincides for the first-order case with "partial instantiations" in [Han94c] and with the "root imitation" rule in [Sny91].

6.5.1 Narrowing Rules for Constructors

In practice, GHRS often have a number of symbols, called constructors, that only serve as data structures. For constructor symbols, we can extract a few simple rules for Lazy

Narrowing. Their main advantage is that their application is deterministic. The rules in Figure 6.4 cover the cases where the root symbol of the left side of a goal is a constructor. Notice that the rules, except for the first, are only possible with oriented goals, where

Deterministic Constructor Decomposition

$$\{\lambda\overline{x_k}.c(\overline{t_n}) \rightarrow^?_{(d)} \lambda\overline{x_k}.c(\overline{t'_n})\} \cup S \quad \Rightarrow \quad \{\overline{\lambda\overline{x_k}.t_n \rightarrow^? \lambda\overline{x_k}.t'_n}\} \cup S$$
if c is a constructor symbol

Deterministic Constructor Imitation

$$\{\lambda\overline{x_k}.c(\overline{t_n}) \rightarrow^?_{(d)} \lambda\overline{x_k}.F(\overline{y_m})\} \cup S \quad \Rightarrow^\theta \quad \{\overline{\lambda\overline{x_k}.t_n \rightarrow^? \lambda\overline{x_k}.H_n(\overline{y_m})}\} \cup \theta S$$
where $\theta = \{F \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$
and $\overline{H_n}$ are new variables

Constructor Clash

$$\{\lambda\overline{x_k}.c(\overline{t_n}) \rightarrow^?_{(d)} \lambda\overline{x_k}.v(\overline{t'_m})\} \cup S \quad \Rightarrow \quad \mathit{fail}$$
if $c \neq v$, where c is a constructor symbol
and v is not a free variable

Figure 6.4: Deterministic Constructor Rules

evaluation proceeds only from left to right. The correctness of the rules in Figure 6.4 follows immediately from the definition of a constructor: if $\lambda\overline{x_k}.c(\theta\overline{t_n}) \xrightarrow{*} t$, then t will have the constructor c as the root symbol.

6.5.2 The Second-Order Case

We examine in this section how the lazy narrowing rules can be refined in the second-order case. The goal is to show that for this case the Lazy Narrowing at Variable rule can be handled more directly by two new rules. The aim is operationally more perspicuous transformation rules. Furthermore, second-order terms suffice in most applications.

For the second-order case, we can refine the Lazy Narrowing rule into three separate rules, as shown in Figure 6.5. The rule Lazy Narrowing at Variable corresponds to two new rules. One of these includes Imitation, the other Projection. For instance, consider the goal

$$G(1) \rightarrow^? 1$$

modulo the rule $f(1) \rightarrow 1$. Then the new rule Lazy Narrowing with Imitation directly simplifies this to

$$1 \rightarrow^? 1, 1 \rightarrow^? 1.$$

The aim of such specialized, but still complete rules is to avoid divergence and to detect failures early.

Definition 6.5.2 System SLN for second-order lazy narrowing consists of the rules in Figure 6.5 plus the rules of second-order unification.

As in Section 6.5 with System LN, we use marked goals to avoid Lazy Narrowing rules before Decomposition has been applied. Notice that only the last rule introduces marked goals.

Lazy Narrowing with Decomposition

$$\{\lambda\overline{x_k}.f(\overline{t_n}) \to^? \lambda\overline{x_k}.t\} \cup S \quad \Rightarrow \quad \{\overline{\lambda\overline{x_k}.t_n \to^? \lambda\overline{x_k}.l_n}, \lambda\overline{x_k}.r \to^? \lambda\overline{x_k}.t\} \cup S$$
where $f(\overline{l_n}) \to r$ is an $\overline{x_k}$-lifted rule

SO-Lazy Narrowing with Imitation

$$\{\lambda\overline{x_k}.H(\overline{t_n}) \to^? \lambda\overline{x_k}.t\} \cup S \quad \Rightarrow^\theta \quad \{\overline{\lambda\overline{x_k}.H_m(\overline{\theta t_n}) \to^? \lambda\overline{x_k}.l_m}, \lambda\overline{x_k}.r \to^? \lambda\overline{x_k}.\theta t\} \cup \theta S$$
where $f(\overline{l_m}) \to r$ is an $\overline{x_k}$-lifted rule and
$\theta = \{H \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$

SO-Lazy Narrowing with Projection

$$\{\lambda\overline{x_k}.H(\overline{t_n}) \to^? \lambda\overline{x_k}.t\} \cup S \quad \Rightarrow^\theta \quad \{\lambda\overline{x_k}.\theta t_i \to^?_d \lambda\overline{x_k}.l, \lambda\overline{x_k}.r \to^? \lambda\overline{x_k}.\theta t\} \cup \theta S$$
where $l \to r$ is an $\overline{x_k}$-lifted rule and
$\theta = \{H \mapsto \lambda\overline{x_n}.x_i\}$

Figure 6.5: Second-Order Lazy Narrowing Rules for System SLN

Completeness of the above rules is easy to see; we only show the differences to the proof of Theorem 6.5.1. Assume a derivation $\theta s \xrightarrow{*}{}^R \theta t$.

The case we have to consider is when some reduction takes place at the head of some term in this sequence. Then we lift the first of these reductions, which must be of the form

$$\theta s = \lambda\overline{x_k}.f(\overline{s_n}) \xrightarrow{*}{}^R \lambda\overline{x_k}.f(\overline{s'_n}) \longrightarrow^R \lambda\overline{x_k}.\theta' r \xrightarrow{*}{}^R \theta t,$$

for some rule $f(\overline{l_o}) \to r$. The only cases that are different from the completeness proof on LN are the last two rules, i.e. if s is of the form $\lambda\overline{x_k}.H(\overline{t_n})$. As $Head(\theta s) = f$, it must either be an imitation binding, as covered by the narrowing rule with imitation, where the consequent decomposition is already performed. Otherwise, the narrowing rule with projection applies as in the original Lazy Narrowing rule.

6.6 Lazy Narrowing with Normalized Substitutions

We examine in this section refinements for lazy narrowing that restrict the solutions considered to normalized substitutions. As in the first-order case, this yields many important optimizations. For convergent HRS R this is not a restriction, as for any substitution there is an equivalent R-normalized one.

Some of the optimizations generalize well-known ideas of the first-order case, e.g. normalization in Section 6.6.4. The results on eager variable elimination in Section 6.6.2 are however new and hold only as we work with directed goals.

6.6.1 Restricting Lazy Narrowing at Variable Positions

We show in this section that narrowing at variables $X(\overline{x_n})$ is not needed for R-normalized substitutions with some HRS R. For patterns reducibility of a term $\theta X(\overline{x_n})$ implies that θ is not R-normalized by Theorem 4.3.5, hence violating the assumption. We conjecture that in practice, as in higher-order logic programming [MP92a], most terms are patterns and hence narrowing at variables is not needed very often.

This results generalizes the first-order case, as for first-order terms narrowing at variable position is not needed. This is the main idea of narrowing with R-normalized solutions. It is an important optimization, as narrowing at variable positions is highly unrestricted and thus may create large search spaces.

For this result the restriction to an HRS with pattern left-hand sides and innermost reductions is necessary. For any solution there exists an innermost reduction, if R is convergent.

Definition 6.6.1 System **LNN** is defined as a restriction of system LN where Lazy Narrowing at Variable is not applied to goals of the form $\lambda\overline{x_n}.X(\overline{y_m}) \rightarrow^? t$.

Completeness follows as for Theorem 6.5.1:

Theorem 6.6.2 *If $s \rightarrow^? t$ has solution θ, $\theta s \xrightarrow{*}{}^R \theta t$ is an innermost reduction, and θ is R-normalized for some HRS R, then $\{s \rightarrow^? t\} \stackrel{*}{\Rightarrow}{}^{\delta}_{LNN} F$ such that δ is more general, modulo the newly added variables, than θ and F is a set of flex-flex goals.*

Proof The proof proceeds as in Theorem 6.5.1; in addition we have to show the invariant that the solutions for all (new) variables are normalized substitutions.

Assume a goal with normalized solution θ. In case of a Projection or Imitation, the partial binding computed maps a variable X to a higher-order pattern of the form $\lambda\overline{x_n}.v(\overline{H_m(\overline{x_n})})$. The new solution constructed (as in the proof of Theorem 4.1.7) maps the newly introduced variables $\overline{H_m(\overline{x_n})}$ to subterms of θX, which are in R-normal form. Hence all $\overline{\theta H_m}$ must be in R-normal form. For the Elimination rule, no new variables are introduced, thus the solution remains R-normalized.

The critical case is when new variables are introduced in the narrowing rule. The narrowing rule is not used if there are rewrite steps in $\theta s \xrightarrow{*} \theta t$ at root position. We consider the first of these, which we assume to be $\theta s \xrightarrow{*} \lambda\overline{x_k}.s_1 \longrightarrow^{l \rightarrow r}_{\epsilon} \lambda\overline{x_k}.t_1$. Hence $s_1 \rightarrow t_1$ must be an instance of $l \rightarrow r$, say with substitution δ, i.e. $\delta l = s_1$. As l is a pattern, all terms in $\mathcal{I}m(\delta)$ are subterms of s_1 (modulo renaming, see Theorem 4.3.5). As the above reduction is innermost, all true subterms of s_1 must be in R-normal form (note that s_1 is not a variable). Hence δ must be R-normalized as well. Since we assume that $l \rightarrow r$ is renamed with new variables, $\theta' = \theta \cup \delta$ is well defined. Then after applying the Lazy Narrowing rule, θ' is a solution of the resulting goal system and is furthermore R-normalized.

Finally, with the invariant that θ' is R-normalized, it is clear from the completeness proof of LN that LNN is complete as there can be no rewrite step in the solution θ of a goal $\lambda\overline{x_n}.X(\overline{y_m}) \rightarrow^? t$ as θX is in R-normal form. □

A few comments are in order:

- A simple consequence of the last result is that System LNN is complete for matching modulo convergent HRS. Unification can be encoded as shown in Section 6.1.1.

- Observe that the restriction to HRS in the last result is essential. If the left-hand sides are non-patterns, then solutions to new variables may be reducible in case of a lazy narrowing step. Assume for instance a is reducible for some GHRS. If an instance $\theta f(G(\lambda x.a))$ of a left-hand side $f(G(\lambda x.a))$ is R-normalized, then θ need not be R-normalized: consider e.g. $\theta \mapsto \lambda x.x(a)$.

- This result implies the following optimization: if a goal $\lambda\overline{x_n}.X(\overline{y_m}) \to^? t$ is unsolvable by pure unification, then we can immediately fail this search path. Since this is an elimination problem as considered in Section 5.1, this is decidable in the second-order case.

The results in this section imply that Lazy Narrowing is not applied to goals of the form $\lambda\overline{x_n}.X(\overline{y_m}) \to^? t$ and their descendants, since such goals are transformed only to goals of this form. A special case of such goals is considered in the next section.

6.6.2 Deterministic Eager Variable Elimination

Eager variable elimination is a particular strategy of general E-unification systems. The idea is to apply the Elimination rule as a deterministic operation whenever possible. That is, when elimination applies to a goal, all other rule applications are not considered.

It is an open problem of general (first-order) E-unification strategies if eager variable elimination is still complete [Sny91]. Interestingly, in [Han94c] the elimination is purposely avoided in a programming language context as it may copy terms whose evaluation can be expensive.

In our case, with oriented goals, we obtain more precise results by differentiating the orientation of the goal to be eliminated. As we consider oriented equations, we can distinguish two cases of variable elimination. In one case elimination is deterministic, i.e. no other rules have to be considered. In other words, eager variable elimination is complete in this case.

Theorem 6.6.3 *System LNN with eager variable elimination on goals $X \to^? t$ with $X \notin \mathcal{FV}(t)$ is complete for convergent HRS R*

Proof We show that the elimination of X reduces the termination ordering in the proof of Theorem 6.5.1: as θ is R-normalized, there can be no rewrite step in $\theta X \to^? \theta t$. Thus $\theta X = \theta t$ follows. Hence for all other goals $s \to^? s'$, $\theta s \to^? \theta s'$ remains unchanged for an elimination step at $X \to^? t$. Binding X to t reduces measure B since the number of bindings decreases. □

In the general case, variable elimination may copy reducible terms with the result that the reductions have to be performed several times. Notice that this case of variable elimination does not affect the reductions in the solution considered, as only terms in normal form are copied: θt must be in normal form.

There are a few important cases when elimination on goals of the form $t \to^? X$ is deterministic:

Theorem 6.6.4 *System LNN with eager variable elimination on goals $t \to^? X$, where t is either*

- *ground and in R-normal form or*

- *a pattern without defined symbols.*

is complete for convergent HRS R

Proof In both cases it is clear that θt is in R-normal form for an R-normalized solution θ. Then elimination of X reduces the termination ordering in the proof of Theorem 6.5.1 as in Theorem 6.6.3. □

As LNN is complete for normalized solutions, the last result yields a refinement for System LNN. Notice that this refinement only holds with directed goals. In an undirected setting, reductions in both directions are possible. For instance, with the HRS $f(X) \to a$ the equation $P =^? f(P)$ can be solved with $\{P \to a\}$.

6.6.3 Avoiding Reducible Substitutions by Constraints

Although system LNN restricts narrowing at variable positions, system LNN can still compute reducible substitutions. For instance, assume the rule $f(a) \to b$ and the goal $H(a) \to^? b$. Then Narrowing at Variable followed by one imitation step with $\{H \mapsto \lambda x.f(H_1(x))\}$ creates the two goals

$$H_1(a) \to^? a, b \to^? b.$$

Performing Imitation on the first goal with $\{H_1 \mapsto \lambda x.a\}$ yields the solution $\{H \mapsto \lambda x.f(a)\}$, which is clearly not normalized.

We can restrict the search for normalized substitutions further by adding constraints as shown in Figure 6.6. The idea of these constraints is to detect reducible substitutions. In the following, we show the restrictions needed for adding such constraints in a safe way.

In the Narrowing at Variable rule, we can avoid a trivial solution $\{H \mapsto \lambda\overline{x_n}.l\}$ to the goal $\lambda\overline{x_n}.H(\overline{t_n}) \to^? \lambda\overline{x_n}.l$, which leads to a reducible solution. Observe that in the first-order case this trivial solution is always possible, unlike in the higher-order case.

Similarly, we can add a constraint in the Imitation rule, if some variable X is partially instantiated by a term $f(\overline{X_n})$. Then for all computed substitutions θ the term $\theta f(\overline{X_n})$ must not be reducible. We denote these constraints by $Irr(\overline{t_n})$, with the intended meaning that $\overline{t_n}$ are not R-reducible.

The important invariant to preserve is that the terms in the constraints are patterns. In essence, the constraints only hold approximations for some variable of the solution to be computed. If the terms are non-patterns, reducibility of a term t in a constraint $Irr(t)$ does not imply that the solution considered, i.e. θt, is reducible. For patterns, however, Lemma 4.3.5 shows that θt is reducible, if t is.

Definition 6.6.5 We define System LNC by replacing the appropriate rules of System LN with the rules in Figure 6.6.

Recall that the Elimination rule is not necessary for completeness of both PT and LN. Thus it is possible to restrict the Elimination rule to patterns in LNC without losing completeness. System LNC can be viewed as a refinement of System LNN with some additional constraints. If some constraint is added, it is clear that the term must not be R-reducible. Furthermore, with the restriction of the elimination rule to patterns it follows,

Elimination

$$\{F \stackrel{?}{\leftrightarrow} t\} \cup S \quad \Rightarrow^{\theta} \quad \{Irr(\theta F)\} \cup \theta S$$
if $F \notin \mathcal{FV}(t)$ and t is a pattern
where $\theta = \{F \mapsto t\}$

Imitation with Constraints

$$\{\lambda\overline{x_k}.F(\overline{t_n}) \stackrel{?}{\leftrightarrow} \lambda\overline{x_k}.f(\overline{t'_m})\} \cup S \quad \Rightarrow^{\theta} \quad \overline{\{\lambda\overline{x_k}.H_m(\overline{\theta t_n}) \stackrel{?}{\leftrightarrow} \lambda\overline{x_k}.\theta t'_m)\}} \cup \{Irr(\theta F)\} \cup \theta S$$
where $\theta = \{F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$
and $\overline{H_m}$ are new variables

Lazy Narrowing with Constraints

$$\{\lambda\overline{x_k}.s \rightarrow^{?} \lambda\overline{x_k}.t\} \cup S \quad \Rightarrow \quad \{\lambda\overline{x_k}.s \rightarrow^{?}_{d} \lambda\overline{x_k}.l\} \cup \{\lambda\overline{x_k}.r \rightarrow^{?} \lambda\overline{x_k}.t\} \cup \{Irr(\mathcal{FV}(l))\} \cup S$$
where $l \rightarrow r$ is an $\overline{x_k}$-lifted rule

Constraint-Failure

$$\{Irr(\overline{t_n})\} \cup S \quad \Rightarrow \quad fail \quad \text{if some } t_i \text{ is } R\text{-reducible}$$

Figure 6.6: System LNC for Lazy Narrowing with Constraints

as for Theorem 4.1.5, that all computed substitutions are patterns. Thus completeness of LNC for normalized solutions follows easily. We will see later that this restriction for the Elimination rule is fulfilled for a special strategy we examine in Section 6.7.2.

It may seem that checking the reducibility constraints is costly, but with normalized substitutions many redundant narrowing attempts can be avoided early. This has been shown in the context of LSE narrowing [BKW93], where also many reducibility conditions have to be checked.

Notice that System LNC may introduce redundant constraints that lead to redundant checks, e.g. if $Irr(t)$ is added and t is a subterm of an existing constraint. All in all, the idea of this section is to show when it is possible to add irreducibility constraints. In applications it may be interesting to add constraints only selectively, as reducibility checks can be costly.

6.6.4 Lazy Narrowing with Simplification

Simplification by normalization of goals is one of the earliest [Fay79] and one of the most important optimizations. Its motivation is to prefer deterministic reduction over search within narrowing. Notice that normalization coincides with deterministic evaluation in functional languages. For first-order systems, functional-logic programming with normalization has shown to be a more efficient control regime than pure logic programming [Fri85, Han92].

The main problem of normalization is that completeness of narrowing may be lost. For first-order (plain) narrowing, there exist several works dealing with completeness of normalization in combination with other strategies (for an overview see [Han94b]). Recall from Section 6.5.1 that deterministic operations are possible as soon as the left-hand side of a goal has been simplified to a term with a constructor at its root. For instance, with the rule $f(1) \to 1$, we can simplify a goal $f(1) \to^? g(Y)$ by

$$\{f(1) \to^? g(Y), \ldots\} \Rightarrow \{1 \to^? g(Y), \ldots\}$$

and deterministically detect a failure.

In the following, we show completeness of simplification for lazy narrowing under some restrictions on the HRS employed. The result is similar to the corresponding result for the first-order case [Han94c]. The technical treatment here is more involved in many respects due to the higher-order case. Using oriented goals, however, simplifies the completeness proof.

For oriented goals, normalization is only complete for goals $s \to^? t$, where θt is in R-normal form for a solution θ. For instance, it suffices if t is a ground term in R-normal form. This is in general no restriction as discussed in Section 6.1.1 and corresponds to the intuitive understanding of directed goals.

Definition 6.6.6 A **simplification step** on a goal $s \to^? t$ is a rewrite step on s, written as $\{s \to^? t\} \Rightarrow_{NLN} \{s' \to^? t\}$ if $s \longrightarrow^R s'$. **Normalizing Lazy Narrowing (NLN)**, is defined as the rules of LN plus arbitrary simplification on goals.

Observe that simplification is not desirable and hence not permitted at marked goals. Furthermore, simplifying the right-hand sides is not desirable. It may produce solutions

repeatedly. For the Constructor Clash rule (see Section 6.5.1) a constructor at the left suffices; normalizing the right-hand side may even evade this search space pruning.

We first need an auxiliary construct for the termination ordering in the completeness result. The **decomposition function** D on goals is defined as

$$\begin{array}{rcl} D(s \to^? t) & = & s \to^? t \\ D(\lambda\overline{x_k}.f(\overline{s_n}) \to_d^? \lambda\overline{x_k}.f(\overline{t_n})) & = & \overline{\lambda\overline{x_k}.s_n \to^? \lambda\overline{x_k}.t_n} \end{array}$$

and is undefined otherwise. The function D extends component-wise to sets of goals. The idea of D is to view marked goals as goals with delayed decomposition. Thus D maps goals to their intended interpretation.

Theorem 6.6.7 (Completeness of NLN) *Assume a confluent HRS R that terminates with order $<^R$. If $s \to^? t$ has solution θ, i.e. $\theta s \xrightarrow{*}{}^R \theta t$ where θt and θ are R-normalized, then $\{s \to^? t\} \stackrel{*}{\Rightarrow}^{\delta}_{NLN} F$ such that δ is more general modulo the newly added variables than θ and F is a set of flex-flex goals.*

Proof Let $<^R_{sub} = <^R \cup <_{sub}$. Assume $\overline{G_n} = \overline{s_n \to^?_{(d)} t_n}$ is a system of goals with solution θ, i.e. $\overline{\theta s_n \xrightarrow{*}{}^R \theta t_n}$. Let $\overline{s'_m \to^? t'_m} = D(\overline{G_n})$. The proof proceeds by induction on the following lexicographic termination order on $(\overline{G_n}, \theta)$:

- A: $<^R_{sub}$ extended to the multiset of $\{\overline{\theta s'_m}\}$,
- B: multiset of sizes of the bindings in θ,
- C: multiset of sizes of the goals $\overline{\theta G_n}$,
- D: $<^R$ extended to the multiset of $\{\overline{s_n}\}$.

By Theorem 4.3.10, item A is terminating. For the proof we need the invariant that all $\overline{t'_m}$ are R-normalized terms. As in Theorem 6.5.1, the following invariant holds for marked goals: if $s \to^?_d t$, then $Head(\theta s) = Head(\theta t)$ is not a free variable and furthermore, no rewrite step at root position occurs in $\theta s \xrightarrow{*}{}^R \theta t$.

In the following we show that normalization reduces this ordering and, furthermore, that for a non flex-flex goal some rule applies that reduces the ordering. In addition, we show in each of these cases that the above invariants are preserved. First, we select some non flex-flex goal $s \to^? t$ from $\overline{G_n}$. If none exists, the case is trivial.

We first consider the case where a simplification step is applied to an unmarked goal, i.e. $s \to^? t$ is transformed to $s' \to t$. We obtain $\theta s \xrightarrow{*} \theta s'$ from Lemma 4.3.4. As θt is in R-normal form, confluence of R yields $\theta s \xrightarrow{*} \theta s' \xrightarrow{*} \theta t$. Thus θ is a solution of $s' \to t$. For termination, we have two cases:

- If $\theta s = \theta s'$, measures A through C remain unchanged, whereas D decreases.
- If $\theta s \neq \theta s'$ measure A decreases.

Clearly, the invariants are preserved.

If no simplification is applied, we distinguish two cases: if $\theta s = \theta t$, then we proceed as in pure unification. Similar to Theorem 4.1.7, one of the rules of higher-order unification applies. In case of the Deletion rule, measure A decreases. For Decomposition on marked goals, A and B remain unchanged, whereas C decreases. On unmarked goals,

Decomposition reduces A. Imitation on marked goals does not change A, but reduces B; on unmarked goals, it reduces A. Projection only decreases B.

As in Theorem 6.6.2, normalization of the associated solution is preserved. Furthermore, the terms $\overline{\theta t'_m}$ do not change under Decomposition and Imitation on marked goals. On unmarked goals, Decomposition and Imitation yield new right-hand sides. These are subterms of θt and are thus R-normalized.

In the remaining case, there must be a rewrite step in $\theta s \xrightarrow{*} \theta t$. In the first case, assume there is no rewrite step at the root position in $\theta s \xrightarrow{*} \theta t$. Hence all terms in this sequence have the same root symbol. Then, similar to the last case, one of the unification rules must apply.

Now consider the case with rewrite steps in $\theta s \xrightarrow{*} \theta t$ at root position. Clearly, $s \to^? t$ cannot be marked. Assume the first of these to be $\theta s \xrightarrow{*} \lambda\overline{y_k}.s_1 \longrightarrow_\epsilon^{l \to r} \lambda\overline{y_k}.t_1$, with the rule $l \to r$. Notice that $s_1 \to t_1$ must be an instance of $l \to r$. Then we apply Lazy Narrowing, yielding the subgoals:

$$s \to_d^? \lambda\overline{y_k}.l, \lambda\overline{y_k}.r \to^? t$$

As there exists δ such that $s_1 = \delta l$ and $t_1 = \delta r$, we can extend θ to the newly added variables: define $\theta' = \theta \cup \delta$. Let $\overline{s_m \to^? l_m} = D(\theta' s \to_d^? \lambda\overline{y_k}.\theta' l)$. Clearly, $s_i <_{sub}^R \theta' s$ holds, and $\theta'\lambda\overline{x_k}.r <_{sub}^R \theta' s$ follows from $\theta' s \xrightarrow{*} \theta'\lambda\overline{y_k}.r$. Thus θ' is a solution of $\overline{s_m \to^? l_m}$ and $\lambda\overline{y_k}.r \to^? t$, that coincides with θ on $\mathcal{FV}(\overline{G_n})$. It remains to show that θ' is in R-normal form. Similar to Theorem 6.6.2, we assume that the reduction is innermost and thus $\theta\overline{l_m}$ are in R-normal form. As l is a pattern, this yields that θ' is R-normalized. □

The termination ordering in this proof is rather complex. For instance, the last item in the ordering is needed in the following example: assume a goal $\lambda x.c(F(x,t)) \to^? \lambda x.c(x)$ with solution $\theta = \{F \mapsto \lambda x, y.x\}$. Here, normalization of t does not change the term $\theta\lambda x.c(F(x,t))$ and thus does not contribute to the solution.

It should be mentioned that the results for deterministic eager variable elimination in Section 6.6.2 can easily be extended to this slightly different completeness proof: measure A is reduced as one goal is removed and the remaining $\overline{\theta s_n}$ do not change.

6.7 Lazy Narrowing for Left-Linear HRS

This section examines refinements possible for left-linear rewrite rules. This is an important class of rewrite rules, as left-linearity is common in functional(-logic) programming languages. The main contribution is a restricted class of goals that suffices for lazy narrowing with left-linear HRS. This class facilitates several optimizations not possible for general HRS.

6.7.1 An Invariant for Goal Systems: Simple Systems

In this section we introduce a particular class of goal systems, Simple Systems, with several interesting properties. For instance, the occurs check is not needed and it is easy to check if the system is solved. We show that this class is closed under the rules of LN for left-linear R. Furthermore, in the second-order case, the syntactic solvability (wrt. the conversions of λ-calculus) is decidable for systems of this class.

The invariant of Simple Systems allows for further optimizations, e.g. a closer analysis of the variables involved and eager variable elimination (Section 6.7.2).

The properties are not specific to the higher-order case and apply to fist-order systems as well. This holds particularly for results on solvability checks in the next section, which can be expensive in an actual implementation.

To introduce Simple Systems, we first define an ordering on goals:

Definition 6.7.1 We write $s \to^? s' \ll t \to^? t'$, if $\mathcal{FV}(s') \cap \mathcal{FV}(t) \neq \{\}$.

This ordering links goals by the variables occurring: e.g. $t \to^? f(X) \ll X \to^? s$. If $G_i \ll G_j$ we say there is a **connection** between these two goals. If two goals have no connection, then they are called **parallel**.

The following properties are essential for Simple Systems.

Definition 6.7.2 A system of goals $\overline{G_n} = \overline{s_n \to^? t_n}$ is called **cycle free** if the transitive closure of $\ll$ is a strict partial ordering on $\overline{G_n}$ and **right isolated** if every variable occurs at most once on the right-hand sides of $\overline{G_n}$.

Now we are ready to define Simple Systems:

Definition 6.7.3 A system of goals $\overline{G_n} = \overline{s_n \to^? t_n}$ is a **Simple System**, if

- all right-hand sides $\overline{t_n}$ are patterns,
- $\overline{G_n}$ is cycle free, and
- $\overline{G_n}$ is right isolated.

For instance, to solve a matching problem $s \to^? t$ we may wlog. assume that t is ground, thus the system is simple.

No corresponding refinement for the first-order case is known to our knowledge. For first-order lazy narrowing in [DMS92][1] right isolation is used. But the invariant there is not completely formalized and serves only a very special purpose. Simple Systems generalize this informal invariant in [DMS92] in several respects. Another fragment of Simple Systems, i.e. the ordering on goals, is used in [CF91] to locate simplification steps after instantiation of variables.

The following properties of variables in Simple Systems follow easily from the definition:

Lemma 6.7.4 *Assume a Simple System $\overline{G_n}$ and a goal $G_i = (s \to^? t)$. Then*

- $\mathcal{FV}(s) \cap \mathcal{FV}(t) = \{\}$.
- *If $X \in \mathcal{FV}(s)$ and G_i is minimal wrt. $\ll^+$, then X occurs on no right-hand side.*
- *If $X \in \mathcal{FV}(t)$ and G_i is maximal wrt. $\ll^+$, then X occurs nowhere else.*

Solving a single goal $l \to^? r$ of a Simple System by pure unification is decidable in the second-order case by Theorem 5.2.1, since r is a linear pattern and l and r share no variables. We extend this to goal systems in Section 6.7.1. Notice that in a Simple System, no occurs check is needed, e.g. $P \to^? c(P)$ cannot occur. This extends to the full system of goals since no cycles are allowed. For instance, a "hidden" occurs check, as in $\{P \to^? c(Q), Q \to^? P\}$, is impossible.

[1] In the proof of Theorem 2.1, the case with left-linear rules.

Simple Systems and Lazy Narrowing

The next theorem shows that Simple Systems are closed under the rules of LN for a left-linear HRS. For the Decomposition rule and the two narrowing rules, the proof follows easily from the form of the goals in Simple Systems and from the restriction on the rules. The imitation and projection bindings introduce new variables, but do not create cycles. The Elimination rule requires a case distinction. For instance, when eliminating a goal of the form $t \to^? P$, the variable P does not occur in any other goal on the right-hand side. Notice that the restriction to patterns on the right-hand side fits nicely with the results in Section 6.5.1: if the left side of a goal has a constructor outside, a deterministic operation applies.

The restriction to left-linear rules is rather standard in functional-logic programming. Similarly, the core of common functional languages such as SML or Haskell consists of left-linear rewrite rules. With directed goals, left-linearity draws a line between matching, as done here, and unification: for equational unification, a non-linear rule $X = X \to \mathit{true}$ must be added. For programming applications full unification is usually not needed. Furthermore, we will see that left-linear rules permit several optimizations.

Theorem 6.7.5 *Assume a left-linear HRS R. If $\overline{G_n}$ is a Simple System, then applying LN with R preserves this property.*

Proof We have the following cases if a goal G_i is transformed:

- Deletion: trivial.
- Decomposition: assume a goal $G_i = f(\overline{t_n}) \to^? f(\overline{t'_n})$ is decomposed to $\overline{G'_n} = \overline{t_n \to^? t'_n}$. Then there can be no connection between some goals in $\overline{G'_n}$. Each of these new goals has at most the connections of G_i and no others. Hence the system remains simple.
- Elimination: we have two cases, depending on the form of G_i:
  - Assume $G_i = X \to^? t$ and let $\{\overline{X'_m}\} = \mathcal{FV}(t)$. There can be at most one goal G_j with X on the right-hand side. When substituting X in G_j, only new connections to G_j are created that are already in $\ll^+$. Substituting t for X on some left-hand side does not introduce new connections as the variables $\overline{X'_m}$ may not occur on the right-hand side of some other goal. As t is a pattern the right sides remain patterns and thus the system remains simple.
  - If $G_i = t \to^? X$, then X may occur only on the left-hand side of some goals. Assume some $G_j = C'(X) \to u$ with $G_i \ll G_j$. We show that no new connections are added. For all G_k with $\theta G_k \ll \theta G_j$, where $\theta = \{X \mapsto t\}$, we have $G_k \ll^+ G_i \ll G_j$ as X may not occur in G_k on the right. Hence $\ll^+$ remains unchanged, as this argument holds for all such goals G_j.
- Imitation: an imitation binding of the form $\{X \mapsto \lambda\overline{x_k}.f(\overline{X_n(\overline{x_k})})\}$ clearly does not change the $\ll$-ordering and furthermore the right-hand sides remain linear patterns. The remainder of this case follows as in the Decomposition case.
- Projection: as in the Imitation case, with the only difference that Projection may eliminate variables, thus removing connections.

- Lazy Narrowing: for replacing a goal $G_i = s \to^? t$ by the goals $s \to^? \lambda\overline{x_k}.l$ and $\lambda\overline{x_k}.r \to^? t$ we assume that the variables in $\lambda\overline{x_k}.l$ are new. Thus the right-hand sides remain patterns with right isolated variables. Then $G_j \ll s \to^? \lambda\overline{x_k}.l$, iff $G_j \ll G_i$ and symmetrically for $\lambda\overline{x_k}.r \to^? t$. Thus no new connections are added and the system is simple.

□

For an implementation it is desirable that the goals are kept in an order compatible with $\ll$. Assume in an implementation goals are kept in a list L which is in an order compatible with some ordering $<$. A transformation T on L **preserves the ordering** $<$, if applying T yields a list in an order compatible to $<$. The following property is easy to see:

Theorem 6.7.6 *System LN applied to a list of goals preserves the $\ll$-ordering for left-linear HRS.*

Proof by an analysis similar to the last proof. □

Solving Simple Systems

In the following, we show that solving second-order Simple Systems by unification is decidable. Furthermore, a particular solved form, which is equivalent to dag-solved form, is easy to detect in Simple Systems.

The following result implies that divergence in Simple Systems only stems from the lazy narrowing rules, as in the first-order case. This is important for practical applications. For instance, it is possible to determine if a Simple System has a syntactic solution before attempting a narrowing step.

For the next result we have to consider weakly second-order terms for the following reason: if a goal contains a second-order bound variable, lifting may yield a weakly second-order term.

Theorem 6.7.7 *Solving a weakly second-order Simple System $\overline{G_n}$ by unification is decidable and yields only a finite number of solutions.*

Proof We iteratively solve maximal (wrt. $\ll^+$) goals with LN. That is, if $s \to^? t$ is a maximal goal, then t is a linear pattern and the free variables in t may not occur elsewhere. Then solving this goal with PT (LN without the narrowing rules) terminates by Theorem 5.2.1 with a set of flex-flex pairs, all of which are of the form

$$\lambda\overline{x_k}.H(\overline{t_n}) \to^? \lambda\overline{x_k}.G(\overline{y_j}),$$

where G does not occur elsewhere. Such pairs can be finitely solved by Theorem 5.2.2. It remains to be seen that this solution preserves the property that the remaining system is simple: all solutions for $F \in \mathcal{FV}(\lambda\overline{x_k}.H(\overline{t_n}))$ are of the form $\{F \mapsto \lambda\overline{x_k}.F'(\overline{z_j})\}$, where $\{\overline{z_j}\} \subseteq \{\overline{x_k}\}$ and F' is a new variable of appropriate type. Hence, when applying this solution to the remaining equations, the system remains simple, as G does not occur elsewhere. □

Simple Systems have the advantage that it is easy to see if a system is in solved form, as we show next. In practice this means that checking whether the system is solved is less expensive. Furthermore, the occurs check is unnecessary as already pointed out.

Definition 6.7.8 A Simple System S is **simplified** if

$$S = \{X_1 \stackrel{?}{\leftrightarrow} t_1, \ldots, X_n \stackrel{?}{\leftrightarrow} t_n\}$$

and all $\overline{X_n}$ are distinct.

Theorem 6.7.9 *Simplified systems are solvable.*

Proof by induction on the number of goals. The base case is trivial. For the induction step, assume a maximal goal from a simplified system $\overline{G_n}$, say $G_n = X_n \stackrel{?}{\leftrightarrow} t_n$. Let $\theta = \{X \mapsto t_n\}$. We show that $\overline{\theta G_{n-1}}$ is simplified. There are two cases when applying the Elimination rule to G_n, depending on the form of G_n.

- If $G_n = t_n \rightarrow^? X_n$ then $\theta\overline{G'_{n-1}} = \overline{G'_{n-1}}$ as X is isolated and does not occur in $\overline{G'_{n-1}}$.
- In the other case, assume $G_n = X_n \rightarrow^? t_n$. Then $\overline{\theta G'_{n-1}} = \overline{X_{n-1} \stackrel{?}{\leftrightarrow} \theta t_{n-1}}$ as all $\overline{X_n}$ are distinct. Thus the system $\theta G'_{n-1}$ is simplified.

Notice that the Elimination rule applies as $X_n \notin \mathcal{FV}(t_n)$. □

A simple corollary is the following.

Corollary 6.7.10 *A Simple System of the form* $\{\overline{t_n \rightarrow^? X_n}\}$ *is solvable.*

It is interesting to compare simplified systems to another well-known solved form: dag-solved form. This form is often used in the first-order case [JK91], but applies to our case as well.

Definition 6.7.11 A system of equations $\overline{X_n =^? t_n}$ is in **dag-solved form** if for all $i < j$, $X_i \neq X_j$ and $X_i \notin \mathcal{FV}(t_j)$.

A system of equations in dag-solved form can be described as

$$\begin{array}{rcl} X_1 & =^? & C_1(X_2, \ldots, X_n) \\ \vdots & & \\ X_i & =^? & C_i(X_i, \ldots, X_n) \\ \vdots & & \\ X_n & =^? & C_n \end{array}$$

where all $\overline{X_n}$ are distinct and $\mathcal{FV}(\overline{C_n}) \cap \{\overline{X_n}\} = \{\}$.

Although simplified systems look very much like systems in dag-solved form, the $\ll$-ordering does not correspond to the ordering needed for dag-solved form. Let us show this by an example: the simplified system

$$\begin{array}{rcl} Y \rightarrow^? f(X) & \ll & X \rightarrow^? f(Z) \\ & \ll & g(X, Y) \rightarrow^? H \end{array}$$

is equivalent to the system (with un-oriented equations)

$$H =^? g(X, Y), Y =^? f(X), X =^? f(Z)$$

This is the only ordering of the above goals to yield a system in dag-solved form. For this reason the following proof is tricky.

Theorem 6.7.12 *A system is simplified if and only if it is in dag-solved form (modulo orientation).*

Proof Clearly, orienting a system $\overline{X_n =^? t_n}$ in dag-solved form to $\overline{t_n \to^? X_n}$ yields a simplified system.

The other direction follows by induction on the number of goals. The base case is trivial. For the induction step, assume a maximal goal from a simplified system $\overline{G_n}$, say $G_n = X_n \stackrel{?}{\leftrightarrow} t_n$. Then by induction hypothesis $\overline{G_{n-1}}$ can be reordered (and reoriented) to dag-solved form, yielding $\overline{G'_{n-1}} = \overline{X'_{n-1} =^? t'_{n-1}}$. Then we again have two cases when applying Elimination to G_n, depending of the form of G_n.

- If $G_n = t_n \to^? X_n$ then X_n does not occur in $\overline{G'_{n-1}}$ as in Theorem 6.7.9. Thus $X_n =^? t_n, \overline{G'_{n-1}}$ is in dag-solved form.
- In the other case, $G_n = X_n \to^? t_n$. Then $\overline{G'_{n-1}}, X_n =^? t_n$ is in dag-solved form, as all variables in $\mathcal{FV}(t_n)$ are isolated and cannot occur in $\overline{G'_{n-1}}$.

□

6.7.2 A Strategy for Needed Narrowing

In this section we develop a new narrowing strategy for Simple Systems, assuming R-normalized solutions, which we call needed lazy narrowing. In essence, we show that certain goals can safely be delayed, which means that computations are only performed when needed.

For this purpose, we first classify the variables occurring in Simple Systems in the next section. Then we show in Section 6.6.2 that the results on eager variable elimination from Section 6.6.2 can be extended in case of Simple Systems. This will reveal that in Simple Systems, one case of variable elimination is not desirable, the other deterministic and always possible.

Variables of Interest

In the following, we classify variables in Simple Systems into variables of interest and intermediate variables. We consider initial goals of the form $s \to^? t$, and assume that only the values for the free variables in s are of interest, neither the variables in t nor intermediate variables computed by LN. For instance, assume the rule $f(a, X) \to g(b, X)$ and the goal $f(Y, a) \to^? g(b, a)$, which is transformed to

$$Y \to^? a, a \to^? X, g(b, X) \to^? g(b, a)$$

by Lazy Narrowing. Clearly, only the value of Y is of interest for solving the initial goal, but not the value of X.

This view is sufficient for solving matching problems, where the right side is ground. A simple example is encoding logic programs with predicates and to start with queries of the form $p(\ldots) \to^? \mathit{true}$. Alternatively, one may consider goals with free variables in the right-hand side that are considered as place holders for some value to be computed. For instance, if a function evaluates to pairs, we may only be interested in one component. Thus, for instance, the oriented query $s \to^? \mathit{pair}(0, X)$ may suffice.

The main result in this section is that Simple Systems allow us to identify variables of interest very easily. Furthermore, we will see how this distinction nicely integrates with our approach to eager variable elimination.

The interesting invariant we will show is that variables of interest only occur on the left, but never on the right-hand side of a goal. We first need to define the notion of variables of interest. Consider an execution of LN. We start with a goal $s \to^? t$ where initially the variables of interest are in s. This has to be updated for each LN step. If X is a variable of interest, and an LN step computes δ, then the free variables in δX are new variables of interest. With this idea in mind we define the following:

Definition 6.7.13 Assume a sequence of transformations $\{s \to^? t\} \stackrel{*}{\Rightarrow}^{\delta}_{LN} \{\overline{s_n \to^? t_n}\}$. A variable X is called a **variable of interest** if $X \in \mathcal{FV}(\delta s)$ and intermediate otherwise.

Now we can show the following result:

Theorem 6.7.14 *Assume a left-linear HRS R, a Simple System $\overline{G_n} = \{\overline{s_n \to^? t_n}\}$ and a set of variables V with $V \cap \mathcal{FV}(\overline{t_n}) = \{\}$. If $\overline{G_n} \Rightarrow^{\delta}_{LN} \{\overline{s'_m \to^? t'_m}\}$, then $((V - \mathcal{I}m(\delta)) \cup \mathcal{R}ng(\delta)) \cap \mathcal{FV}(\overline{t'_m}) = \{\}$.*

Proof For all rules of LN, except the Elimination rule, the claim is trivial. For the Elimination, consider first a goal of the form $t \to^? X$. In this case $X \notin V$ and X may not occur on any other right-hand side. Hence variables from V in t are only copied to some other left-hand side.

After the elimination of a goal $X \to^? t$ with $X \in V$, the right isolated free variables in t are in $\mathcal{R}ng(\delta)$, but do not occur in $\mathcal{FV}(\overline{t'_m})$. If $X \notin V$, nothing remains to show as $\mathcal{FV}(t) \cap V = \{\}$. □

Then the desired result follows easily:

Corollary 6.7.15 (Variables of Interest) *Assume a left-linear HRS R and assume solving a Simple System $s \to^? t$ with system LN. Then variables of interest only occur on the left, but never on the right-hand side of a goal.*

Notice that variables from the right may be shifted by the Elimination rule to some left-hand side.

The Two Cases of Variable Elimination

As we consider oriented equations, we can distinguish two cases of variable elimination and we will handle variable elimination appropriately in each case. In the first case,

$$X \to^? t,$$

the variable X can be a variable of interest. Thus the elimination of X is desirable for computational reasons and is deterministic for normalized solutions, as shown in Section 6.6.2. Notice that elimination is always possible on such goals in Simple Systems, as $X \notin \mathcal{FV}(t)$. In the context of Simple Systems we can refine the result for eager variable elimination in Section 6.6.2 by an additional failure case. Assume a goal $X \to^? t$ of a Simple System with an R-normalized solution θ. We have two cases:

- If t is in R-normal form, then elimination is deterministic by Theorem 6.6.3.

- If t is R-reducible, then the goal is unsolvable. As t is a pattern and the solution for X, i.e. θX, is R-normalized, $\theta X = \theta t$ must hold. This is impossible, as Theorem 4.3.5 entails that θt is reducible.

This observation shows the intuitive reason why Elimination is deterministic: in this case Elimination does not copy terms to be evaluated, t must be in normal form. In the other case of variable elimination, i.e.

$$t \to^? X,$$

elimination may not be deterministic and is not desirable, as we argue below.

Needed Lazy Narrowing

The results on intermediate variables and eager variable elimination in mind, we develop a new narrowing strategy. The idea is to delay goals of the form $t \to^? X$. This simple strategy has some interesting properties, which we will examine in the following.

We first view this idea in the context of a programming language. Let us for instance model the evaluation (or normalization) $f(t_1, t_2)\downarrow_R = t$ by Lazy Narrowing, assuming the rule $f(X, Y) \to g(X, X)$:

$$\{f(t_1, t_2) \to^? t\} \Rightarrow_{LN} \{t_1 \to^? X, t_2 \to^? Y, g(X, X) \to^? t\}$$

Now with the optimizations considered so far, variable elimination and normalization, we can model the following evaluation strategies.

Eager evaluation is obtained by performing normalization on the goals t_1 and t_2, followed by eager variable elimination on $t_1\downarrow_R \to^? X$ and $t_2\downarrow_R \to^? Y$. The disadvantage is that eager evaluation may perform unnecessary evaluation steps.

Lazy evaluation is obtained by immediate eager variable elimination on $t_1 \to^? X$ and on $t_2 \to^? Y$. It has the disadvantage that terms are copied, e.g. t_1 here as X occurs twice in $g(X, X)$. Thus expensive evaluation may have to be done repeatedly.

Needed (lazy) evaluation is an evaluation strategy that can be obtained by delaying the goals $t_1 \to^? X$ and $t_2 \to^? Y$, thus avoiding copying. Then t_1 and t_2 are only evaluated when X or Y are needed for further computation.

Needed lazy evaluation models equationally lazy evaluation with sharing copies of identical subterms [BvEG+87], i.e. the delayed equations may be viewed as shared subterms. It should be noted that the strategy may not be optimal as defined in [HL91], neither concerning the number of R-reductions nor β-reductions. The notion of need considered here is similar to the notion of call-by-need in [Wad71].

Let us now come back from evaluation to the context of narrowing. Consider for instance the Lazy Narrowing step with the above rule

$$\{f(t_1, t_2) \to^? g(a, Z)\} \Rightarrow_{LN} \{t_1 \to^? X, t_2 \to^? Y, g(X, X) \to^? g(a, Z)\}$$

In contrast to evaluation as in functional languages, solving the goals $t_1 \to^? X, t_2 \to^? Y$ may have many solutions. Whereas in functional languages, eager evaluation can be more efficient, this is unclear for solving equations or functional-logic programming. Thus we propose the following approach:

Definition 6.7.16 Needed Lazy Narrowing is defined as Lazy Narrowing where goals of the form $t \to^? X$ are delayed, if no deterministic operation, i.e. Constructor Imitation (Section 6.5.1) or Elimination (Theorem 6.6.4), applies.

For instance, in the above example, decomposition on $g(X, X) \to^? g(a, Z)$ yields the goals $X \to^? a, X \to^? Z$. Then deterministic elimination on $X \to^? a$ instantiates X, thus the goal $t_1 \to^? a$ has to be solved, i.e. a valued for t_1 is needed. In contrast, $t_2 \to^? Y$ is delayed.

This new notion of narrowing for Simple Systems and left-linear HRS is supported by the following arguments: **Needed Lazy Narrowing**

is complete, or safe, in the sense that when only goals of the form $\overline{t_n \to^? X_n}$ remain, they are solvable by Theorem 6.7.10. Since the strategy is to delay such goals, this result is essential.

delays intermediate variables only. As shown in the last section, we can identify the variables to be delayed: a variable X in a goal $t \to^? X$ cannot be a variable of interest.

avoids copying, as shown above, variable elimination on intermediate variables possibly copies unevaluated terms and duplicates work. Thus intermediate goals of the form $t \to^? X$ are only considered if X is instantiated, i.e. if a value is needed.

Sharing, as modeled equationally in the Needed Lazy Narrowing strategy, is often considered on an implementational level. In contrast, we have a more abstract view of sharing, which may lead to the same implementation: since each variable occurs only once on the right, it is sensible to view the delayed goals as a context of delayed terms. In an implementation, an intermediate variable can be associated with a pointer to the corresponding delayed goal. If the variable occurs repeatedly, this corresponds to sharing.

The notion of safe delaying stated above can be illustrated by an example. In practice, not only completeness but also (early) detection of failure is important. For instance, assume two goals

$$a \to^? X, b \to^? X,$$

where a and b are in normal form. Then with delaying both goals, the apparent unsolvability will never be detected. This will not occur with the above strategy in Simple Systems, as these are right isolated. Hence a variable X in a delayed goal $t \to^? X$ may occur on some left-hand side, but not on two right sides.

Next we examine a problem that occurs in the higher-order case when Needed Lazy Narrowing is employed. Two kinds of equations are delayed:

- (A) flex-flex goals of the form $\lambda\overline{x_k}.X(\overline{t_n}) \to^? \lambda\overline{x_k}.Y(\overline{y_m})$
- (B) goals of the form $t \to^? X$

A system of such goals can be unsolvable in general. Consider for instance

$$\begin{aligned} \lambda x.Y(x) &\to^? \lambda x.F, \\ \lambda x.f(x) &\to^? Y \end{aligned}$$

which is unsolvable. Thus the delayed goals have to be solved by narrowing. We conjecture that such cases are rare. Furthermore, in many cases such goals are solvable:

Proposition 6.7.17 *Assume a second-order Simple System consisting of a set of flex-flex goals* $\overline{G_m}$ *and a set of goals* $\overline{t_n \to^? X_n}$*. Such a system is solvable if the following condition holds for all* i *and* j*:* $t_i \to^? X_i \ll G_j$ *implies that* X_i *is first-order.*

Proof The strategy of the proof is to eliminate goals from $\overline{t_n \to^? X_n}$ until only flex-flex goals remain. We show that each such Elimination transforms the goals into another set of goals of the above form. It clearly terminates, as the number of goals reduces.

For an elimination of $t_i \to^? X_i$, the variable X_i cannot occur on some other right-hand side. We consider two cases. If there is no G_j with $t_i \to^? X_i \ll G_j$, then X_i does not occur in some flex-flex goal and thus $\overline{G_m}$ does not change.

Otherwise, if

$$t_i \to^? X_i \ll G_j = \lambda\overline{x_k}.X(\overline{y_o}) \to^? \lambda\overline{x_k}.Y(\overline{z_m}),$$

then there are again two cases: if $X_i \neq X$, then G_j remains flex-flex, and the case is trivial. In the remaining case, we have $X_i = X$ and $n = 0$ as X and t_i are first-order by assumption. Binding X to t_i yields $\lambda\overline{x_k}.t_i \to^? \lambda\overline{x_k}.Y(\overline{z_m})$. As t is first-order, Theorem 5.1.4 for System EL applies, yielding the solution $\{Y \mapsto \lambda\overline{z_m}.t\}$. For simplicity, we only apply $\theta = \{Y \mapsto \lambda\overline{z_m}.Y'\}$ for a new variable Y', which yields an equation where Elimination applies. Thus θG_j is not flex-flex, but of the form of the $\overline{t_n \to^? X_n}$ goals. Furthermore, the other flex-flex goals remain flex-flex when applying θ. As this holds for all G_j, we obtain a smaller set of goals where the induction hypothesis applies. □

6.8 Lazy Narrowing with Conditional Equations

Adding conditions to equations is very common for rewrite systems. Although, at least in the first-order case, this may not increase the expressive power [BT87], conditions are often convenient. For instance, consider the rules

$$\begin{array}{lcll} \mathit{fib}(X) & \to & \mathit{fib}(X-1)+\mathit{fib}(X-2) & \Leftarrow X > 1 \\ \mathit{fib}(X) & \to & 1 & \Leftarrow X \leq 1 \end{array}$$

In the following section, we develop a general notion of conditional narrowing with unrestricted conditions. Much research has been dedicated to narrowing with conditional equations. This has led to an abundance of different classes of conditional term rewrite systems and many different results. There exist various restrictions on the variables occurring in the conditions, for instance in [MH94] a hierarchy of four classes of conditional rules can be found. Combining these with the known strategies for (plain) narrowing led to an abundance of results in the first-order case, see for instance [MH94].

One of the problems with conditional rewriting is that termination of the associated rewrite relation does not imply the termination of conditional rewriting: rewriting the conditions proceeds recursively and may diverge without any actual reduction performed on the main goal. Thus most termination criteria for first-order conditional term rewriting need additional restrictions that assure that the reductions in the conditions are decreasing some termination ordering. For termination criteria that include conditional higher-order rewrite systems see for instance [LS93]. Another problem, addressed below, is that solving the conditions may require reducible substitutions, which renders many first-order strategies with plain narrowing incomplete.

We will discuss in Section 6.8.2 that in our functional approach many of these problems can be avoided due to the higher-order setting.

6.8.1 Unrestricted Conditional Equations

In the following we introduce an unrestricted notion of conditional rules. For instance, the conditions may have variables not occurring in the rule itself. These are called extra variables. We will see in Section 7.4 that conditions with extra variables are useful for some examples.

Definition 6.8.1 A **higher-order conditional rule** is of the form $l \to r \Leftarrow \overline{l_n \to r_n}$, where l is a higher-order pattern of base type and not η-equivalent to a free variable. A conditional HRS is a set of such rules and is abbreviated by **CHRS.**

In the literature, there exist different notions of conditional rewriting. They differ in the way the equations are to be solved. Either requiring $\overline{l_n \longrightarrow r_n}$, which is called normal equality [DO90] or $\overline{l_n \downarrow r_n}$, called join equality. In the latter the logical equality induced by R is considered. The former is more tailored for programming languages, where evaluation is of interest, and usually the right-hand sides of the conditions are assumed to be ground R-normal forms, which we consider in the following section.

Definition 6.8.2 Assuming a rule $(l \to r \Leftarrow \overline{l_n \to r_n}) \in R$ and a position p in a term s in long $\beta\eta$-normal form, a **conditional rewrite step** from s to t is defined recursively as

$$s \longrightarrow_{p,\theta}^{l \to r \Leftarrow \overline{l_n \to r_n}} t \quad \Leftrightarrow \quad s \longrightarrow_{p,\theta}^{l \to r} t \ \wedge \ \overline{\theta l_n \xrightarrow{*}{}^{R} \theta r_n}$$

Lifting rewrite rules over a set of variables extends to conditional rules by applying the lifter as well to the conditions.

Lazy Narrowing with Conditions

$$\{\lambda\overline{x_k}.s \to^{?} \lambda\overline{x_k}.t\} \cup S \quad \Rightarrow \quad \{\lambda\overline{x_k}.s \to_d^{?} \lambda\overline{x_k}.l,\ \overline{\lambda\overline{x_k}.l'_n \to^{?} \lambda\overline{x_k}.r'_n},\ \lambda\overline{x_k}.r \to^{?} \lambda\overline{x_k}.t\} \cup S$$

where $l \to r \Leftarrow \overline{l'_n \to r'_n}$ is an $\overline{x_k}$-lifted rule

Figure 6.7: System CLN for Conditional Lazy Narrowing

Definition 6.8.3 We define **Conditional Lazy Narrowing (CLN)** as the unification rules of system LN plus the Lazy Narrowing with Conditions rule in Figure 6.7.

Define the **length of a conditional reduction** as

$$len(s_1 \longrightarrow_{\sigma}^{l \to r \Leftarrow \overline{l_n \to r_n}} s_2 \xrightarrow{*} s_n) = 1 + len(s_2 \xrightarrow{*} s_n) + \Sigma_{i=1,\ldots,n} len(\sigma l_i \xrightarrow{*} \sigma r_i)$$

if $n > 1$ and $len(s_1) = 0$ if $n = 1$.

This notion of the length of a reduction reflects the problem with termination of conditional rewriting, since the rewrite steps for the conditions are included. As mentioned above, a conditional rewrite relation may itself terminate, but there may be reductions with infinite length.

Theorem 6.8.4 (Completeness of CLN) *Assume a CHRS R. If $s \to^? t$ has solution θ, i.e. $\theta s \xrightarrow{*}{}^R \theta t$, then $\{s \to^? t\} \overset{*}{\Rightarrow}{}^{\delta}_{CLN} F$ such that δ is more general modulo the newly added variables than θ and F is a set of flex-flex goals.*

Proof We only add a few changes and generalizations to the proof of System LN. We assume the setup and invariants of Theorem 6.5.1.

We use induction on the following termination ordering for a system of goals $\overline{G_n} = \overline{s_n \to t_n}$ with solution θ.

- A: The sum of the lengths of the conditional R-reductions in all goals $\overline{\theta G_n}$.
- B: Multiset of the sizes of the bindings in θ.
- C: Multiset of the sizes of the goals $\overline{G_n}$.

Only the lazy narrowing step differs from System LN: consider the first rewrite step at root position, which is assumed to be $\theta s \xrightarrow{*} \lambda\overline{x_k}.s_1 \longrightarrow_\epsilon^{l \to r \Leftarrow \overline{l'_o \to r'_o}} \lambda\overline{x_k}.t_1$. Hence $s_1 \to t_1$ must be an instance of $l \to r \Leftarrow \overline{l'_o \to r'_o}$ such that the conditions are solvable. Therefore, there exists a substitution δ with $\delta l = s_1$ and $\delta r = t_1$ such that $\overline{\delta l'_o \longrightarrow \delta r'_o}$. Let m be the number of (conditional) reductions in $\overline{\delta l'_o \longrightarrow \delta r'_o}$. Thus the size of this conditional rewrite step is $m+1$. Hence applying Lazy Narrowing with Constraints reduces A, as one conditional reduction of size $m+1$ is replaced by new goals with conditional reductions of a total size m. As in Theorem 6.5.1, $\theta \cup \delta$ is a solution to the newly added goals. □

It may seem tempting to examine conditional narrowing with normalized substitutions as in Section 6.6.1, but it is difficult to show that the solutions for the extra variables in the conditions are normalized. In the first-order case, this is a known problem as mentioned in the beginning of this section (see for instance [Han94b]). We therefore discuss conditional narrowing for a restricted class of rules in the following section, where the optimizations for unconditional rules of Section 6.5 can be adapted.

6.8.2 Normal Conditional Rules

In this section, we discuss narrowing for a restricted class of conditional rewrite rules, which we argue to be sufficient for programming purposes. We examine how these restrictions can be utilized for the optimizations developed for unconditional lazy narrowing. The restrictions will allow to use Simple Systems with conditional rules. Our restrictions and invariants are stronger than the ones currently used in some first-order functional-logic languages. This is possible as some operational constructs, which we disallow, are often simpler expressed directly in a higher-order framework (see also Section 2.6).

Definition 6.8.5 A **normal conditional HRS (NCHRS)** R is a set of conditional rewrite rules of the form $l \to r \Leftarrow \overline{l_n \to r_n}$, where $l \to r$ is a rewrite rule and $\overline{r_n}$ are ground R-normal forms.

Note that there is no difference between normal equality and joinability in our case as the right-hand sides of the rules are in ground R-normal form. Thus oriented goals suffice for proving the conditions as $\overline{\theta l_n \xleftrightarrow{*}{}^R \theta r_n}$ is equivalent with $\overline{\theta l_n \longrightarrow^R r_n}$.

The definition of NCHRS may seem too restrictive, as no variables are allowed in the right sides of the conditions. As already discussed in Section 2.6, this is not needed

for higher-order programming languages. We permit extra variables on the left sides of conditions, as these are needed to embed logic programs (for an example see Section 2.6). Extra variables on the right are often used to model local variables, which can be done here by "where" or "let" constructs of functional programming languages. These can easily be described by higher-order rules, such as

$$\mathsf{let}\ X\ \mathsf{in}\ T \to T(X).$$

For instance, when writing a quick-sort program, the main rule will be of the form

$$qs(O_{\leq}, S) \to merge(qs(O_{\leq}, S_1), qs(O_{\leq}, S_2)) \Leftarrow split(O_{\leq}, S) \to (S_1, S_2),$$

where the S_i represent lists and (S_1, S_2) is a pair of lists. This is already not a first-order rule, as the ordering used for sorting is given as a parameter, here written as $O_{\leq}$. In our framework, we can write this as in a functional language:

$$\begin{array}{rl} qs(O_{\leq}, S) \to & \mathsf{let}\ pair(s_1, s_2) \quad = \quad split(O_{\leq}, S) \\ & \mathsf{in}\ \lambda s_1, s_2.merge(qs(O_{\leq}, s_1), qs(O_{\leq}, s_2) \end{array}$$

assuming a let rule for pairs, as shown in Section 7.3.

We believe that β-reduction is more appropriate than the instantiation of extra variables in the conditions. For instance, with depth-first search, as e.g. in Prolog, instantiations are recorded for possible backtracking. In contrast, β-reduction is a deterministic operation.

First we show that Simple Systems are invariant under the rules of CLN for a left-linear NCHRS R.

Theorem 6.8.6 *Assume a left-linear NCHRS R. If G is a Simple System then applying CLN with R preserves this property.*

Proof Building upon Theorem 6.7.5, we only consider the case of Conditional Lazy Narrowing. In this case, the right-hand sides of the conditions are ground terms and the new variables in the left sides of the conditions occur only on the left. Thus the system remains simple. □

Similarly we get the following result as in Section 6.7.1:

Theorem 6.8.7 *System CLN with a left-linear NCHRS applied to goals in a list preserves the $\ll$-ordering.*

Normalized Solutions and Variable Elimination

As we disallow variables on the right in conditions, it is easy to extend the results for narrowing with normalized solutions in Section 6.6.1 to normal conditional rules. For extra variables in the conditions it is often necessary to consider reducible solutions, making this important optimization impossible. In the following, we adapt the results of Sections 6.6.1 through 6.7 to conditional narrowing.

Definition 6.8.8 System **CLNN** is defined as the restriction of System CLN where Conditional Lazy Narrowing is not applied to goals of the form $\lambda\overline{x_n}.X\,(\overline{y_m}) \to^{?} t$.

Theorem 6.8.9 *Assume a convergent NCHRS R. If $s \to^? t$ has solution θ, i.e. $\theta s \xrightarrow{*}{}^R \theta t$, and θ is R-normalized, then $\{s \to^? t\} \overset{*}{\Rightarrow}{}^{\delta}_{CLNN} F$ such that δ is more general modulo the newly added variables than θ and F is a set of flex-flex goals.*

Proof As in Theorem 6.6.2, we have to show the invariant that (intermediate) solutions are R-normalized. The problem here are new variables in the conditions of the rewrite rules. Thus, we first construct a new reduction $\theta s \xrightarrow{*}{}^R \theta t$, which differs only in the rewrite proofs of the conditions.

In a conditional rewrite step in $\theta s \xrightarrow{*}{}^R \theta t$ it is possible that the substitution is not R-normalized for some new variables in the reduction in a condition. Consider e.g. $\theta s \xrightarrow{*} s_1 \longrightarrow^{l \to r \Leftarrow \overline{l_n \to r_n}}_{\delta} s_2$ with $\overline{\delta l_n \xrightarrow{*} r_n}$. As in Theorem 6.6.2, we can assume that $\delta|_{\mathcal{FV}(l)}$ is R-normalized since the reduction is innermost. Let $V = \mathcal{FV}(\overline{l_n}) - \mathcal{FV}(l)$. As $\delta|_V$ may not be R-normalized, we construct a new and equivalent reduction $s_1 \longrightarrow^{l \to r \Leftarrow \overline{l_n \to r_n}}_{\delta'} s_2$, where $\delta' = \delta\downarrow_R$. Furthermore, we can assume that the reduction $\overline{\delta' l_n \xrightarrow{*} r_n}$ is innermost since R is confluent and all $\overline{r_n}$ are in ground R-normal form. This can be repeated recursively for all conditional reductions in $\theta s \xrightarrow{*}{}^R \theta t$.

The remainder of the proof proceeds as in Theorem 6.8.4. In addition, R-normalization of the intermediate solutions is shown as in Theorem 6.6.2. This assumes the newly constructed reduction for the Conditional Narrowing Rule. □

As no variables in the right-hand sides of the conditions are allowed, it is easy to see that the results for variables of interest, variable elimination of Section 6.7.2, and narrowing strategies of Section 6.7.2 hold in this context. Only simplification is more involved, as shown next.

Normalization for Normal Conditional Narrowing

We show how the results for lazy narrowing with simplification in Section 6.6.4 can be adapted to normal conditional narrowing. The basis for this is the restriction to normalized solutions, as elaborated above.

A termination ordering $<^R$ is a **decreasing termination ordering** for an NCHRS R, if $\theta l'_i <^R \theta l$ for any θ and for all $l_i \in \{\overline{l_n}\}$ and $l \to r \Leftarrow \overline{l_n \to r_n} \in R$.

Decreasing termination orders as defined here originate from the (first-order) definition in [DOS88] and imply termination of conditional rewriting (similar to [DOS88]). This is easy to show since for any rewrite step, the left-hand sides of the conditions are smaller in the ordering. Thus it suffices to consider a multiset of reductions. Then a conditional rewrite step performs one reduction and adds only smaller elements, i.e. the conditions, to the multiset.

Definition 6.8.10 System **NCLN** is defined as System CLNN plus arbitrary simplification steps.

Theorem 6.8.11 *Assume a confluent NCHRS R with a decreasing termination ordering $<^R$. If $s \to^? t$ has solution θ, i.e. $\theta s \xrightarrow{*}{}^R \theta t$ where θt is in R-normal form, then $\{s \to^? t\} \overset{*}{\Rightarrow}{}^{\delta}_{NCLN} F$ such that δ is more general modulo the newly added variables than θ and F is a set of flex-flex goals.*

Proof To adapt the completeness result for NLN in Theorem 6.6.7, it suffices to consider a Conditional Lazy Narrowing step, all other cases are as in Theorem 6.6.7. Let $\overline{G_n} =$

$\overline{s_n \to^?_{(d)} t_n}$ be a system of goals with solution θ. Consider

$$\begin{array}{lcll} \{G_i\} & = & \{\lambda\overline{x_k}.s \to^? \lambda\overline{x_k}.t\} & \Rightarrow_{CLN} \\ \{\overline{G'_m}\} & = & \{\lambda\overline{x_k}.s \to^?_d \lambda\overline{x_k}.l, \overline{\lambda\overline{x_k}.l_o \to \lambda\overline{x_k}.r_o}, \lambda\overline{x_k}.r \to^? \lambda\overline{x_k}.t\} & \end{array}$$

Let $\overline{s'_m \to^? t'_m} = \overline{D(\theta G'_m)}$. In this case, there are only two differences to the completeness result for NLN in Theorem 6.6.7:

- First, it is to assure that all $\overline{t'_m}$ are R-normalized terms. This holds for the newly added conditions $\overline{\lambda\overline{x_k}.l_o \to^? \lambda\overline{x_k}.r_o}$, as $\lambda\overline{x_k}.\overline{r_o}$ are ground terms in R-normal form.
- The above Narrowing step reduces measure A of Theorem 6.6.7, as $\overline{\theta l_o <^R \theta s}$.

□

Notice that there is a good reason for not simplifying the right-hand side of goals in Simple Systems. Rewriting with rules where free variables occur repeatedly on the right can destroy an invariant of Simple Systems: right isolation. For simplification on the left, in contrast, it is trivial that the invariant is preserved.

6.9 Narrowing on Patterns with Constraints

We have seen in Section 6.2 that the well-developed first-order notion of plain narrowing is problematic when going beyond higher-order patterns. Although lazy narrowing solves most of these problems, it would be nice to integrate some of the ideas of the former approach.

An approach that allows to use plain narrowing in the higher-order case is presented in this section. The idea is to factor out the complicated case, narrowing at variable positions, into constraints and work with the simpler pattern part as shown in Section 6.3. The idea is similar to [Pfe91], where non-pattern unification problems are delayed in a higher-order logic programming language. In contrast to the latter, we also have to solve the constraints modulo R.

The rules NC in Figure 6.8 work on a pair (t, C), where t is a goal, in which non-pattern subterms can be shifted to the goals C with rule Flatten. These can be solved with lazy narrowing as in NC or any comparable method. Then on t, narrowing at or below variable positions is not needed. The assumption is that in many applications, most (sub-)terms are patterns, such that the pattern part performs the large part of the computation.

For instance, to solve a goal $f(F(f(a))) \to^? g(a)$ wrt. R_0 as in Example 6.4.1, we flatten the left-hand side to $(f(F') \to^? g(a), \{F(f(a)) \to^? F'\})$. Then the flattened term can be handled with first-order techniques, possibly yielding $\{F' \mapsto f(a)\}$. Solving the remaining constraint $F(f(a)) \to^? f(a)$ is simple, and it may not even be desirable to compute all its solutions.

The rule Pattern Narrow applies only at subterms that have been flattened to patterns. Hence the unification needed in rule Narrow is pattern unification. The main advantage of this version of narrowing is that we achieve a system, where we can work similar to the first-order case on the pattern part.

Solve

$(t \to^? t', C) \quad \Rightarrow^\theta \quad (t' \to^? t', \theta C)$ if $\theta t = t'$

Flatten

$(t \to^? t', C) \quad \Rightarrow \quad (t[X'(\overline{x_k})]_p \to^? t', \{\lambda\overline{x_k}.X(\overline{t_n}) \to^? X'\} \cup C)$
if p is a rigid path in t such that $t|_p = X(\overline{t_n})$
is not a pattern, where $\overline{x_k} = \mathcal{BV}(t,p)$

Pattern Narrow

$(t \to^? t', C) \quad \Rightarrow^\theta \quad (s \to^? t', \theta C)$ if p is a rigid path in t,
$t|_p$ is a pattern, and
$t \leadsto^R_{p,\theta} s$

Constraint Solving

$(t \to^? t', C) \quad \Rightarrow^\theta \quad (\theta t \to^? \theta t', C')$ if $C \Rightarrow^\theta_{LN} C'$

Figure 6.8: System NC for Narrowing with Constraints

To prove completeness we first need a more technical lemma. The problem is that the two methods integrated here are based on very different proof strategies. The next lemma shows more precisely which rewrite steps are handled by lazy narrowing in the constraints and which are modeled directly. When working with NC we will call the goal t in a tuple (t, C) the pattern part, although it may not be a pattern goal.

Lemma 6.9.1 *Assume a convergent HRS R, two terms s and t where t is a ground R-normal form, an R-normalized substitution θ, and a set of constraints $\overline{G_n} = \{\overline{u_n \to^? u'_n}\}$ such that*

- $\theta s \xrightarrow{*}{}^R t$,
- $\overline{\theta u_n \xrightarrow{*}{}^R \theta u'_n}$

Then $(s \to^? t, \{\overline{G_n}\}) \overset{*}{\Rightarrow}{}^\delta_{NC} (s' \to^? t, \{\overline{G'_m}\})$ such that there exists θ' with

- $\theta s' = t$,
- $\theta =_{\mathcal{FV}(s)} \theta'\delta$,
- $\theta s \xrightarrow{*}{}^R \theta' s'$,
- *θ' is R-normalized and*
- *θ' is a solution of $\overline{G'_m}$.*

Proof by induction on $\longrightarrow^R$ on θs, which is terminating. We maintain the last four claims as invariants. Assume $\theta s \longrightarrow_p^{l\to r} t_1 \xrightarrow{*}{}^R t$ is an innermost reduction with some appropriately lifted rule $l \to r \in R$. We have the following two cases depending on p. Since θ is normalized, p cannot occur below a pattern subterm $X(\overline{y_m})$ in s.

If p is not a position on a rigid path in s, the reduction is modeled in the constraints: let q be a minimal prefix of p such that $s|_q$ is of the form $X(\overline{t_n})$. Let $\overline{x_k} = \mathcal{BV}(t, q)$. Since θ is R-normalized, $\lambda\overline{x_k}.X(\overline{t_n})$ cannot be a pattern. Apply flatten to obtain a new constraint $G_0 = \lambda\overline{x_k}.X(\overline{t_n}) \to^? \lambda\overline{x_k}.X'(\overline{y_n})$ for a new variable X'. Let further $s' = s[X'(\overline{y_n})]q$ and $\theta' = \theta \cup \{X' \mapsto \lambda\overline{x_k}.(\theta s|_q)\downarrow_R\}$. As R is convergent, we obtain $\theta' s' \xrightarrow{*} t$ via an innermost reduction. Since $\theta s \xrightarrow{*}{}^R \theta' s'$, the induction hypothesis applies with $(s', G_0 \cup \overline{G_n})$ and θ'.

In case p is on a rigid path in s, we apply Flatten at all (maximal) non-pattern subterms $\overline{s_m}$ of $s|_p$. This yields s' and some new constraints $\overline{G'_m} = \overline{t_m \to^? X_m}$ and a new associated solution θ_1. As described in the last case, to obtain θ_1, θ has to be extended at each Flattening step. Since the reduction is innermost, all $\overline{\theta s_m}$ are in R-normal form and hence the new solutions added for $\overline{X_m}$ are R-normalized. Thus we have $\theta s|_p = \theta_1 s'|_p$.

Then the rule Pattern Narrow applies and the proof proceeds similar to Theorem 6.3.2: as $s_1|_p$ is a pattern and $\theta_1 s_1|_p$ is an instance of l, there exists a most general unifier δ of $s_1|_p$ and l and there exits θ' such that $\theta_1 = \theta'\delta$. Then the Pattern Narrow step yields $(s' \to^? t, \{\overline{\delta G'_m}, \overline{\delta G_n}\})$, where $s' = \delta s_1[r]_p$. It follows as in Theorem 6.3.2 that θ' is R-normalized. Clearly, θ' is a solution for all constraints $\overline{\delta G'_m}, \overline{\delta G_n}$. As $\theta s \xrightarrow{*}{}^R \theta_1 s_1 \xrightarrow{*}{}^R \theta' s'$, it remains to apply the induction hypothesis with θ' and $(s' \to^? t, \{\overline{\delta G'_m}, \overline{\delta G_n}\})$.

□

Now the completeness of NC follows easily. We only have to lift rewrite steps that occur in the primary goal, the others are handled by lazy narrowing in the constraints.

Theorem 6.9.2 (Completeness of NC) *Assume a convergent HRS R. If $s \to^? t$ has the solution $\theta s \xrightarrow{*}{}^R t$ where θ is R-normalized and t is a ground R-normal form, then $(s \to^? t, \{\}) \overset{*}{\Rightarrow}{}^{\delta}_{NC} (t \to^? t, C)$ such that $\delta s = t$ and δ is more general modulo the newly added variables than θ and the goals in C are flex-flex.*

Proof First, apply Lemma 6.9.1 to $(s \to^? t, \{\})$, yielding a pair $(s' \to^? t, \{\overline{G_n}\})$ that is solvable by some substitution θ' with $\theta \le \theta'$. Thus the Solve rule applies with some substitution $\delta \le \theta'$. It remains to solve the constraints $\{\delta\overline{G_n}\})$ by System LN. □

It is interesting to examine how rewrite steps in a solution $\theta s \longrightarrow^R s_1 \xrightarrow{*}{}^R t$ are modeled in the pattern part in above completeness result. There are two possibilities for a rewrite step $\theta s \longrightarrow^R s_1$ at a position p:

- If there exists a prefix q of p such that $s|_q$ is a non-pattern term, then this non-pattern subterm is flattened into the constraints and replaced by a new variable, say X. The solution associated to this (intermediate) variable X is the R-normal form of $s|_q$, thus the flattening step shifts the normalization of a full subterm into the constraints.

- Otherwise, the subterm is flattened to a pattern. Then a single narrowing step is lifted and this step takes place at the same position and with the same rule as the narrowing step. As in the first-order case, we have a one-to-one correspondence of the rewrite step in θs and the narrowing step in s.

With the last observation in mind, we conjecture that narrowing strategies for first-order rewrite systems can be lifted to the pattern part in a modular way. There are two reasons for this. First, most first-order strategies only lift particular derivations, e.g. innermost reductions (basic narrowing [Hul80]) or leftmost innermost reductions (LSE narrowing [BKW93]). A reduction is leftmost, if for each step no rewrite step at a position left to it applies. Secondly, for leftmost innermost solutions $\theta s \xrightarrow{*}{}^{R} t$, it seems that the above completeness result can be extended to show that the reductions in the pattern part form a leftmost innermost subsequence of the $\theta s \xrightarrow{*}{}^{R} t$ reduction.

Chapter 7

Applications of Higher-Order Narrowing

This section presents examples for higher-order rewriting and narrowing. As most of these applications are oriented towards programming, left-linear rewrite rules and thus Simple Systems suffice. Only the examples on program transformation in Section 7.2 and type inference in Section 7.4 go beyond programming and more expressiveness is needed. For other examples on the utility of higher-order constructs, we refer to [Nad87, PM90] for natural language parsing, [Nip91a] for formalizing logics and λ-calculi, and for Process Algebras to [Pol94].

7.1 Symbolic Computation: Differentiation

In this section we present an example for modeling symbolic differentiation. Symbolic differentiation is a standard example in many text books on Prolog [SS86]. In contrast to first-order programming, we can easily formalize the chain rule for differentiating nested functions, e.g. $\lambda x.sin(cos(x))$. This requires a notion of bound variables and is hence excluded in the first-order versions.

The naive approach to specify differentiation with an equation $diff(\lambda x.F) = \lambda x.0$ fails, as the equation is not of base type. With rules of higher type, our notion of rewriting does not capture the corresponding equational theory [Nip91a]. The idea is to define a function *diff* such that $diff(\lambda x.v, X)$ computes the value of the differential of $\lambda x.v$ at X. When abstracting over this X, we can express the differential of a function again as a function. Although the former version seems slightly more elegant, it would require a more complex notion of rewriting, not to mention narrowing.

Figure 7.1 shows the rules of R_d for symbolic differentiation with left-linear, second-order equations of base type. Observe that we do not formalize the chain rule explicitly, as this would require nested free variables. Our goal is to have patterns as left-hand sides, i.e. the left-hand side of the chain rule would be of the form $\lambda x.diff(F(G(x)))$. Notice that the right-hand sides of the rules of R_d in Figure 7.1 are non-patterns, hence rewriting a pattern term may yield a non-pattern.

We first show termination of the rules by the method developed in [Pol94]. As in [Pol94], we use natural numbers with the usual ordering as the domain for the in-

$$
\begin{array}{lcl}
\mathit{diff}(\lambda y.F, X) & \to & 0 \\
\mathit{diff}(\lambda y.y, X) & \to & 1 \\
\mathit{diff}(\lambda y.\mathit{sin}(F(y)), X) & \to & \mathit{cos}(F(X)) * \mathit{diff}(\lambda y.F(y), X) \\
\mathit{diff}(\lambda y.\mathit{cos}(F(y)), X) & \to & -1 * \mathit{sin}(F(X)) * \mathit{diff}(\lambda y.F(y), X) \\
\mathit{diff}(\lambda y.F(y) + G(y), X) & \to & \mathit{diff}(\lambda y.F(y), X) + \mathit{diff}(\lambda y.G(y), X) \\
\mathit{diff}(\lambda y.F(y) * G(y), X) & \to & \mathit{diff}(\lambda y.F(y), X) * G(X) + \\
 & & \mathit{diff}(\lambda y.G(y), X) * F(X) \\
\mathit{diff}(\lambda y.\mathit{ln}(F(y)), X) & \to & \mathit{diff}(\lambda y.F(y), X)/F(X) \\
X * 1 & \to & X \\
1 * X & \to & X \\
X * 0 & \to & 0 \\
0 * X & \to & 0 \\
0 + X & \to & X \\
X + 0 & \to & X \\
0 / X & \to & 0
\end{array}
$$

Figure 7.1: Rules R_d for Symbolic Differentiation

terpretation. Then the following interpretations[1] are strictly monotonic on the positive natural numbers:

$$
\begin{array}{rcl}
[\![\mathit{diff}]\!] & = & \lambda f, x.(f(x))^2 * X + 1 \\
[\![0]\!] = [\![1]\!] & = & 1 \\
[\![*]\!] = [\![+]\!] & = & \lambda x, y.x + y + 4 \\
[\![\mathit{sin}]\!] = [\![\mathit{cos}]\!] & = & \lambda x.x + 4 \\
[\![/]\!] & = & \lambda x, y.x + y \\
[\![\mathit{ln}]\!] & = & \lambda x.x + 1 \\
[\![-]\!] & = & \lambda x.x
\end{array}
$$

Notice that the symbols, e.g. $+$, on the left refer to the defined symbols of R_d, whereas the identical ones on the right denote the usual operations on numbers.

Next we show that R_d is confluent via critical pair analysis. It is easy to see that the first rule overlaps with all remaining rules for *diff* and the remaining rules only have trivial critical pairs. All of these are joinable; consider for instance

$$0 \longleftarrow \mathit{diff}(\lambda x.F * G, X) \longrightarrow \mathit{diff}(\lambda x.F) * G + \mathit{diff}(\lambda x.G) * F \xrightarrow{*} 0$$

Thus R_d is a convergent, left-linear HRS and we can apply Simple Systems with normalization. As an example, we attempt to solve the query

$$\lambda x.\mathit{diff}(\lambda y.\mathit{ln}(F(y)), x) \to^{?} \lambda x.\mathit{cos}(x)/\mathit{sin}(x).$$

The solution $\{F \mapsto \lambda x.\mathit{sin}(x)\}$ can be found with the pattern narrowing sequence

$$
\begin{array}{ll}
\lambda x.\mathit{diff}(\lambda y.\mathit{ln}(F(y)), x) & \longrightarrow \\
\lambda x.\mathit{diff}(\lambda y.F(y), x)/F(x) & \rightsquigarrow \\
\lambda x.\mathit{cos}(F'(x)) * \mathit{diff}(\lambda y.F'(y), x)/\mathit{sin}(F'(x)) & \overset{*}{\rightsquigarrow} \\
\lambda x.\mathit{cos}(x)/\mathit{sin}(x). &
\end{array}
$$

[1] Developed jointly with Jaco van de Pol.

as all term occurring are patterns.

Lazy narrowing provides a more goal directed search in this example, as unification can be used earlier for simplification:

$$\begin{array}{ll}
\{\lambda x.diff(\lambda y.ln(F(y)), x) \rightarrow^? \lambda x.cos(x)/sin(x)\} & \stackrel{*}{\Rightarrow} \\
\{\lambda x.diff(\lambda y.F(y), x)/F(x) \rightarrow^? \lambda x.cos(x)/sin(x)\} & \stackrel{*}{\Rightarrow} \\
\{\lambda x.diff(\lambda y.F(y), x) \rightarrow^? \lambda x.cos(x), & \\
\lambda x.F(x) \rightarrow^? \lambda x.sin(x)\} &
\end{array}$$

Now the solution is obtained by first solving the second goal by deterministic variable elimination and then by simplifying the first goal.

Although this example only uses higher-order pattern, it is easy to imagine non-pattern goals, e.g. $diff(\lambda y.sin(F(cos(y))), X) \rightarrow^? cos(X)/sin(X)$. When solving such a goal with system NC, we first flatten the pair

$$(diff(\lambda y.ln(F(cos(y)))), X) \rightarrow^? cos(X)/sin(X), \{\})$$

to

$$(diff(\lambda y.ln(F'(y))), X) \rightarrow^? cos(X)/sin(X), \{\lambda y.F(cos(y)) \rightarrow^? \lambda y.F'(y)\}).$$

Then a simple strategy is to perform narrowing on the pattern term and after each step check if the constraints are solvable. Only if they are not solvable, lazy narrowing should be applied.

7.2 Program Transformation

The utility of higher-order unification for program transformations has been shown nicely by Huet and Lang [HL78] and has been developed further in [PE88, HM88]. This example for unfold/fold program transformation is taken from [FH88]. We assume the following standard rules for lists

$$\begin{array}{lcl}
map(F, [X|R]) & \rightarrow & [F(X)|map(F, R)] \\
foldl(G, [X|R]) & \rightarrow & G(X, foldl(G, R))
\end{array}$$

Now assume writing a function $g(F, L)$ by

$$g(F, L) \quad \rightarrow \quad foldl(\lambda x, y.plus(x, y), map(F, L))$$

that first maps F onto a list and then adds the elements via the function *plus*. This simple implementation for g is inefficient, since the list must be traversed twice. The goal is now to find an equivalent function definition that is more efficient. We can specify this with higher-order terms in a syntactic fashion by one simple equation:

$$\lambda f, x, l.g(f, [x|l]) = \lambda f, x, l.B(f(x), g(f, l))$$

The variable B represents the body of the function to be computed and the first argument of B allows to use $f(x)$ in the body. The scheme on the right only allows recursing on l for g.

To solve this equation, we add a rule $X = X \to \mathit{true}$ as described in Section 6.1.1, and then apply narrowing, which yields the solution $\theta = \{B \mapsto \lambda fx, rec.plus(fx, rec)\}$ where

$$g(f, [x|l]) = \theta B(f(x), g(f, l)) = plus(f(x), g(f, l)).$$

This shows the more efficient definition of g. In this example, simplification can reduce the search space for narrowing drastically: it suffices to simplify the goal to

$$\lambda f, x, l.plus(f(x), foldl(plus, map(f, l))) = \lambda f, x, l.B(f(x), foldl(plus, map(f, l))),$$

where narrowing with the newly added rule $X = X \to \mathit{true}$ yields the two goals

$$\begin{aligned} \lambda f, x, l.plus(f(x), foldl(plus, map(f, l))) &\to^? \lambda f, x, l.X(f, x, l), \\ \lambda f, x, l.B(f(x), foldl(plus, map(f, l))) &\to^? \lambda f, x, l.X(f, x, l). \end{aligned}$$

These can be solved by pure higher-order unification. It should be noted that our notion of oriented goals requires an additional rule for equality. A clever implementation will hide such details from the user.

7.3 Higher-Order Functional-Logic Programming

Our approach to functional-logic programming is oriented towards functional languages. This is in contrast to most first-order approaches that often aim at extending Prolog by functions. Our goal is to extend a functional core language by logical variables as in Prolog. The core of functional languages such as SML [MTH90] or Haskell [HJW92] essentially is higher-order term rewriting, no matter if the language employs lazy or eager evaluation. Relational programming as in logic programming can be embedded as shown in Section 2.6.

We show by several examples that left-linear, normal conditional HRS suffice for programming and allow computing in Simple Systems. As we do not allow extra variables on the right-hand side of the conditions, local variables as in functional programming are created via let-constructs, as for instance shown in Section 2.6. For example, we show how the let-construct for pairs from Section 2.6 can be formulated by higher-order rewrite rules. This common notation for let can be defined by

$$\mathsf{let}\ pair(xs, ys) = X\ \mathsf{in}\ F(xs, ys) =^{def} let\ X\ in\ \lambda xs, ys.F(xs, ys)$$

Notice that in the tuned notation on the left, $pair(xs, ys)$ serves as a binder for xs and ys. The higher-order rewrite rule for this construct is

$$let\ pair(Xs, Ys)\ in\ \lambda xs, ys.F(xs, ys) \to F(Xs, Ys).$$

The idea behind this modeling is that in $let\ t\ in\ \lambda xs, ys.t'$, the term t is evaluated to a pair of the form $pair(Y, Z)$ and then the rewrite rule applies.

Several of the following examples assume an equality predicate = on natural numbers. There are two ways to formalize such a predicate: either simply by a rule $X = X \to \mathit{true}$, which goes beyond Simple Systems, or by encoding strict equality on numbers, as shown in Section 6.1.1. For instance, the rules

$$\begin{aligned} s(X) = s(Y) &\to X = Y \\ 0 = 0 &\to \mathit{true} \end{aligned}$$

suffice for the constructors s and 0 for natural numbers.

As we will see, strict equality suffices for most applications. The disadvantage of strict equality is that for instance $\lambda x.x = x$ is not provable. It is however possible to add a rule $X = X \to true$ for simplification only, as suggested in [Han94c].

Recall that we sometimes write p for a rule $p \to true$ or a goal $p \to^? true$. Furthermore, we use in the examples some common abbreviations, e.g. $1 = s(0)$ etc.

7.3.1 "Infinite" (Data-)Structures and Eager Evaluation

Infinite data structures are one of the nice features of lazy functional programming, e.g. [Tur86, HJW92]. For this reason, some functional-logic languages, e.g. [MNRA92], support non-terminating rules. We show in the following that such infinite structures can be modeled within functional-logic programming while retaining eager evaluation.

Consider the example of an infinite list of ones, defined by:

$$ones \to [1|ones]$$

This rule, together with lazy evaluation, can be used with rules such as:

$$\begin{aligned} first([X|R]) &\to X \\ rest([X|R]) &\to R \\ sum\_n(0, L) &\to [] \\ sum\_n(s(n), [X|R]) &\to X + sum\_n(n, R) \end{aligned}$$

Lazy evaluation yields for instance

$$sum\_n(4, ones) \longrightarrow 4$$

This model of lazy computation has the disadvantage that non-terminating rules, here $ones \to [1|ones]$, have to be used carefully to avoid divergence.

We can model such infinite structures with terminating rules in our setting. We simple reverse the rule generating infinite objects:

$$ones([1|R]) \to [1|ones(R)]$$

The technique for working with this definition is to imagine, given an object of appropriate size, how to compute the solution. Thus, terminating rules suffice and eager reduction is possible.

Using the above definition, we can state the query

$$sum\_n(4, ones(Y)) \to^? 4,$$

which has the desired solution

$$\{Y \mapsto [1, 1, 1, 1|Y'], \ldots\}.$$

Thus the term $ones(Y)$ represents an "infinite" list.

Lazy Needed Narrowing is particularly useful in this example, as it solves goals only when needed. In the above example, intermediate goals of the form $ones(Y) \to X$ are

simply delayed. Only if X is instantiated, the goal is simplified and possibly delayed again.

The above example only models functional programming, which aims at evaluating expressions to unique values. Compared to functional programming, this approach also models search as in logic programming. For instance, lists where each element is a one or a two are easy to model. This is not possible with the first functional approach, as it would require non-confluent rules.

A simple example for this scheme is computing ancestors:

$$\begin{array}{lcl}
father(mary) & \to & john \\
mother(john) & \to & amy \\
father(john) & \to & art \\
\\
prim_rel(father) & & \\
prim_rel(mother) & & \\
\\
anchestor_rel(R) & & \Leftarrow prim_rel(R) \\
anchestor_rel(comp(R_1, R_2)) & \to & anchestor_rel(R_2) \\
 & & \Leftarrow prim_rel(R_1) \\
comp(R_1, R_2)(X) & \to & R_2(R_1(X))
\end{array}$$

The function *comp* composes two functions. It is equally possible to write the rule for *anchestor_rel* as $anchestor\_rel(R_2(R_1)) \to \ldots$, which has the disadvantage of not being a pattern. With these rules, the query

$$anchestor\_rel(R), R(mary) \to^{?} amy$$

has the solution

$$\{R \mapsto comp(father, mother)\}.$$

This technique for modeling infinite functional structures will reappear in some of the following examples.

7.3.2 Functional Difference Lists

Difference lists are a standard technique [SS86] for implementing lists in logic programming such that appending two lists can be done in linear time. A difference list is a pair, where the first element is the actual list of interest and the second element is the tail of the first list, typically a free variable. For instance, to represent the list $[a, b, c]$ as a difference list, we use the pair $([a, b, c \mid R], R)$ for some variable R.

Concatenating two difference lists is done in functional-logic programming by the function

$$append((X\,,\,Y),(\,Y\,,R)) = (X\,,R)$$

and in plain logic programming by the corresponding predicate. A principal problem with this representation is that a concrete variable is used to represent the end of the list. Thus when copying a difference list, a "predicate" is needed to introduce a new variable at the end of the copied list, as e.g. shown in [Red94]. The drawback is that this takes linear time.

The functional equivalent is to abstract over this variable representing the rest of the list. Thus we use functions from lists to lists as "functional difference lists", i.e. $\lambda x.[a, b, c \mid x]$ instead of $[a, b, c \mid X]$. This idea was introduced by Hughes [Hug86] and compared to the logic approach by Burton [Bur89] and Reddy [Red94]. We believe that this representation is much clearer than using free variables, which must be "new" for each copy. For instance, appending two functions is straightforward by β-reduction:

$$append1(L, R) = \lambda x.L(R(x))$$

A nice result on this approach in higher-order logic programming was shown in [BR91]: a naive reverse function on lists can be linear.

To formulate these ideas in our framework with rules of base type, we model a functional list as $flist(\lambda x.[X \mid R(x)])$ with an additional constructor *flist*. Then the higher-order rewrite rule for append reads as

$$append\_f(flist(L), flist(R)) \rightarrow flist(\lambda x.L(R(x)))$$

7.3.3 A Simple Encryption Problem

This example deals with a simple method for authorization. Assume the following method for the authorization of a client and some server. Both parties share an encryption function f. To authorize, a client sends some name a and its encryption $f(a)$ to the server. This value $f(a)$ can be viewed as a "password".

For several authorization steps, the channel between the client and the server transmits a list of names and a list of the corresponding passwords. Since the channel is unsafe, the client and the server use the following method to change the password of a name after each use. For simplicity, we assume names are natural numbers. If the client uses the name n and its password $f(n)$, both parties compute a new encryption function f' from f by: $f'(n) = f(n+1)$ and $f'(n+1) = f(n)$. That is, two passwords are swapped.

In the following program, the function $encode(F, [X \mid Rest])$ computes a list of passwords from a list of names of the communications on the channel. It maps the encryption function F to the first element of a stream, and updates the encryption function for the rest of the list.

$$\begin{array}{lcl}
comp(F, G)(X) & \rightarrow & G(F(X)) \\
\\
swap(X, Y, Z) & \rightarrow & \mathit{if}\ Z = X\ \mathit{then}\ Y\ \mathit{else\ if}\ Z = Y\ \mathit{then}\ X\ \mathit{else}\ Z \\
\\
encode(F, [X \mid Rest]) & \rightarrow & [F(X) \mid encode(comp(swap(X, X+1), F), Rest)] \\
encode(F, []) & \rightarrow & [] \\
\\
\mathit{if\ true\ then}\ X\ \mathit{else}\ Y & \rightarrow & X \\
\mathit{if\ false\ then}\ X\ \mathit{else}\ Y & \rightarrow & Y
\end{array}$$

For instance, if the initial encoding is the identity function, we obtain:

$$encode(\lambda x.x, [1, 2, 2]) = [1, 1, 3]$$

Now we consider the following situation. Some spy on the channel does not know the initial encryption function, but the method for the update. Now if the spy observes some

communication, his goal is to infer passwords. For instance, assume the spy observes the names $[1,2,2]$ and the corresponding passwords $[a,a,b]$. Then if 3 is sent as the fourth name, we can compute the fourth password with the goal

$$\lambda f.encode(f,[1,2,2,3]) \rightarrow^? \lambda f.F(f)$$

with solution

$$\theta = \{F \mapsto \lambda f.[f(1),f(1),f(3),f(3)]\}.$$

Clearly $f(1) = a$ and $f(3) = b$ and the spy can infer the fourth password.

The encryption in this example is rather simple. It is clearly possible to model more complicated authorization strategies with this approach.

7.3.4 Eight-Queens Generalized

In the following example we model techniques of object-oriented programming by higher-order functions. As this is done in a functional-logic setting, this is a sketch for integrating object-oriented programming and logic programming.

For modeling functional object-oriented programming (see e.g. [Red88, Wan87]), objects are represented by records, which we adopt here in a very simple fashion. In a more advanced representation, as pursued in [Gro94], an object is a function that essentially consists of a case-statement dispatching the incoming messages.

In a functional setting of object-oriented programming, objects have no internal, mutable state. Furthermore, we do not address other important issues of object-oriented programming, such as inheritance.

We extend the classical eight-queens problem in the following, straightforward way: not only queens but arbitrary (chess) pieces are considered. We view chess pieces as objects consisting of a position and a function that determines if the piece attacks another position. These represent the "instance variables" and the "methods" of an object, to speak in object-oriented terminology. For a "message call" we simply select the appropriate function from the object and apply it. Alternative versions that are closer to object-oriented programming would require a more tuned syntax, which we avoid for simplicity.

Positions on a chess board are represented as pairs, e.g. $pair(4,5)$ represents column 4, row 5. We assume a function *next* to compute the next position on the chess board and an extended let-construct. Furthermore, a general "attacking function" for each kind of piece is assumed to take two positions and determines if the piece placed on the first position attacks the second.

Pieces are created by the constructor *piece*, i.e. $o = piece(F, Pos)$, where Pos is a position and F is a function, such that $F(pos1)$ determines if the piece attacks some position $pos1$. We assume the following destructors: *get_pos*, returning the position, and *get_attack_fun* which returns the attacking function of a piece. For instance, $get\_pos(o) = Pos$.

The main function in the program in Figure 7.2 is $position(Lf, Lp, Pos)$, taking a list of attacking functions Lf for pieces to be placed, a list Lp of already placed pieces and a position Pos. The list Lf characterizes the pieces to be placed, as shown below. The function $no\_attacksL(F, Lo)$, determines if the piece F attacks some piece in the list Lo. Notice that partial application serves in the expression $piece(F(Npos), Npos)$ to turn a

$$
\begin{array}{lcl}
get\_pos(piece(F, Pos)) & \rightarrow & Pos \\
get\_attack\_fun(piece(F, Pos)) & \rightarrow & F \\
 & & \\
no\_attacksL(O, []) & \rightarrow & true \\
no\_attacksL(O, [O1|R]) & \rightarrow & if\ (get\_attack\_fun(F))(get\_pos(O1))\ then \\
 & & \quad false\ else\ no\_attacksL(O, R) \\
 & & \\
position([], L, pair(X, Y)) & \rightarrow & L \\
position([F|R], L, Pos) & \rightarrow & \mathsf{let}\ Npos = next(Pos) \\
 & & \quad\ \ Nobj = piece(F(Pos), Pos) \\
 & & \mathsf{in}\ if\ no\_attacksL(Nobj, L)\ then \\
 & & \quad position(R, [Nobj|L], Npos) \\
 & & \quad else\ position([F|R], L, Npos)
\end{array}
$$

Figure 7.2: Rules for the Eight-Queens Problem

general attacking function into the attack function for a new piece with fixed position.

In order to apply the above rules, we write for instance the functions *queen_attacks* and *knight_attacks* as below. Both functions take two positions and check the appropriate attacking.

$$
\begin{array}{lcl}
queen\_attacks(pair(X, Y), pair(Sx, Sy)) & \rightarrow & X = Sx \vee Y = Sy \vee \\
 & & |X - Sx| = |Y - Sy| \\
knight\_attacks(pair(X, Y), pair(Sx, Sy)) & \rightarrow & (|X - Sx| = 1 \wedge |Y - Sy| = 2) \vee \\
 & & (|X - Sx| = 2 \wedge |Y - Sy| = 1)
\end{array}
$$

In order to position one knight and two queens on some board, the query

$$position([queen\_attacks, queen\_attacks, knight\_attacks], [], pair(1, 1))$$

suffices, assuming that $pair(1, 1)$ is the initial position.

Furthermore, the power of higher-order unification permits other queries: given a set of positions on the chess board, which pieces can be placed on these positions such that they do not attack each other.

The general idea of this example is that higher-order functions are used to represent "heterogeneous" information, e.g. arbitrary chess pieces. For the pure eight queens problem, it is easy to devise special data-structures (e.g. [Bra90]), in contrast to our generalized version. Furthermore, the object-oriented version is also easier to extend, e.g. by other (chess) pieces.

7.4 Higher-Order Abstract Syntax: Type Inference

In this section, we consider the problem of polymorphic type reconstruction as it occurs in functional languages, see for instance [CDDK86, NP99]. For simplicity, we only consider the core constructs of such a language, i.e. typed λ-calculus. The syntax of the language includes atoms $const(x)$, application $app(t, t')$, and abstraction $abs(\lambda x.t)$.

The set of polymorphic types is generated by some base types, type variables, and the function type constructor $->$, written in infix notation as $\sigma \; -> \; \tau$. We chose the symbol $->$ instead of the common $\rightarrow$, in order to avoid confusion with term rewriting. For instance, a term $succ(5)$, where $succ$ is a function on integers, is represented as

$$app(const(succ), const(5)).$$

As usual for type inference systems, we store the type of atoms in a context E. For instance, compared to the typing rules of simply typed λ-calculus in Section 3.2, the judgment $x : \tau$ in the rule

$$\frac{x : \tau \quad s : \tau'}{(\lambda x.t) : (\tau \; -> \; \tau')}$$

is represented in a context.

In violation of our conventions, we write free variables over types by Greek letters σ and τ as usual for type inference systems. The standard rules for type inference can easily be expressed as conditional equations:

$$\begin{array}{lcl}
update(E, T, X, Y) & \rightarrow & if \;\; Y = X \;\; then \;\; T \;\; else \;\; E(Y) \\
 & & \\
type_of(E, const(X)) & \rightarrow & E(X) \\
type_of(E, app(T, T')) & \rightarrow & type(\tau) \\
 & & \Leftarrow \;\; type_of(E, T) \rightarrow type(\sigma \; -> \; \tau), \\
 & & \quad\;\; type_of(E, T') \rightarrow type(\sigma) \\
type_of(E, abs(\lambda x.T(x))) & \rightarrow & type(\sigma \; -> \; \tau) \\
 & & \Leftarrow \;\; \lambda v.type_of(update(E, type(\sigma), v), T(const(v)) \\
 & & \quad\;\; \rightarrow \lambda v.type(\tau)
\end{array}$$

The function $update(E, T, X)$ creates a new context where X has the type T. Notice that the last two rules have the extra free variables σ and τ that do not occur on the left-hand side.

In the last rule for typing an abstraction, typically for higher-order abstract syntax, a local constant v serves to explore the type of $\lambda x.T(x)$. The local binder for v corresponds to a $\forall$-quantifier, i.e. a goal $\lambda v.t \rightarrow^? \lambda v.s$ is equivalent to $\forall v.s \rightarrow^? t$.

For example, a term $f(\lambda x.plus(x, y))$, where f and y are polymorphic atoms, is represented as

$$t = app(const(f), abs(\lambda x.app(app(const(plus), x), const(y)))).$$

Type inference for t is done by the following query

$$type\_of(E, t) \rightarrow^? type(\beta),$$

where $E(f) = \alpha \; -> \; \alpha$, $E(y) = \gamma$, and $E(plus) = int \; -> \; int \; -> \; int$. This goal has the solution

$$\{\beta \mapsto int \; -> \; int, \alpha \mapsto int \; -> \; int, \gamma \mapsto int, \ldots\}.$$

In the above rules, the extra variables are purposely introduced to compute "local" types. For instance, the variable σ in the abstraction rule only serves for computing the type of the subterm T. Thus rewriting requires computing solutions to for these variables in the conditions. This can be done by narrowing.

Furthermore, this example requires full unification, although it is not immediate: in the conditions of the third rule, the variable σ occurs on both right-hand sides. Simple systems cannot express this problem, as they do not need full unification (e.g. no occurs check). Hence this example requires the general completeness result in Section 6.8. Thus if a term has a type, then Conditional Lazy Narrowing will compute it.

Unfortunately, current methods for proving convergence (see Section 4.3) do not suffice for this example. Confluence is a delicate matter if extra variables exist in the conditions. Confluence in this example would entail a desirable property of type inference: unique, most general types exist in this case.

It is interesting to compare this formulation with the similar specification in λ-Prolog [PE88, MP92b]. In λ-Prolog, a predicate *type_rel* replaces *type_of* and defines defines a relation between a term and a type. Thus it is not possible to speak directly about most general types.

Chapter 8

Concluding Remarks

This work was led by the idea that higher-order equations can be used in practical systems for equational reasoning and functional-logic programming. Towards this goal we first examined decidable classes of higher-order unification. We have shown that for many practical purposes, higher-order unification is not only a powerful tool, but also terminates for several classes of terms. The main restriction needed is linearity, which is common for programming. It also explains to some degree that higher-order unification in logic programming [NM88] and higher-order theorem proving [Pau94, AINP90] rarely diverges.

Secondly, we have developed a first framework for solving higher-order equations by narrowing. We have seen that some approaches such as plain narrowing are not suitable for the higher-order case. For lazy narrowing, in contrast, we were able to develop many important refinements, such as normalization and eager variable elimination for normalized solutions. Of similar practical importance are the extensions to conditional equations.

The work on left-linear rewrite systems for programming applications led to Simple Systems, which is an important class of goals for equational programming. This class enjoys many useful properties, for instance solved forms are easy to detect. Furthermore, in the second-order case, unification remains decidable for Simple Systems. Simple Systems are a large class of goals where the occurs check is not needed. Interestingly, in most implementations of Prolog, the occurs check is missing for computational reasons. This suggests to define languages where the occurs check is redundant. More importantly, it indicates that the problems solvable with Prolog implementations correspond to such a class of problems. The main result for Simple Systems is the strategy of Needed Lazy Narrowing, where intermediate goals can be identified and can safely be delayed, which leads to a needed computation strategy.

Altogether, we believe that the results for normalized solutions and Simple Systems are a major step towards high-level programming languages, where efficiency and a simple operational model are significant. This leads to a novel approach to functional-logic programming that is oriented towards higher-order functional languages. Whereas most other approaches aim at extending logic programming by functions, the main idea here is to extend a higher-order functional language by logic or free variables as in Prolog. This approach facilitates several operational optimizations that are not possible in other approaches oriented towards extending logic programming.

Another observation is that oriented goals turned out to be a particularly useful

restriction on equational goals. Oriented goals do not limit the expressiveness, i.e. full unification can be encoded, and simplify the technical treatment. Furthermore, we have seen that for left-linear HRS, there is a difference between matching, as performed by oriented goals, and full unification. In the former, Simple Systems suffice for narrowing and the decidability of second-order unification is maintained.

This work also contributes to equational reasoning in higher-order theorem provers. For this, we have provided complete calculi for higher-order narrowing for unrestricted equations. Although for most of the optimizations we considered, some restrictions were needed, they apply to general theorem proving as well. As another application, Section 5 can be the basis for further investigation into the decidability of second-order R-matching problems. For instance, Curien [Cur93] presents first results on second-order E-matching for first-order E.

8.1 Related Work

Recently, there have been several works on higher-order narrowing, but none covering the full higher-order case. Qian [Qia94] lifted the completeness of first-order narrowing strategies to higher-order patterns for first-order rules. Higher-order patterns are an important subclass of λ-terms, which include bound variables, but behave almost as first-order terms in most respects. However, patterns are often too restrictive, as obvious from the examples in Sections 2.5 and 7 (see also [Pre94b, MP92a]). In particular, they do not suffice for modeling higher-order functional programs. Examples are the function *map* in Section 2.4.1 and the definition of the let-construct by a higher-order rewrite rule

$$\mathsf{let}\ X\ \mathsf{in}\ T \to T(X)$$

where the right-hand side is no pattern. Thus rewriting or narrowing with this rule may introduce non-pattern terms. For a discussion on this issue in a logic-programming context see [MP93].

The approach to higher-order narrowing in [LS93, ALS94a] aims at narrowing with higher-order functional programs and does not limit rules to higher-order patterns. Rules with pattern left-hand sides are used for narrowing on quasi-first-order terms. (These are slightly more general than quasi first-order terms defined here.) This guarantees that the resulting term is still quasi-first-order. Although this seems to be an interesting compromise, it has strong restrictions: higher-order variables in the left-hand sides of rules may occur only directly below the outermost symbol. For instance, the function $map(F, cons(X, Y)) = \ldots$, fulfills this requirement only if X and Y are first-order. Roughly speaking, when narrowing with such a rule, narrowing and rewriting coincide for these higher-order variables as they occur only at depth one on the left-hand side.[1]

Higher-order logic programming [NM88] has two major extensions of first-order logic programming: first, higher-order terms are used and, secondly, hereditary Harrop formulas, which generalize horn clauses. The latter, roughly speaking, allow for "local" rules in the contexts. In contrast to the first, this nice extension cannot be modeled directly by (conditional) narrowing.

[1]This result is a bit more subtle, since the rewrite rules are lifted for narrowing. Lifting turns a first-order term into a higher-order term. This problem is however not addressed in [LS93, ALS94a].

Compared to higher-order logic programming, the functional approach with Simple Systems lies between λ-Prolog [NM88], where full higher-order terms are used, and Elf [Pfe91],[2] where non-patterns are just delayed as constraints. The main advantage of this functional approach is that a decidable class of second-order unification instead of pattern unification can be used. The problem is that higher-order patterns cover large classes of terms occurring in practice, but are not sufficient in general. Thus strategies have been developed in [MP93] to handle such cases effectively.

Another difference to (higher-order) logic programming is that predicates and terms are not separated. Higher-order λ-terms are used for data structures and do not permit higher-order programming as in functional languages. For instance, the function *map* in Section 2.4.1 cannot be written directly by higher-order logic programming. Nadathur [Nad87] reports similar problems with variables over predicates for modeling *map* in higher-order logic programming. In this respect, our approach is more general and allows for an arbitrary integration of data and functions. Notice that in most Prolog implementations a quasi second-order predicate "apply" exists, that applies a variable function symbol to some arguments. Use of this built-in predicate obviously destroys completeness in the usual sense.

The many higher-order extensions of functional-logic and logic languages [BG86, CKW89, GMHGRA92, Loc93, She90] are, to our knowledge, limited to first-order unification and are not complete in a higher-order sense. For instance, the work in [Loc93] uses higher-order variables, but only (first-order) narrowing on first-order terms plus β-reduction as the operational model. Since higher-order rules such as *map* are used, higher-order terms can be created which cannot be handled by this approach. The work in [GMHGRA92] on SFL, an extension of the language BABEL [MNRA92], similarly permits higher-order variables. Completeness of narrowing with first-order unification is claimed w.r.t. particular denotational and operational semantics for partial objects.

8.2 Open Problems and Further Work

Simple Systems can also be employed to improve existing first-order languages. For this purpose, it is desirable to extend Simple System to conditional narrowing with extra variables on the right side of the conditions. This is easily possible with linearity restrictions on the right sides of the conditions. For a conditional rule

$$l \to r \Leftarrow \overline{l_n \to r_n}$$

it is required that

$$X \to^? l, \overline{l_n \to^? r_n}$$

is a Simple System for some new variable X. This holds if no variable occurs more than once in $r_1, \ldots, r_n$ and l (and in addition all r_n are patterns). This is not overly strict, as exemplified in Section 2.6, since variables on the right side of conditions are usually used for local variables.

Developing efficient implementations for higher-order programming is another essential step. Projects in this direction are an abstract machine for higher-order logic programming [NJW93] and a compiler for higher-order logic programming [BR92].

[2] It should be noted that Elf has a much more expressive type system.

More expressive type systems, such as polymorphism and type classes in current functional languages [NP99], have not been considered here. An extension to polymorphism faces the problem that higher-order unification with polymorphism is infinitely branching [Nip91b]. In practice, such cases are rare as experience with the Isabelle system [Pau94] shows.

An interesting application is to model calculi for distributed systems, e.g. the π-calculus [MPW92a, MPW92b], by higher-order rewriting, thus obtaining an executable version. The major obstacle for this approach is that most of the rules in this calculus apply modulo associativity (A) and commutativity (C). Rewriting modulo AC has been extensively studied in the first-order case, for the higher-order case there exist first results on AC-unification [QW94, MW94].

Bibliography

[AEH94] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.

[AINP90] Peter B. Andrews, Sunil Issar, Dan Nesmith, and Frank Pfenning. The TPS theorem proving system. In M.E. Stickel, editor, *Proc. 10th Int. Conf. Automated Deduction*, pages 641–642. LNCS 449, 1990.

[ALS94a] J. Avenhaus and C. A. Loría-Sáenz. Higher-order conditional rewriting and narrowing. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, München, Germany, 7–9 September 1994. Springer-Verlag.

[ALS94b] J. Avenhaus and C. A. Loría-Sáenz. On conditional rewrite systems with extra variables and deterministic logic programs. In *LPAR '94*, Lecture Notes in Computer Science, vol. 822, Kiev, Ukraine, July 1994. Springer-Verlag.

[Bac91] Leo Bachmair. *Canonical Equational Proofs.* Progress in Theoretical Computer Science. Birkhäuser, 1991.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North Holland, 2nd edition, 1984.

[Bax76] L. D. Baxter. *The complexity of Unification.* PhD thesis, University of Waterloo, Waterloo, Canada, 1976.

[BG86] P. G. Bosco and E. Giovannetti. IDEAL: An ideal deductive applicative language. In Gary Lindstrom and Robert M. Keller, editors, *Symposium On Logic Programming*, pages 89–96. IEEE, 1986.

[BG89] Hubert Bertling and Harald Ganzinger. Completion-time optimization of rewrite-time goal solving. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 45–58. Springer LNCS 355, 1989.

[BGM88] P. G. Bosco, E. Giovanetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[BKW93] A. Bockmayr, S. Krischer, and A. Werner. An optimal narrowing strategy for general canonical systems. In M. Rusinowitch and J. L. Remy, editors, *Conditional Term Rewriting Systems: Proc. of the Third Internati onal Workshop (CTRS-92)*, pages 483–497. Springer-Verlag, Berlin, Heidelberg, 1993.

[BR91] Pascal Brisset and Olivier Ridoux. Naïve reverse can be linear. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 857–870, Paris, France, 1991. The MIT Press.

[BR92] P. Brisset and O. Ridoux. The architecture of an implementation of lambda-prolog: Prolog/mali. In *Proc. Workshop on LambdaProlog, Philadelphia*, 1992. PA, USA.

[Bra75] D. Brand. Proving theorems with the modification method. *SIAM Journal of Computing*, 4:412–430, 1975.

[Bra90] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Wokingham, 2nd edition, 1990.

[Bro88] M. Broy. Equational specification of partial higher order algebras. *Theoretical Computer Science*, 57:3–45, 1988. Also in: Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 36, 1987.

[BS94] F. Baader and J. Siekmann. Unification theory. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, 1994.

[BT87] J. A. Bergstra and J. V. Tucker. Algebraic specifications of computable and semicomputable data types. *Theoretical Computer Science*, 50:137–181, 1987.

[Bur89] F. W. Burton. A note on higher-order functions versus logical variables. *Information Processing Letters*, 31:91–95, 1989.

[BvEG+87] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE: Parallel Architectures and Languages Europe (Volume 2: Parallel Languages)*, pages 141–158. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 259.

[CAB+86] Robert Constable, S. Allen, H. Bromly, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics With the Nuprl Proof Development System*. Prentice-Hall, New Jersey, 1986.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.

[CF91] P. H. Cheong and L. Fribourg. Efficient integration of simplification into prolog. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91, Passau, Germany*, number 528 in Lecture Notes in Computer Science, pages 359–370. Springer Verlag, August 1991.

[CKW89] W. Chen, M. Kifer, and D. S. Warren. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114, Cleveland, Ohio, USA, 1989.

[CM84] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, 2nd edition, 1984.

[Cur93] Régis Curien. Second-order E-matching as a tool for automated theorem proving. In *EPIA '93.* Springer LNCS 725, 1993.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[DFH+93] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide version 5.8. Technical Report 154, INRIA, May 1993.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. Elsevier, 1990.

[DL86] Doug DeGroot and Gary Lindstrom. *Logic Programming Functions, relations, and Equations.* Prentice-Hall, 1986.

[DM79] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[DMS92] N. Dershowitz, S. Mitra, and G. Sivakumar. Decidable matching for convergent systems (preliminary version). In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, LNAI 607, pages 589–602, Saratoga Springs, New York, USA, June 15–18, 1992. Springer-Verlag.

[DO90] N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75(1):111–138, 1990.

[DOS88] N. Dershowitz, M. Okada, and G. Sivakumar. Canonical conditional rewrite systems. In *Proc. of the 9th International Conference on Automated Deduction.* Springer LNCS 310, 1988.

[Dow92] Gilles Dowek. Third order matching is decidable. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 2–10, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press.

[Dow93] Gilles Dowek. Personal communication. 1993.

[DW88] Michael R. Donat and Lincoln A. Wallen. Learning and applying generalised solutions using higher order resolution. In E. Lusk and R. Overbeek, editors, *9th International Conference On Automated Deduction*, pages 41–61. Springer-Verlag, 1988.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.

[Far88] W. M. Farmer. A unification algorithm for second-order monadic terms. *Annals of Pure and Applied Logic*, 39:131–174, 1988.

[Far91] W. M. Farmer. Simple second-order languages for which unification is undecideable. *Theoretical Computer Science*, 87:25–41, 1991.

[Fay79] M. Fay. First order unification in equational theories. In *Proc. 4th Conf. on Automated Deduction*, pages 161–167. Academic Press, 1979.

[Fel92] Amy Felty. A logic-programming approach to implementing higher-order term rewriting. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Extensions of Logic Programming, Proc. 2nd Int. Worksho p*, pages 135–158. LNCS 596, 1992.

[FH86] F. Fages and Gérard Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43:189–200, 1986.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, Wokingham, 1988.

[FH91] Ulrich Fraus and Heinrich Hußmann. A narrowing-based theorem prover. In *Rewriting Techniques an Applications*, pages 435–436. LNCS 488, April 1991.

[Fri85] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Symposium on Logic Programming* [IEE85], pages 172–184.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[GMHGRA92] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. On the completeness of narrowing as the operational semantics of functional logic programming. In E. Börger, G. Jäger, H. Kleine Büning, S. Martini, and M.M. Richter, editors, *Computer Science Logic. Selected papers from CSL'92*, LNCS, pages 216–231, San Miniato, Italy, September 1992. Springer-Verlag.

[Gol81] W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[Gor88] Michael J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle et al., editor, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Press, 1988.

[Gou66] W. E. Gould. A matching procedure for ω-order logic. Scientific Report 4, Air Force Cambridge Research Laboratories, 1966.

[Gro94] R. Grosu. *A Formal Foundation for Concurrent Object Oriented Programming*. PhD thesis, Institut für Informatik, TU München, Arcisstr. 21, D–80290 München, Germany, November 1994.

[GTW89] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T. Yeh, editor, *Current trends in programming methodology*, volume 3, Data structuring, pages 80–149. Prentice-Hall, 1989.

[Hag91a] Masami Hagiya. From programming-by-example to proving-by-example. In *International Conference on Theoretical Aspects of Computer Software*, pages 387–419, 1991.

[Hag91b] Masami Hagiya. Synthesis of rewrite programs by higher-order and semantic unification. *New Generation Computing*, 8, 1991.

[Han91] M. Hanus. Efficient implementation of narrowing and rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pages 344–365. Springer LNAI 567, 1991.

[Han92] M. Hanus. Improving control of logic programs by using functional logic languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 1–23. Springer LNCS 631, 1992.

[Han94a] M. Hanus. Combining lazy narrowing and simplification. In *Proc. 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 370–384. Springer LNCS 844, 1994.

[Han94b] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[Han94c] M. Hanus. Lazy unification with simplification. In *Proc. 5th European Symposium on Programming*, pages 272–286. Springer LNCS 788, 1994.

[Har90] Masateru Harao. Analogical reasoning based on higher-order unification. In S. Arikawa, S. Goto, S. Ohsuga, and T . Yokomori, editors, *Algorithmic Learning Theory*, pages 151–163. Springer-Verlag, 1990.

[HJW92] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HL91] Gérard Huet and Jean-Jacques Lévy. Computations in orthogonal rewriting systems, I. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–414. MIT Press, Cambridge, MA, 1991.

[HM88] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In *Fifth International Logic Programming Conference*, pages 942–959, Seattle, Washington, August 1988. MIT Press.

[Höl88] Steffen Hölldobler. From paramodulation to narrowing. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 327–342, Seatle, 1988. ALP, IEEE, The MIT Press.

[Höl89] Steffen Hölldobler. *Foundations of Equational Logic Programming.* LNCS 353, 1989.

[HS86] J.R. Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[Hsi85] J. Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, 25:255–300, 1985.

[Hue73] Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22:257–267, 1973.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hue76] Gérard Huet. *Résolution d'équations dans les languages d'ordre 1,2,...ω.* PhD thesis, University Paris-7, 1976.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27:797–821, 1980.

[Hug86] R. J. M. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, pages 141–144, 1986.

[Hul80] Jean-Marie Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings of 5th Conference on Automated Deduction*, pages 318–334. Springer-Verlag, LNCS 87, 1980.

[Huß93] H. Hußmann. *Nondeterminism in Algebraic Specifications and Algebraic Programs.* Birkhäuser, 1993.

[IEE85] IEEE Computer Society, Technical Committee on Computer Languages. *Symposium on Logic Programming*. The Computer Society Press, July 1985.

[JK86] Jean-Pierre Jouannaud and Claude Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.

[JK91] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 257–321. MIT Press, 1991.

[JP76] D. Jensen and T. Pietrzykowski. Mechanizing ω-order type theory through unification. *Theoretical Computer Science*, 3, 1976.

[KB70] Donald E. Knuth and P.B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[Klo80] Jan Willem Klop. *Combinatory Reduction Systems*. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

[Klo92] Jan Willem Klop. Term rewriting systems. In Samson Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 2–116. Oxford University Press, 1992.

[LLFRA93] Rita Loogen, Francisco López-Fraguas, and Mario Rodríguez-Artalejo. A demand driven strategy for lazy narrowing. In *PLILP*, LNCS, Tallin, Estonia, 1993. Springer-Verlag.

[Llo87] John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[Llo94] John Wylie Lloyd. Combining functional and logic programming languages. In *ILPS 1994*, 1994. To appear.

[Loc93] Hendrik C.R Lock. *The Implementation of Functional Logic Languages*. Oldenbourg Verlag, 1993.

[LS93] C. A. Loría-Sáenz. *A Theoretical Framework for Reasoning about Program Construction Based on Extensions of Rewrite Systems*. PhD thesis, Univ. Kaiserslautern, December 1993.

[Luc72] C. L. Lucchesi. The undecidability of the unification problem for third order languages. Technical Report CSRR 2059, University of Waterloo, Waterloo, Canada, 1972.

[Mak77] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik*, 32:129–198, 1977.

[MH94] A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *J. of Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994. Short version appeared at ALP '92.

[Mil91a] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1:497–536, 1991.

[Mil91b] Dale Miller. Unification of simply typed lambda-terms as logic programming. In P.K. Furukawa, editor, *Proc. 1991 Joint Int. Conf. Logic Programming*, pages 253–281. MIT Press, 1991.

[MN94] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. Technical report, Institut für Informatik, TU München, 1994.

[MNRA92] Juan Jose Moreno-Navarro and Mario Rodriguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *The Journal of Logic Programming*, 12(1, 2, 3 and 4):191–223, 1992.

[Möl86] B. Möller. Algebraic specifications with higher-order operators. In *IFIP TC2 Working Conference on Program Specification and Transformation, Bad Tölz*, pages 367–392. North-Holland, 1986.

[MP92a] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In Dale Miller, editor, *Proceedings of the Workshop on the Lambda Prolog Programming Language*, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania.

[MP92b] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Extensions of Logic Programming, Proc. 2nd Int. Workshop*, pages 299–344. LNCS 596, 1992.

[MP93] Spiro Michaylov and Frank Pfenning. Higher-order logic programming as constraint logic programming. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 221–229, Newport, Rhode Island, April 1993. Brown University.

[MPW92a] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I. *Information and Computation*, 100(1):1–40, 1992.

[MPW92b] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part II. *Information and Computation*, 100(1):41–77, 1992.

[MRM89] A. Martelli, G. F. Rossi, and C. Moiso. Lazy unification algorithms for canonical rewrite systems. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Vol. 2, Rewriting Techniques.* Academic Press, 1989.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MW94] Olaf Müller and Franz Weber. Theory and praxis of minimal modular higher-order E-unification. In *Automated Deduction — CADE-12*. Springer LNAI 814, 1994.

[Nad87] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming*. PhD thesis, University of Pennsylvania, Philadelphia, 1987.

[Nar89] P. Narendran. Some remarks on second order unification. Technical report, Institute of Programming and Logics, Dep. of Computer Science, State Univ. of New York at Albany, 1989.

[Nip91a] Tobias Nipkow. Higher-order critical pairs. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 342–349, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.

[Nip91b] Tobias Nipkow. Higher-order unification, polymorphism, and subsorts. In S. Kaplan and M. Okada, editors, *Proc. 2nd Int. Workshop Conditional and Typed Rewriting Systems*, volume 516 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1991.

[Nip93a] Tobias Nipkow. Functional unification of higher-order patterns. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74, Montreal, Canada, 19–23 June 1993. IEEE Computer Society Press.

[Nip93b] Tobias Nipkow. Orthogonal higher-order rewrite systems are confluent. In M.A. Bezem and Jan Friso Groote, editors, *Proc. Int. Conf. Typed Lambda Calculi and Applications*, pages 306–317. LNCS 664, 1993.

[NJW93] Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Duke University, 1993.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λ-Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proc. 5th Int. Logic Programming Conference*, pages 810–827. MIT Press, 1988.

[NP99] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *J. Functional Programming*, 199? To appear. Short version appeared in POPL '93.

[Oos94] Vincent van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, 1994. Amsterdam.

[OR94] Vincent van Oostrom and Femke van Raamsdonk. Weak orthogonality implies confluence: the higher-order case. In A. Nerode, editor, *Logical Foundations of Computer Science*, volume 813 of *Lect. Notes in Comp. Sci.*, pages 379–392. Springer-Verlag, 1994.

[Pad94] Vincent Padovani. Personal communication. 1994.

[Pau86] Lawrence C. Paulson. Natural deduction proof as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proc. SIGPLAN '88 Symp. Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM-Press.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[Pie73] T. Pietrzykowski. A complete mechanization of second-order type theory. *J. of ACM*, 20:333–364, 1973.

[PM90] Remo Pareschi and Dale Miller. Extending definite clause grammars with scoping constructs. In David H. D. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 373–389. MIT Press, June 1990.

[Pol94] Jaco van de Pol. Termination proofs for higher-order rewrite systems. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic and Term Rewriting*, volume 816 of *Lect. Notes in Comp. Sci.*, pages 305–325. Springer-Verlag, 1994.

[Pre94a] Christian Prehofer. Decidable higher-order unification problems. In *Automated Deduction — CADE-12*, LNAI 814. Springer-Verlag, 1994.

[Pre94b] Christian Prehofer. Higher-order narrowing. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 507–516. IEEE Computer Society Press, 1994.

[Pre94c] Christian Prehofer. On modularity in term rewriting and narrowing. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, München, Germany, 7–9 September 1994. Springer-Verlag.

[Qia93] Zhenyu Qian. Linear unification of higher-order patterns. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 391–405, Orsay, France, April 1993. Springer-Verlag LNCS 668.

[Qia94] Zhenyu Qian. Higher-order equational logic programming. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, Portland, 1994.

[QW94] Zhenyu Qian and Kang Wang. Modular AC unification of higher-order patterns. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, München, Germany, 7–9 September 1994. Springer-Verlag.

[Red85] U. S. Reddy. Narrowing as the operational semantics of functional languages. In *Symposium on Logic Programming* [IEE85], pages 138–151.

[Red86] U. S. Reddy. On the relationship between logic and functional languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 3–36. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Red88] U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 289–297, July 1988.

[Red94] U. S. Reddy. Higher-order aspects of logic programming. In *ICLP'94*. MIT Press, Cambridge, MA, 1994. To appear.

[RW69] G. A. Robinson and L. T. Wos. Paramodulation and theorem proving in first order theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 133–150. American Elsevier, 1969.

[SD94] D. De Schreyeand and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19:199–260, 1994.

[SG89] Wayne Snyder and Jean Gallier. Higher-order unification revisited: Complete sets of transformations. *J. Symbolic Computation*, 8:101–140, 1989.

[She90] Yeh-Heng Sheng. HIFUNLOG: Logic programming with higher-order relational functions. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 529–545, Jerusalem, 1990. The MIT Press.

[SJ92] F. S. K. Silbermann and B. Jayaraman. A domain-theoretic approach to functional and logic programming. *Journal of Functional Programming*, 2(3):273–321, 1992.

[Sla74] J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 1974.

[Smo86] Gert Smolka. Fresh: A higher-order language based on unification. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming Functions,relations,and Equations*, pages 469–525. Prentice-Hall, 1986.

[Sny90] Wayne Snyder. Higher order E-unification. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 573–587, Berlin, Heidelberg, 1990. Springer LNAI 449.

[Sny91] Wayne Snyder. *A Proof Theory for General Unification.* Birkäuser, Boston, 1991.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.

[Ste90] G. L. Steele. *Common LISP: The Language (Second Edition).* Digital Press, Burlington, MA, 1990.

[Tho90] W. Thomas. Automata on infinite objects. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, pages 134–191. Elsevier, 1990.

[Tur86] D. Turner. An overview of miranda. *Sigplan Notices*, 21(12):158–160, 1986.

[vR93] Femke van Raamsdonk. Confluence and superdevelopments. In *Rewriting Techniques an Applications*, pages 168–182. LNCS 690, June 1993.

[Wad71] Christopher Peter Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* Phd thesis, University of Oxford, Oxford, September 1971.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proceedings, Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, 22–25 June 1987. The Computer Society of the IEEE.

[Wol93] D. A. Wolfram. *The Clausal Theory of Types.* Cambridge Tracts in Theoretical Computer Science 21. Cambridge University Press, 1993.

[Wol94] D. A. Wolfram. A semantics for λProlog. *Theoretical Computer Science*, 136(1), 1994.

Index