OBJECT-LEVEL SUBSTITUTION, UNIFICATION AND GENERALIZATION IN META-LOGIC

CHUCK C. LIANG

A DISSERTATION

in

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

1995

Dale Miller

Supervisor of Dissertation

Peter Buneman

Graduate Group Chairperson

ABSTRACT

OBJECT-LEVEL SUBSTITUTION, UNIFICATION AND GENERALIZATION IN META-LOGIC

CHUCK C. LIANG
Supervisor: DALE MILLER

Meta-programming in logic has had difficulty with a class of problems, which includes polymorphic type inferencing and Knuth-Bendix completion. At the core of this difficulty is that logic programming, in particular higher-order logic programming, is often too dependent on its inherent unification and resolution algorithms in providing declarative formulations of object-level theories. To free meta-programming in logic from this dependency, we provide logic programming formulations of substitution and unification on object-level expressions. This will provide a set of basic tools for meta-programming in a higher-order logic programming language, which supports the use of higher-order abstract syntax. In particular, the techniques presented here can be used in a very simple meta-language, the L_λ restriction of λProlog.

Acknowledgments

I would like to thank Dale Miller foremost for his patience and guidance during the years I've been a graduate student. I would also like to thank all members of my thesis committee for helpful comments and suggestions. Special thanks goes to Sandip Biswas and Jawahar Chirimar for invaluable conversation regarding topics of this thesis. This research is supported in part by the following grants: ONR N00014-93-1-1324, NSF CCR-94-00907, NSF CCR-92-09224, and DARPA N00014-85-K-0018.

Contents

Acknowledgments iii

1 Introduction 1

1.1 Overview . . . 3

1.2 Notations, Conventions, and Prerequisites . . . 3

1.3 Choice of Meta-Languages . . . 5

1.3.1 Hereditary Harrop Formulas and Uniform Provability . . . 6

1.3.2 The L_λ Restriction of λProlog . . . 8

1.4 Representing Object-Level Syntax . . . 10

1.4.1 Higher-Order Abstract Syntax For Object-Level Expressions . . . 12

1.4.2 The Ground Representation in Abstract Syntax . . . 13

2 Substitutions as Logic Programming 15

2.1 Copy Clauses . . . 15

2.2 The Substitution Encoding Theorem and Corollaries . . . 21

2.3 Object-level Matching and Unifiability . . . 25

2.4 WOLOG: A Technique for Automated Theorem Proving . . . 26

3 Unification and Generalization 32

3.1 The Necessity of Object-Level Unification . . . 32

3.2 Various Routes to Unification . . . 36

3.2.1 The Algorithmic Approach . . . 37

3.2.2 A Short Cut to Unification . . . 43

3.3 Forcing $\forall$-Introduction . . . 52

3.4 A Prolog Meta-Interpreter 55
3.4.1 Substitutions with New Variables 57
3.4.2 Object-level Depth-First Interpretation 58
3.4.3 Adding Partial Evaluation and Generalization 60
3.5 Polymorphic Type-Inferencing 64
3.5.1 The Purely Declarative Approach 65
3.5.2 Algorithm $\mathcal{W}_\lambda^*$ 78
3.5.3 A Quick and Simple Alternative 85
3.5.4 Comparison with Other Research 90
3.5.5 Explanation-Based Generalization with Principal Type Schemes . . 92

4 Meta-Programming Higher-Order Formalisms **95**
4.1 Background: Simplifying Higher-Order Unification 96
4.2 Translating λProlog to L_λ 98
4.2.1 Polarizing Copy Clauses 98
4.2.2 The Translation Algorithm 106
4.3 Unification Under a Mixed Prefix 110
4.3.1 Object-Level Higher-Order Unifiability 110
4.4 Object-level L_λ Unification: The Algorithmic Approach 111
4.4.1 Raising 113
4.4.2 The L_λ $\exists\forall$-Unification Program 114
4.5 `setsub` Under A Mixed Prefix 124
4.5.1 Object-Level L_λ Resolution 128
4.6 Finding Higher-Order Critical Pairs 131
4.6.1 Preserving α-Equivalence 133
4.6.2 Rewrite Rules in Abstract Syntax 136
4.6.3 Sample Application: Local-Confluence of $\beta\eta$-Reduction 139
Conclusion 142

A Complete Program Codes **150**

List of Figures

3.1 The Damas-Milner Calculus 34
3.2 The Vanilla interpreter (in Prolog syntax) 56
3.3 Basic Operations on Substitutions 58
3.4 A Meta-Interpreter for First-Order Horn Clauses 59
3.5 Prolog Meta-Partial Evaluator 63
3.6 Forming Generalizations from Meta-Partial Evaluation 64
3.7 MLtyper program 66
3.8 Algorithm $\mathcal{W}$ 79
3.9 Algorithm $\mathcal{W}^*$ 81
3.10 Algorithm $\mathcal{W}^*_\lambda$ 82
3.11 Generalization of Type Expressions Through `setsub` 86
3.12 ML-QuickTyper 87
4.1 Mixed-Prefix to $\exists\forall$-Unification Transformation 114
4.2 L_λ Main Transformations, Part 1 116
4.3 L_λ Main Transformations, Part 2 117
4.4 L_λ Top-Level Control 118
4.5 L_λ Critical-Pair Program 138

Chapter 1

Introduction

Meta-programming, or the ability to specify and manipulate an object-level language or formalism in a meta-level formalism, is a fundamental concept in computer science. The most obvious areas of its application include compilers, program transformers, knowledge representation and automated deduction, but it is in fact manifested in many other ways as well. In the theory of computation, for example, we often speak of a Turing machine accepting another Turing machine as input. In hardware design, we speak of virtual memory and virtual machine architectures. In short, computer science, perhaps more than any other discipline, must be precise about the relationship between the object-level world and how it is being represented and manipulated at the meta-level.

Prolog, as a symbolic language, has generally been regarded as an ideal candidate for meta-programming. Much has been accomplished in logic programming in the last twenty years. Prolog-like languages have been used to specify many operations declaratively, with the benefit that the correctness of programs can be shown directly. There are still many problems, however, that remained to be solved. Limitations of pure logic programming has led to the use of many extra-logical constructs, such as `assert`, `retract`, and `gensym`. Significant improvements have been made in this aspect with the introduction of logic programming languages based on logical formalisms richer than first-order Horn-clause resolution. It remains interesting, however, how logic programming has still not been able to define certain operations easily, in ways one would think it was intended for. Many inherently logic-oriented operations have been surprisingly lacking in satisfactory treatments in logic programming. Efforts have been made by meta-programmers (see [23]) to solve these problems, though only in a first-order setting, by adding built-in imperatives to the meta-language. A specialized semantics for meta-programs must also be adapted. Techniques were also borrowed from other programming disciplines, such as modules (see [24]). The result of this style of programming is that meta-programs looks much like programs in imperative languages. The original principle of programming-in-logic is weakened if not lost. Stronger *logical* tools are needed in order to give operations such as polymorphic type-inferencing and critical-pair completion satisfactorily declarative treatments. Particularly lacking in this respect is the field of higher-order logic programming. Although the idea is promising since higher-order terms allow more declarative specifications, it has not reached a practical level of maturity. Most of the experiments up to now have relied too heavily on the internal implementation of some unification algorithm such as Huet's [27]. Unlike first-order unification, the higher-order case lacks the notion of unique most general unifiers and is undecidable. The higher-order unification algorithm contains a high degree of nondeterminism in choosing substitutions (unifiers). Because of this, the algorithm can be easily misused or over-used.

The aim of this thesis is to provide general techniques to facilitate the use of higher-order logic programming in formulating a class of problems that has traditionally been difficult to meta-program. This class of problems requires an explicit treatment of object-level free variables. Central to these techniques is the ability to specify and apply substitutions. The starting point of this research is the work of Miller in [37], where the concept of *copy clauses* was introduced. Copy clauses explicitly encode substitution for simply typed lambda terms. By this mechanism, the nondeterminism of the higher-order unification algorithm can be moved from the level of implementation detail to the level of *logic programming specifications*, giving us more control over the *exact* higher-order operations desired. This paper thus proposed to replace a logic programming language (λProlog) using Huet's higher-order unification algorithm with a language (L_λ) using a much simpler form of unification. In this thesis we will formalize this transformation. In addition to easing the complexity of using higher-order terms, copy clauses allow the domain of substitutions to be no longer restricted to meta-level logic variables. Using logic variables to represent object-level variables (as opposed to using constant symbols) may constrain or distort the semantics of the object-level formalism. Copy clauses provide a means of using higher-order unification in a logic programming context while avoiding its pitfalls.

1.1 Overview

In the remainder of the introductory chapter, we will give some background material on meta-programming in logic, and on using higher-order logic programming languages. In Chapter 2, we develop the theory of copy clauses for use in meta-programming object-level substitutions. This leads directly to formulations of matching and unifiability. A sample application of object-level matching is given. Chapter 3 begins the formulation of the most difficult meta-programming issue of generalization over free variables. Two general methods are given to deal with this problem in the first-order setting. A Prolog meta-interpreter is given to illustrate the techniques. The meta-interpreter is then modified to conduct partial-deduction and explanation-based learning. The sections on polymorphic type inferencing also presents new techniques for dealing with object-level free variables. In Chapter 4, we deal with the case where the object-theory is itself higher-ordered. The first-order techniques of Chapter 3 are upgraded to a restricted higher-order case. This restricted case, however, still suffices for all formulations. A higher-order extension of Knuth-Bendix style critical-pair completion is implemented to demonstrate the techniques.

1.2 Notations, Conventions, and Prerequisites

The following conventions and notations will be adopted in this thesis, unless otherwise explicitly noted. Others will be introduced in the appropriate sections:

- The symbols $\in$ and $\notin$ should be read as "appears free in" and "does not appear free in" respectively where appropriate.

- α-conversion is assumed whenever there is the possibility of bound-variable capture, except where noted.

- $[t/x]$ denotes the substitution t replacing x.

- $t\sigma$ stands for the substitution σ applied to t.

- Composition of functions (namely substitutions) $f \circ g$ should be read right to left in application, with the intended meaning that $(f \circ g)(x) = g(f(x))$. When the functions are substitutions, we will write application postfix-style: $x(f \circ g) = (xf)g$.

- $\overline{X_n}$ is to be read as $X_1, \ldots, X_n$, and $\overline{X_n} : \overline{T_n}$ is to be read as $X_1 : T_1, \ldots, X_n : T_n$. The substitution $[\overline{t_n}/\overline{x_n}]$ stands for $[t_1/x_1, \ldots, t_n/x_n]$.

- $=_r$ is to stand for equality modulo some rule r. For example, $=_\alpha$ represents α-convertibility.

- Expressions such as Σ, R or $\Sigma + R$ are to represent list or set Σ appended with, or unioned with the singleton element R.

- For list notation, we use *nil* for the empty list and :: for cons. We use λProlog's built-in list structures for convenience only. List structures can be easily defined independently as well.

- Expressions in mathematical font, such as $(A\ x)$, may be interchanged with literal logic programming expressions, in this case `(A x)`. This helps to stress whether an expression it is being regarded meta-mathematically or as part of actual program code. The logical meaning remains the same.

In this thesis, definitions and results from other sources are often used and cited. We sometimes take the liberty of slightly restating the borrowed results to better incorporate them here.

We assume basic familiarity with the following concepts:

- First-order Horn Clause Prolog [31, 51] and first-order unification [49, 33].

- The simply typed λ-calculus [3, 25].

- Gentzen-style sequent calculus and natural deduction for intuitionistic logic, and the cut-elimination result [16].

Summaries of much of this basic material can be found in [35], which also discusses the relationship between these concepts and the higher-order logic programming language in which most of the examples here are implemented.

1.3 Choice of Meta-Languages

The principle techniques of this dissertation are given for a class of meta-languages supporting what is commonly known as *higher-order abstract syntax,* [48, 38] which we will explain in the next section. First-order Horn Clauses are insufficient for our tasks. We require that the logical connectives $\forall$ and $\Rightarrow$ be unrestricted in their use in forming program clauses and queries (goals). This requires the classical semantics of Herbrand models be dropped in favor of a direct, constructive, proof-theoretic semantics. Such a semantics also gives rise to a logical notion proof search known as *uniform provability*, studied in [41].

It is undesirable to restrict the applications of our definitions and tools to one particular meta-language. We will therefore show how the techniques developed can be formulated in the simplest language that is capable of supporting our definitions, namely the L_λ sub-language [36] of the better known λProlog [43]. This means that our techniques can be easily adopted to work in a spectrum of languages (or *logical frameworks*) that contain L_λ, including λProlog, Elf [46], Forum [40], and their variants. Although these richer languages may simplify the writing of meta-programs for particular tasks (such as natural deduction in Elf), it is desirable to show how meta-programs can be written using a minimal set of meta-language features.

A particular variation among meta-languages is the treatment of types. The meta-language can be simply typed, dependently typed, polymorphically typed, or untyped altogether. Type information may be highly useful or necessary in the object-level formalisms we seek to represent. This does not mean, however, that the meta-logic itself has to be richly typed. Certainly there are examples where a typed meta-theory is desirable. In the programs we will write, however, meta-level type information rarely plays a more useful role than as comments for program clauses. There are even cases where the meta-level typing discipline hampers the flexibility of the meta-language to deal with different classes of object-level syntax. We would like to present our techniques as independently of this issue as possible. In this we are limited by the fact that all programs are written in a currently-available interpreter for λProlog (based on [11]), which is simply-typed (though it also allows some degree of polymorphism). Almost all programs we write are simply-typed at the meta-level. Only trivial list operations such as `append` and `member` will have polymorphic meta-level types. Our formulations will thus remain valid in richer-typed or untyped meta-languages. In program codes presented, we will give the meta-level type information only if we feel it can aide in the better understanding of the code. This is true particularly early on, as we introduce the reader to the sample meta-programs using higher-order abstract syntax.

1.3.1 Hereditary Harrop Formulas and Uniform Provability

First-order Horn clauses, as used in Prolog, are insufficient for our purposes (as will be demonstrated). We require that the meta-language be based on *higher-order hereditary harrop formulas* (HoHH). λProlog is the original language designed to support these formulas. Simply typed lambda terms replaces first order terms, and higher-order unification replaces first order unification. The λProlog syntax for λ-term $\lambda x.p$ is `(x\p)`. Two terms are equal if they are $\alpha\beta\eta$-convertible to each other. HoHH formulas allow universal quantification in goal clauses. The operational interpretation of this type of quantification, written `(pi x\ (A x))` is to prove `(A x)` with a fresh constant symbol, or *eigenvariable,* `v` (of the appropriate type) replacing `x`. The *signature* (or language) is now *dynamically* extended with the a constant symbol `v`. Implications can also appear in goals. The interpretation of `(A => B)` is to prove `B` assuming clause `A`, with `A` visible only in the proof of `B`. In this way the known program clauses can also be dynamically extended without resorting to imperative constructs such as `assert` and `retract`.

λProlog goal clauses and program clauses, also called *definite clauses* or D-clauses, are defined by the following. An atomic formula $(f\ g_1, \ldots, g_n)$ is *rigid* if f is not a free variable. Let A range over atomic formulas and A_r range over rigid atomic formulas. Let G, G_1, G_2 and D, D_1, D_2 range over goal clauses and definite clauses respectively. Quantifications are defined relative to a type τ. In the actual interpreter however, the type of the quantified variable is inferred.

- $G = \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau x\ G \mid \forall_\tau x\ G \mid D \Rightarrow G$
- $D = A_r \mid D_1 \wedge D_2 \mid \forall_\tau x\ D \mid G \Rightarrow A_r$

Non-logical constants are defined by a signature of declarations. Types are defined using the keywords `kind` and `type`, as in `(kind atype type)`. Constants are assigned types in the manner of `(type t atype)`.

A logical notion of proof search is made possible by the *uniform proof* property of HoHH formulas. A uniform proof is a cut-free proof where the last inference rule of the proof is always a R-right rule, where R is the top level logical connective (such as $\forall$, $\wedge$) of the conclusion, if the conclusion is non-atomic. In case the conclusion is an atomic formula, *backchaining,* (similar to backchaining in Horn clause proof search) must be the last rule of the proof.

Formally, a *signature* is a list of typing declarations $\{a_1 : \sigma_1, \ldots, a_n : \sigma_n\}$ for distinct constants $a_1, \ldots, a_n$ and simple types $\sigma_1, \ldots, \sigma_n$. For signature Σ a term t is "in Σ" (alternatively, a "Σ-term") if all constants in t are declared in Σ. Let $\mathcal{P}$ be a set (or conjunction) of D-clauses in a signature Σ, i.e., a logic program. Define $\mid \mathcal{P} \mid_\Sigma$ to be the smallest set of pairs (Γ, D), for D a (Σ) D-clause and Γ a set of (Σ) D-clauses, where:

- If $D \in \mathcal{P}$, then $(\langle\rangle, D) \in\mid \mathcal{P} \mid_\Sigma$.
- If $(\Gamma, D_1 \wedge D_2) \in\mid \mathcal{P} \mid_\Sigma$, then $(\Gamma, D_1) \in\mid \mathcal{P} \mid_\Sigma$ and $(\Gamma, D_2) \in\mid \mathcal{P} \mid_\Sigma$.
- If $(\Gamma, \forall_\tau x\ D) \in\mid \mathcal{P} \mid_\Sigma$, then $(\Gamma, D[t/x]) \in\mid \mathcal{P} \mid_\Sigma$ for all Σ-terms t of type τ.
- If $(\Gamma, G \Rightarrow D) \in\mid \mathcal{P} \mid_\Sigma$, then $(\Gamma \cup \{G\}, D) \in\mid \mathcal{P} \mid_\Sigma$

Uniform provability can now be defined by the following properties, which immediately form the search rules of a non-deterministic interpreter for logic program queries of the form $\Sigma; \Gamma \vdash G$ (where $\vdash$ can be interpreted as intuitionistic provability):

INSTANCE ($\exists$-right): $\Sigma; \mathcal{P} \vdash \exists_\tau x\ G$ if and only if for some Σ-term $t : \tau$, $\Sigma; \mathcal{P} \vdash G[t/x]$.

AND ($\wedge$-right): $\Sigma; \mathcal{P} \vdash G_1 \wedge G_2$ if and only if $\Sigma; \mathcal{P} \vdash G_1$ and $\Sigma; \mathcal{P} \vdash G_2$.

GENERIC ($\forall$-right): $\Sigma; \mathcal{P} \vdash \forall_\tau x\ G$ if and only if $\Sigma, c; \mathcal{P} \vdash G[c/x]$ for constant symbol (eigenvariable) $c \notin \Sigma$.

AUGMENT ($\Rightarrow$-right): $\Sigma; \mathcal{P} \vdash D \Rightarrow G$ if and only if $\Sigma; \mathcal{P}, D \vdash G$

BACKCHAIN ($\Rightarrow$-left): $\Sigma; \mathcal{P} \vdash A$ if and only if for some $(\Gamma, A) \in\mid \mathcal{P} \mid_\Sigma$ and for all $G \in \Gamma$, $\Sigma; \mathcal{P} \vdash G$.

Two aspects of the above description make the interpreter unacceptably non-deterministic. One is the construction of $\mid \mathcal{P} \mid_\Sigma$, which is generally infinite, and the other is the INSTANCE rule, which would require the interpreter to "guess" some term t. Thus the second component of λProlog's search strategy is a higher-order (pre-)unification algorithm. For every signature there are denumerably many *logic variables* of each type. The goal $\exists v G$ is manifested as $G[X_v/v]$ for a logic variable X_v. Unification is then used in the proof to find an answer substitution for X_v. Similarly, BACKCHAIN would use unification to determine which pair $(\Gamma, D[t/x])$ in the hypothetical $\mid \mathcal{P} \mid_\Sigma$ to resolve with.

The crucial result for uniform provability is that it is sound and complete with respect to intuitionistic provability. Provability in this thesis will always mean intuitionistic provability. Background on intuitionistic logic can be found in [52]. The detailed theory of uniform proofs is given in [41]. For more information on the λProlog language, please consult [35, 43].

1.3.2 The L_λ Restriction of λProlog

An important sub-language of λProlog is L_λ. Instead of full higher-order unification, L_λ requires only a very restricted form of unification, which is decidable and yields unique most general unifiers (mgu). The feasibility of meta-programming complex systems in the simple L_λ language is a part of the aim of this thesis, and we will discuss this thoroughly in Chapter 4. For now, we will give its formal definition. Please consult [36] for the full technical details of the L_λ language.

In a given formula, a subformula occurrence is *positive* if it occurs on the left of an even number of implications ($\Rightarrow$'s), and *negative* if it occurs on the left of an odd number of implications. A variable is *essentially universal* if it is bound by a term-level λ-abstraction, or a positive occurrence of an universal quantifier in a goal clause, or a negative occurrence of an universal quantifier in a program clause. A variable is *essentially existential* if it is bound by a positive occurrence of an existential quantifier or a negative occurrence of an universal quantifier in a goal clause, or a positive occurrence of an universal quantifier in a program clause. Effectively, the essentially universal variables are constant symbols in their scope, while the essentially existential variables are the so-called logic variables.

Terms where all higher-order essentially existential variables M appear in subterms of the form $(M\ x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ $(0 \leq n)$ are distinct, essentially universal variables are called L_λ *terms*[1]. A β_0 *redex* is a β-redex of the form $(\lambda x.T)y$ where y is an essentially universal variable. Unification of β-normalized L_λ terms will only generate new β_0 redices. β_0-reduction is an exceedingly simple task of renaming the λ-bound variables with the universal variables being applied to. Only avoidance of bound-variable capture needs to be assured (just as in α-conversion). Unlike general β-reduction, β_0-reduction will terminate regardless of well-typedness.

The following is a sample L_λ program clause:

```
pi A\ pi H\  (pred (x\ (A x)) R :- pi y\ (pi z\ (H y z))).
```

And the following are λProlog program clauses that are not in L_λ:

```
pi A\ pi T\  (pred A T x\(A T x)).
pi A\  (pred A :- pred2 (A 0)).
pi A\ pi B\  (pred A :- pred2 (A B)).
```

Presently, an interpreter for L_λ exists only in the sense that its parent langugage λProlog has an interpreter. The actual λProlog interpreter also allows the omission of positive `pi` quantifiers in top-level program clauses, just as in most Prolog interpreters. L_λ still properly contains all first-order Horn Clauses (without negation-as-failure, which is commonly adopted by Prolog programmers). In concrete code presented as above, we use captial letters for essentially existential variables and lower case letters for essentially universal variables. Keep in mind, however, that the status of these variables also depend on whether they appear inside a program or goal clause.

Points to Note

To further familiarize the reader with some key λProlog and L_λ "features" used in this thesis, we briefly note the following:

- The formula $(A \wedge B) \Rightarrow C$ is logically equivalent to $A \Rightarrow B \Rightarrow C$ ($\Rightarrow$ associates to the right). Thus conjunction $\wedge$ need not be used in forming program and goal clauses (this is true for Horn Clauses also).

[1] L_λ terms have also been called *higher-order patterns.*

- Because program clauses can appear "hereditarily" in goal clauses, *free variables* or existentially quantified varibles may become part of the program. If `p` is a predicate on integers then the query `sigma X\((p X) => (p 1, p 2))` will fail, assuming there is no other clause for `p`. Here, `sigma` denotes existential quantification, just as `pi` denotes universal quantification. `(pi X\(p X)) => (p 1, p 2)` will always succeed.

- In goals, negatively $\forall$-quantified variables are the same as positively $\exists$-quantified variables. $\exists$-quantification can be mixed with $\forall$-quantification in goals. A $\exists$-quantified variable x can appear inside the scope of some $\forall$-quantified variables $v_1, \ldots, v_n$, and instantiations of x can not contain these free. To avoid using `sigma` explicitly in goals, we can replace x with $(x' \ v_1, \ldots, v_n)$ where x' is a new, negatively $\forall$-quantified variable. This process is called "raising," which we will explain further in section 4.4.1.

- Goals and Program clauses can also appear as *data* - of designated type o. For example, the clause

  ```
  update-and-solve D G :- (p :- D) => G
  ```

  defines a predicate that updates the predicate `p` with a new success-condition `D`. The goal `G` is then solved, "continuation-passing style," in this updated context.

1.4 Representing Object-Level Syntax

Central to meta-programming is the issue of how the syntax of object-level expressions is represented at the meta-level. It is possible to identify three different approaches to the representations of object-level syntax [38]. First, *concrete syntax* is the linear, string representation of expressions that humans are usually comfortable reading and typing into computers. What is good for humans, however, is not necessarily good for computation. *Parse trees*, the second representation of syntax, is generally represented in most programming languages using first-order terms (as in Prolog) or as linked structures (as in Pascal or Lisp). This representation is clearly superior to concrete syntax since the later contains much information that is not important to the intended meaning of an expression. For example, the amount of white space in an expression, and whether or not an operator is infix or prefix, are seldom relevant to the meaning of an expression. Also, information that is important for manipulating expressions, such as the relationship between a function and its arguments, is not represented conveniently in concrete syntax. With parse trees, it is immediate to locate the subtrees representing the arguments of a function symbol.

Just as concrete syntax is an unsatisfactory basis for directly supporting computations, parse trees can be similarly unsatisfactory. Many objects, including programs, logical formulas and type expressions, involve scoping constructs. In particular, many involve the use of bound variables. For example, consider representing the expression (given as concrete syntax) $\forall x \exists y(r(x,y) \vee q(y))$ as a first-order term in Prolog. The choices here seem to be terms of one of the following forms:

```
all(X,exists(Y,or(r(X,Y),q(Y)))), all("x",exists("y",or(r("x","y"),q("y"))))
```

That is, the object-level bound variables x and y are either represented as Prolog logic variables (as in the first term), or by using meta-level closed data structures, such as strings or constant symbols. In meta-programming terminology these are known as the *non-ground* and *ground* representations respectively (see [23] for a survey of meta-programming in first-order logic programming languages). Although both representations can be used to form the basis of a program that manipulates logical expressions (as in a theorem prover), forcing object-level syntax with bound variables into first-order terms can produce undersirable consequences similar to those resulting from forcing tree structured first-order terms into linear strings. First, the names of bound variables are generally not important with respect to the intended meaning of expressions. Unfortunately, programmers using the parse tree representation generally must worry about names for bound variables in various different ways. Instantiation or substitution for such expressions is also difficult to implement correctly. A correct implementation requires, for example, checks that variables are not captured, and a procedure for generating new variable names. Various techniques, such as the use of nameless dummies [2], can be implemented as auxiliary procedures to deal with these aspects of bound variables. It is important to realize, however, that bound variables and the operations related to them have well-understood logical treatments.

With a logic programming language, it is natural to ask for a declarative treatment of bound variables. It is in this sense that we speak now of the third presentation of syntax. By *higher-order abstract syntax* we mean expressions that contain λ-abstractions and which are considered equal if they are related by α-conversion. In such a setting, the meta-language need not inquire about the names of bound variables since α-conversion renders them fictitious.

1.4.1 Higher-Order Abstract Syntax For Object-Level Expressions

To illustrate the use of abstract syntax, consider representing first-order formulas as data structures. In λProlog, object-level languages are declared with a signature of type and constant declarations. The primitive types `term` and `form` classify first-order terms and formulas. The logical connectives are then declared using these types:

```
kind term, form      type.
type truth           form.
type neg             form -> form -> form.
type and, or, imp    form -> form -> form.
type all, exists     (term -> form) -> form.
```

Non-logical constants must also be declared. The following non-logical object-level constants will be used in later examples.

```
type a, b            term.
type g               term -> term.
type adj, path       term -> term -> form.
```

Here, these declarations are of two object-level constants, one unary function symbol, and two binary predicate symbols. Closed meta-level terms (over these constants) of type `term` will denote object-level, closed first-order terms; similarly, closed meta-level terms of type `form` will denote object-level, closed first-order formulas. The only aspect of this encoding that requires explanation is the representation of quantified formulas. For this, a λ-abstraction is used, as illustrated by the formula

```
all x\(exists y\(or (adj x y) (adj y x))).
```

That is, object-level quantified formulas are represented by meta-level λ-abstractions, which have types such as `term -> form` (term abstractions over formulas). We use variable names in writing this expression (just as we use blank spaces and parenthesis) but these are not part of the meaning of the term.

The above example illustrates the representation of an object-level *first-order* formula in higher-order abstract syntax. We can of course also use meta-level higher-order abstract syntax to represent higher-order expressions at the object-level. Below is a signature for object-level λ-terms:

```
type app term -> term -> term.
type abs (term -> term) -> term.
```

Object-level λ-abstraction is represented with meta-level λ-abstraction, prefixed by the constructor `abs` to indicate that it belongs to the category of object-level *terms.* The term $\lambda x.(g\ x)$ is thus represented by `(abs x\(app g x))`.

1.4.2 The Ground Representation in Abstract Syntax

The preceeding has demonstrated that as a consquence of this logical treatment of variables, we need not be concerned with the distinction between the ground and non-ground forms of representation when only object-level quantified formulas are involved. The issue of ground v.s. non-ground representation, however, re-surfaces when object-level *free* variables are needed.

Many of the problems of meta-programming in logic concerns the discrepancy between semantics at the object- and meta-levels. At the core of this discrepancy is how *object-level free variables* are represented and how properties such as equality and unifiability between object-level expressions are specified. Using meta-level logic variables to represent object-level free variables directly may be inconsistent with the intended meaning of object-level expressions, since the meaning of logic variables is fixed at the meta-level as either existentially quantified in goals or universally quantified in program clauses. For example, say that we would like to have a well-ordering relation *ord* defined on some object-level variables. If logic variables are used to represent these variables, then the queries $ord\ X\ Y$ and $ord\ Y\ X$ (where X and Y are logic variables) will either *both* succeed or both fail, clearly an undesirable consequence for a well-ordering relation. We therefore consider a so-called ground representation, where object-level free variables are mapped to meta-level *constant symbols*. It should be noted that in the context of higher-order abstract syntax the words "ground" and "non-ground" are not entirely appropriate. *Signatures* and *mixed quantifier prefixes* [39] are more appropriate concepts for describing the style of representation here. We will, however, also respect the standard meta-programming terminology.

Central to the declarative manipulation of object-level expressions containing free variables is the encoding and application of substitutions. One difficulty in programming in the ground representation is much of the inherent substitution and unification facilities of the meta-level logic programming language will be made redundant. We would like to have methods which allow us to program in the ground representation, independently of the fixed semantics of the meta-language, while still taking advantage of the inherent abilities of the meta-language as much as possible. The next chapters address these issues.

Chapter 2

Substitutions as Logic Programming

The Rules of type checking for simply-typed λ-terms are specified inductively on the structure of types. Let H represents a context or *environment* of type assumptions for variables. The rule are as follows:

$$\frac{H \vdash M : \sigma \to \tau \quad H \vdash N : \sigma}{H \vdash (M\ N) : \tau}\ \mathbf{app} \qquad\qquad \frac{H, x : \sigma \vdash M : \tau}{H \vdash \lambda x.M : \sigma \to \tau}\ \mathbf{abs} \qquad x \notin H$$

The relationship between a term and its type can be captured by a logical sentence. For example, that term f has type $i \to j$ can be specified by

$$\forall x(hastype\ x\ i \Rightarrow hastype\ (f\ x)\ j).$$

In [37], it was shown that this specification of type checking can be extended in a natural way to formulate $\beta\eta$-equality between λ-terms, thereby making possible the explicit specification of equality and substitution in logic programming. This is accomplished by extending the type checking predicate to involve a pair of terms and their type.

2.1 Copy Clauses

Suppose there is a predicate symbol $copy\sigma$ for each type σ. Given terms T and S (of some type), let $T^{\beta\eta}$ and $S^{\beta\eta}$ be the $\beta\eta$ normal forms of T and S, respectively. Define the following (partial) function inductively on the structure of simple types:

Definition 2.1 *(Simple-type copy clauses, [37])*

$$[\![T,\ S : a]\!] = (copya\ T^{\beta\eta}\ S^{\beta\eta})\ \textit{if } a \textit{ is a primitive type.}$$

$$[\![T,\ S : b \rightarrow c]\!] = \forall x \forall u([\![x,\ u : b]\!] \Rightarrow [\![(T\ x),\ (S\ u) : c]\!])$$

$[\![T,\ S : \sigma]\!]$ can be viewed as type-checking two terms in parallel.

In the following, we will refer to $[\![]\!]$ clauses as *copy clauses.* Copy clauses for first-order types are horn clauses. However, if we are to use them for arbitrary order types, and as goals as well as program clauses, then Horn Clauses do not suffice. Higher-order hereditary Harrop formulas are needed in general. As an example of a copy clause, $[\![t,\ s : a \rightarrow b \rightarrow c]\!]$ is equal to:

$$\forall A \forall B\ (copya\ A\ B \Rightarrow \forall C \forall D(copyb\ C\ D \Rightarrow copyc\ ((t\ A)\ B)\ ((s\ C)\ D))).$$

It is valid to put such formulas into prenex-normal form. This clause is thus equivalent to, in λProlog notation,

```
pi A\ pi B\ pi C\ pi D\
     (copya B D => (copyb A C => copyc ((t A) B) ((s C) D)))
```

or simply

```
copyc (t A B) (s C D) :- copya A C, copyb B D.
```

Although this is a Horn clause, the formula $[\![t,\ s : (a \rightarrow b) \rightarrow c]\!]$ is equivalent to:

```
copyc (t G) (s H) :- pi x\ (pi y\ (copya x y => copyb (G x) (H y))).
```

All copy clauses can be considered as either goal clauses *or* program clauses in λProlog. Furthermore, all copy clauses, with some modification, can also be represented in the L_λ language. We will detail these modifications in Chapter 4.

Copy clauses effectively represents substitutions for simply typed terms (of arbitrary order). To formalize the role of copy clauses, we introduce the following results.

First, we have to show that the definition of copy clauses is well formed.

Lemma 2.1 *(Well-formedness of simple-type copy clauses)*
Under any signature Σ and (simple) type σ, for any Σ-terms M, N, M', N' such that $M =_{\beta\eta} M'$ and $N =_{\beta\eta} N'$,

$$[\![M,\ N : \sigma]\!] =_\alpha [\![M',\ N' : \sigma]\!].$$

Proof: by induction on the structure of σ (straightforward). $\square$

The following technical lemma will be required in many places. This lemma also demonstrates the basic format of proofs by induction on the height of uniform proofs, which are used prolifically in this thesis.

Lemma 2.2 *(Same-Variable)*

Under any signature Σ, for some correctly typed Σ-terms $a_1 : \tau_1, \ldots, a_n : \tau_n$ and $b_1 : \tau_1, \ldots, b_n : \tau_n$, let Γ be the set of clauses

$$\{[\![a_1,\ b_1 : \tau_1]\!], \ldots, [\![a_n,\ b_n : \tau_n]\!]\}.$$

For terms M and N of type $\sigma \to \tau$, and constant symbols $x : \sigma$ and $y : \sigma$ not appearing free in Γ, M or N,

$$\Sigma; \Gamma \vdash [\![x,\ y : \sigma]\!] \Rightarrow [\![(M\ x),\ (N\ y) : \tau]\!]$$

if and only if

$$\Sigma; \Gamma \vdash [\![x,\ x : \sigma]\!] \Rightarrow [\![(M\ x),\ (N\ x) : \tau]\!].$$

Proof: the forward direction is by $\forall$-Right and instantiating (using cut) x for both quantified variables. The reverse direction is by induction on the height of uniform proofs:

> The last step of the proof is Augment by definition of uniform provability, so we are really showing (by induction) that
>
> $$\Sigma; \Gamma, [\![x,\ x : \sigma]\!] \vdash [\![(M\ x),\ (N\ x) : \tau]\!]$$
>
> implies
>
> $$\Sigma; \Gamma, [\![x,\ y : \sigma]\!] \vdash [\![(M\ x),\ (N\ y) : \tau]\!].$$
>
> **Base Case:** Either $Mx = x$ and $Nx = x$ or $Mx = s$ and $Nx = t$ for $[\![s,\ t : \tau]\!] \in \Gamma$. In the first case, since $x \notin N$, $Nx = x$ implies $Ny = y$. For the second case, since x does not appear free in t, it can not appear free in Nx, and so $Nx = Ny = t$.
>
> **Augment Case:**[1] If the last rule was Augment, then we have:
>
> $$\Sigma, u : \rho, v : \rho; (\Gamma, [\![u,\ v : \rho]\!]), [\![x,\ x : \sigma]\!] \vdash [\![Mxu,\ Nxv : \tau']\!]$$

[1] To be precise, this case involes some applications of the **Generic** rule followed by **Augment**. By the definition of uniform provability, it is valid to combine them into one case.

where $\tau = \rho \rightarrow \tau'$ and u and v are distinct new constants. Let $M' = \lambda y.Myu$ and $N' = \lambda y.Nyv$. By inductive hypothesis (ranging over the extended Γ), we have

$$\Sigma, u : \rho, v : \rho; (\Gamma, [\![u,\ v : \rho]\!]), [\![x,\ y : \sigma]\!] \vdash [\![M'x,\ N'y : \tau']\!].$$

The result then follows by $\Rightarrow$-Right and the definition of copy clauses.

Backchain Case:[2] Since backchaining only occurs on atomic formulas, assume $Mx =_{\beta\eta} (k\ f_1, \ldots, f_n)$ and $Nx =_{\beta\eta} (k'\ g_1, \ldots, g_n)$ such that $[\![k,\ k' : i_1 \rightarrow \ldots \rightarrow i_n \rightarrow \tau]\!]$ exists in $(\Gamma, [\![x,\ x : \sigma]\!])^3$. This amounts to showing that for each $j \in 1 \ldots n$,

$$\Sigma; \Gamma, [\![x,\ x : \sigma]\!] \vdash [\![f_j,\ g_j : i_j]\!].$$

So by inductive hypothesis,

$$\Sigma; \Gamma, [\![x,\ y : \sigma]\!] \vdash [\![f_j,\ g_j[y/x] : i_j]\!].$$

The result then follows.

$\square$

The following two theorems formally establish the relationship between copy clauses and simple typing.

Theorem 2.3 *(Parallel Typing)*
Under any signature Σ, for correctly typed Σ-terms $a_1 : \tau_1, \ldots, a_n : \tau_n$ and $b_1 : \tau_1, \ldots, b_n : \tau_n$, let Γ be the set of clauses

$$\{[\![a_1,\ b_1 : \tau_1]\!], \ldots, [\![a_n,\ b_n : \tau_n]\!]\}$$

For Σ-terms M, N and type σ, if $\Sigma; \Gamma \vdash [\![M,\ N : \sigma]\!]$, then M and N have type σ in Σ.

Proof: by induction on the height of (uniform) proofs:

- The base case follows by the preconditions of the theorem.
- If the last step of the proof was augment, then $\sigma = \gamma \rightarrow \delta$ and we have

$$\Sigma, u : \gamma, v : \gamma; \Gamma, [\![u,\ v : \gamma]\!] \vdash [\![Mu,\ Nv : \delta]\!]$$

[2] Again to be precise, this case involves a $\forall$-left rule (which alternatively can be interpreted as looking up $|\ \mathcal{P}\ |_\Sigma$) followed by **Backchain**.

[3] Note here that k and k' are not necessarily constants. The term $(k\ f_1, \ldots, f_n)$ and $(k'\ g_1, \ldots, g_n)$ are not necessarily in normal form.

for some new constants u, v. The inductive hypotheses applies (on the extended signature) and so $Mu : \delta$ is well-typed in $\Sigma, u : \gamma, v : \gamma$, and in fact in $\Sigma, u : \gamma$ since v does not appear in M. Similarly, $Nv : \delta$ is well-typed in $\Sigma, v : \gamma$. By rule **abs** for simple typing, we therefore have $M : \gamma \to \delta$ and $N : \gamma \to \delta$ in Σ.

- If the last step of the proof was Backchain, then the goal must be atomic, and so we have

$$\Sigma; \Gamma \vdash copy\delta\ (k\ f_1, \ldots, f_n)^{\beta\eta}\ (k'\ g_1, \ldots, g_n)^{\beta\eta}$$

where

$$[\![k,\ k' : \gamma_1 \to, \ldots, \gamma_n \to \delta]\!]$$

exists in Γ. This means

$$\Sigma; \Gamma \vdash [\![f_i,\ g_i : \gamma_i]\!]$$

holds for each $i \in 1 \ldots n$ at a lower height. Thus by inductive hypothesis each $f_i : \gamma_i$ and $g_i : \gamma_i$ are well typed in Σ. Since by the assumption of

$$[\![k,\ k' : \gamma_1 \to, \ldots, \gamma_n \to \delta]\!]$$

being in Γ means $k : \gamma_1 \to, \ldots, \gamma_n \to \delta$ and $k' : \gamma_1 \to, \ldots, \gamma_n \to \delta$ are correctly typed, we have by the simple typing rule **app** that $(k\ f_1, \ldots, f_n) : \delta$ and $(k'\ g_1, \ldots, g_n) : \delta$ are correctly typed.

□

Theorem 2.4 *(Typing Recovery)*
For signature $\Sigma = \{k_1 : \sigma_1, \ldots, k_n : \sigma_n\}$, let K be the set of clauses

$$\{[\![k_1,\ k_1 : \sigma_1]\!], \ldots, [\![k_n,\ k_n : \sigma_n]\!]\}.$$

Then for Σ-term M and type τ, $\Sigma; K \vdash [\![M,\ M : \tau]\!]$ if and only if $M : \tau$ is well-typed in Σ.

Proof: The forward direction is an instance of the Parallel Typing theorem. The reverse direction is by induction on the simple typing rules:

- In the base case, $H \vdash x : \sigma$ if $x : \sigma \in H$, where H is a simple typing environment. This holds since $[\![x,\ x : \sigma]\!] \in K$.

- For typing rule **app**, by inductive hypothesis $\Sigma; K \vdash [\![M,\ M : \sigma \to \tau]\!]$ and $\Sigma; K \vdash [\![N,\ N : \sigma]\!]$. But this means

$$\Sigma; K \vdash \forall u \forall v [\![u,\ u : \sigma]\!] \Rightarrow [\![M u,\ M v : \tau]\!].$$

By instantiating (using cut) N for u and v, we get

$$\Sigma; K \vdash [\![N,\ N : \sigma]\!] \Rightarrow [\![(M\ N),\ (M\ N) : \tau]\!].$$

By modus ponens (also cut in sequent calculus) with $\Sigma; K \vdash [\![N,\ N : \sigma]\!]$, we have

$$\Sigma; K \vdash [\![(M\ N),\ (M\ N) : \tau]\!].$$

- For typing rule **abs**, by inductive hypothesis we have $\Sigma; K, [\![x,\ x : \sigma]\!] \vdash [\![M,\ M : \tau]\!]$ for a new variable x . Let $M' = \lambda x.M$ so that $M't = M[t/x]$. In (intuitionistic) sequent calculus, $\Gamma, A \vdash B$ if and only if $\Gamma \vdash A \Rightarrow B$. Thus we have

$$\Sigma; K, [\![x,\ x : \sigma]\!] \vdash [\![M'x,\ M'x : \tau]\!]$$

Now by the Same-Variable lemma, we have

$$\Sigma; K, [\![x,\ y : \sigma]\!] \vdash [\![M'x,\ M'y : \tau]\!]$$

for two distinct new variables x and y. The result then follows by $\Rightarrow$-Right and $\forall$-Right on x and y.

□

Since copy clauses are necessarily "directionalized," the parallel typing theorem can not have an "if and only if" result, and thus the form of the Typing Recovery theorem. But Parallel Typing is more general since there are less restrictions on the form of the copy clause assumptions. Note that one consequence of Parallel Typing is the *consistency*, in the classical sense, of a set of copy clauses with respect to a valid signature.

Meta-Level Types and Object-Level Types

In the preceding, definitions and proofs are given mathematically, without referring to how the terms and types are to be represented in a meta-language. When a copy clause $[\![M,\ N : \sigma]\!]$ is to be represented in the meta-logic, we have the choice of representing σ as one of the meta-logic's own (simple) type expressions, or as an *object-level* type. Object-level types are represented as terms (or data-structures) in the meta-logic. In the sample meta-programs presented here, we will use the meta-logic's (λProlog's) own simple types in writing copy clauses for convenience. However, one consequence of the above results is that the burden of type-checking can now be moved from the meta-level to the object-level when using copy clauses. This means the meta-language can in fact be untyped, as in Prolog. For example, instead of a set of meta-level universal quantifiers $\forall_\sigma$ ranging over all types σ (or some polymorphic operator) we can simply have one quantifier $\forall$, and replace each expression $\forall_\sigma x.(A\ x)$ with $\forall x.[\![x,\ x:\sigma]\!] \Rightarrow (A\ x)$. This would amount to moving all type information from the meta-level to the object level. If we used meta-level types in writing copy clauses, however, the role of copy clauses for type inferencing becomes redundant. This dual role is only useful when copy clauses are written for object-level types, which are encoded in the meta-language as terms.

2.2 The Substitution Encoding Theorem and Corollaries

The following result and its corollaries are the backbone of this chapter. The conclusion of this result formulates substitution as β-reduction, and we will adopted this equivalence liberally in this thesis. That is, $(\lambda x.M)T$ and $M[T/x]$ are often used interchangeably, since their normal forms are the same. This theorem was first proposed by Miller in [37]. Here we give a modified version and offer a formal proof. The theorem is stated in a way as to make clear its role in a "ground representation" style of meta-programming. That is, the signature Σ is divided into two sets of constants: $k_1, \ldots, k_m$ which presumably will represent object-level predicate, function and constant symbols, and $c_1, \ldots, c_n$ representing object-level *free variables* which are subject to substitution. The fact that the $k_1, \ldots, k_m$ constants are invariant under (object-level) substitution is represented by the clauses K below.

Theorem 2.5 *(Substitution Encoding) (originally proposed in [37])*
Let signature $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m, c_1 : a_1, \ldots, c_n : a_n\}$. *For* Σ*-terms* $t_1 : a_1, \ldots, t_n : a_n$, *let* $C = \{[\![c_i,\ t_i : a_i]\!] \mid 0 \leq i \leq n\}$ *and let* $K = \{[\![k_i,\ k_i : s_i]\!] \mid 0 \leq i \leq m\}$. *For* Σ*-terms* M *and* N *of some type* τ,
$\Sigma; K, C \vdash\ [\![M,\ N : \tau]\!]$ *if and only if* $(\lambda c_1 \ldots \lambda c_n.M)t_1, \ldots, t_n$ *is* $\beta\eta$*-convertible to* N.

Proof:

Forward Direction: By induction on the height of uniform proofs:

- Base Case. An atomic clause $[\![M,\ N : \tau]\!]$ is derivable in one step iff it is in either K or C. The result follows directly in either case.
- If the last rule was Augment, then we have for new constants x, y,

$$(\Sigma, x : i, y : i); K, (C, [\![x,\ y : i]\!]) \vdash [\![Mx,\ Ny : j]\!]$$

where $\tau = i \to j$. But by the inductive hypotheses on the extended signature and C clauses,

$$(\lambda c_1 \dots \lambda c_n \lambda x.Mx)t_1, \dots, t_n y =_{\beta\eta} Ny.$$

By η-conversion, we also have

$$(\lambda c_1 \dots \lambda c_n.M)t_1, \dots, t_n, y =_{\beta\eta} Ny.$$

And therefore

$$\lambda y[(\lambda c_1 \dots \lambda c_n.M)t_1, \dots, t_n]y =_{\beta\eta} \lambda y.Ny$$

which, again by η-conversion, means

$$(\lambda c_1 \dots \lambda c_n.M)t_1, \dots, t_n =_{\beta\eta} N.$$

- If the last rule was Backchain, then we have

$$\Sigma; K, C \vdash \mathit{copy}\sigma\ (k\ f_1, \dots, f_n)^{\beta\eta}\ (k'\ g_1, \dots, g_n)^{\beta\eta}$$

where $[\![k,\ k' : \vartheta]\!]$ appears in K or C. If it is in K, then $k = k'$; and if a clause in C is used, then $k = c_j$ and $k' = t_j$ for some i. In either case, this amounts to proving $[\![f_1,\ g_1 : \sigma_1]\!], \dots, [\![f_n,\ g_n : \sigma_n]\!]$ at lower heights. So by inductive hypothesis, for each $i \in 1 \dots n$,

$$(\lambda c_1 \dots \lambda c_n.f_i)t_1, \dots, t_n =_{\beta\eta} g_i.$$

This then implies

$$(\lambda c_1 \dots \lambda c_n.(k\ f_1, \dots, f_n))t_1, \dots, t_n =_{\beta\eta} (k'\ g_1, \dots, g_n).$$

Reverse Direction: By induction on n, the number of the c constants in Σ. By the well-formedness lemma, it suffices to show in each case that

$$\Sigma; K, C \vdash [\![M,\ (\lambda c_1 \dots \lambda c_n.M)t_1, \dots, t_n : \sigma]\!].$$

- Base Case. $n = 0$. M is assumed to be well-typed, so by the Typing Recovery theorem, $\Sigma; K \vdash [\![M,\ M : \sigma]\!]$ must hold.

- Inductive Case. Let c_{n+1} be a new constant symbol not appearing in Σ and let t_{n+1} be any $(\Sigma, c_{n+1} : a_{n+1})$-term of some type a_{n+1}. If $M : \sigma$ is a $(\Sigma, c_{n+1} : a_{n+1})$-term, then $\lambda c_{n+1}.M$ is a Σ-term. But by the inductive hypotheses,

$$\Sigma; K, C \vdash [\![\lambda c_{n+1}.M,\ (\lambda c_1 \ldots \lambda c_n \lambda c_{n+1}.M)t_1, \ldots, t_n : a_{n+1} \rightarrow \sigma]\!]$$

holds on the shorter signature. This means, renaming bound variables if necessary,

$$\Sigma; K, C \vdash \forall x \forall y([\![x,\ y : a_{n+1}]\!] \Rightarrow [\![M[x/c_{n+1}],\ M[t_1/c_1, \ldots, t_n/c_n, y/c_{n+1}] : \sigma]\!])$$

Furthermore, this also holds under the extended signature $\Sigma, c_{n+1} : a_{n+1}$. Now by instantiating x with c_{n+1} and y with t_{n+1},[4] we have

$$\Sigma, c_{n+1} : a_{n+1}; K, (C, [\![c_{n+1},\ t_{n+1} : a_{n+1}]\!]) \vdash [\![M,\ M[t_1/c_1, \ldots, t_n/c_n, t_{n+1}/c_{n+1}] : \sigma]\!],$$

and this completes the proof.

□

Another way to view the conclusion of this theorem is that

$$\Sigma; K, C \vdash\ [\![M,\ M\theta : \tau]\!]$$

if and only if θ is the substitution $[t_1/c_1, \ldots, t_n/c_n]$.

The practical result of this theorem is that when a copy clause formula $[\![x,\ t : \sigma]\!]$ is assumed, it *encodes* the substitution of t for x, where x is a variable *of any order* in some object-level formula. And when a copy clause is solved (in the sense of a logic programming goal or query), it *applies* these encoded substitutions. Substitution (or β-conversion) is thus captured explicitly as logic programming.

The following critical corollaries follow immediately from the theorem, and formalizes the relationship between $\beta\eta$-conversion and copy-clauses.

Corollary 2.5.1 *(Equality)*
Let signature $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m\}$ *and let* $K = \{[\![k_i,\ k_i : s_i]\!] \mid 0 \leq i \leq m\}$. *For* Σ*-terms* M *and* N *of type* τ,
$\Sigma; K \vdash\ [\![M,\ N : \tau]\!]$ *if and only if* M *is* $\beta\eta$*-convertible* ($=_{\beta\eta}$) *to* N.

[4] Since by "instantiating" we mean applying the **cut** rule on the above sequent and the conclusion of a $\forall$-left rule on the instantiated formula, cut-elimination is also a key factor in this proof.

Corollary 2.5.2 *(Substitution)*

Let signature $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m\}$ *and let* $K = \{[\![k_i,\ k_i : s_i]\!] \mid 0 \leq i \leq m\}$*. For* Σ*-terms* M *of type* $\sigma \rightarrow \tau$*,* t *of type* σ*,and* N *of type* τ*,*

$\Sigma; K \vdash\ \forall x([\![x,\ t : \sigma]\!] \Rightarrow [\![M\ x,\ N : \tau]\!])$ *if and only if* $(M\ t)$ *is* $\beta\eta$*-convertible to* N*.*

Corollary 2.5.3 *(Unification)*

Let signature $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m, c_1 : a_1, \ldots, c_n : a_n\}$*, and let* $K = \{[\![k_i,\ k_i : s_i]\!] \mid 0 \leq i \leq m\}$*. For* Σ*-terms* M *and* N *of some type* τ*,*

$\Sigma; K \vdash\ \exists T_1 : a_1, \ldots, T_n : a_n \exists R : \tau([\![c_1,\ T_1 : a_1]\!], \ldots, [\![c_n,\ T_n : a_n]\!]) \Rightarrow ([\![M,\ R : \tau]\!] \wedge [\![N,\ R : \tau]\!])$ *if and only if for some substitution* θ*,* $M\theta = N\theta$*. (In fact,* $\theta = [t_1/c_1, \ldots, t_n/c_n]$*, where* $t_1, \ldots, t_n$ *are the existential witnesses to* $T_1, \ldots, T_n$*.)*

The first corollary says that copy clauses formulates $\beta\eta$-equality in the presence of empty substitutions. The second corollary, which can obviously be extended to substitute for multiple terms as in $(M\ t_1, \ldots, t_n)$, allows copy clauses to simulate $\beta\eta$-conversion. But more importantly, it suggests that arbitrary β-redices of the form $(M\ t_1, \ldots, t_n)$ can be replaced by a much simpler form, namely $(M\ x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ is a list of distinct λ-bound variables. These are exactly L_λ terms. They greatly simplify the process of normalization and unification for λ-terms, leading to the programming language L_λ. The last corollary is the most interesting for our meta-programming endeavors and will be discussed at length below. Note that the copy clause formulation does not restrict substitutions to be idempotent. This property must be maintained by other conditions. In the Substitution corollary for example, since x is a new eigenvariable, it follows immediately that it can not appear free in t.

It is important to note the presence of η-conversion in these results. Technically, it is evident from the proof of the Substitution Encoding theorem the crucial role of η-conversion. Intuitively, one can simply note that the Equality corollary clearly assumes extentionality, i.e, $f = g$ iff for arbitrary $x, y,\ x = y$ implies $(f\ x) = (f\ y)$, which is exactly the definition of $[\![f,\ g : \sigma \rightarrow \tau]\!]$. In extensions of copy clauses beyond simple types, such as the *dependent-type copy clauses* of [32], it is important for $\beta\eta$-reduction to remain confluent and strongly normalizing in the extended type theory. Simple type copy clauses suffice for our purposes here.

We stress that the critical implication of the above theorem is that all constructs of the object-level formalism, including constant, function and predicate symbols, and especially *variables,* are to be represented by *constant symbols* (of possibly higher-order) in the meta-logic. Specifically, the "fixed" $k_1, \ldots, k_m$ constants are to represent object-level constant, function and predicate symbols, etc..., and the $c_1, \ldots, c_n$ constants are to represent object-level variables. This prevents the misrepresentation of object-level semantics as is possible when meta-logic variables are used directly.

Sometimes we may wish to represent two substitutions such as $[t_1/c]$ and $[t_2/c]$. Multiple substitutions can be encoded by different *copy* predicates, such as *acopy* and *bcopy*. The Substitution Encoding theorem obviously holds for any such predicate. In practice, however, these additional predicates are seldom needed. Recall that the corollaries of the Substitution Encoding theorem assume that $[\![k_i,\ k_i : \sigma]\!]$ holds for each object-level non-logical constant k_i. Each k_i is thus invariant under any substitution. For the constants $c_1, \ldots, c_n$ representing object-level variables, however, copy clauses are introduced *dynamically* under the scope of intuitionistic implication ($\Rightarrow$). Thus separate substitutions can be encoded inside independent scopes.

2.3 Object-level Matching and Unifiability

The corollaries of the Substitution Encoding theorem almost-immediately suggest logic programming formulations for equality, matching, and unifiability.

- Equality is modeled by empty substitutions, i.e, no $c_1, \ldots, c_n$ constants. The copy clauses $K = \{\ldots, [\![k_i,\ k_i : s_i]\!], \ldots\}$ for "fixed" constants $k_1, \ldots, k_m$ can be generated from their type declarations. Under the K clauses, $[\![M,\ N : \sigma]\!]$ is provable if and only if $M =_{\beta\eta} N$.

- For matching, given a logic program with the appropriate copy clauses for the fixed constants, i.e, $\{[\![k_i,\ k_i : s_i]\!] \mid 0 \leq i \leq m\}$, the query

  $$\exists T_1 \ldots \exists T_n([\![c_1,\ T_1 : a_1]\!] \Rightarrow \ldots \Rightarrow [\![c_n,\ T_n : a_n]\!] \Rightarrow [\![M,\ N : \sigma]\!])$$

  succeeds only if there is a substitution θ (namely $[T_1/c_1, \ldots, T_n/c_n]$) such that $M\theta = N$.

- Unifiability can be similarly determined by

  $$\exists R \exists T_1 \ldots \exists T_n ([\![c_1,\ T_1 : a_1]\!] \Rightarrow \ldots \Rightarrow [\![c_n,\ T_n : a_n]\!] \Rightarrow ([\![M,\ R : \sigma]\!] \wedge [\![N,\ R : \sigma]\!])),$$

  which succeeds if and only if an answer substitution can be found for $T_1, \ldots, T_n$. All ground instances of this substitution are thus solutions to the unification problem.

The correctness of these specifications is immediate by the corollaries.

In all these formulations, the inherent unification procedure of the logic programming language is used in a slightly indirect manner. (We will refer to this level of unification as meta-level unification). Operationally, one can think of the copy clause assumptions

$$[\![c_1,\ T_1 : a_1]\!], \ldots, [\![c_n,\ T_n : a_n]\!]$$

as "lifting" the closed or ground representation into a non-ground representation, which can take advantage of meta-level unification directly. This is not due to some programming trick. It follows from the natural interpretation of existential quantification as specified in the Unification corollary, and to the expressiveness of hereditary Harrop formulas. Thus the copy-clause formulation of *object level* unifiability and matching is a *safe* way to use the underlining unification procedure of the logic programming language.

2.4 WOLOG: A Technique for Automated Theorem Proving

In this section we will give a sample application of object-level matching. There are certainly other applications for matching. Here we introduce a technique especially suitable for meta-logic programming given the previous work of Felty on specifying theorem provers [12]. The techniques given here can also stand independently of logic programming and are potentially applicable to a varity of tactic-style theorem-provers such as NuPrl [4] and HOL [17].

Often the same proof of a conjecture can also be used as the proof of similar conjectures. "Without loss of generality" (wolog) is a statement that is often made when the proof of one case of a proposition is in some sense symmetrical to the proof of another case. Such a statement in general, however, can be made vaguely, even carelessly. We shall formalize one form of this notion that has a sound logical foundation. This is an interesting example of *reusing* proofs in that no analogous proof is actually carried out: the existence of the symmetrical proof follows automatically by consideration of the syntactic structure of the two cases.

wolog-right

The use of a wolog-like argument is evident in proving goals such as

$$(A \wedge B \rightarrow B \wedge A) \wedge (B \wedge A \rightarrow A \wedge B)$$

That the second case of this proposition follows from the first is evident by observation, without need to consider the actual structure of the proof.

The formal schema for rule wolog-right (in intuitionistic sequent calculus form) is as follows, which we will explain presently:

$$\frac{\Gamma \vdash \Gamma_v\theta \quad \Gamma \vdash A}{\Gamma \vdash A \wedge B}\; wolog-right$$

where $A_v\theta = B$ (B is an instance of A).

The meaning of this rule is as follows. This part of the discussion is independent of copy clauses and meta-programming. Assume that for each function and predicate symbol, as well as each variable (in some arbitrary domain) q, there exists an uniquely associated variable q'. This variable is intended to "raise" q to a *parameter* subject to substitution. Define T_v as the term or formula T in which all occurrences of symbols s in set v are replaced by s'.

In the rule above, Γ_v and A_v stand for terms Γ and A where the non-logical parameters in set v are considered as the domain of substitution θ (we also require that θ does not contain any parameter variables in its range). If symbols in v also occur in B, they are effectively "frozen" - i.e., regarded as constants not subject to substitution. It is in this way that we can say that $(A \wedge B)$ is an instance of $(B \wedge A)$. Note that unification can not be used to define this notion of symmetry since that would produce $(A \wedge A) \rightarrow (A \wedge A)$.

The *intuitive* correctness of wolog-right is understood as: every proof step in the case for $\Gamma \vdash A$ is transformable to a proof step of $\Gamma \vdash B$. That is, if information concerning A in Γ is referred to in the proof, then that reference could also be made for B via θ.

Furthermore, this transformation on Γ from a source of information about A to a source of information about B is "safe" in the sense that $\Gamma_v\theta$ still follows from Γ.

The formal correctness of wolog-right is based on the following:

$$\dfrac{\Gamma \vdash A \qquad \dfrac{\Gamma \vdash \Gamma_v\theta \quad \Gamma_v\theta \vdash A_v\theta(= B)}{\Gamma \vdash B}\, CUT}{\Gamma \vdash A \wedge B}\, \wedge - right$$

All that remains to be established is that $\Gamma \vdash A$ implies $\Gamma_v\theta \vdash A_v\theta$. The proof of this is by induction on the height of proofs for all v and θ. In fact, for first order logic the result follows from the "adequacy" result for representing first-order logic in the LF logical framework [22] and the Transitivity property of LF.

The practical justification of having wolog-right as a rule are due to:

1. No higher than second order terms are needed to represent first order formulas

2. First and second order matching, which are used to determine if θ exists, are both decidable.

3. Only the proof of A needs to be carried out.

The goal $\Gamma \vdash \Gamma_v\theta$, however, might not be trivial, and may in fact involve more work than proving $\Gamma \vdash B$. Moreover, the optimal domain of substitution v is hard to extract. We therefore define the following restricted but more practical version of wolog-right:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \wedge B}\, wolog - right$$

where $A_v\theta = B$, $\Gamma_v\theta = \Gamma$, *and* $v =$ *the set of all non-logical constants in* A.

In appropriate environments (such as an interactive theorem prover), v can also be given by the user. Note that a particularly common case of this restriction is when no symbol in v occurs free in Γ.

This eliminates the subgoal $\Gamma \vdash \Gamma_v\theta$, which may be overly-complicated or even false. This restriction allows us to specify the wolog rule in λProlog purely declaratively using object-level matching. Logic variables obviously can not be used to represent parameters because of the need to "freeze" parameters on one side of a matching problem while treating them as variables on the other side. We need to use constants symbols and object-level substitutions in order to implement the technique correctly. Following the manner of specifying sequent-calculus theorem provers of [12], the following program can be given. Here, (seq G A) represents the sequent $\Gamma \vdash A$. Given a list of parameters Params, (wolog Params) specifies a *tactic* that transforms one sequent goal to a simpler subgoal. We will use copy for a generic first-order copy clause regardless of type. The program easily generalizes to allow higher-order parameters, since copy clauses are valid at all orders.

```
wolog Params (seq G (and A B)) (seq G A) :-
  seq-formula G G2,
  wolog2 Params G2 (and A B).

wolog2 (P::Ps) G C :- copy P X => wolog2 Ps G C.
wolog2 nil G (and A B) :- copy A B, copy G G.
```

seq-formula merely converts a list of formulas to a conjunction of formulas (i.e., to one formula). Note that (copy G G) holds only if the substitution is idempotent on G, i.e., $\Gamma_v\theta = \Gamma$.

wolog-left

Just as wolog-right corresponds to the $\wedge$-right rule, wolog-left below corresponds to the $\vee$-left rule in an analogous fashion:

$$\frac{A, \Gamma \vdash C \quad \Gamma_v\theta \Rightarrow C_v\theta \vdash \Gamma \Rightarrow C}{A \vee B, \Gamma \vdash C}\ wolog - L$$

where $A_v\theta = B$.

The validity of this schema is similar to that of wolog-right once we note the fact that $\Gamma, A \vdash B$ is provable if and only if $\Gamma \vdash A \Rightarrow B$ is provable:

$$\frac{A, \Gamma \vdash C \qquad \dfrac{A_v\theta(=B) \vdash \Gamma_v\theta \Rightarrow C_v\theta \quad \Gamma_v\theta \Rightarrow C_v\theta \vdash \Gamma \Rightarrow C}{A_v\theta(=B), \Gamma \vdash C}\ Cut}{A \vee B, \Gamma \vdash C}\ \vee L$$

We can also require that $(\Gamma, C)\theta = \Gamma, C$ as a restriction to eliminate the second premise. A special instance in which this rule can be useful is when C is of the form $\exists x.D(x)$. Then for some term t, let the parameters v contain some constants in t, but *not* in D, so that

$D(t)\theta = D(t\theta)$. Then the following inference would be valid:

$$\dfrac{\dfrac{A, \Gamma \vdash D(t)}{A, \Gamma \vdash \exists x.D(x)}\ \exists R \qquad \dfrac{\dfrac{\Gamma \vdash \Gamma_v\theta \quad A_v\theta, \Gamma_v\theta \vdash D(t_v\theta)}{(B =)A_v\theta, \Gamma \vdash D(t_v\theta)}\ Cut}{B, \Gamma \vdash \exists x.D(x)}\ \exists R}{A \vee B, \Gamma \vdash \exists x.D(x)}\ \vee L$$

Since $A, \Gamma \vdash D(t)$ implies $(B =)A_v\theta, \Gamma_v\theta \vdash D(t_v\theta)$ $(= D(t)_v\theta)$, we therefore have the following special instance of $wolog - L$:

$$\dfrac{A, \Gamma \vdash D(t) \quad \Gamma \vdash \Gamma_v\theta}{A \vee B, \Gamma \vdash \exists x.D(x)}\ wolog - \exists, \qquad A_v\theta = B,\ D(y)\theta = D(y)\ for\ y \notin v.$$

We can also choose to add the restriction that $\Gamma_v\theta = \Gamma$ (the same restriction suggested for wolog-right), which will eliminate the premise $\Gamma \vdash \Gamma_v\theta$ as well.

Yet another variation is

$$\dfrac{A, \Gamma \vdash C \quad \Gamma, C_v\theta \vdash C}{A \vee B, \Gamma \vdash C}\ wolog - L' \qquad A_v\theta = B,\ \Gamma_v\theta = \Gamma$$

which has a similar justification ($\Rightarrow L$ with $\Gamma \vdash \Gamma_v\theta$).

The choice of which variation to use must be made by other heuristics. However, since most theorem provers allow user interaction, the human user may choose the appropriate tactic. Say one needs to prove that $\forall m \forall n \exists b.b < m \wedge b < n$ under the usual axioms for integers. The human user mentally make the argument *"without loss of generality, assume $m \leq n$."* This argument means discharging the axiom $\forall x \forall y.x \leq y \vee y \leq x$ and having to solve the goal

$$\Gamma, m \leq n \vee n \leq m \vdash \exists b.b < m \wedge b < n.$$

At this point the user may choose rule $wolog - L'$ with $v = \{m, n\}$. Now $(m \leq n)_v\theta = (n \leq m)$ with $\theta = [n/m_v, m/n_v]$. Since m, n do not appear free in the context Γ, $\Gamma_v\theta = \Gamma$ holds trivially. The derivation can now be as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, m \leq n \vdash (m-1) < m \wedge (m-1) < n}}{\Gamma, m \leq n \vdash \exists b.b < m \wedge b < n}\ \exists R \qquad \dfrac{}{\Gamma, \exists b.b < n \wedge b < m \vdash \exists b.b < m \wedge b < n}\ \textbf{AutoTactic}}{\Gamma, m \leq n \vee n \leq m \vdash \exists b.b < m \wedge b < n}\ wolog - L'$$

That is, the user presumably has decided that his/her theorem prover can prove the goal

$$\Gamma, \exists b.b < n \wedge b < m \vdash \exists b.b < m \wedge b < n.$$

automatically (given the commutativity of $\wedge$ perhaps). Now the only remaining goal is

$$m \leq n \vdash \exists b.b < m \wedge b < n,$$

which the user can prove by instantiating b with $m-1$ (this subproof can be still be quite involved, depending on how integer arithmetics is encoded and axiomatized in the theorem prover). The symmetrical proof for the case $n \leq m$ need never be constructed.

These wolog-left rules can be given similar copy clause specifications. For example, $wolog - \exists$ can be implemented by the following. The constructor `andg` (borrowed from [12]) declares a pair of subgoals left to be proved.

```
wolog-E V (seq ((or A B)::G) (exist D))
         (andg (seq (A::G) (D T)) (seq G GV)) :-
  seq-formula G G2,
  wolog-E2 V G2 GV (or A B) .

wolog-E2 (V::Vs) G GV (or A B) :- copy V XV => wolog-E2 Vs G GV (or A B).
wolog-E2 nil G GV (or A B) :- copy A B, copy G GV.
```

Chapter 3

Unification and Generalization

While matching and unifiability already suffice for a range of applications, there remains a class of problems which the tools of the last chapter do not address. These are problems that require expressions with free-variables to be *generalized* into quantified, closed expressions. This chapter addresses this problem.

3.1 The Necessity of Object-Level Unification

So long as object-level free variables are represented by meta-level constant symbols, generalization of free-variables into quantified expressions is not a difficulty. For example, let `all` be of type `term -> form -> form`, just as in section 1.4.1. We can also use `exists`; the exact object level meaning is not relevant here. The intent is to create a closed expression. A `close` predicate can be defined as follows. Recall that the Substitution Encoding theorem and its corollaries assume that the clauses $[\![k_i,\ k_i : \sigma_i]\!]$ for each object-level fixed constant k_i are always visible. Copy clauses representing substitutions for object-level variables can be introduced dynamically under the scope of $\Rightarrow$.

```
close (V::Vs) A (all B) :-
  pi u\ (copyterm V u => close Vs A (B u)).
close nil A B :- copyform A B.
```

Given a list `L` of symbols representing free variables, `(close L A B)` implies `B` is `A` with all variables in `L` quantified by `all`, i.e, closed by meta-level λ-abstraction. This technique will be used throughout this thesis.

Certain object-level operations, however, require generalized solutions to be formed through unification. Recall that in the unifiability formulation of the previous chapter, the meta-level constants $c_1, \ldots, c_n$ are "copied" into meta-level free variables $T_1, \ldots, T_n$ so that meta-level unification can be used. The meta-level unification algorithm then finds the most general solution for $T_1, \ldots, T_n$. The problems is that some *meta-level* free variables will remain in this solution. We can only say that for all ground instances $t_1, \ldots, t_n$ of $T_1, \ldots, T_n$, respectively, $[t_1/c_1, \ldots, t_n/c_n]$ represents an object level unifier. But to form the object-level *most general* unifier, the logic variables in the meta-level unifier must be replaced by meta-level constant symbols that represent object-level variables, as opposed to any ground instance. To illustrate this point, consider the example of unifying the object-level terms $f(u, w)$ and $f(f(a, w), v)$ under the signature

$$\Sigma = \{a : term, f : term \rightarrow term \rightarrow term, u : term, v : term, w : term\},$$

and clauses

$$K = \{[\![a,\ a : term]\!], [\![f,\ f : term \rightarrow term \rightarrow term]\!]\}.$$

That is, the Σ-constants u, v, w represent object-level variables while the constants a and f respectively represent an object-level constant symbol and a binary function symbol. To determine if $(f\ u\ w)$ is unifiable with $(f\ (f\ a\ w)\ v)$, we can solve the goal

$$\Sigma; K \vdash \exists R \exists U \exists V \exists W ([\![u,\ U : term]\!] \Rightarrow [\![v,\ V : term]\!] \Rightarrow [\![w,\ W : term]\!]$$

$$\Rightarrow ([\![(f\ u\ w),\ R : term]\!] \wedge [\![(f\ (f\ a\ w)\ v),\ R : term]\!]),$$

which in λProlog (L_λ) syntax is equivalent to

```
?- copyterm u U => copyterm v V => copyterm w W =>
       (copyterm (f u w) R, copyterm (f (f a w) v) R).
```

This query will yield the answer substitution that binds the meta-logic variables `U` to `(f a V)` and `W` to `V`. This corresponds to a substitution that maps object-level variables `u` to `(f a V)`, `v` to `V`, and `w` to `V`. Unfortunately, the latter mapping does not quite describe an object-level substitution since the range of this mapping contains meta-level open terms.

$$\textbf{Proj} \qquad \overline{H \vdash x : T}, \quad x : T \in H$$

$$\textbf{abs} \qquad \frac{H, x : s \vdash M : t}{H \vdash \lambda x.M : s \rightarrow t}$$

$$\textbf{app} \qquad \frac{H \vdash M : s \rightarrow t \quad H \vdash N : s}{H \vdash (M\ N) : t}$$

$$\textbf{let} \qquad \frac{H \vdash M : S \quad H, x : S \vdash N : t}{H \vdash \texttt{let}\ x = M\ \texttt{in}\ N\ :\ t}$$

$$\Pi\textbf{-Intro} \qquad \frac{H \vdash M : T \quad a \text{ not free in } H}{H \vdash M : \Pi a.T}$$

$$\Pi\textbf{-Elim} \qquad \frac{H \vdash M : \Pi a.T}{H \vdash M : T[s/a]}$$

(s, t represent unquantified types; S, T represent arbitrary type schemes)

Figure 3.1: The Damas-Milner Calculus

We would like to form the substitution mapping `u` to `(f a v)`, `v` to `v` and `w` to `v` (or some variable-renaming of this substitution). This process of generating object-level most general unifiers we shall call *object-level* unification.

Resolution, where two universally quantified formulas are resolved, yielding another universally quantified formula is an obvious example requiring object-level unification [1]. Logic variables can not be used since they may appear in the resolvent, which then can not be closed. The Knuth-Bendix completion algorithm [29] for term-rewriting systems also illustrates this point well. Variables in rewrite rule-schemes have universal interpretations. Once a critical pair is found by unification and resolved by some fixed well-ordering, a new rule-scheme is formed. Variables in this new rule must also retain an universal interpretation, lest the meaning of the original rewrite system be skewed. If logic variables were used to determine critical pairs by Prolog's internal unification, their status as existentially quantified variables will be retained in the new rule-scheme. It is also likely that the resulting rewrite system will loose strong-normalization, since a logic variable can be rewritten to anything.

Yet another example of the need for explicit generalization is polymorphic type inferencing (for functional programming languages such as ML).

[1] This is sometimes referred to as forward resolution, to distinguish it from the backward resolution in Prolog interpreters.

Figure 3.1 contains the *Damas-Milner* proof system or calculus [7] for typing basic expressions in ML. The symbol H is a *type environment* containing type assignment to variables (or constants) of the form $x : T$. One might infer that from these rules, type inferencing can be defined directly. Assume that `tm` is the meta-level type for object-level terms (or ML program expressions) and `poly` is the type for object-level types. The constructors `abs` and `app` represent object level λ-abstraction and application respectively (as in section 1.4.1). Assume that `integer` and `real` are primitive types and `arr` is the object-level arrow type constructor. It is important to emphasize the distinction between meta-level types and object-level types here. Object-level types are represented in the meta-logic as *terms.* These terms are assigned meta-level type `poly`. For simplicity we will not make the distinction between (object-level) types and type schemes at the meta-level (so `poly` categorizes both). The following program is an attempt at defining type inference:

```
kind poly type.
kind tm type.
type integer  poly.
type real     poly.
type arr      poly -> poly -> poly.
type abs      (tm -> tm) -> tm.
type app      tm -> tm -> tm.
type zero     tm.

typing zero integer.
typing (app M N) T :- typing M (arr S T), typing N S.
typing (abs M) (arrow S T) :- pi x\ (typing x S => typing (M x) T).
```

Assuming the above clauses, the query `typing (abs x\zero) T` will succeed with mgu

```
T = (arr S integer)
```

where `S` is some new logic variable. This can not be used to represent a proper polymorphic type since not both `(arr integer integer)` and `(arr real integer)` can be instances of `(arr S integer)` at once. Furthermore, consider the ML `(let (y\(B y)) X)` construct that binds the local variable `y` to term `X`. The type of `X` must be universally generalized before the type of the `let` expression can be derived. The seemingly obvious program

```
type let   (tm -> tm) -> tm -> tm.
```

```
typing (let B X) T :-
  typing X S, pi x \ (typing x S => typing (B x) T).
```

will not be able to derive a type for `(let (abs y\y) (x\(app x x)))` since `x` can not have both `A` and `(arr A A)` as types.

In any case, the above program can at most be called a *type checker,* when `T` is instead some closed type expression. In [19], Hannan specified type inferencing as logic programming, up to the limitations described here. In [21], Harper defined several type inferencing proof systems in the Edinburgh LF logical framework [22], which has a close relationship with λProlog. Unification, however, was not present in any of his formulations, and none of the formulations can lead directly to a practical logic program. One would have to non-deterministically *guess* the correct form of type expressions.

The common thread through all these examples is the need for object-level unification. *It would allow the construction of most general solutions independently of the meta-logic's semantics of variables.* Logic programming specifications of object-level unification is therefore the backbone of this chapter.

3.2 Various Routes to Unification

In cases where logic programmers have encountered this problem, two approaches were generally taken:

1. Force the issue: i.e, add some built-in, extra-logical predicate to the programming language that changes expressions with meta-level free variables into closed, quantified expressions. These "hot-wiring" techniques, see [9, 26, 15], vary in sophistication and their effect on the declarative meaning of programs is problematic.

2. Another approach is to design a new logic programming language for the specific purpose of meta-programming in the ground representation, with a specialized semantics for dealing with unification (and resolution) at the object- and meta-levels. This is the approach of [24, 6]. These languages, in particular Gödel [24], require a

large library of built-in operations (such as substitution). There are also no known attempts to extend these experiments beyond first-order Horn clauses (which are insufficient for higher-order abstract syntax). Furthermore, these approaches conflict with the aim of this thesis, which is to show that general meta-programming is possible within a higher-order logic programming language *without* specialization for that express purpose.

In the next sections we will solve this problem of *object-level unification* in two different ways. The first approach is completely within the meta-logic (L_λ), while the second approach uses the common Prolog construct `!` (cut) in a very simple way.

3.2.1 The Algorithmic Approach

One possible solution to this problem is to completely re-implement some known unification algorithm - i.e, build a "meta-circular" unification program. The algorithm we implement here is the non-deterministic one of Martelli and Montanari [33]. In this algorithm, unification is presented as solving a set of difference pairs, which are transformed using rules similar to the Gaussian elimination method for solving systems of linear equations. When a unique solution t is found for a variable x (provided x is not free in t), the pair (x, t) is considered *solved.* This pair is then *eliminated* from the set of difference pairs. The algorithm succeeds when all difference pairs are solved. The final set of elminated solved pairs represents the most general unifier. This unifier is also idempotent.

When implementing a non-trivial algorithm such as unification, limitations of logic programming often leads to the use of extra-logical constructs. Since negation (as failure) is not generally available, there is often no way to implement mutually exclusive rules without cut (`!`). While it is easy to define `member` declaratively, defining `remove` (removing *all* elements of a certain form from a list) becomes much more difficult. Similarly, though the subset relation for arbitrary finite sets is easily definable, proper-subset is not. As a result, mundane operations such as eliminating duplicate entries in a list are not available to logic programmers who do not wish to use extra-logical constructs. Fortunately, by using the ground representation (in the context of higher-order abstract syntax), some of these problems can be solved.

Several aspects of the unification algorithm facilitate a declarative logic programming implementation:

- The algorithm is specified by a set of mutually exclusive transformations (relative to a difference pair).
- The order in which individual difference pairs are solved does not matter.
- The condition for the termination of these transformations is clear (when the difference pair list is nil).

These non-deterministic features mean that we can always "concentrate" on solving the first difference pair, until all pairs are solved. The chief difficulty in specifying this alorithm is to identify variables and rigid terms, and to distinguish distinct variables. We do this by imposing an ordering relation on the constant symbols representing free variables. (Recall there is no way to define such an ordering to open terms.) The following clauses impose an ordering relation `vrd` on a list of distinct variables before solving the goal `G`:

```
order (V::Vs) G :- (pi X\ (vrd V X :- member X Vs)) => order Vs G.
order nil G :- G.
```

In the scope of G, then, variables can be distinguished by

```
distinct-vars X Y :- vrd X Y; vrd Y X.
```

We give the unification algorithm for a sample object-level signature consisting of

```
kind poly     type.
type integer  poly.
type arr      poly -> poly -> poly.
```

where `poly`, recall, categorizes *terms* that can be interpreted to mean object-level types and type schemes. This suffices to demonstrate a wide range of examples, and will be needed to implment the ML type inferencers of section 3.5. The formulation easily generalizes to other signatures.

Although copy clauses encode substitutions, sometimes it is desirable to have a more explicit representation of object-level substitutions. The unification algorithm will build substitutions (mgu) using the following declarations.

```
kind substitution type.
type emp substitution.
type sub poly -> poly -> substitution -> substitution.
```

A structure of the form (sub a S (sub b T emp)) will represent the substitution $[S/a, T/b]$.

Certain basic operations on substitutions are needed, such as sub_apply, which applies a substitution to a term (of type poly). Its definition is

```
type sub_apply poly -> substitution -> poly -> o.
sub_apply A emp A.
sub_apply A (sub V T Ss) B :-
  ((copypoly V T, (pi Y\ (copypoly Y Y :- distinct-vars Y V))) =>
    copypoly A C), sub_apply C Ss B.
```

The correctness of this definition again follows from the Substitution Encoding theorem. The only extra complication is that we must apply the substitution for one variable at a time, because not every variable may be in the domain of the substitution, and so we add the trivial substitution (copy v v) to the domain. We also require the following predicates, which all have straightforward definitions:

- diff-subst Sub D1 D2 applys a substitution Sub to a list of difference pairs D1, yielding D2.
- sub-subst Sub S1 S2 applys a substitution Sub to all terms (in domain and range) of another substituion S1, yielding S2.

Difference pairs are represented using structures of the form

```
((diff A B)::(diff C D)::(diff E F)::nil)
```

The unification program is given by the following:

```
kind difference type.
type unify1 (list poly) -> (list poly) -> poly -> poly -> substitution -> o.
type unify (list poly) -> poly -> poly -> substitution -> o.
type diff poly -> poly -> difference.
type rigid poly -> o.
type var poly -> o.
```

```
type occur-check poly -> poly -> o.
type transform (list difference) -> (list difference) -> substitution
               -> substitution -> o.
type copypoly poly -> poly -> o.
type unify0 (list difference) -> substitution -> substitution -> o.

copypoly integer integer.
copypoly (arr A B) (arr C D) :- copypoly A C, copypoly B D.
rigid integer.
rigid (arr A B).

transform ((diff X X)::S) S Sub Sub :- var X.
transform ((diff integer integer)::S) S Sub Sub.
transform ((diff (arr A B) (arr C D))::S)
          ((diff A C)::(diff B D)::S) Sub Sub.
transform ((diff X T)::S) S2 Sub (sub X T Sub2) :-
  var X, occur-check X T,
  diff-subst (sub X T emp) S S2,
  sub-subst (sub X T emp) Sub Sub2.

occur-check X integer.
occur-check X (arr A B) :- occur-check X A, occur-check X B.
occur-check X Y :-  distinct-vars X Y.

unify (V::Vs) A B Sub :-
  (var V, (pi Z\ (vrd V Z :- member Z Vs))) => unify Vs A B Sub.
unify nil A B Sub :-
  unify0 ((diff A B)::nil) emp Sub.

unify0 ((diff A B)::R) Sub Sub2 :-
  var B, rigid A, unify0 ((diff B A)::R) Sub Sub2.
unify0 (D::Ds) Sub Sub3 :-
  transform (D::Ds) S Sub Sub2, unify0 S Sub2 Sub3.
unify0 nil Sub Sub.
```

Given (object-level) terms `A` and `B` whose free variables (constant symbols) are in the list `Vs`, `unify Vs A B U` will succeed only if `U` represents a most general unifier of `A` and `B`. The first clause of `unify` orders variables so they can be distinguished apart. It designates object-level variables with the meta-logic predicate `var`. The second clause then creates the initial difference-pairs list using the two terms. The first clause of the predicate `unify0` switches the order of a rigid-variable pair. The second clause recursively applies some *transformation* to the first difference pair until the difference-pairs list is nil. The first three clauses of `transform` implements *term decomposition* [2]. The last `transform` clause implements *variable elimination.*

The correctness of this program is of course due to the correctness of the known unification algorithm. It is straightforward to show that these clauses implement the algorithm correctly. We only need to note that every successful `unify0` goal implies that the set of difference pairs (and substitution) is updated by some transformation, unless there are no more difference pairs.

Variations of the program

The substitution constructed by this algorithm does not contain trivial substitutions of the form `(sub x x ..)`. In using copy clauses to encode substitutions, however, these trivial mappings must also be made explicit. Each variable must be "copied" to something; we can not simply assume that variables not in the "support" of the unifier stay invariant. We therefore require that the substitution constructed contains entries for every variable in the given (finite) domain. This can be accomplised by using the `add-trivials` clauses below.

```
add-trivials nil S S.
add-trivials (V::Vs) S S2 :- in-domain V S, add-trivials Vs S S2.
add-trivials (V::Vs) S (sub V V S2) :-
  not-indomain V S, add-trivials Vs S S2.

in-domain V (sub V X Y).
in-domain V (sub U X Y) :- in-domain V Y.
```

[2] We've made the case of a pair of identical variables part of term decomposition. This choice was largely arbitrary.

```
not-indomain V emp.
not-indomain V (sub U X Y) :- (vrd V U; vrd U V), not-indomain V Y.

unify Vs A B U :- unify1 Vs Vs A B U.

unify1 O (V::Vs) A B Sub :-
  (var V, (pi Z\ (vrd V Z :- member Z Vs))) => unify1 O Vs A B Sub.
unify1 O nil A B Sub :-
  unify0 ((diff A B)::nil) emp Sub1,
  add-trivials O Sub1 Sub.
```

Another shortcomming of this program is that while mgu's are equivalent up to renaming, this program will only construct one renaming. But this is a property of the original algorithm itself, and not of the implementation here. The particular renaming is determined by the order in which difference pairs are elminated. With the program as it is, we can not say that `unify V A B U` succeeds *if and* only if `U` is a mgu of `A` and `B`. If we must recognize other renamings, we can eliminate the condition `rigid A` in the first clause of `unify0`, so that multiple solutions are yielded by the program. We must also permute the list of difference pairs before applying each transformation. Permutation (of a list) is a easily defined predicate in Prolog:

```
permute nil nil.
permute (X::Xs) Y :- permute Xs Ys, insert-somewhere X Ys Y.
insert-somewhere X A (X::A).
insert-somewhere X (A::B) (A::C) :- insert-somewhere X B C.
```

This technique suffices for recognizing all mgu's where the variables in the range of the substitution is a subset of the original domain. In most object-theories, however, a denumerable set of variables is assumed. Fortunately, in first-order unification, it is unnecessary to consider mgu's involving new variables (the same is not true for higher-order or equational unification). In fact, with most applications deriving one renaming of the mgu would suffice. The limitations of the existing program (even without permuting the difference pairs) are superficial in practice, there being no example where the extra ability is needed.

Since we already have a declarative and simple formulation of object-level unifiability, we can check if two terms are unifiable independently of deriving the mgu. This alternative would eliminate the need for an explicit occur-check. The variable ordering would still be needed for adding trivial substitutions. One alternative to this would be to begin with a trivial substitution ($[a/a, b/b, c/c, \ldots]$) for each (object-level) free variable in the given terms, and build the mgu by composing substitutions.

A final possible variation concerns the representation of object level terms. Note that this program is signature-dependent more than just with respect to copy clauses. With the simple signature we've chosen this did not present itself as a problem. In general however, the transformation that decomposes terms (as well as occur-check) will have to be given for each object-level constructor. We can correct this situation by changing the way object-level applications of constructor symbols to arguements is represented. We have so far used (meta-level) expressions such as `(arr integer integer)` to represent the application of the `arr` constructor to two arguments. That is, meta-level application is used for object-level application. We may also choose to represent object-level application more explicitly by using a meta-level constructor `app` that takes a list of arbitrarily many arguements. Expressions of the form `(app arr (integer::integer::nil))` can then be decomposed uniformily, whithout separate clauses for each constructor. For illustrational purposes, this representation is more cumbersome than the one used above. In the next chapter, when we deal with object-level L_λ unification, this alternate representation will be used to greater advantage.

3.2.2 A Short Cut to Unification

The unification program of the previous section is technically "purely declarative" because it did not resort to any extra-logical constructs. It was also defined completely in the L_λ language. There are however, two aspects of the formulation that remain troubling.

First, since the correctness of the program requires the exclusive use of (meta-level) closed terms, only matching occurs at the meta-level. While it is interesting that meta-level matching and uniform provability for HoHH formulas suffice to define an object-level unification procedure, it would be desirable (and faster) if the meta-language's unification procedure can be used to full advantage.

Secondly, although the above program used our facilities of copy clauses and abstract representations as much as possible, conventional data-structures and programming techniques, such as the list of difference pairs, were still needed. This should not be surprising, since the program we gave is not a specification of *unification*, but of a specific *unification algorithm,* which itself calls for these techniques. As a basic principal of logic programming, however, algorithmic detail should be relegated to the underlining interpreter as much as possible; i.e, it should be implicit, not explicit. The algorithmic approach to object-level unification above contrasts with the formulations of object-level matching and unifiability of the previous chapter. These formulations are correct by virtue of a meta-level theorem concerning substitutions which does not refer to any specific algorithmic detail. They are formulations of matching and unifiability as logical concepts. In this sense, they can be considered "even more declarative."

There is no known solution which will completely satisfy both points. In this section, we introduce a technique for object-level unification that uses meta-level unification, but which is not purely declarative in the technical sense because it makes one use of Prolog's cut (`!`). However, because the role of this cut can be made clear, and its relationship to the purely logical part of the method is modular, programs using this technique can still have a highly declarative character. (We will of course make precise this notion of "highly declarativeness.")

Recall that the formulation of unifiability fails to construct most general unifiers exclusively because logic variables may be introduced during meta-level unification. That is, in using the clauses

$$[\![c_1,\ T_1 : \tau_1]\!] \Rightarrow \ldots \Rightarrow [\![c_n,\ T_n : \tau_n]\!] \Rightarrow \ldots$$

we have effectively left the ground representation. In the case for matching, the solution to the logic variables $T_1, \ldots, T_n$ are necessarily closed, and hence represents a proper object-level substitution. For unification in general however, some of these variables $(T_1, \ldots, T_n)$ may remain in the (meta-level) most general solution. If these remaining logic variables can be mapped back to object-level variables, however, then the resulting unifier would in fact be most general. The following clause accomplishes this:

```
setsub X X :- !.
setsub X Y.
```

That is, after the *intermediate* unifier $[T_1/c_1, \ldots, T_n/c_n]$ is constructed, solving `setsub` c_1 $T_1, \ldots,$ `setsub` c_n T_n will complete the construction of the *object-level* most general unifier. The unification program using `setsub` is exceedingly simple:[3]

```
quick-unify (V::Vs) A B ((sub V XV)::Subs) :-
  (copy V XV => quick-unify Vs A B Subs), setsub V XV.
quick-unify nil A B nil :- copy A N, copy B N.
```

The simplicity of this program is due to the fact that the inherent unification procedure of the meta-programming language is used to construct the mgu. The `setsub` goals only change the representation from non-ground to ground.

The cut in the first clause of `setsub` renders the two `setsub` clauses mutually exclusive. Since cut only has meaning in a left-to-right, depth-first implementation of a logic programming interpreter, it is *extra-logical* to higher-order hereditary Harrop formulas and uniform provability. Without the cut, the second clause of `setsub` may also resolve with free logic variables. In this case, `unify Vs A B U` will succeed for *any* unifier `U`, and only the first one will be most general, by virtue of the order in which clauses are resolved with in actual interpreters. Also, because the behavior of `setsub` is dependent on the order of the variables fixed by the list representation, only one possible renaming of the mgu is recognized by the `quick-unify` program. We can alleviate this situation somewhat by permuting the list of variables representing the domain of unification, but in practice this is not necessary.

Since cut does not have a declarative meaning within the meta-logic, technically `setsub` belongs to the "hot-wiring" family of solutions to object-level generalization. However, cut has been used extensively and is well-studied. There are several logical formulations of the control structure of Prolog that characterizes the behavior of cut (see [53] for example). This use of cut is minor in the sense that its role in the unification program is clear. It is modular with respect to the copy-clause formulation of substitution, equality and unifiability. Its effect on the declarativeness of the program is therefore minimal, if "declarativeness" can be taken to mean a measure of clarity of programs. Technically,

[3]This program will construct solutions containing trivial substitutions $[x/x]$, but these can be filtered out with a very slight complication of the above program. In practice, in using the ground representation for variables, these trivial substitutions are needed.

using this `!` is a compromise of our principle of a minimal meta-logic. Except in sections 3.2.2, 3.5.3 and 4.5, however, the thesis can stand completely independent of `!`. On the other hand, adopting one use of this construct can lead to the simplification of many meta-programs. We will spend some time further discussing this use of cut below, after we have proved the correctness of the technique with respect to object-level unification and the depth-first search interpretation of logic programs.

Correctness of the `setsub` Technique

The correctness of the `quick unify` program is due in large part to the correctness of λProlog's own higher-order unification algorithm on simply-typed lambda terms. It is important that the higher-order unification algorithm adds no complication when performing first-order unification. It performs the same operations as standard first-order unification; it always terminates and returns a most general unifier if it succeeds. This should be clear from the presentation of higher-order unification given in [50]. One other crucial property of λProlog's meta-level unification is that only *idempotent* substitutions are constructed as unifiers.

The proof follows the operational account of the unification program given above. All substitutions here are idempotent ($t\sigma\sigma = t\sigma$ for all terms t). To facilitate argumentation, we rewrite the last clause of `quick-unify` as follows:

```
quick-unify nil A B nil :- copy A N, copy B M, N = M.
```

This clause has the same meaning as the original clause.

Let $v_1, \ldots, v_n$ be the constant symbols representing object-level variables in A and B, the terms to be unified. Assume we are at the stage where each constant v has been associated with a logic variable X_v by `copy V XV`, i.e, at the second clause of `quick-unify`. `copy A N (copy B M)` now makes a "copy" of `A (B)` with all v_i constants replaced by their uniquely associated logic variables. This holds by virtue of the Substitution Encoding theorem. `N` and `M` now contain only logic variables.

Next, `N = M` succeeds if and only if there is an idempotent meta-level most general unifier θ_m such that $N\theta_m = M\theta_m$. We want to construct an object-level mgu θ from θ_m (and record it in the fourth argument of `quick-unify`).

Let ϕ be the substitution $[\overline{X_{v_n}}/\overline{v_n}]$ and φ be the substitution $[\overline{v_n}/\overline{X_{v_n}}]$. Trivially, $\phi \circ \varphi$ equals the identity substitution. Let $\theta = \phi \circ \theta_m$, i.e, $\theta = [(X_{v_i})\theta_m/v_i]$ for each constant symbol v_i (θ is the substitution "stored" in the assumptions `copy v1 Xv1, ... , copy vn Xvn`). Then $A\theta = N\theta_m = M\theta_m = B\theta$, so θ is already an unifier for A and B. We will show that after each call to `setsub V XV`, the substitution $\theta\downarrow = \theta \circ \varphi$ is an (object-level) mgu for A and B. That is, we will show that each step carries out an imitation up to renaming of the meta-level unifier.

To prove that $\theta\downarrow$ is initially most general for A and B, we will make the further assumption that the variables in the codomain of θ_m is a subset of $X v_1, \ldots, X v_n$. This assumption is actually unnecessary, as we will discuss later, but it clarifies the proof somewhat, and is consistent with real λProlog interpreters[4]. Thus intially we have

$$\theta\downarrow = \phi \circ \theta_m \circ \varphi.$$

Assume θ' is another (object-level) unifier for A and B. Then since $N = A\phi$ and $M = B\phi$, $\varphi \circ \theta'$ unifies N and M. Since θ_m is most general for N and M, there is some substitution θ'_m such that

$$\theta_m \circ \theta'_m = \varphi \circ \theta'$$

so

$$\theta' = \phi \circ \theta_m \circ \theta'_m$$

but then

$$\theta\downarrow \circ \theta' =$$

$$\phi \circ \theta_m \circ \varphi \circ \phi \circ \theta_m \circ \theta'_m =$$

$$\phi \circ \theta_m \circ \theta_m \circ \theta'_m$$

since θ_m is idempotent, we finally have

$$\theta\downarrow \circ \theta' = \phi \circ \theta_m \circ \theta'_m = \theta'$$

[4]Another way of considering the current argument is to regard logic variables and constant symbols $v_1, \ldots, v_n$ as having equal status in a unifier θ. That is, if $\theta(x) = Y$ for some logic variable Y and $\theta'(x) = y$ for some constant symbol y, then θ is *not* considered more general than θ' (and vice versa). This way, we can argue that the unifier constructed by `quick-unify` differs from the meta-level unifier only up to renaming.

and $\theta\!\downarrow$ is therefore most general for A and B.

We now show that after each call to `setsub`, which possibly binds free logic variables in the codomain of θ to constant symbols, $\theta \circ \varphi = \theta\!\downarrow$ is still an mgu for A and B. The argument is inductive (on n, the number of v_i variables). There are two cases, depending on which clause of `setsub` is resolved with:

If for a constant symbol v in $v_1, \ldots, v_n$, `setsub v XV` fails to resolve with `setsub X X :- !`, then θ does not change at this step. All free logic variables remain free. (This is recorded by setting the substitution of `v` to `Xv` $= Xv\theta_m$ in the fourth argument of `quick-unify`,so that it is indeed an accurate construction of θ). Thus $\theta\!\downarrow$ trivially remains an mgu.

If the first clause of `setsub` succeeds for `setsub V XV`, then $Xv\theta_m$ *was* still an uninstantiated logic variable ($Xv\theta_m$ can not be v since N and M contains only logic variables). In this case $Xv\theta_m$ is now bound to v. Since $\theta = \phi \circ \theta_m$, both θ_m and θ are modified so that now $v\theta = v$ instead of Xv[5]. To show that $\theta\!\downarrow$ is most general, there are two sub-cases:

1. If $Xv\theta_m = Xv$ (before `setsub` was called), then almost trivially $\theta\!\downarrow$ is still a mgu for A and B, since $\theta\!\downarrow$ was an mgu in the previous step (by inductive hypothesis). That is, $\theta \circ \varphi$ in previous step is the same as $\theta \circ \varphi$ after the `setsub` in the current step, since `setsub` binds Xv to v.

2. But what if $Xv\theta_m = Xw$, Xw being the logic variable associated with a variable w distinct from v? In this case, both Xv and Xw will be bound to v. The *entire* θ_m (and θ) unifier is effectively modified. For every constant symbol v_i, if previously $v_i\theta = t[Xw]$, now $v_i\theta = t[v]$. In particular, the substitution of Xw for Xv in θ_m is now "switched," in θ:

$$w\theta = v,$$
$$v\theta = v,$$
$$z\theta = t[v] \ \text{ if previously } z\theta = t[Xw].$$

This seems less than ideal, since if we had instead:

$$v\theta = w,$$

[5]We do not want to say that θ is transformed to another substitution θ' because it is still θ that is recorded by `quick-unify`.

$w\theta = w,$

$z\theta = t[w]$ if previously $z\theta = t[Xv]$ (since $Xv\theta_m = Xw$),

then $\theta\downarrow$ remains an mgu by the same argument as in the first case. But this is not the case. What takes place instead is a more complicated renaming. We need to show that, given that in the previous step, $\theta \circ \varphi$ was an mgu, $\theta \circ \varphi$ after the current step is still an mgu despite the *variable switching*. This amounts to proving the following:

Lemma 3.1 *(Variable Switching)*
For all terms A and B, if for some σ, $A\sigma = B\sigma$ with $x\sigma = y$ for some variables x and y occurring in A or B, let

$\sigma' = \sigma \circ [x/y]$

(so effectively $y\sigma' = x$, $x\sigma' = x$),

then $A\sigma' = B\sigma'$.
Furthermore, σ' remains an idempotent mgu of A and B if σ was.

Proof:

First we have to show that σ' is a valid definition, i.e., σ' is idempotent. This amounts to showing that for all z, z appears free in $z\sigma'$ implies $z\sigma' = z$. So suppose $z\sigma'$ contains z for some variable z. z can not be y since $y\sigma' = x$. If $z = x$, then the result is trivial since $x\sigma' = x$. So assume z is distinct from x and y. But then z appears free in $z\sigma$ as well. Since σ is idempotent, $z\sigma = z$ and thus $z\sigma[x/y] = z\sigma' = z$.

Next we show that σ' unifies A and B - by induction on the structure of A and B:

- If both are variables, assume $A = v_i$, $B = v_j$ for some variables v_i and v_j. Then $v_i\sigma' = v_i\sigma[x/y] = v_j\sigma[x/y] = vj\sigma')$.
- If A is a variable, and B is of form $(p\ t_1, \ldots, t_n)$ where p is defined in the signature (of the appropriate type)[6]. Then this case doesn't apply since there won't be variables x and y such that $x\sigma = y$.
- The only other case where A and B are unifiable by σ is

  $$A = (p\ t_1, \ldots, t_n), \quad B = (p\ s_1, \ldots, s_n)$$

  But then $t_i\sigma = s_i\sigma$ for all $i\epsilon\{1 \ldots n\}$. So by inductive hypotheses, $t_i\sigma' = s_i\sigma'$, so $A\sigma' = B\sigma'$.

[6] We can regard constants as functions of zero arity.

Assuming σ is a mgu, i.e, for any unifier μ of A and B, there exists a substitution σ_2 such that $\sigma \circ \sigma_2 = \mu$. We may further assume that $x\sigma_2 = x$ since x does not appear in $t\sigma$ for any term t. Define σ_2' to be:

$$\sigma_2' = [y/x] \circ \sigma_2$$

so effectively $x\sigma_2' = y\sigma_2$. This definition is valid since:
If x occurs in $x\sigma_2'$ then x occurs in $y\sigma_2$. But then x occurs in $x\mu$, so it must be that $x\mu = x$ since μ is idempotent. Since $x\sigma\sigma_2 = x\mu$, $y\sigma_2 = x$, so $x\sigma_2' = x$. If $z\sigma_2'$ (for z a variable different from x) contains an occurrence of z, this means $z\sigma_2' = z[y/x]\sigma_2 = z\sigma_2$ contains z. But then $z\sigma_2 = z$ since σ_2 is idempotent, so $z\sigma_2' = z$.

We now show that $\sigma' \circ \sigma_2' = \mu$:

- $x\mu = x\sigma\sigma_2 = y\sigma_2 = x\sigma_2' = x\sigma'\sigma_2'$.
- For z distinct from x, $z\sigma'\sigma_2' = z\sigma'[y/x]\sigma_2 = z\sigma[x/y][y/x]\sigma_2$. $z\sigma$ does not contain x, so $z\sigma[x/y][y/x] = z\sigma$ (because applying $[y/x]$ after $[x/y]$ will not affect any x already in $z\sigma$, since there are none). So finally $z\sigma[x/y][y/x]\sigma_2 = z\sigma\sigma_2 = z\mu$.

σ' is therefore still most-general, i.e., it differs from σ only up to renaming. This completes the proof of the lemma.

□

With this result, $\theta\!\downarrow$ remains an mgu of A and B even though it involves a "switch" on $\theta\!\downarrow$ of the previous step. Finally, when all `setsub` goals have been solved, there will be no more free logic variables in the codomain of θ. So at the end $\theta\!\downarrow = \theta \circ \varphi = \theta$, which is the final mgu constructed by the program.

This concludes the sub-case $\theta_m(X\,v) = X\,w$ and the proof of soundness. Full completeness is not possible because of the cut. But because the meta-level unification is complete, we know that *some* renaming of the mgu will be constructed (recognized) by the program. To formally show completeness one only need to note that all `setsub` goals will succeed (since the second clause of `setsub` can never fail), and therefore the `quick-unify` program will always terminate.

New Variables in the Unifier

The `quick-unify` program in fact works even if the meta-level unification algorithm introduced new logic variables into θ_m, the meta-level unifier. For example, it is conceivable that x and y can be unified by $[z/x, z/y]$ for some new variable z. But this is not a problem since a new logic variable X_{new} can only appear in θ_m if $X_v\theta_m = X_{new}$ for some X_v introduced by the `quick-unify` program for some constant symbol v. To see this, we know that there is *some* mgu ϑ_m that does *not* contain new logic variables. Then for some δ_m, $\vartheta_m \circ \delta_m = \theta_m$ since ϑ_m is also most general. Since θ_m and ϑ_m can only differ up to the renaming of variables, for all variables z, $z\theta_m = t[X_{new}]$ (for some t) implies $z\vartheta_m = t[Xv]$ for some original logic variable Xv. But then $Xv\delta_m = X_{new}$, which in turn means $Xv\theta_m = X_{new}$ since $Xv\vartheta_m = Xv$ for idempotent ϑ_m.

Also, since θ_m is idempotent, we know that $X_{new}\theta_m = X_{new}$ and Xv does not appear anywhere in the codomain of θ_m. It follows that X_{new} is superfluous; we could have used Xv in its place. That is, define

$$\vartheta'_m = \theta_m \circ [Xv/X_{new}] \text{ for all } x,$$

then θ'_m is a valid variable-renaming of θ_m, and is therefore also an mgu of N and M. The object-level unifier $\theta\downarrow$ can be defined as $\phi \circ \theta'_m \circ \varphi$. When `setsub` is solved for Xv, Xv *and* X_{new} would be bound to v (assuming this is the first such Xv `setsub` is called on). So it is this version of $\theta\downarrow$ that will be constructed by the `quick-unify` program.

These details were omitted in the above proof for sake of clarity. To include them in the original proof, we would have to extend the $\downarrow$ operation to first "rename" all new logic variables back to original logic variables. This would involve a choice operation since there could be more than one original variable Z such that $Z\theta_m = X_{new}$. This choice operation is defined by the order in which the original logic variables were introduced, since that determines the order the `setsub` goals are solved. The first such Z encountered will determine the meta-level unifier $\theta\downarrow$ will imitate. A third sub-case, where $Xv\theta_m = X_{new}$, also needs to be added in addition to the cases $Xv\theta_m = Xv$ and $Xv\theta_m = Xw$. This case would basically be the same as the first sub-case where $Xv\theta_m = Xv$ since no variable switching would be involved: for idempotent substitutions we can not have both $Xv\theta_m = Xw$ and $Xv\theta_m = X_{new}$.

Adding these details unnecessarily complicates the proof. The original proof stands on its own since it is not unreasonable to assume that the meta-level unification algorithm does not introduce new variables (for first-order unification). But it is a stronger result that the `quick-unify` program is correct for any implementation of meta-level unification that constructs idempotent mgu's[7].

This completes the proof of the soundness (and completeness) of our algorithm for first-order unification. We finalize our result with the following. Assume that `?-` represents uniform provability amended with the standard Prolog left-to-right, depth-first search strategy that recognizes `!` (i.e, the query prompt of the λProlog interpreter).

Theorem 3.2 *(Soundness and semi-completeness of* `quick-unify`*)*
Let signature $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m, c_1 : a_1, \ldots, c_n : a_n\}$, *and let* $K = \{[\![k_i,\ k_i : s_i]\!] \mid 0 \leq i \leq m\}$. *The following holds:*

1. *If* $\Sigma; K ?-$ `quick-unify` $(c_1, \ldots, c_n)\ A\ B\ U$ *where* A *and* B *are (closed)* Σ*-terms, then* U *is a most general unifier of* A *and* B.

2. *If* U *is a mgu of (closed)* Σ*-terms* A *and* B, *then for some renaming* U' *of* U, $\Sigma; K ?-$ `quick-unify` $(c_1, \ldots, c_n)\ A\ B\ U'$ *holds.*

3.3 Forcing $\forall$-Introduction

The object-level unification problem, and especially the `setsub` technique, addresses a critical, yet subtle problem in meta-programming. In fact, this problem itself makes *meta*-programming necessary. When a goal G succeeds and yields a mgu θ, we can say that "all" ground instances of $G\theta$ must also succeed. It is therefore "valid," in the sense that it's consistent with the existing program, to add the clause $\forall V.G\theta$ (where V are the free variables in $G\theta$) to the program. But then why isn't any simple construct that generalizes over free variables and forms a universally quantified formula a valid addition to the logic programming language?

The problem is that this use of "all" and this notion of "validity" exist at the level of human reasoning (the ultimate meta-level). It is not the $\forall$ used in defining logic program clauses. Consider the following line of reasoning about the answer substitution `T = (arr S integer)` from the query to the attempted definition of type inferencing of section 3.1 (`typing`).

[7] Idempotence is *not* an unnecessary assumption.

> *Since* `S` *can be instantiated to anything, it can be instantiated to some arbitrary constant symbol. This symbol can then be* generalized *over to form a valid, polymorphic type. In other words, apply* $\forall$*-introduction (in the sense of natural deduction) on the constant symbol, which represents an object-level variable.*

The reasoning process above is valid because we, as human beings, know how to apply $\forall$-introduction at the level of mathematical discourse. It is therefore valid for *us* to interpret the answer substitution with free logic variables as a "general solution." The problem is how to implement or *reflect* this reasoning process in a logic programming language. This is what `setsub` accomplishes, with the use of `!`. Unfortunately, there are no purely declarative specifications of any predicate that performs this kind of generalization. For example, assume Gen is a predicate that takes a program clause A and a goal G such that $(Gen\ A\ G)$ holds if and only if $(\forall V.A) \Rightarrow G$ holds, where V represents all free variables in A.

Proposition: *Gen is not definable in the language of hereditary Harrop Formulas.*

Proof:

Consider the following program

```
kind i type.
type p i -> o.
type a i.
type var i -> o.

var X :- gen (p X) (pi y\ (pi y)).
```

where `gen` represents Gen. The variable-check `var` predicate is well-known to be impossible. The query `var X, X = a` succeeds, but the query `X = a, var X` will fail. This is clearly a non-logical consequence.

□

Similar proofs can be given to show that predicates similar to Gen, such as Dietzen and Pfenning's `rule` construct, are also not declarative. One may complain that this theorem and proof only holds for HOHH languages. Note, however, that unless a construct (such as $\Rightarrow$) exists to incorporate generalized formulas into the existing program, a generalization operation would not be very useful. The $\forall$-goal used in var_{mf} can be simulated in any language by reserving a unique symbol representing the eigenvariable that would be introduced during uniform proof-search.

Unfortunately, there is no result that shows that all *Gen*-like predicates are unspecifiable in HoHH. One can always define a predicate "on paper" in a variety of ways, with a variety of restrictions. For example, one can require that a goal containing free variables must be solved first before these free variables can be generalized. There may even be extremely restricted cases (such as with very small signatures) where closing a meta-logic variable is not inconsistent. There are no known, general techniques to show how each sufficiently-general *Gen*-like predicate leads to logical inconsistencies.

If we accept the premise that generalization over free variables is impossible from within a meta-logic, however, then we must also accept the necessity of *meta-programming* the object-theory and operations needed for generalization. That is, since it is valid to generalize over an derived expression with free variables, but this operation is not itself definable within the logic programming language, the generalization must take place at the *meta-level,* as an operation of a meta-language. But because we wish this meta-language itself to be a logic programming language, we face the problem at the meta-level as well. One way to avoid this problem entirely is to use the ground representation exclusively, as we have done in implementing the Martelli-Montanari unification algorithm. Another method is to adopt a non-logical construct, but to make the effects of this construct clear so that meta-programs can remain declarative in the qualitative sense.

We claim that `setsub` is the simplest and most modular such construct. The cut in its definition is necessary to ensure that free logic variables in the solution *will be* instantiated to constant symbols instead of something else. If the above reasoning process can be defined purely in the logic of higher order hereditary Harrop formulas without the cut, we would need to be able to determine when something is not a free logic variable. This is well-known to be impossible.

The `setsub` technique implicitly uses this ability to distinguish non-variables (since the second clause of `setsub` is used only when `Y` is not a free logic variable). The extra-logicality, however, is clearly identified and isolated as that one use of cut. This type of cut is needed often in many standard operations in Prolog. Consider the following program for removing *all* occurrences of some given term in a list:

```
remove X nil nil.
remove X (X::L) M :- !, remove X L M.
remove X (Y::L) (Y::M) :- remove X L M.
```

Without the cut in the second clause, this will not be a correct definition since

```
remove 1 (1::nil) (1::nil)
```

will succeed by resolving with the third clause. This use of cut is similar to the cut in `setsub` in that it renders the second and third clauses mutually exclusive. Without the cut there is no way to determine when two arbitrary expressions (for arbitrary signatures) are distinct. The cut in `setsub` can also be viewed in the same way: `setsub V XV` succeeds by resolving with the first clause if `XV` and `V` *can* be considered the same, which is the case only if `XV` is still a free variable. This needs to be distinguished from the case where `XV` and `V` can not be considered the same, which is the case when `XV` has already been instantiated into something else (i.e., "became a non-variable"). Furthermore, the effect of the cut in the `remove` program on its declarativeness is minimal in the same sense the cut in `setsub` is. From this perspective especially, the fact that unification can be specified with just one cut is interesting, since even many straightforward, mundane operations such as `remove` also require a similar cut.

The foregoing suffices to distinguish `setsub` from an arbitrary "programming trick." In short, we have seen why it is needed, what its role is, and what its consequences are. The purely-declarative alternative is also always available.

3.4 A Prolog Meta-Interpreter

A classic example of meta-programming is the specification of an object-level interpreter for first-order Horn clauses. It embodies many of the critical issues of meta-programming.

One early attempt to specify Prolog within Prolog is the so-called Vanilla interpreter in Figure 3.2 [23]. We also included the definition of the `path` program in Figure 3.2, which denotes the transitive and reflective closure of the predicate `adj`. The Prolog query

```
interpret(true).
interpret((G,H)) :- interpret(G), interpret(H).
interpret(A)     :- clause(A,G), interpret(G).

clause(adj(a,b),  true).
clause(adj(b,c),  true).
clause(path(X,X), true).
clause(path(X,Z), (adj(X,Y),path(Y,Z))).
```

Figure 3.2: The Vanilla interpreter (in Prolog syntax)

`?- interpret(path(a,X))` returns three answer substitution, namely, those that bind `X` to either `a`, `b`, or `c`. Note the use of the so-called "non-ground" representation: meta-level free variables are used to denote object-level free variables, and meta-level universally quantified variables are used (in the `clause` clauses) to denote object-level universally quantified variables. Since the meta-level notions of substitution, and unification are directly used to specify the corresponding notions at the object-level, the Vanilla interpreter is exceedingly simple to define.

There are, however, various problems with this interpreter. The most serious of these is that it represents object-level programs not as proper values within the meta-logic but as part of the meta-program itself. Object-level answer substitutions are also embedded as meta-level substitutions. It is desirable to have object-level programs and substitutions represented as first-class citizens in the way lists and binary trees can be represented and manipulated. This will, for example, allow us to derive generalizations from the meta-level for the purpose of forward chaining that is otherwise not possible within the object-level language itself.

One approach to addressing the shortcomings of the Vanilla interpreter is to represent object-level programs and goals using conventional data-structures in the context of Prolog. That is, if we are determined to remain in a first-order meta-language, then object-level program clauses need to be represented using their parse trees via first-order terms. In this setting, a meta-interpreter would take an object-level program as a term, but the meta-language would have to provide constructs for many additional procedures. For example, creating new instances of a clause represented this way requires a procedure to produce new variables. One way to handle this complexity would be to provide direct support for substitution and unification in the underlining implementation of the meta-language. The Gödel programming language [24], for example, takes this approach by providing a large collection of data structures and procedures to handle aspects of the parse tree representation of object-level expressions.

Using higher-order abstract syntax in the context of HOHH formulas, we can achieve a high-level specification of an interpreter, but with object-level programs as proper values in the meta-language. Recall the signature for first-order formulas of section 1.4.1. The path program can be represented with:

```
type prog form -> o.
prog (and (adj a b) (and (adj b c) (and (all x\(path x x))
          (all x\(all y\(all z\(imp (and (adj x y) (path y z))
                                     (path x z)))))))).
```

3.4.1 Substitutions with New Variables

We will now present an implementation of an object-level interpreter for first-order Horn clauses that constructs explicit object-level answer substitutions. The interpreter will construct substitutions of the form `(sub X Y Z)` as used in the unification programs. However, since arbitrarily many new free variables may be introduced during resolution, we will need the following construct to represent them:

```
type newvar (term -> subst) -> subst.
```

This construct declares new free variables that may appear in the range of the substitution. That is, since in any given signature there are only finitely many constant symbols, we must dynamically generate new eigenvariables using `pi`-goals. Eigenvariables are λ-bound variables in the meta-logic; they may not appear free in the static signature. This means using meta-level λ-abstractions (prefixed by the constructor `newvar`), which are always readily available, to represent new object-level variables that can not be statically declared. An alternative to using λ-bound variables is to use recursive data structures, such as strings and nested lists. We can then define a `gensym` predicate to generate new variable names within the existing signature. Such methods are always possible, though they will be unduely cumbersome and are clearly inconsistent with the concept of higher-order

```
type apply_sub             form -> subst -> form -> o.
type apply_sub_term        term -> subst -> term -> o.
type compose_subs, mapsub  subst -> subst -> subst -> o.

apply_sub M nul N         :- copyform M N.
apply_sub M (sub V VT Ss) N :- copyterm V VT => apply_sub M Ss N.
apply_sub_term M nul N    :- copyterm M N.
apply_sub_term M (sub V VT Ss) N :-
  copyterm V VT => apply_sub_term M Ss N.
compose_subs M nul M.
compose_subs M (newvar N) (newvar U) :-
  pi z\(compose_subs M (N z) (U z)).
compose_subs M (sub V VT Ns) U :- mapsub M (sub V VT Ns) U.
mapsub nul S nul.
mapsub (sub V VT Ms) S (sub V UT Us) :-
  apply_sub_term VT S UT, mapsub Ms S Us.
```

Figure 3.3: Basic Operations on Substitutions

abstract syntax. The technique of representing new object-level free variables with meta-level bound variables will lead to greater flexibility in meta-programming, and will be of central importance in this thesis.

We also require the basic operations on substitutions given in Figure 3.3. The signature for substitutions was given in section 3.2.1.

The predicates `apply_sub`, `compose_subs`, and `mapsub` specify standard operations on substitutions. The correctness of `apply_sub` (and consequently `compose_subs` and `mapsub`) follow easily from the Substitution Encoding theorem.

3.4.2 Object-level Depth-First Interpretation

The interpreter for first-order Horn clauses is defined in Figure 3.4. In the clauses for `interp Cs Vs A U`, `Cs` represents the program clauses and `Vs` is a list of constant symbols representing object-level variables in query `A`, for which answer substitution `U` is to be found. For example, given the path program of Section 3.4, and the declarations

```
atom (adj X Y).
atom (path X Y).
```

indicating the atomic formulas of the language, the query

```
type interp      form -> list term -> form -> subst -> o.
type interp_aux  form -> list term -> subst -> form -> subst -> o.
type backchain   form -> list term -> form -> form -> subst -> o.
type resolve     list term -> form -> form -> form -> subst -> o.

interp Cs Vs A U :- atom A, backchain Cs Vs Cs A U.
interp Cs Vs (and A B) U :- interp Cs Vs A M, interp_aux Cs Vs M B U.
interp_aux Cs Vs (newvar M) B (newvar U) :-
  pi z\(interp_aux Cs (z::Vs) (M z) B (U z)).
interp_aux Cs Vs M B U :- (M = nul; M = (sub X Y Z)),
  apply_sub B M B2,  interp Cs Vs B2 N, compose_subs M N U.

backchain Cs Vs (and C D) A U :-
  backchain Cs Vs C A U; backchain Cs Vs D A U.
backchain Cs Vs (all D) A (newvar U) :-
  pi z\(backchain Cs (z::Vs) (D z) A (U z)).
backchain Cs Vs D A U :- atom D, resolve Vs (imp truth D) A N U.
backchain Cs Vs (imp G D) A U :- resolve Vs (imp G D) A NewG M,
  interp Cs Vs NewG N, compose_subs M N U.

resolve (V::Vs) (imp G D) A NewG (sub V XV Us) :-
  (copyterm V XV => resolve Vs (imp G D) A NewG Us), setsub V XV.
resolve nil (imp G D) A NewG nul :-
  copyform D N, copyform A N, copyform G NewG.
```

Figure 3.4: A Meta-Interpreter for First-Order Horn Clauses

```
?- prog P, interp P (w::nil) (path w c) U
```

will succeed with the solutions `U = (sub w a nul)`, `U = (sub w b nul)`, and `U = (sub w c nul)` (the actual program also generates some vacuous `newvar` abstractions, which can be easily eliminated).

The top-level `interp` clause breaks conjunctive goals, and when atomic goals are reached, it calls `backchain`. Solutions for conjunctive goals are combined using `compose_sub`. Since the backchain clauses may recursively call `interp` for new goals, we must include the program `Cs` twice. That is, in `backchain Cs Vs Ds A U`, `Cs` is the original program, `Vs` are the free variables in atomic goal `A`, `Ds` represents the clause in `Cs` being backchained on, and `U` represent the most-general answer-substitution for `A`. When an universally quantified clause is backchained on, new eigenvariables are introduced for each quantifier, and these new variables may appear in the answer substitution under `newvar` abstractions[8]. When an implication clause in the program is backchained on, the `resolve` clauses uses unification to create the mgu, and gives the new goal (`NewG`) to be recursively solved.

In this program we have used the `setsub` technique for unification. If we wished to be "purely" declarative, we can replace the `resolve` clauses with:

```
resolve Vs (imp G D) A NewG Mgu :- unify Vs D A Mgu, apply_sub G Mgu NewG.
```

But this would require the unification program of section 3.2.1.

The correctness of this interpeter is of course mostly due to the correctness of resolution theorem proving. The other meta-programming details are straightforward given the previous results concerning object-level substitution and unification.

3.4.3 Adding Partial Evaluation and Generalization

In this section we will give a follow-up application of the meta-interpreter built above. Because closed answer substitutions are returned by the program, we can generalize over the remaining (object-level) free variables to form a new, universally-quantified program clause.

By *partially evaluating* or *partially deducing* a Prolog program we mean solving a goal until only subgoals of a certain pre-designated form remain (see [30] for a tutorial). For instance, consider a program containing the clauses:

```
happy X :- loves S X, not-in-debt X.
not-in-debt X :- bank-balance X Y, Y > 0.
bank-balance harry 100.
```

Here, we have a predicate `not-in-debt`, which is easy to determine, and another predicate `loves`, which may not be so easy to determine. However, we still would like at least a *partial* answer to harry's happiness. This can be achieved by designating goals of the form `(loves A B)` as *operational*, i.e, the derivation can be stopped when only operational subgoals remain. The query `happy harry` will be solved, with the subgoal or constraint

[8] Because of this, intermediate "newvar" variables may appear in the domain of the final unifier. These can be filtered out, however, by applying the substitution on the original variables `Vs`.

`loves S harry` remaining. What's more, if the variable `S` is universally generalized over to form the clause

```
pi S\ (happy harry :- loves S harry)
```

then this clause follows from the original program, and can be added to the program without changing its meaning.

Bundy and Harmelen noted in [20] that partial evaluation in the context of Prolog is closely related to another concept, that of *explanation-based generalization* (EBG). This idea has its roots in AI [42]. Basically, instead of forming a generalization in an ad hoc manner from a multitude of instances, we can instead examine a specific instance, or *training example* of how a conclusion was derived or "explained." Then we can extract out the contingent elements from the derivation and resulting conclusion and form a generalization. This generalization is guaranteed to be correct, since it creates a generalized *proof* (or explanation) for its generalized conclusion. In [28] it was noted that a Prolog interpreter can be augmented in a natural way so that each proof search will give rise to an EBG. This concept has also been studied for a HoHH language in [8]. Bundy and Harmelen noted that this augmentation of a Prolog interpreter is essentially the same as for partial evaluation. They gave the following program, which is like the Vanilla interpreter, except it also derives constraint formulas for the specific goal solved. It was given in Prolog syntax:

```
peval-ebg(GenA,GenA) :- operational(GenA), not not call(GenA).
peval-ebg(GenA,GenB),(GA,GB)) :-
   peval-ebg(GenA,GA), peval-ebg(GenB,GB),
peval-ebg(GenA,G) :-
  clause(GenA,GenB),
  peval-ebg(GenB,G).
```

The Prolog query `peval-ebg A B` will solve `A` and yield operational goals `B` as constraints. The use of double negation-as-failure is to prevent (logic) variables from being bound after successful resolution, so the constraints can be "most general." Even without `not not`, this is clearly not a declarative program for generalization, for the same reasons we've been discussing. Even if it were possible to program this in the ground-representation in languages such as Gödel, there is still no declarative way to incorporate a generalized clause into a program in a first-order Prolog setting. By *meta-programming* first-order Prolog in a higher-order logic programming language, the problems concerning declarativeness can be solved. We now show how the Prolog meta-interpreter of the previous section can be modified to implement partial evaluation and generalization. Figures 3.5 and 3.6 contain the code for the modified meta-interpreter. The clauses for `compose_subs`, `apply_sub`, and `resolve` are the same as in the meta-interpreter. `form-trivials` takes a list of variables $x_1, \ldots, x_n$ and forms the trivial substitution $[x_1/x_1, \ldots, x_n/x_n]$.

Since the new variables introduced during interpretation can also appear in the constraint goals being collected, the constraint (operational) goals must also come under the scope of the `newvar` abstractions. Thus we introduced a new type `sfpair`, for "substitution-formula pair" and a constructor `pe` that forms such a pair, that can come under `newvar` declarations. `compose-pe`, in addition to composing the component substitutions as before, will also apply the new substitution to the first list of formulas and then append the two list of formulas. The `partial` predicate replaces `interp` by constructing not just a unifier but a "sfpair" `(pe U F)` with the constraint goals as well. In the first clause of `partial`, an operational goal becomes a constraint goal, along with the trivial substitution for the variables in this goal. Double negation-as-failure is replaced by the extra `backchain` goal, which allows interpretation of the operational goal without recording any implied instantiation. In the second `partial` clause, resolution is allowed on non-operational goals, and thus the solution from backchaining is the solution of this clause. If partial-evaluation succeeds, a new Horn clause (the explanation-based generalization) can be formed by applying the unifier to the original goal, and adding the conjunction of all the formulas in the `sfpair` as conditions. Free variables are of course to be abstracted to form `all` clauses. This is accomplised by the `form-clause` predicate, which transforms `newvar` abstractions into `all`-quantifications for Horn clauses, and also abstract over original free variables in the list `Vs` by first copying them to unique eigenconstants. The `ebg-partial` predicate combines `partial` with `form-clause` to yield the top-level predicate for partial evaluation or explanation-based generalization.

In the `path` program defined previously, let `adj`-goals be considered operational, and other goals be non-operational. The query

```
?- prog P, ebg-partial P (u::w::nil) (path w u) NP.
```

will yield among other solutions a generalization

```
kind sfpair type.
type pe          subst -> (list form) -> sfpair.
type newvar      (term -> sfpair) -> sfpair.
type compose-pe  sfpair -> sfpair -> sfpair -> o.
type partial     form -> list term -> form -> sfpair -> o.
type backchain   form -> list term -> form -> form -> sfpair -> o.

compose-pe (pe M G) (newvar N) (newvar U) :-
  pi z\ (compose-pe (pe M G) (N z) (U z)).
compose-pe (pe M G) (pe N H) (pe U K) :-
  compose_subs M N U, list-app G N G2, append G2 H K.

partial Cs Vs A (pe TV (A::nil)) :-
  atom A, operational A, form-trivials Vs TV, backchain Cs Vs Cs A U.
partial Cs Vs A U :- atom A, non-operational A, backchain Cs Vs Cs A U.
partial Cs Vs (and A B) U :- partial Cs Vs A M, partial_aux Cs Vs M B U.
partial_aux Cs Vs (newvar M) B (newvar U) :-
  pi z\(partial_aux Cs (z::Vs) (M z) B (U z)).
partial_aux Cs Vs (pe M Gs) B U :-
  apply_sub B M B2, partial Cs Vs B2 N, compose-pe (pe M Gs) N U.
backchain Cs Vs (and C D) A U :-
  backchain Cs Vs C A U; backchain Cs Vs D A U.
backchain Cs Vs (all D) A (newvar U) :-
  pi z\(backchain Cs (z::Vs) (D z) A (U z)).
backchain Cs Vs D A (pe M nil) :- atom D,
  resolve Vs (imp truth D) A N M.
backchain Cs Vs (imp G D) A U :- resolve Vs (imp G D) A NewG M,
  partial Cs Vs NewG N, compose-pe (pe M nil) N U.

form-trivials nil emp.
form-trivials (X::Xs) (sub X X Z) :- form-trivials Xs Z.
```

Figure 3.5: Prolog Meta-Partial Evaluator

```
all x\ all y\ (imp (adj x y) (path x y))
```

which is indeed another valid way to find a path.

The only significant difference between partial evaluation and EBG is that, with EBG, we supply a *training goal* and a generalized *target goal* (or target concept), and build in parallel a proof for the training instance and a proof for the generalized concept. This will serve to guide the formation of the generalization to the same search path used in proving the training example. With the partial-evaluator program, all search paths will be attempted, yielding a multitude of generalizations. Incorporating the training goal and training proof is straightforward, though cumbersome, since it requires `partial` to have

```
type abstract-over (list term) -> form -> form -> o.
type ebg-partial form -> (list term) -> form -> form -> o.
type form-clause (list term) -> form -> subst -> form -> o.

form-clause Vs G (newvar U) (all N) :-
  pi x\ (form-clause Vs G (U x) (N x)).
form-clause Vs G (pe U nil) G3 :-
  apply_sub G U G2, abstract-over Vs G2 G3.
form-clause Vs G (pe U (F::nil)) G3 :-
  apply_sub G U G2, abstract-over Vs (imp F G2) G3.
form-clause Vs G (pe U (X::Y::Z)) F :-
  form-clause Vs G (pe U ((and X Y)::Z)) F.

abstract-over (V::Vs) C (all NewC) :-
  pi z\ (copyterm V z => abstract-over Vs C (NewC z)).
abstract-over nil C NewC :- copyform C NewC.

partial-ebg Cs Vs G (and N Cs) :-
  partial Cs Vs G U, form-clause Vs G U N.
```

Figure 3.6: Forming Generalizations from Meta-Partial Evaluation

another set of parameters representing a parallel resolution.

To conclude this section, we must re-emphasize the subtle difference between generalization at the meta- and object-levels. We've shown that it is not possible to define generalization from *within* a logic programming language. Yet, such generalizations would be valid additions to a logic program, if they can be formed. What we are doing here is formulating a *meta-level* predicate for *object-level* generalization. This predicate has a logical definition in the meta-language. We are not adding this predicate to the object-level Prolog interpreter itself (in the sense that no formula in an object-level program can refer to it). Thus it is still not possible to define variable-check in the object-level language. This demonstrates conclusively the effectiveness *and necessity* of meta-programming with respect to problems requiring unification and generalization.

3.5 Polymorphic Type-Inferencing

One of the original motivations for developing object-level unification is polymorphic type-inferencing for functional programming languages (namely ML). No purely declarative solution has ever been given - in abstract syntax or otherwise. Existing declarative solutions are either restricted to the propositional, non-polymorphic case, are merely type checkers, or are highly nondeterministic. In this section we provide two programs which, using the technique of object-level unification, will derive the most general type for simple ML programs containing `app` for application, `abs` for λ-abstraction, and `let` for local binding. The first program will use the algorithmic but purely declarative unification program of section 3.2.1, and the second one will use `setsub` (though in a broader context than object-level unification).

3.5.1 The Purely Declarative Approach

The typing program will have three central clauses, one each for the `app`, `abs`, and `let` expressions. `let`-expressions are of the form `(let x\(R x) T)`, which has the meaning

$$\textit{let val } x = T \textit{ in } (R\ x) \textit{ end.}$$

The signature for object-level ML expressions and types was given in section 3.1. We also wish to avoid confusing meta-level types such as `tm -> tm` with object-level types such as `(arr integer integer)`. Meta-level type declarations are omitted from the displayed code (they can be found in the complete codes in the Appendix).

In addition to the need for object-level unification, a few other problems needs to be addressed. Before we do so, however, we will present the program first. We will only give the core code for type inferencing the three kind of expressions, and explain the meanings of the auxillary predicates below for sake of presentation (see Appendix for full code). Figure 3.7 contains the code for the main clauses. Initial (object-level) type-assumptions for three sample term constants `zero`, `succ` and `op`, are also given for illustration. Quantifier-free types are predicated as *monotypes.* In addition to being entirely free of non-logical constructs, the program is also properly contained in the simple language L_λ. Polymorphic types are represented in the form `(all T)`, where `all` stands for the usual Π-quantification over type variables. We will use the simple signature of `integer` and `arr` of the object-level unification program of section 3.2.1, which was customized with this purpose in mind. A polymorhic type T for term N is inferred by the goal[9]

```
polytype nil M N T.
```

[9]The meta-level type of predicate `polytype` is `(list tm) -> substitution -> tm -> poly -> o.`

```
monotype integer.
monotype (arr A B) :- monotype A, monotype B.

polytype A emp zero integer.
polytype A emp succ (arr integer integer).
polytype A emp op (all x\ (arr x x)).

polytype NV (newvar Mgu) (app M N) (all Ty) :-
pi tv\ (sigma Tn\ (sigma Mg\ (sigma Mg2\ (monotype tv => (
  polytype NV Mt1 M Ar1, fill-out Mt1 Ar1 Mt Artype,
  polytype NV Nt1 N T1,  fill-out Nt1 T1 Nt T,
  merge-type T tv Tn, typeunify (tv::NV) Tn Artype Mg,
  append-sub Nt Mt MN, merge-sub MN Mg Mg2,
  resolve-sub (tv::NV) (tv::NV) Mg2 (Mgu tv),
  typesub tv (Mgu tv) (Ty tv)))))).

polytype NV (newvar Mgu) (abs M) (all Typ) :-
  pi x\ (pi a\ (sigma A2\ (sigma B\ (sigma Mg\ (sigma Ty\ (monotype a => (
    ((pi Any\ (polytype Any emp x a))
                 => polytype (a::NV) Mg (M x) B),
    resolve-sub (a::NV) (a::NV) Mg (Mgu a),
    merge-type a B Ty, typesub Ty (Mgu a) (Typ a)))))))).

polytype NV M (let R T) C :-
  polytype NV Mt T Ty,
  pi x\ ((pi A\ (polytype A Mt x Ty)) => polytype NV Mr (R x) B),
  append-sub Mr Mt Mrt, resolve-sub NV NV Mrt M,
  typesub B M C.
```

Figure 3.7: MLtyper program

The first argument to `polytype` is a list of constants representing new free variables (introduced by the `abs` case) that are not bound in the type expression. The second argument is a substitution (unifier) that may contain the new free variables in its domain. The third and fourth arguments are the term and type respectively. We do not assume that the initial type enviornment contains free type variables, thus `nil` is always used in the query (it is not difficult to extend the program to allow initial free variables).

Initial typing assignments for constants are declared by meta-program clauses of the form

```
pi A\ (polytype A emp N T)
```

for term `N` and `type T`. The basic technique for typing an `(app M N)` expression is to first recursively infer the types of `M` and `N` as `Ar` and `Tn`, respectively. Then, introduce a new type variable `tv`, and unify the types `(arr Tn tv)` and `Ar`. The substitution for `tv` resulting from this unification is the type of `(app M N)`. The basic technique for typing an `(abs M)` expression is to first introduce a new type-variable `a` and a new term-variable `x`. Then, under the assumption that `x` has type `a`, infer the type for `(M x)` as `B`. Then, apply the unifier resulting from this inference to `(arr a B)` to form the type of `(abs M)`. The implementation of these basic techniques, however, must be consistent with the case for infering the type of `let` expressions. This requires some elaborate mechanisms, which we now turn to.

To Close Or Not To Close

A natural dilemma exists when designing a type inferencing algorithm: when do we close the type expressions. That is, given a type T with free type variables $v_1, \ldots, v_n$, when do we form the closed, polymorphic type $\Pi v_1 \ldots \Pi v_n.T$. Typing the `let` expression requires that the type of the locally-bound term be closed. On the other hand, typing the `abs` and `app` expressions require unification on free variables, which suggests that inferred type expressions not be closed immediately. For example, assume f has pre-assigned type $\Pi v.v \rightarrow v$. If we wish to type $\lambda x.(f\ x)$, we first introduce a new type variable a to represent the supposed type of the λ-bound variable x. We then *recursively* infer the type for $(f\ x)$ as v, for some free type variable v. In the meantime, a would also be bound to this v, forming the final type expression $\Pi v.v \rightarrow v$. But if we were to generalize all types as soon as they are inferred, then $(f\ x)$ will have type $\Pi u.u$ instead. How then would we know that the v bound to a must be resolved with this closed type expression? If we assumed that v is an arbitrary variable, then we will form the wrong type, $\Pi v \Pi u.v \rightarrow u$, for $\lambda x.(f\ x)$.

This problem becomes especially acute in the context of logic programming and higher-order abstract syntax. Because of the impossibility of a *Gen* operation, we use meta-level constant symbols of a finite signature to represent type variables. As in the case of the Prolog meta-interpreter, however, type inferencing may introduce arbitrarily many new variables. Recall that we used the `newvar` construct to represent these new variables in the meta-interpreter. That is, meta-level λ-abstraction is used to declare and represent new object-level variables. In this case, this technique effectively means that whenever we (recursively) infer a type expression containing new variables not in the current signature, this type expression must be closed! We are therefore forced to deal with closures at all times[10]. How then, can we infer `app` and `abs` expressions that require unification on recursively-inferred type expressions?

Prefix-Compatible Types and Substitutions

The `abs` case presents the most difficulty because not only does it require a recursive type inference, it also requires this inference to return the most general substitution for the new type variable a it introduced (to represent the domain type). Both the substitution and the type returned (by the recursive inference) can be closed by abstractions representing new variables. To complicate matters even further, the variable a itself must *not* be captured by abstraction until the type of the `abs` expression that introduced it is formed. We call such variables *fugitive*.

What we will require is that the *ordering* of abstractions in the type and the substitution created by each inference be compatible so that we can introduce the same eigenvariable for both sets of abstractions and ensure the correctness of the substitution for the fugitive variables. Formally, let $\cap$ denote abstraction over substitutions (i.e, `newvar`) and Π abstraction (quantification) over types (i.e, `all`). Let @ denote list-append. Each inference (goal) of (`polytype`) for term M and type T will be relative to a set (list) of free type variables NV and a substitution (mgu) U (that contains NV in its domain).

Definition 3.1 *(Prefix-Compatibility)*
We say that a type expression Ty and a substitution Un are **prefix-compatible** *with respect to a set of fugitive type variables NV under the valid inference*

$$\texttt{polytype}\ NV\ Un\ M\ Ty$$

for some term M if

$$Ty = \Pi v_1 \ldots \Pi v_n \ldots \Pi v_{n+m}.T, \qquad Un = \cap u_1 \ldots \cap u_n \ldots \cap u_{n+l}.U$$

[10]It may be possible to navigate this problem by indicating if a λ-bound variable is to be interpreted as free or bound at the object-level. But the way free variables are traditionally treated in type-inferencing makes this unnecessarily difficult. Here we introduce a new technique for type inferencing through a more uniform treatment of all type variables.

where $v_1, \ldots, v_{n+m}, u_1, \ldots, u_{n+l}$ *and the variables in* NV *are all distinct, and the following holds. For every* a *in* NV*, if the type-inference of* T *for* M *implies the substitution* $[S/a]$*, then the type variables in* S *are in* $v_1, \ldots, v_n @ NV$ *and* U *contains* $[(S[u_1/v_1, \ldots, u_n/v_n])/a]$*. That is, every variable in* S*, but not in* NV*, is quantified in the same position in* Un *as in* Ty*.*

Given this prefix-compatibility invariance for all valid `(polytype NV Un M Ty)` inferences, we can then type an `(abs M)` expression as follows:

1. For a fresh term-level variable x, introduce a new type (eigen)variable a.

2. Under the assumption x has type a, infer the type of $(M\ x)$ - resulting in type Ty and substitution Un. Assume that Ty and Un has the quantified forms above. Since Ty and Un are prefix-compatible with respect to a and this valid inference, we can do the following:

3. Form the type
$$Tz = \Pi v_1 \ldots \Pi v_n \ldots \Pi v_{n+m}.a \rightarrow T.$$
Now apply U to Tz by merging the Π and $\cap$-prefixes as far as possible, yielding (assume without loss of generality that $l \leq m$):
$$\Pi v_1 \ldots \Pi v_{n+m}.(a \rightarrow T)(U \circ [v_1/u_1, \ldots, v_n/u_n])$$
as the type for `(abs M)`.

We also need to ensure that closed type expressions genuinely represent polymorphic types in that multiple "copies" of them can be made. Sometimes, as in the `abs` case, we need to merge prefixes to allow for substitution, while at other times we need to keep the prefixes apart to reflect (possibly) different occurrences of terms of polymorphic type. For example, while inferring an `(app M N)` expression, we need to first infer the types of `M` and `N` separately, yielding the mgu's `Mu` and `Nu`. But `M` and `N` may share a subterm of type $\Pi v_1 \ldots \Pi v_n.T$ in common, and their inference may imply different substitutions for free type variables which needs to be resolved. Thus in combining the solutions for `M` and `N` we can not merge the prefixes, but must `append` them (by introducing different eigenvariables).

The final type inferred for `(app M N)`, however, still must be made prefix-compatible with the final mgu.

In other words, in the standard "algorithm W" for type inference, sometimes the types are open (subject to mutation via substitution), and sometimes they must be closed. Here, this dichotomy is captured by the difference between *merging* and *appending* abstraction prefixes.

To ensure the prefix-compatible invariance for every successful type-inference, we must take care in defining the `polytype` clauses so that each type created has the necessary abstraction-prefix in the same order as the the mgu. We first note the meaning of the following predicates, all having relatively straightforward definitions, which can be found in the full MLtyper module in the Appendix. The correctness of their specifications is also straight forward to show, given the correctness of copy clauses and the object-level unification program.

- (`typesub` $A\ Un\ Ty$) holds for type expression A if (assume without loss of generality that $m \leq n$)

  $$A = \Pi x_1 \ldots \Pi x_n . A', \qquad Un = \cap v_1 \ldots \cap v_m . U$$

  and

  $$Ty = \Pi x_1 \ldots \Pi x_n . A'(U \circ [x_1/v_1, \ldots, x_n/v_n]).$$

  That is, `type-sub` preserves the abstraction prefix by merging the Π-prefix in the type and the $\cap$-prefix in the substitution as far as possible. (Presumably, of course, this is only used when the type and substitution are prefix-compatible.)

- (`merge-type` $T_1\ T_2\ T_3$) holds if: $m \leq n$, and

  $$T_1 = \Pi v_1 \ldots \Pi v_n . S_1, \quad T_2 = \Pi z_1 \ldots \Pi z_m . S_2$$

  and

  $$T_3 = \Pi v_1 \ldots \Pi v_n . S_1 \rightarrow S_2[v_1/z_1, \ldots v_m/z_m].$$

  If $n \leq m$, switch the roles of n and m and each v_i and z_i in T_3. That is, `merge-type` creates an arrow-type by merging the prefixes of T_1 and T_2 as far as possible.

- (`merge-sub` U_1 U_2 U_3) behaves much like `merge-type`, in the it combines the $\cap$-prefixes of U_1 and U_2 as far as possible. That is, assume without loss of generality that $m \leq n$, let

  $$U_1 = \cap v_1 \ldots \cap v_n.V, \quad U_2 = \cap u_1 \ldots \cap u_m.U$$

  Then

  $$U_3 = \cap v_1 \ldots \cap v_n.V @ (U \circ [v_1/u_1, \ldots, v_m/u_m]).$$

  Composition of substitutions and conflict-resolution are handled by `resolve-sub` (see below).

- (`type-unify` V T_1 T_2 Un) holds if V is a list of free type variables,

  $$T_1 = \Pi v_1 \ldots \Pi v_n.S_1, \quad T_2 = \Pi z_1 \ldots \Pi z_m.S_2$$

  such that V, $v_1, \ldots, v_n$ and $z_1, \ldots, z_m$ are all distinct (renaming bound variables if necessary to ensure this), and

  $$Un = \cap v_1 \ldots \cap v_n \cap z_1 \ldots \cap z_m.U$$

  such that U is the most general unifier of S_1 and S_2 under the free variables[11]

  $$V \cup \{v_1, \ldots, v_n, z_1, \ldots, z_m\}.$$

- (`append-sub` U_1 U_2 U_3) functions exactly like `merge-sub` except it does not merge prefixes, but rather concatenate them in the manner of `type-unify`, keeping the abstractions distinct.

- (`resolve-for` A V U_a U_b) for some free type variable A uses unification by resolving conflicting solutions for A in U_1. The free variables V are needed to complete the necessary domain of unification. That is, let

  $$U_a = \cap u_1 \ldots \cap u_n.[T_1/A, \ldots T_m/A] \circ U_1$$

  where $[T_1/A, \ldots T_m/A]$ are all the substitutions for A in $[T_1/A, \ldots T_m/A] \circ U_1$. Let θ be the mgu for $T_1, \ldots, T_n$ under free variables $V \cup \{A, u_1, \ldots, u_n\}$, then

  $$U_b = \cap u_1 \ldots \cap u_n.[T_1\theta/A] \circ (U_1 \circ \theta).$$

[11] Note that here we necessarily assume that the unification algorithm does not create new variables in the range of the mgu.

If $T_1, \ldots, T_n$ are not unifiable, then the goal fails (and the entire type inference fails).

- (`resolve-sub` $NV\ V\ U_a\ U_b$) calls resolve-for for each variable in Ua, including NV. It applies the resolved, most general solution for each variable to the substitution, thereby effectively also carries out composition of substitutions. `resolve-sub` also adds the trivial substitution $[a/a]$ for each a in NV if a does not already exist in the domain of U_a, this being needed by the copy clause formulation of substitutions. `resolve-sub` also preserves the order of the $\cap$-prefix.

- (`fill-out` $N\ A\ M\ B$) takes a substitution N and a type expression A and makes their $\cap$ and Π prefixes to be of the same length, by appending the prefixes with vacuous abstractions where necessary.

Assuming these predicates, we can now show the following:

Lemma 3.3 *(Prefix-Compatibility of* `polytype`*)*
Under the program `MLtyper`*, for each successful inference of (*`polytype` $NV\ Un\ M\ Ty$*), Ty and Un are prefix-compatible with respect to all variables in NV.*

Proof: We will show that this holds for every `polytype` clause in the program, by simultaneous induction on the size of term M.

- . Since all initially assumed typing relations for constants are declared by clauses of the form

  ```
  pi A\ (polytype A emp M Ty)
  ```

  no substitution for any variable in `A` is implied. Thus for any successful goal resolving with this clause, `Ty` and `emp` are trivially prefix-compatible.

- For the `app` case, a successful inference must have the form

  ```
  polytype NV (newvar Mgu) (app M N) (all Ty).
  ```

  A new free variable t_v is introduced. By inductive hypotheses, the type-substitution pairs $Artype, Mt$, and T, Nt are prefix-compatible with respect to NV and the respective inferences

  (`polytype` $NV\ Mt1\ M\ Ar1$), and (`polytype` $NV\ Nt1\ N\ T1$).

Filling out the substitutions and types with vacuous abstractions clearly does not affect this property. Because $Artype, Mt$, and T, Nt are prefix-compatible and "filled out", we can assume that they have the following forms:

$$Artype = \Pi v_1 \ldots \Pi v_m.A \rightarrow B, \qquad Mt = \cap v_1 \ldots \cap v_m.U_M,$$

$$T = \Pi w_1 \ldots \Pi w_n.S, \qquad Nt = \cap w_1 \ldots \cap w_n.U_N.$$

Now, (**merge-type** $T\ t_v\ Tn$) forms an arrow type **Tn** of the form

$$Tn = \Pi w_1 \ldots \Pi w_n.S \rightarrow t_v.$$

Now (**typeunify** $(t_v :: NV)\ Tn\ Artype\ Mg$) implies that Mg is of the form (after possible α-conversion)

$$Mg = \cap w_1 \ldots \cap w_n \cap v_1 \ldots \cap v_m.U_t$$

with U_t containing $[B/t_v]$. (The prefixes are appended instead of merged to keep polymorphic type variables apart.) This solution must be resolved with the solutions from typing M and N, so (**append-sub** $Nt\ Mt\ MN$) produces

$$MN = \cap w_1 \ldots \cap w_n \cap v_1 \ldots \cap v_m.U_N @ U_M,$$

and (**merge-sub** $MN\ Mg\ Mg2$) produces

$$Mg2 = \cap w_1 \ldots \cap w_n \cap v_1 \ldots \cap v_m.U_N @ U_M @ U_t.$$

(**resolve-sub** $(t_v :: NV)\ (t_v :: NV)\ Mg2\ (Mgu\ t_v)$) now combines and resolves the conflicting solutions U_M, U_N, and U_t to form the substitution of this type inference:

$$(Mgu\ t_v) = \cap w_1 \ldots \cap w_n \cap v_1 \ldots \cap v_m.U.$$

The final **type-sub** goal for the type variable t_v then forms the type (after possible α-conversions)

$$(Ty\ t_v) = \Pi w_1 \ldots \Pi w_n \Pi v_1 \ldots \Pi v_m.B(U).$$

$(Ty\ t_v)$ and $(Mgu\ t_v)$ are therefore prefix-compatible with respect to $t_v :: NV$. Finally, in the derived type and mgu for **(app M N)**, t_v is quantified in front: $\Pi t_v.(Ty\ t_v)$ and $\cap t_v.(Mgu\ t_v)$. Thus they are prefix-compatible for NV.

- for the **abs** case, goals have the form

```
polytype NV (newvar Mgu) (abs M) (all Typ)
```

New term variable x and type variable a are introduced. Since x is assumed to have type a, a and the empty substitution emp are prefix-compatible, and so

```
pi Any\ (ploytype Any emp x a)
```

is a valid `polytype assumption`. Thus by inductive hypothesis,

$$\texttt{polytype}\ (a :: NV)\ Mg\ (M\ x)\ B$$

implies that B and Mg are prefix-compatible with respect to $(a :: NV)$ (for this inference). In this case, $(\texttt{resolve-sub}\ \ NV\ NV\ Mg\ (Mgu\ a))$ can only add trivial substitutions for variables in NV if they do not exist in Mg, since M, being part of a valid type-inference, must already be resolved. Thus $(Mgu\ a)$ remains prefix-compatible with B, with respect to $a :: NV$. Assume B and $(Mgu\ a)$ are of the forms

$$B = \Pi v_1 \ldots \Pi v_n.\tau, \qquad (Mgu\ a) = \cap u_1 \ldots \cap u_m.U,$$

assuming without loss of generality that $m \leq n$. Now $(\texttt{merge-type}\ \ a\ B\ Ty)$ implies

$$Ty = \Pi v_1 \ldots \Pi v_n.a \rightarrow \tau,$$

and the final goal $\texttt{type-sub}\ \ Ty\ (Mgu\ a)\ (Typ\ a)$ implies

$$(Typ\ a) = \Pi v_1 \ldots \Pi v_n.(a \rightarrow \tau)(U[v_1/u_1, \ldots v_m/u_m]).$$

This type is prefix-compatible (by the definition of prefix-compatibility) with $(Mgu\ a)$ with respect to $a :: NV$, since

$$U = U[v_1/u_1, \ldots v_m/u_m][u_1/v_1, \ldots u_m/v_m].$$

Finally, the derived type and mgu for `(app M N)`, $\Pi a.(Typ\ a)+$ and $\cap a.(Mgu\ a)$, necessarily quantifies a in front, and so are prefix-compatible with respect to NV.

- In a `let` typing, `(polytype NV M (let R T) C)`, one only needs to observe that if Ty and Mt are prefix compatible with respect to NV, then `(pi A\ polytype A Mt x Ty)` is a valid type assumption. Therefore (by inductive hypotheses), $(\texttt{polytype}\ \ NV\ Mr\ (R\ x)\ B)$ implies B and Mr are prefix compatible with respect to NV. Assume the following forms:

$$Ty = \Pi v_1 \ldots \Pi v_m.T, \qquad Mt = \cap v_1 \ldots \cap v_{m'}.U_t,$$

$$B = \Pi w_1 \ldots \Pi w_n.S, \qquad Mr = \cap w_1 \ldots \cap w_{n'}.U_r.$$

We still need to append Mt to the end of Mr, since x may be vacuous in R. Since `append-sub` keeps prefixes apart (respecting polymorphic types), and `resolve-sub` does not change the $\cap$-prefix, (`append-sub` $Mr\ Mt\ Mrt$) and (`resolve-sub` $NV\ NV\ Mrt\ M$) means that

$$M = \cap w_1 \ldots \cap w_{n'} \cap v_1 \ldots \cap v_{m'}.U.$$

Assume without loss of generality that $n \leq n'$, (`typesub` $B\ M\ C$) implies that (after possible α-conversion)

$$C = \cap w_1 \ldots \cap w_{n'} \cap v_1 \ldots \cap v_{m'}.S(U).$$

Thus C is prefix-compatible with M with respect to NV.

□

Sample Inferences

Before further discussion on the correctness of the program, it is important to see it in action. In the aforementioned example of typing $\lambda x.(f\ x)$ (where f has type $\Pi v.v \rightarrow v$), the subterm $(f\ x)$ can be assigned type $\Pi w \Pi u.u$, with the accompanying unifier $\cap t \cap v.[v/a, v/v, v/t]$. (This is done by unifying $\Pi v.v \rightarrow v$ with $a \rightarrow t$ for some new variable t, which becomes vacuous in the type and substitution.) Since the unifier and type are prefix-compatible with respect to a, we can create the type $\Pi w \Pi u.a \rightarrow u$ and then apply the unifier to it while merging the prefixes (through `typesub`) to form $\Pi w \Pi u.u \rightarrow u$, which is the correct type for $\lambda x.(f\ x)$, the vacuous quantification over w notwithstanding.

For an example of a `let` expression, consider *(let val $x = \lambda y.y$ in $(x\ x)$ end),* or in our notation

```
polytype nil U (let (x\ (app x x)) (abs y\y)) T.
```

First, the type of $\lambda y.y$ is inferred as $(\Pi v.v \rightarrow v)$ with the accompanying, prefix-compatible substitution $\cap v.[v/v]$. Then x is assumed to have this type, along with the substitution. Now we infer the type of $(x\ x)$. The type of x is inferred twice as $(\Pi v.v \rightarrow v)$ and $(\Pi w.w \rightarrow w)$ with the respectively accompanying mgu's $\cap v.[v/v]$ and $\cap w.[w/w]$. There is no need to fill out the length of any of the Π or $\cap$ prefixes with vacuous abstractions in this case. Now a new type variable t_v is introduced, and the type

$$(\Pi w.(w \rightarrow w) \rightarrow t_v)$$

is formed (by `merge-type`). This type is unified (through `typeunify`) with ($\Pi v.v \rightarrow v$) by concatenating the Π-prefixes. The prefixes are not merged here to respect that fact that both are polymorphic types which can yield arbitrarily many instances or "copies." This unification results in the mgu

$$\cap w \cap v.[(w \rightarrow w)/v, (w \rightarrow w)/tv, w/w].$$

Appending the substitutions $\cap w.[w/w]$ and $\cap v.[v/v]$ yields a structure respecting the prefix of this unifier: $\cap w \cap v.[w/w, v/v]$. This is then merged with the unifier to form

$$\cap w \cap v.[(w \rightarrow w)/v, (w \rightarrow w)/tv, w/w, w/w, v/v].$$

This structure is seen to be consistent by `resolve-sub` (the multiple instances of w/w will lead to redundant search paths during interpretation, but this does not affect correctness). This unifier is then applied to tv (by `typesub`), which moves the $\cap$-prefix to a Π-prefix in the same order, yielding ($\Pi w.\Pi v.w \rightarrow w$). Finally, the variable t_v is abstracted over (even though it became bound to something else) to form the type

$$(\Pi t_v \Pi w.\Pi v.w \rightarrow w)$$

as the type for $(x\ x)$, and hence as the type for *(let val* $x = \lambda y.y$ *in (x x) end).* In this case, since x was not vacuous in the body of the `let`, appending the mgu from the derivation of $\lambda y.y$ was not necessary, but this is harmless in any event (because `append-sub` keeps prefixes apart). At worst more vacuous quantifiers would be added to the type.

Let's now turn to an expression that's not typable. Assume p has type $\Pi v.v \rightarrow v \rightarrow v$. Consider the expression

```
(abs x\ (let (y\ (app y y)) (app p x))).
```

For the top level `abs` clause, a new fugitive variable a is assumed as the type for x. In the `let` expression, $(p\ x)$ will be inferred as having type $\Pi t_v.\Pi v.v \rightarrow v$, which is prefix-compatible with the mgu $\cap t_v \cap v.[v/a, (v \rightarrow v)/t_v]$. (The other possibility of a substituting v is less interesting; it fails too easily.) The type of y in the body of the `let`-expression will therefore be assumed as $\Pi t_v.\Pi v.v \rightarrow v$, along with this unifier. Now, typing `(app y y)` will produce two copies of this type and unifier:

$$\Pi t_v.\Pi v.v \rightarrow v, \qquad \cap t_v \cap v.[v/a, (v \rightarrow v)/t_v],$$

(we omitted the trivial substitutions for simplicity) and

$$\Pi t_w.\Pi w.w \to w, \qquad \cap t_w \cap w.[w/a, (w \to w)/t_w].$$

Another type variable t_z is introduced, and `merge-type` forms

$$\Pi t_w.\Pi w.(w \to w) \to t_z.$$

This is unified with $\Pi t_v.\Pi v.v \to v$, resulting in the mgu

$$\cap t_w \cap w \cap t_v \cap v.[(w \to w)/v, (w \to w)/t_z].$$

Again the prefixes are appended instead of merged to reflect polymorphisms. But then this substitution must be resolved with that resulting from appending the two substitutions from the recursive inferences for y (renaming the bound-variables for emphasis):

$$\cap t'_w \cap w' \cap t'_v \cap v'.[w'/a, (w' \to w')/t'_w, v'/a, (v' \to v')/t'_v],$$

This time we must merge the $\cap$-prefixes, since the list of variables forming the domain of the above unification is the same as the variables from the two recursive type inferences, appended in the same order. Thus `merge-sub` yields the structure:

$$\cap t_w \cap w \cap t_v \cap v.[(w \to w)/v, (w \to w)/t_z, w/a, (w \to w)/t_w, v/a, (v \to v)/t_v].$$

This can not be resolved since a would have to have both w and $w \to w$ as solutions. In other words, the fugitive variable a really represents a *monotype.* Ideally, new variables in the substitution for a should not be Π-quantified until the end. But since we are forced to use abstractions to represent new variables, they must be quantified-over anyway. The only way to distinguish them from other variables is through the fact that they are bound to a fugitive variable. It is critical for the fugitive variable a to remain free in the substitutions (and types) from the recursive type inferences. If they were also λ-abstracted, then appending the prefixes would create two copies of them! Specifically, if the two copies of the unifier from the typing of the two instances of y in $(y\ y)$ were instead

$$\cap a \cap t_v \cap v.[v/a, (v \to v)/t_v] \quad \text{and} \quad \cap a \cap t_w \cap w.[w/a, (w \to w)/t_w],$$

appending them will result in

$$\cap a \cap t_w \cap w \cap a' \cap t_v \cap v[w/a, (w \to w)/t_w, v/a', (v \to v)/t_v].$$

There would then be no difficulty in resolving for a and a'. By virtue of abstract syntax, it was more natural to write the program as it is, which prevents any variable in NV from being captured by abstractions. In section 3.5.3, however, we will see how this subtle problem prevents us from adopting a more straightforward approach to type-inference using `setsub`.

Undoubtedly some parts of the MLtyper program given above are redundant, leading to redundant search paths during actual interpretation. For example, the `resolve-sub` goal in the `app` case only serves to add trivial substitutions. We probably can correct this situation by using `newvar`-abstracted trivial substitutions when we assume type assignments. In fact, it may be valid just resolve substitutions for the free variables in NV. Also, we conjecture that the last `typesub` goal in the `let` case is not needed. Referring to the code for `let` case, we included the mgu Mt instead of *emp* in assuming the typing of x. Thus as long as x appears in $(R\ x)$, any substitution for a in NV carried by Mt must already be applied when forming B. This program was written as it is to guarantee its correctness, and to facilitate the proof of the prefix-compatibility invariance. We now discuss the correctness of the implicit algorithm implemented in the context of known algorithms for type inferencing.

3.5.2 Algorithm $\mathcal{W}_\lambda^*$

The design of a new type inferencing algorithm was not the aim of this thesis. We stress that the most critical problem for formulating a type inferencer in logic programming is still that of object-level unification. The program of this section is an illustration of its necessity. In writing the program, however, it became clear that the algorithm implicitly used is not the original *Algorithm* $\mathcal{W}$ of Milner [7], given in Figure 3.8. That algorithm is easy to define "on paper." The presence of the *close* operation, however, entails a *Gen*-like predicate that not only abstracts over free variables, but must ensure a maximality condition of the type particularly difficult in logic programming (as in proper subset). Furthermore, Algorithm $\mathcal{W}$ assumes the availability of an inexhaustible supply of new variable names. There are several differences between the algorithm implied by `polytype` and Algorithm $\mathcal{W}$, some of them more critical than others.

Let H be a function (a type environment) from term-level variables to types. Let L be a term and σ be a substitution on type variables in H. $\mathcal{W}(H;\ L)$ returns a pair (σ, t) such that L has type t under the type environment $H \circ \sigma$. The algorithm is defined inductively on the structure of L.

$\boldsymbol{\mathcal{W}(H;\ x)} =$ (where x is a term-variable), let $H(x) = \Pi a_1 \ldots \Pi a_n.t$. For a new set of variables $b_1, \ldots, b_n$, return $(\emptyset,\ t[b_1/a_1, \ldots, b_n/a_n])$.

$\boldsymbol{\mathcal{W}(H;\ \lambda x.M)} =$ let a be a new type variable, and $\mathcal{W}(H + (x \mapsto a);\ M) = (\sigma, t)$. Return $(\sigma,\ (a \to t)\sigma)$.

$\boldsymbol{\mathcal{W}(H;\ (M\ N))} =$ let $\mathcal{W}(H;\ M) = (\sigma_1, t_1)$ and let $\mathcal{W}(H \circ \sigma_1;\ N) = (\sigma_2, t_2)$. For a new type variable t_v, find σ, the mgu of $t_1\sigma_2$ and $t_2 \to t_v$. Return $(\sigma_1 \circ \sigma_2 \circ \sigma,\ t_v\sigma)$.

$\boldsymbol{\mathcal{W}(H;\ \textit{let}\ x = M\ \textit{in}\ N)} =$ let $\mathcal{W}(H;\ M) = (\sigma_1, t_1)$ and let

$$\mathcal{W}(H \circ \sigma_1 + (x \mapsto close(H \circ \sigma_1;\ t_1)));\ M) = (\sigma_2, t_2).$$

Return $(\sigma_1 \circ \sigma_2,\ t_2)$.

$\boldsymbol{close(H;\ t)} =$ $\Pi v_1 \ldots \Pi v_n.t$, where $v_1, \ldots, v_n$ are all the free variables in t that are not free in the range of H.

Figure 3.8: Algorithm $\mathcal{W}$

Algorithm $\mathcal{W}$ maintains an explicit *environment* H of type assignments to variables for each type inference. In our program, this environment is encoded by the `polytype` meta-logic program clauses. In typing `(app M N)`, the pair (σ, t_1) is first found for M. But then σ is applied to the current environment before the type (and substitution) of N is found. Since a (closed) meta-logic program clause can not be mutated, we can not apply the substitution to the encoded environment. Thus we find the solutions for M and N in the same environment, and delay the composition and resolution of these substitutions. In the `let` case, we similarly do not mutate the given environment. Instead, the type environment in our case contains not only mappings from term variables to types, but also prefix-compatible substitutions for the types. These serve the same purpose as "updating" the environment, since the solutions in these substitutions will be resolved with those already in the environment. These differences, however, are minor. One way to eliminate them will be to code the environment explicitly using list-like structures instead of relying of meta-program clauses. The program will need to carry this environment as an extra argument. Instead of using $\Rightarrow$, we would append the environment before a recursive inference call. This variation is not only unnecessary, it would be of no help in resolving the most important distinction between our approach and Algorithm $\mathcal{W}$.

Eager Polymorphism

In the `let` case, $\mathcal{W}$ uses the *close* operation, which distinguishes not only variables free from not free, but also free variables that appear only in the type from those that appear in the environment and type. Here, all type variables are λ-bound by default, except for the fugitives. Even a fugitive variable however, can become substituted by λ-bound variables representing new type variables. Recall that in the sample inference that (correctly) failed, the fugitive a for x was given the substitution v/a where v was λ-bound. In typing the body of the `let` expression, two copies of this v, v and w, were necessarily made when the prefixes where appended. Type inference was allowed to continue where in $\mathcal{W}$ it would have failed: v was unified with $w \rightarrow w$. This mistake was only corrected when the substitutions were resolved, revealing that the two solutions v/a and w/a are inconsistent if $v = w \rightarrow w$. We may represent the type environment in a variety of ways, but as long as λ-abstraction (as opposed to strings) are used to represent new variables, this problem will persist. Indiscriminate quantification over variables means that more general (or more polymorphic) types will be derived[12]. In case these types can be resolved, then the variables that *should* represent free variables in the environment can remain rightfully quantified, since the final principal type will quantify over all variables[13].

This method of *eager quantification, delayed resolution,* is shared by another known type-inferencing algorithm: the *Algorithm* $\mathcal{W}^*$ of Appel and Shao [1], given in Figure 3.9. Although the motivation for this algorithm was entirely different from the need for λ-abstraction, it shares several essential characteristics of our approach. For this reason, let the algorithm implicit in the `polytype` program be known as *Algorithm* $\mathcal{W}^*_\lambda$, where λ represents the use of λ-abstraction to represent new type variables. Algorithm $\mathcal{W}^*_\lambda$ is formulated in Figure 3.10 in the style of the other algorithms. Given the Prefix-Compatibility lemma, it is straightforward to show that the `polytype` clauses correctly implement $\mathcal{W}^*_\lambda$.

The motivation for Algorithm $\mathcal{W}^*$ was to support separate compilation, where the types of operators are not always available until link-time. Operators and variables are all given the most general type possible: a free type variable. Terms are assigned a series of possible

[12] It's possible that a constrained type discipline is an alternative to this.

[13] We consider only initial environments without free type variables. If they do contain free variables (initial fugitives), then we would have to resolve for them at the end, and regard all λ-bound variables that appear in any substitution for them as free variables.

An *assumption environment* A is a mapping from term variables to *sets* of types (represented as a list of (x, t) pairs). $A \backslash \{x\}$ represents A minus the mapping for x (or all pairs (x, t)). For term L, $\mathcal{W}^*(L)$ returns a pair (t, A) where t is a type and A is an assumption environment.

$\boldsymbol{\mathcal{W}^*(x)} =$ (where x is a term-variable) let a be a new type variable, return $(a,\ \{(x, a)\})$.

$\boldsymbol{\mathcal{W}^*(\lambda x.M)} =$ let a be a new type variable and let $\mathcal{W}^*(M) = (t, A)$. Let substitution $\sigma = MonoUnify(a, A(x))$ (defined below). Return $((a \rightarrow t)\sigma,\ (A \backslash \{x\}) \circ \sigma)$.

$\boldsymbol{\mathcal{W}^*(M\ N)} =$ let $\mathcal{W}^*(M) = (t_1, A_1)$ and $\mathcal{W}^*(N) = (t_2, A_2)$. For a new type variable t_v, form σ, the mgu of t_1 and $t_2 \rightarrow t_v$. Return $(t_v\sigma,\ (A_1 \cup A_2) \circ \sigma)$.

$\boldsymbol{\mathcal{W}^*(let\ x = M\ in\ N)} =$ let $\mathcal{W}^*(M) = (t_1, A_1)$ and $\mathcal{W}^*(N) = (t_2, A_2)$. Let $TV =$ all type variables in t_1 and A_1. Let $(\sigma, A) = PolyUnify((TV, t_1, A_1), A_2(x))$ (defined below). Return

$$(t_2\sigma,\ A_1 \cup [(A \cup A_2 \backslash \{x\}) \circ \sigma]).$$

$\boldsymbol{MonoUnify(a, T)} =$ assume $T = \{t_1, \ldots, t_n\}$ (a set of types). Return the mgu of the set of unification problems (difference pairs) $(a, t_1), (a, t_2), \ldots, (a, t_n)$.

$\boldsymbol{PolyUnify((TV, t, A), T)} =$ assume $TV = \{a_1, \ldots, a_n\}$ (set of type variables) and $T = \{t_1, \ldots, t_m\}$ (set of types). Begin with $B = \emptyset, P = \emptyset$. Iterate the following for *each* t_i in T, from $i = 1$ to m:

- Pick a set of new variables $\{b_1, \ldots, b_n\}$.
- Let $B = B \cup (A \circ [b_1/a_1, \ldots, b_n/a_n])$
- Let $P = P \cup \{(t_i, t[b_1/a_1, \ldots, b_n/a_n])\}$.

Form σ, the mgu of the set of unification problems P. Return (σ, B).

$\boldsymbol{Match(H, A)} =$ H is a regular type environment. Assume A is of the form $\{(x, t_1), \ldots (x, t_m)\}$ (because of the way it's called), and assume $H(x) = \Pi a_1 \ldots \Pi a_n.t$. Begin with $P = \emptyset$. Iterate the following from $i = 1$ to m:

- Pick a set of new variables $\{b_1, \ldots, b_n\}$.
- Let $P = P \cup \{(t_i,\ t[b_1/a_1, \ldots, b_n/a_n])\}$.

Return the mgu of P.

Figure 3.9: Algorithm $\mathcal{W}^*$

Let H_e, an *extended type environment,* be a mapping from term variables to a pair (σ, t) where t is a type and σ is a substitution. A *candidate substitution* is a structure resulting from the appending of several idempotent substitutions, with therefore possibly multiple solutions for type variables: $[t_1/a, \ldots, t_n/a]$. For a term M, $\mathcal{W}^*_\lambda(H_e;\ M)$ returns a new pair (σ, t). Assume H_e does not contain free type variables initially.

$\boldsymbol{\mathcal{W}^*_\lambda(H_e;\ x)} =$ $H_e(x)$, for term-variable x.

$\boldsymbol{\mathcal{W}^*_\lambda(H_e;\ \lambda x.M)} =$ let a be a new type variable, and let

$$\mathcal{W}^*_\lambda(H_e + (x \mapsto (\emptyset, a));\ M) =_\alpha (\cap\overline{v_n}.\sigma,\ \ \Pi\overline{v_n}.t).$$

Return $(\cap a \cap \overline{v_n}.\sigma,\ \ \Pi a \Pi\overline{v_n}.(a \rightarrow t)\sigma)$.

$\boldsymbol{\mathcal{W}^*_\lambda(H_e;\ (M\ N))} =$ let

$$\mathcal{W}^*_\lambda(H_e;\ M) =_\alpha (\cap\overline{v_n}.\sigma_1,\ \ \Pi\overline{v_n}.t_1), \quad \text{and} \quad \mathcal{W}^*_\lambda(H_e;\ N) =_\alpha (\cap\overline{u_m}.\sigma_2,\ \ \Pi\overline{u_m}.t_2).$$

For t_v a new type variable, let θ be the mgu of t_1 and $t_2 \rightarrow t_v$. Let $\sigma = Resolve(\theta @ \sigma_2 @ \sigma_1)$. Return $(\cap t_v \cap \overline{u_m} \cap \overline{v_n}.\sigma,\ \ \Pi t_v \Pi\overline{u_m}\Pi\overline{v_n}.t_v\sigma)$.

$\boldsymbol{\mathcal{W}^*_\lambda(H_e;\ let\ x = M\ in\ N)} =$ let

$$\mathcal{W}^*_\lambda(H_e;\ M) =_\alpha (\cap\overline{v_n}.\sigma_1,\ \ t_1), \quad \text{and}$$

$$\mathcal{W}^*_\lambda(H_e + (x \mapsto (\cap\overline{v_n}.\sigma_1,\ \ t_1));\ N) =_\alpha (\cap\overline{u_m}.\sigma_2,\ \ \Pi\overline{u_m}.t_2).$$

Let $\sigma = Resolve(\sigma_2 @ \sigma_1)$. Return $(\cap\overline{u_m} \cap \overline{v_n}.\sigma,\ \ \Pi\overline{u_m}\Pi\overline{v_n}.t_2\sigma)$.

$\boldsymbol{Resolve(S)} =$ for candidate substitution S: for each a in the domain of S, iterate the following: (S also contain trivial substitutions $[a/a]$ for each a in the range of S.)

- For $S = [t_1/a, t_2/a]@S'$ (modulo ordering), let θ be the mgu of t_1 and t_2. Let $S = [t_1\theta/a]@(S' \circ \theta)$.
- For $S = [t_1/a]@S'$ such that t_1 is the only substitution for a in S, let $S = [t_1/a]@(S' \circ [t_1/a])$, and end iteration for this a.

Return the resolved S.

Figure 3.10: Algorithm $\mathcal{W}^*_\lambda$

types, in an *assumption environment,* which are then resolved when the actual types of the operators are known. The algorithm consists of two parts: the $\mathcal{W}^*$ function, which does not take in any environment as input, and *Match*, which takes a regular type environment (as in $\mathcal{W}$) and an assumption environment (a set of possible, most-general type assignments generated by the $\mathcal{W}^*$ function), and resolves them with the type environment. *Match* returns a substitution representing the real types of the terms. Note that all recursive calls to $\mathcal{W}^*$ can be made in parallel. Composition and resolution of substitutions must take place *post-hoc.* The use of assumption environments corresponds to the *candidate substitutions* of $\mathcal{W}^*_\lambda$. The candidate substitutions are appendings of substitutions in preparation for resolving them. They include possibly conflicting solutions, such as $[v/a, w/a, (w \rightarrow w)/v]$. The role of the *MonoUnify* operation of $\mathcal{W}^*$ corresponds exactly to that of *Resolve* in $\mathcal{W}^*_\lambda$. That is, both algorithms ensures that each fugitive variable can have only one solution by resolving conflicting solutions *after* they have been generated. $\mathcal{W}^*$ does this only in the `abs` case, while $\mathcal{W}^*_\lambda$ does the resolution as soon as possible (recall that the `resolve-sub` goal of the `abs` case only serves to add trivial substitutions). No resolution is performed at all in the `app` case of $\mathcal{W}^*$. All conflicting solutions are recorded in the assumption environment. The *MonoUnify* operation in the `abs` case is sufficient to block all inconsistencies. This suggests that parts of our algorithm are redundant, but this difference is largely cosmetic[14].

Most significantly, note the lack of *close* in the `let` case of $\mathcal{W}^*_\lambda$. Maximal (possibly more than maximal) polymorphism in the body of `let` expressions are ensured by *PolyUnify*. Consider the expression $let\ x = M\ in\ N$. The purpose of *PolyUnify* is to match each possible solution (type) t_i for x, found by independently typing N, with a fresh instance of the type t derived for M. By fresh instance we of course mean replacing each variable in t with a new variable, thus allowing polymorphism. Note **especially** that the set of type variables TV given to *PolyUnify*, for which fresh variables are to be "made from" consists of *all* variables in t, the type for M, *and* the accompanying assumption environment to the derivation of t. No distinction is made between variables free in the type alone and those that might be associated with a fugitive (and would be kept free by $\mathcal{W}$'s *close* operation). Thus polymorphism is allowed on all variables. The mistake is only to be caught later. In $\mathcal{W}^*_\lambda$, there is no need for *PolyUnify* to grant polymorphism. The necessary λ-abstraction

[14] The author was not aware of the existence of $\mathcal{W}^*$ when writing the `polytype` program.

of new variables and the appending of abstraction prefixes serve this purpose. *Resolve* (or `resolve-sub` in the implementation) alone suffices.

Of course there are differences between $\mathcal{W}^*$ and $\mathcal{W}^*_\lambda$ as well. In our case there is no need to apply *Match* afterwards. We can begin with a type environment (with no free type variables). The most pronounced difference in this respect is the appending of the (extended) type environment with the type and substitution resulting from the typing of the binding term (M) in $\mathcal{W}^*_\lambda$. This is needed since conflicting substitutions are to be resolved when typing the body of the `let`, and not entirely afterwards. We conjecture with confidence that $\mathcal{W}^*$ would remain correct if a type environment is given as input, and in the base case for a term variable x, a fresh copy of the type of x in the environment is returned. In other words, $\mathcal{W}$ and $\mathcal{W}^*$ represent two extreme ways of type inferencing. $\mathcal{W}$ makes sure that variables quantified during `let`-typing are those that can be, and resolves conflicts immediately. $\mathcal{W}^*$ allows nearly everything to be typed with the most general types possible, and delays conflict resolution as much as possible. In this way $\mathcal{W}^*_\lambda$ is a hybrid algorithm between $\mathcal{W}$ and $\mathcal{W}^*$. Despite appearance, however, the differences $\mathcal{W}^*_\lambda$ has with $\mathcal{W}$ are more fundamental than those with $\mathcal{W}^*$.

While the foregoing does not constitute a formal proof, it should render strong confidence in the correctness of $\mathcal{W}^*_\lambda$. The formal proof of its correctness is certain to be saturated with technical detail, and will likely be similar to Appel and Shao's proof of the correctness of $\mathcal{W}^*$ relative to $\mathcal{W}$. Slight modifications of the algorithm and implementation (such as moving the resolution of substitutions to different places) will also be needed to facilitate a proof. This will be done eventually[15]. The aim of this thesis, however, is to show that complex formulations such as principal type-inferencing are feasible in the context of higher-order abstract syntax and a HOHH meta-logic. For this we have provided and demonstrated the techniques of object-level unification and prefix-compatibility, and therefore believe that this aim has been satisfied.

[15] At the time of final submission of this thesis, proofs of soundness and completeness have been found and will appear in a future manuscript.

3.5.3 A Quick and Simple Alternative

The difficulties of the previous section, namely the explicit handling of object-level substitutions and keeping the prefixes compatible, can be avoided if one can accept a singular use of `!`. In this section we will introduce another solution to ML type-inferencing by taking full advantage of the logic programming interpreter's built-in unification and resolution algorithms. We will also need the fact that if `T` is typable then typing `(let R T)` is the same as typing `(R T)`. Unfortunately, adopting this equivalence means (among other things out of scope here) that we can no longer remain in the simple L_λ language, at least not in a direct way.

First we will extend the use of `setsub` beyond unification, to implement a kind of *Gen* predicate (which we already know it is capable of). We will do this for the signature of type expressions used in the previous section, but it can clearly be extended to (first-order) expressions of any signature. Given an expression `T` and a list `V` such that all free (logic) variables appearing in `T` also appear in `V`, we want to close the type into a quantified expression, i.e, an abstract-syntax representation. We can do this by using `setsub` to force logic variables in `V` back into closed structures, and then quantifying those closed structures. This operation is simpler than unification (`quick-unify`), were we needed to prove that `setsub` preserved the maximum generality of the unifier with respect to an original domain of unification. Here, there is no "original domain" of unification to consider. Any new closed structure can be used to close off the type expression.

Since there are arbitrarily many distinct logic variables in `V`, we would need an denumerable supply of new closed structures to "`setsub`" the logic variables. Unfortunately we can not use dynamically generated eigenvariables for this purpose, since these variables can not appear in `T`, the top level expression. The closed structures we use are only an intermediate representation; they will be abstracted over to form quantified type expressions, abstract-syntax style. A variety of data structures can be used for this task, such as nested lists of increasing depth. We choose to use a constructor `vc` of type `int -> poly` that takes an integer and returns a closed term, i.e, `(vc 1)`, `(vc 2)`, etc.[16]. The code for the predicate `close-type` is given in Figure 3.11. A type `T` can then be closed in two

[16] Natural numbers can easily encoded using a zero symbol and a successor function symbol. In the code we use λProlog's built in `int` type for convenience only.

```
type vc int -> poly.
type close-type (list poly) -> poly -> poly -> o.
type close-type1 int -> (list poly) -> poly -> o.
type close-type2 int -> (list poly) -> poly -> poly -> o.

close-type V T M :-  close-type1 0 V T, close-type2 0 V T M.

close-type1 N nil T.
close-type1 N (V::Vs) T :-
  setsub (vc N) V, Nn is (N + 1), close-type1 Nn Vs T.

close-type2 N nil M C :- copypoly C M.
close-type2 N (V::Vs) M (all C) :-
  pi a\ (copypoly a (vc N) =>
   (Nn is (N + 1), close-type2 Nn Vs M (C a))).

setsub X X :- !.
setsub X Y.
```

Figure 3.11: Generalization of Type Expressions Through `setsub`

stages. First, in `close-type1`, each member of the list `V` is "setsubed" to a distinct `(vc N)` structure (so `N` is never larger than the length of this list. Thus the logic variables in `V` are instantiated to closed structures, while non-logic variables are left alone. Then, in `close-type2`, the `(vc N)` structures are abstracted over by copying each `(vc i)` to a unique corresponding eigenvariable (`a`). Since this eigenvariable can not appear free in `C`, `(all C)` is an abstraction over `(vc i)`.

This operation, plus `setsub`, are the only auxiliary operations required by the new type-inferencing program, which is given in figure 3.12. Assume it shares a signature with the program of the previous section.

Types are inferred by solving `(quick-infer N T)` for term `N`. The predicate `quicktype` corresponds to `polytype` of the previous section. Each `quicktype` clause keeps track of any possibly-free variables that may occur in the inferred type in a list. Initial polymorphic type assignments are assumed by clauses of the form

```
pi X1\...pi Xn\ (quicktype (X1::...Xn::nil) N T)
```

for term `N` and type `T` with type variables `X1,...,Xn`. That is, meta-level `pi`-quantification is used to directly represent object-level `all`-quantification, though the program still derives types of the object-level form. If we wish to re-incorporate derived polymorphic types of

```
type quicktype (list poly) -> tm -> poly -> o.
type quick-infer tm -> poly -> o.

quicktype nil zero integer.
quicktype nil succ (arr integer integer).
quicktype (X::nil) op (arr X X).

quicktype (B::Bv) (app M N) B :-
  quicktype Mv M (arr A B), quicktype Nv N A,
  append Mv Nv Bv.

quicktype (A::VB) (abs M) (arr A B) :-
  pi x\ (quicktype (A::nil) x A => quicktype VB (M x) B).

quicktype VM (let R T) B :-
  quicktype M T Sometype, quicktype V (R T) B,
  append V M VM.

quick-infer N T :- quicktype M N S, close-type M S T.
```

Figure 3.12: ML-QuickTyper

the form (all Ty) back into the program, we can do so by transforming this closed type expression (assigned to a term N) into such a program clause (pi A) by the following predicate:

```
assume-type Vs N (all T) (pi A) :-
  pi x\ (monotype x => assume-type (x::Vs) N (T x) (A x)).
assume-type Vs N T (quicktype Vs N T) :- monotype T.
```

Solving

```
assume-type nil N (all Ty) (pi A)
```

transforms a closed type expression (assigned to a term N) into a meta-level program clause of the form of quicktype assumption clauses.

At this point, one might be tempted to use assume-type to re-define the let clause as follows:

```
quicktype V (let R T) B :-
  pi x\ (sigma PiS\ (
    quicktype M T S, close-type M S CS,
    assume-type nil x CS PiS,
```

```
    PiS => quicktype V (R x) B)).
```

This method, however, will not work. Recall that in the MLtyper program of the previous section, the fugitive type variables NV introduced in typing `abs` clauses are not Π- or $\cap$-bound in the recursive calls to `polytype`. They are only quantified when the type of the `abs` expressions are formed. Yet, these fugitives are still *variables,* in that they are subject to substitution. In using logic variables as type variables, however, there is no way to distinguish the class of fugitive variables from variables introduced by resolving with a polymorphic type assumption. In the above clause for `quicktype`, the goal `close-type M S CS` will abstract over *all* logic variables in `M`. The result is that some expressions which are not typable are assigned types anyway. The example of the previous section

```
(abs x\ (let (y\ (app y y)) (p x)))
```

is not typable because `(p x)` should have *monotype* $v \to v$, which is bound to a, the fugitive introduced as the type for x. Using the above `quicktype` clause however, `y` will be assumed to have the closed type $\Pi v.v \to v$, and the `let` expression will be incorrectly inferred as having type $\Pi v.v \to v$. This is one instance which the approach of the previous section is superior to the current one, even discounting the use of `setsub`. There does not seem to be a way around this problem using the logic variables approach, except to adopt the alternative way of typing `let`, which in a way is not to bother with `let` at all.

Comparing the Approaches

Clearly, `quick-infer` is a much simpler program than MLtyper of the previous section, which required a series of auxiliary operations to deal with object-level substitutions, including unification. In the program here, the process of generating new (logic) variables and of finding and resolving substitutions are pushed to the meta-level. In fact, apart from maintaining the list of possibly-free variables, the `quicktype` clauses are not much different from the type-checker of section 3.1. There is also no dilemma of when to consider a type expression with variables as a closure. In the MLtyper program, the default was that types are closed (except for the ephemeral fugitives), which required the prefix-compatibility property to be maintained to allow for substitutions to remain consistent.

Here, the default is that types are open, and are only closed at the very end. However, the two programs also share basic similarities in structure. The list of (possible) logic variables in the `quicktype` clauses corresponds to the mgu-substitution carried by each `polytype` clause. They represent the same relationship between the substitution for variables and the type expression being derived. The difference is that `polytype` must maintain the substitution explicitly, while in `quicktype` the substitution is implicitly kept in the meta-level, and thus only a list of variables is needed. Similarly, the relationship between this list of variables and the type expression is kept automatically, while `polytype` needs to maintain prefix-compatibility. The corresponding invariance in `quicktype` to prefix-compatibility in `polytype` is simply the following:

Proposition: *For each successful inference of* `(quicktype L N T)`*, every free variable in* `T` *also appears as an individual member of the list* `L`*.*

This fact is trivial to verify by examining each `quicktype` clause.

It is straightforward to show that the `quick-infer` program is correct, despite the cut. The trade-off is that this correctness would be with respect to to `?-` (depth-first interpretation), and not to the logical notion of intuitionistic provability. To make sure non-logical consequences such as variable-check can not result from `close-type` we can require that no clause other than the one for `quick-infer` is visible in any other program [17]. We should also require that all uses of `quick-infer` involve only (meta-level) closed-terms. The fact that a type-inferencing predicate is available does not by itself destroy the declarativeness of other programs, since this predicate can also be given a purely declarative formulation.

Eliminating Vacuous Quantifiers

Both the `quicktype` and the `polytype` programs yield types with vacuous Π-quantifiers. In fact, the prefix-compatibility technique requires that vacuous abstractions not be eliminated in the middle of type-inferencing. Although it should be possible to define a predicate that eliminates vacuous quantification while preserving prefix-compatibility, this is not necessary for our purposes. Vacuous type-quantifiers from an object-level polymorphic type expression (or type scheme) can be eliminated after the type of the top-level term

[17] perhaps by a module system.

is inferred. The following `remvac` predicate removes vacuous `all`-quantifiers from a type expression:

```
remvac (all x\A) B :- remvac A B.
remvac (all A) (all B) :-
  pi a\ (monotype a => (appear-in a (A a), remvac (A a) (B a))).
remvac A A :- monotype A.

appear-in A (all B) :- pi x\ (appear-in A (B x)).
appear-in A A :- monotype A.
appear-in A (arr B C) :- appear-in A B; appear-in A C.
```

In the first clause for `remvac`, since `A` is quantified outside of the scope of the bound variable `x`, `x\A` must be vacuous for `x`.

3.5.4 Comparison with Other Research

Ideally, one would like to have a direct formulation of the ML typing system (e.g, the Damas-Milner calculus) independently of meta-programming principles. That is, to interpret typing rules directly as logic program clauses, such as Hannan did in [19]. However, the type schemes derived by such programs can not be interpreted as principal types *within* the meta-logic for the reasons we have discussed. Furthermore, Hannan had to bypass the difficulties of `let`-typing in the same manner as our `quicktype` program. Full β-reduction was used instead of β_0-reduction.

To allow `let`-expressions to be typed naturally, in [21] Harper defined an "algorithmic" version of the Damas-Milner calculus for the express purpose of allowing the modified typing rules of the new calculus to become (Elf) logic programs that yield principal type schemes. He defined a predicate called *witnessed* that formulates, proof-theoretically, the maximality condition of Algorithm $\mathcal{W}$'s *close* operation. Specifying the *witnessed* predicate directly as logic programming, however, entails an unacceptable degree of non-determinism. In other words, it still amounts to having to use a *Gen*-like predicate. Harper's conclusion was that "meta-programming is unavoidable," such as in the use of an extra-logical construct. In the `polytype` program we bypassed the specification of the maximality condition of *close* with the method of "eager quantification, delayed resolution." We also did not require extra-logical constructs by formulating object-level unification (the purely declarative version).

The use of `!` in the `quicktype` program implements a *Gen*-like predicate for meta-level free variables in types, so they can be interpreted as type schemes from within the meta-language. In practice, this allows the meta-logical control of the static type-checking of programs without re-typing a symbol every time it appears. Because the `quicktype` program uses full β-reduction to type `let`-expressions, however, retyping of `let`-bound local terms would still occur. The `polytype` program avoids both retyping and the `!` (though it would require streamlining before practical application).

Type Checking and Type System Formulation

The type-inferencing programs developed in this section can not be used *directly* for type checking. This is because the *meta-level* structure of type schemes such as `all T\ (arr T T)` is very different from that of types such as `(arr integer integer)`. Also, the order of Π-quantifiers in a type expression matters not to the meaning of the type, but matters significantly in the `polytype` program. Only one ordering is recognized by the programs. The typing programs have the same restrictions as the object-level unification programs, for the same reason of using the ground representation. Fortunately, only slightly more meta-programming is necessary to allow the type-inferencing programs to also perform type-checking.

Given a principal type scheme $\Pi v_1 \ldots \Pi v_n.T$ and a type $\Pi u_1 \ldots \Pi u_m.S$, all that's required is object-level matching to ensure that S is an instance of T with $v_1, \ldots, v_n$ replaced by new (meta-logic) variables.

```
subsume-type (all T) S :-
  pi v\ (montype v => copypoly v Xv => subsume-type (T v) S).
subsume-type T (all S) :-
  monotype T, pi v\ (montype v => subsume-type T (S v)).
subsume-type T S :- monotype T, monotype S, copypoly T S.
```

By formulating a type-inferencing algorithm as a logic program, and by slightly altering it for type checking, we have arrived, albeit indirectly, at a logic programming, i.e. *proof-theoretic* formulation of the ML typing system. The `has-type N S` relation below holds *if and only if* the type judgment $N : S$ holds under the type environment encoded by `polytype` clauses.

```
has-type N S :-  polytype nil Mgu N T, subsume-type T S.
```

3.5.5 Explanation-Based Generalization with Principal Type Schemes

As a sample application of specifying type-inferencing in meta-logic, consider the following typing declarations (or type environment):

$$a : \Pi X.\Pi S.(nodebt\ X) \to (loves\ S\ X) \to (happy\ X)$$
$$b : \Pi X.\Pi Y.(greater\ Y\ 0) \to (bankbalance\ X\ Y) \to (nodebt\ X)$$
$$t_1 : (greater\ 100\ 0)$$
$$t_2 : (bankbalance\ harry\ 100)$$
$$t_3 : (loves\ mary\ harry).$$

That is, the terms *harry*, 100, etc..., are considered as types while terms such as *nodebt* and *loves* are considered as type constructors. We will use the `quicktype` program for this illustration. The above definitions can be represented by:

```
quicktype (X::S::nil) a
  (arr (not-in-debt X) (arr (loves S X) (happy X))).
quicktype (X::Y::nil) b
  (arr (greater Y zip) (arr (bank-balance X Y) (not-in-debt X))).
quicktype nil t1 (greater one-hundred zip).
quicktype nil t2 (bank-balance harry one-hundred).
quicktype nil t3 (loves mary harry).
```

Inferring the type for the term $(a\ (b\ t_1\ t_2)\ t_3)$, or

```
quicktype V (app (app a (app (app b t1) t2)) t3) S, close-type V S T.
```

will yield

```
T = happy harry.
```

This is not surprising given the well-known correspondence between λ-terms and proof terms, or the general principal of formulas-as-types [18]. What is interesting is that pure λ-terms, by which we mean terms not annotated with type information, naturally embody a form of generalization in their principal type schemes. If we had "abstracted" from the above proof term the *training-instance* declaration that "mary loves harry," represented by the term t_3, we will derive the following principal type assignment:

$$\lambda t_3.(a\ (b\ t_1\ t_2)\ t_3): \quad \Pi S.(loves\ S\ harry) \rightarrow (happy\ harry).$$

This is exactly the explanation-based generalization (or partial deduction) of this proof object relative to the "operational" clause ($loves\ mary\ harry$). Furthermore, we know that this is the *maximal* generalization derivable from the proof, since it is a *principal* type. Thus the criterion of operationality can be captured in proof terms by λ-abstractions. For another example, if we had abstracted out all three training-instance declarations and form the (proof) term

$$\lambda t_1 \lambda t_2 \lambda t_3.(a\ (b\ t_1\ t_2)\ t_3),$$

we will arrive at the following generalization:

$$\Pi X.\Pi Y \Pi S.(greater\ Y\ 0) \rightarrow (bankbalance\ X\ Y) \rightarrow (loves\ S\ X) \rightarrow (happy\ X).$$

Indeed, if we had abstracted over a and b as well, we will arrive at a propositional tautology.

Note also that object-level unification is key to the successful meta-programming of both EBG and type inference.

Studying EBG and partial deduction in the framework of the formulas-as-types principle can be conducted naturally in logic programming. We have just provided some basic tools for this. Results from research in one field could lead to insight into unsolved problems and possible extensions in the other. For example, free type variables in the type environment would correspond to existentially quantified statements. Principal type-inference would therefore also yield $\exists$-quantified generalizations. Although principal types represent only a restricted class of logical formulas (there can not be embedded Π's), all Horn clauses (and more) can be represented. One possible extension of this technique is to use

[18] The `let` construct in this context naturally corresponds to proving a general lemma before the proof a main theorem. The lemma can be used repeatedly in the main proof, each time with a different instantiation.

a dependent type system, such as used in the LF logical framework [22], that do allow embedded Π-quantification. The Elf programming language [46] directly operationalizes this type theory in the same way Prolog operationalizes Horn clauses. Solving a query G in the language represents finding a proof-term for G interpreted as a type. Unfortunately, Elf proof terms are not "pure" in the sense that they are annotated with type information (e.g, $\lambda x : \sigma$ instead of simply λx). Additional meta-programming is necessary to derive generalizations. Furthermore, Dowek has shown in [10] that LF-typability for pure terms is undecidable. This should not deter the finding of restricted cases that are decidable, however.

Chapter 4

Meta-Programming Higher-Order Formalisms

In the previous chapters we mainly developed tools for meta-programming first-order formalisms in a higher-order meta-language. Even though the type-inferencing section dealt with higher-order terms representing simple functional programs, the types were still mainly first-order structures, requiring only first-order object-level unification and first-order copy clauses. We now turn our attention to the case where the object-level formalism is itself higher-order, and requires object-level higher-order unification. In general, the higher-order case presents more problems of how to represent object-level terms at the meta-level. In addition to variables, the problem of normalization also must be addressed explicitly. This requires an object-level formulation of λ-abstraction, β-conversion, as well as type-checking in case the object-theory is typed. Fortunately many of these additional problems have been addressed by previous research on higher-order logic programming, notably that of Felty [12]. The work here is built on top of these earlier efforts.

We will naturally address the problem of higher-order object-level unification. The definition of copy clauses and the Substitution encoding theorem are valid for first and higher-order terms, but the unification algorithms of the Chapter 2 only work for the first-order case. In particular, the `setsub` technique is only valid for first order unification, which has the property that variables in the most general unifier is a subset, up to renaming, of the original variables in the domain of unification. This property fails in the higher-order case. To ease (or rather, modularize) the task of meta-programming object-level higher-order systems, we will confine ourselves to the L_λ language and its simplified form of unification.

4.1 Background: Simplifying Higher-Order Unification

Because of the excessive non-determinism inherent in the algorithm for higher-order unification [27, 50], the algorithm can be easily misused, or over-used. In [37], Miller proposed that higher-order unification problems can be decomposed into more manageable components. He noted through the Substitution Corollary of Chapter 2 that a unification problem involving an arbitrary β-redex $(M\ T)$ can be replaced by a new variable N and the logic programming query

$$\forall x.[\![x,\ T : \sigma]\!] \Rightarrow [\![(M\ x),\ N : \tau]\!]$$

(where M has type $\sigma \rightarrow \tau$.) This query contains a much simpler β-redex $(M\ x)$, called a β_0-redex, where x is a universally quantified bound variable, which in a goal clause effectively amounts to a constant symbol.

The intuition behind simplifying higher-order unification is that once terms are reduced to β-normal form, they are not unlike first-order terms in structure. The exceptions are that there may be λ-bound expressions and subterms of the form $(M\ N)$ where M is a free, higher-order variable. In β-reduced terms, λ-bindings are not much of a problem. They are merely scoping constructs over constant symbols, and unification would just have to respect the scopes of these constants when forming solutions. Intuitively, we know that the term-tree for $(M\ N)$ looks like the term-tree for M with N appended onto some of its leafs. Instead of finding the solution to the composite term $(M\ N)$, we can first try find what out M is, by pruning the N subterms off from $(M\ N)$. This can be done by applying an arbitrary constant symbol x to M, yielding $(M\ x)$. When the solution for $(M\ x)$ and N have been found (recursively), we can re-attach N to M by substituting it for x, which occupied the same positions in the term tree for M as N should. What remains is to find solutions to first-order structures with λ-bindings. This can be done by upgrading the first-order unification algorithm to deal with scoped constants. This is L_λ unification. In this chapter, we will repeatedly see how even higher-order object-level expressions are usually treated as first-order structures at the meta-level. The "higher-ordered" aspect of the meta-language can be restricted to its ability to deal with object-level scoping constructs, i.e, in supporting so-called "higher-order" abstract syntax.

By replacing regular higher-order unification problems with copy clause goals of the form in the conclusion of the Substitution corollary, the nondeterminism of the higher-order unification algorithm is captured explicitly by copy clauses. Choices between substitutions are now choices between copy clauses. Specifically, the $[\![x,\ t:\sigma]\!]$ clause leads to *projections*, while the $[\![k_i,\ k_i:s_i]\!]$ clauses lead to *imitations* (see [50] for the best description of higher-order unification).

The Substitution corollary alone, however, is not enough to provide a means to transform all λProlog expressions into L_λ beyond second order. The problem is that copy clauses can themselves introduce new, non-β_0 redices. Suppose T is a free logic variable of type $i \to i$ and M is of type $(i \to i) \to i$. Then β redex $(M\ T)$ in a goal can be replaced with a new variable N and the condition $\forall x([\![x,\ T:i\to i]\!] \Rightarrow [\![(M\ x),\ N:i]\!]$. But

$$[\![x,\ T:i\to i]\!] = \forall U \forall V(\mathit{copyi}\ U\ V \Rightarrow \mathit{copyi}\ (x\ U)\ (T\ V)),$$

which as a definite clause (as is its role in the condition of N) is not properly L_λ since T is applied to another essentially existential variable.

In [37] it was originally suggested that this problem can be solved by nested application of the transformation suggested by the Substitution corollary on these new non-β_0 redices. Since each time the type of the nested copy clauses is lowered, this process will terminate.

Although this process will yield a L_λ clause, it is not correct (it does not preserve all the solutions of the untranslated form). This was observed[1] with the example $(P\ h) = (g\ h)$ where g is of type $((i \to i) \to i) \to i$ and h is of type $(i \to i) \to i$ and P is a free logic variable. The translated query $\forall x([\![x,\ h:(i\to i)\to i]\!]'' \Rightarrow [\![(P\ x),\ (g\ h):i]\!])$, where $[\![x,\ h:(i\to i)\to i]\!]''$ is transformed to a L_λ clause by nested application of the translation, will produce the answer $P = \lambda x.(g\ h)$, but not $P = g$. In the next section we fully

[1] by Fernando Pereira

demonstrate how λProlog can be translated to the simple L_λ language.

4.2 Translating λProlog to L_λ

Observe that in the Substitution corollary the conclusion holds only under the set of assumptions $K = \{[\![k_i,\ k_i : s_i]\!] \mid 0 \leq i \leq m\}$ where the $k_i's$ represent all constant symbols of the signature. With λProlog, the signature necessarily expands when a `pi`-goal is solved by introducing an eigenvariable. The problem is that when an eigenvariable $y : \sigma$ is added to the signature, the corresponding copy clause $[\![y,\ y : \sigma]\!]$ is *not*. This is the root cause of the failure of the translation described above.

If one is to program strictly in L_λ, then it is not enough to merely extend the signature, but also the assumed program clauses during $\forall$-introduction (in a uniform proof). Thus one method is to change the interpretation of `pi`-goals in L_λ. Since the meaning of universal quantification is different depending on whether it appears in a definite clause or a goal clause, this amounts to *polarizing* the definition of `pi`-quantification. We say that a `pi`-quantifier is *negative* if it occurs at the outermost level of a definite clause, and *positive* if it occurs at the outermost level of a goal clause. This modified L_λ can still be defined within λProlog where goals of the form $\forall^+ G$ have the meaning

$$\forall x.[\![x,\ x : \sigma]\!] \Rightarrow (G\ x).$$

In other words, we merely include the clause

$$\forall^+_\sigma G \Leftarrow \forall x([\![x,\ x : \sigma]\!] \Rightarrow (G\ x)).$$

This uniform approach will solve the problems of third-order (or higher) copy clause translations. Since copy clauses also involve universal quantification, this necessarily also alters the definition of copy clauses. This amounts to moving the level of polarization from that of $\forall$-quantifiers in general to the level of copy clauses. Naturally, we will have to show that the relationship between copy clauses and substitutions is preserved by the polarization.

4.2.1 Polarizing Copy Clauses

From the above discussion, the straightforward way to define polarized copy clauses would be as follows:

Definition 4.1 *(Polarized copy clauses, 1st version)*

$$
\begin{aligned}
&[\![M,\ N:\sigma]\!]^{+} = [\![M,\ N:\sigma]\!]^{-} = \mathit{copy}\sigma\ M^{\beta\eta}\ N^{\beta\eta}\\
&\quad \textit{for primitive type } \sigma.\\
&[\![M,\ N:\sigma\to\tau]\!]^{-} = \forall u \forall v([\![u,\ v:\sigma]\!]^{+} \Rightarrow [\![(M\ u),\ (N\ v):\tau]\!]^{-})\\
&[\![M,\ N:\sigma\to\tau]\!]^{+} =\\
&\quad \forall u \forall v([\![u,\ u:\sigma]\!]^{-} \Rightarrow [\![u,\ v:\sigma]\!]^{-} \Rightarrow [\![v,\ v:\sigma]\!]^{-} \Rightarrow [\![(M\ u),\ (N\ v):\tau]\!]^{+})
\end{aligned}
$$

This definition is correct (in that it preserves the meaning of copy clauses) but not efficient. The extra copy clauses will mean redundant search paths in the λProlog interpreter (though the solutions will be α-convertible to each-other). Not all three copy clauses $[\![x,\ x:\sigma]\!]$, $[\![x,\ y:\sigma]\!]$ and $[\![y,\ y:\sigma]\!]$ are needed. The following is an improved version of polarized copy clauses[2]:

Definition 4.2 *(Polarized copy clauses, 2nd version)*

$$
\begin{aligned}
&[\![M,\ N:\sigma]\!]^{+} = [\![M,\ N:\sigma]\!]^{-} = \mathit{copy}\sigma\ M^{\beta\eta}\ N^{\beta\eta}\\
&\quad \textit{for primitive type } \sigma.\\
&[\![M,\ N:\sigma\to\tau]\!]^{-} = \forall u \forall v([\![u,\ v:\sigma]\!]^{+} \Rightarrow [\![(M\ u),\ (N\ v):\tau]\!]^{-})\\
&[\![M,\ N:\sigma\to\tau]\!]^{+} = \forall x([\![x,\ x:\sigma]\!]^{-} \Rightarrow [\![(M\ x),\ (N\ x):\tau]\!]^{+})
\end{aligned}
$$

The following formal results establish the correctness of this definition and its uses. Analogous results would be required to prove the 1st version of polarized copy clauses correct, but we will forego that in favor of studying the more efficient version. The Same-Variable lemma of Chapter 1 is needed here, since it establishes the following "soundness and completeness" results.

Lemma 4.1 *(Polarization soundness)*
Under any signature, for any terms $M:\tau$ and $N:\tau$,

$$\vdash\ [\![M,\ N:\tau]\!]^{-} \Rightarrow [\![M,\ N:\tau]\!]^{+}.$$

Proof: by induction on the type τ.

[2] This was originally suggested by Amy Felty as a correction to the copy clause definition in [37], though it alone does not correct the so-called ″-translation.

The primitive-type case is by definition. If $\tau = i \to j$, then we have

$$[\![M,\ N:\tau]\!]^- = \forall x \forall y([\![x,\ y:i]\!]^+ \Rightarrow [\![Mx,\ Ny:j]\!]^-)$$

and

$$[\![M,\ N:j]\!]^+ = \forall x([\![x,\ x:i]\!]^- \Rightarrow [\![Mx,\ Nx:j]\!]^+).$$

Let c be a fresh eigenvariable. The proof is by the following derivation:

$$\dfrac{\dfrac{\dfrac{[\![c,\ c:i]\!]^- \vdash [\![c,\ c:i]\!]^+ \qquad [\![c,\ c:i]\!]^-, [\![Mc,\ Nc:j]\!]^- \vdash [\![Mc,\ Nc:j]\!]^+}{([\![c,\ c:i]\!]^+ \Rightarrow [\![Mc,\ Nc:j]\!]^-), [\![c,\ c:i]\!]^- \vdash [\![Mc,\ Nc:j]\!]^+}\Rightarrow L}{[\![c,\ c:i]\!]^+ \Rightarrow [\![Mc,\ Nc:j]\!]^- \vdash [\![c,\ c:i]\!]^- \Rightarrow [\![Mc,\ Nc:j]\!]^+}\Rightarrow R}{\forall x, y([\![x,\ y:i]\!]^+ \Rightarrow [\![Mx,\ Ny:j]\!]^-) \vdash \forall x([\![x,\ x:i]\!]^- \Rightarrow [\![Mx,\ Nx:j]\!]^+)}\forall R, \forall L, \forall L$$

The remaining premises of this derivation hold by the inductive hypotheses on the shorter types i and j.

□

Theorem 4.2 *(Polarization Completeness)*

Under any signature Σ, for some correctly typed (Σ-)terms $a_1 : \tau_1, \ldots, a_n : \tau_n$ and $b_1 : \tau_1, \ldots, b_n : \tau_n$, let Γ be the set of (unpolarized) copy clauses

$$\{[\![a_1,\ b_1:\tau_1]\!], \ldots, [\![a_n,\ b_n:\tau_n]\!]\}.$$

Let $\Gamma^- =$

$$\{[\![a_1,\ b_1:\tau_1]\!]^-, \ldots, [\![a_n,\ b_n:\tau_n]\!]^-\}.$$

For any Σ-terms M and N of type τ,
$\Gamma \vdash [\![M,\ N:\tau]\!]$ *if and only if* $\Gamma^- \vdash [\![M,\ N:\tau]\!]^+$

Proof: by induction on the height of proofs.

The base case is immediate since the polarized definitions converge at atomic copy clauses. For Augment ($\Rightarrow$-introduction followed $\forall$-introduction), we have the assumption that

$$\Gamma, [\![u,\ v:i]\!] \vdash [\![Mu,\ Nv:j]\!],$$

for eigenvariables u and v, is provable at a lower height. By the Same-Variable lemma (preceded by a $\Rightarrow R$), we also have:

$$\Gamma, [\![u,\ u:i]\!] \vdash [\![Mu,\ Nu:j]\!],$$

Thus by inductive hypotheses (which range over extensions of Γ),

$$\Gamma^-, [\![u,\ u : i]\!]^- \vdash [\![Mu,\ Nu : j]\!]^+.$$

$\forall R$ now applies to yield the desired conclusion $\Gamma^- \vdash \forall u([\![u,\ u : i]\!]^- \Rightarrow [\![Mu,\ Nu : j]\!]^+)$.

In the Backchain case, the atomic copy clause $[\![M,\ N : j]\!]$ must be of the form

$$copy\ (a\ f_1, \ldots, f_m)^{\beta\eta}\ (b\ g_1, \ldots, g_m)^{\beta\eta}$$

for some $[\![a,\ b : \sigma_1 \to \ldots \to \sigma_m \to j]\!] \in \Gamma$ where j is primitive. But this means that each $[\![f_i,\ g_i : \sigma_i]\!]$ for $i \in 1 \ldots m$ is provable from Γ at a lower height. But then by the inductive hypotheses,

$$\Gamma^- \vdash [\![f_i,\ g_i : \sigma_i]\!]^+$$

for each $i \in 1 \ldots m$, and thus

$$\Gamma^- \vdash copy\ (a\ f_1, \ldots, f_m)^{\beta\eta}\ (b\ g_1, \ldots, g_m)^{\beta\eta}.$$

The other direction follows by the same arguments.

$\square$

The "soundness" result is so named for a formula should always follow from its assumption. Polarization must respect this basic principle. "Completeness" is so named because it allows the recovery of the core results of the original copy clauses, namely the Substitution Encoding theorem and its corollaries. The *polarized versions* of those results are obtained in the obvious way by replacing $A \vdash B$ with $A^- \vdash B^+$, and $[\![M,\ N : \sigma]\!] \Rightarrow [\![P,\ Q : \tau]\!]$ with $[\![M,\ N : \sigma]\!]^- \Rightarrow [\![P,\ Q : \tau]\!]^+$. We thus have:

Corollary 4.2.1 *(Substitution Encoding, Polarized)*
Let signature $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m, c_1 : a_1, \ldots, c_n : a_n\}$. For Σ-terms $t_1 : a_1, \ldots, t_n : a_n$, let $C^- = \{[\![c_i,\ t_i : a_i^-]\!] \mid 0 \le i \le n\}$ and let $K^- = \{[\![k_i,\ k_i : s_i^-]\!] \mid 0 \le i \le m\}$. For Σ-terms M and N of some type τ,
$\Sigma; K^-, C^- \vdash\ [\![M,\ N : \tau]\!]^+$ if and only if $(\lambda c_1 \ldots \lambda c_n.M)t_1, \ldots, t_n$ is $\beta\eta$-convertible to N.

Corollary 4.2.2 *(Equality, Polarized)*
Let signature $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m\}$ and let $K^- = \{[\![k_i,\ k_i : s_i]\!]^- \mid 0 \le i \le m\}$. For Σ-terms M and N of type τ,
$\Sigma; K^- \vdash\ [\![M,\ N : \tau]\!]^+$ if and only if M is $\beta\eta$-convertible ($=_{\beta\eta}$) to N.

Corollary 4.2.3 *(Substitution, Polarized)*

Let signature $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m\}$ *and let* $K^- = \{[\![k_i,\ k_i : s_i^-]\!] \mid 0 \leq i \leq m\}$. *For* Σ*-terms* M *of type* $\sigma \rightarrow \tau$, t *of type* σ,*and* N *of type* τ,

$\Sigma; K^- \vdash\ \forall x([\![x,\ t : \sigma]\!]^- \Rightarrow [\![M\ x,\ N : \tau]\!]^+)$ *if and only if* $(M\ t)$ *is* $\beta\eta$*-convertible to* N.

Corollary 4.2.4 *(Unification, Polarized)*

Let signature $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m, c_1 : a_1, \ldots, c_n : a_n\}$, *and let* $K^- = \{[\![k_i,\ k_i : s_i]\!]^- \mid 0 \leq i \leq m\}$. *For* Σ*-terms* M *and* N *of some type* τ,

$\Sigma; K^- \vdash\ \exists T_1 : a_1, \ldots, T_n : a_n \exists R : \tau([\![c_1,\ T_1 : a_1]\!]^-, \ldots, [\![c_n,\ T_n : a_n]\!]^-) \Rightarrow ([\![M,\ R : \tau]\!]^+ \wedge [\![N,\ R : \tau]\!]^+)$ *if and only if for some substitution* θ, $M\theta = N\theta$.

Once the polarized version of the Substitution corollary is established, we can examine the effect of applying its implied translation recursively on possible non-β_0 redices. This recursive translation can be axiomatized by a collection of $subst_{\sigma\rightarrow\tau}$ clauses, specifying the substitution (as β-reduction) of a σ-typed expression into a τ-type one. Each $subst_{\sigma\rightarrow\tau}$ has (meta-level) type $(\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau$. The definition is inductive on σ. Assume σ has the form $i_1 \rightarrow \ldots \rightarrow i_l \rightarrow j$ where j is primitive. This allows us to perform a series of substitutions at once. Assume also that `M` and `N` are $\beta\eta$-normalized.

Definition 4.3 *(Substitution Clauses)*

$subst_{\sigma\rightarrow\tau}\ M\ R\ N \Leftarrow \forall x((copy_\sigma\ x\ R) \Rightarrow [\![(M\ x),\ N : \tau]\!]^+)$ *for primitive* σ.

$$
\begin{aligned}
& subst_{(i_1\rightarrow\ldots\rightarrow i_l\rightarrow j)\rightarrow\tau}\ M\ R\ N \Leftarrow \\
& \quad \forall x[\ (\forall H \forall u_1, \ldots, u_l \forall v_1, \ldots, v_l (copy_j\ (x\ u_1, \ldots, u_l)\ H \Leftarrow \\
& \qquad\qquad subst^l_{i_1\rightarrow\ldots\rightarrow i_l\rightarrow j}\ R\ v_1, \ldots, v_l\ H \wedge [\![u_1,\ v_1 : i_1]\!]^+ \wedge \ldots \wedge [\![u_l,\ v_l : i_l]\!]^+)) \\
& \qquad\qquad\qquad\qquad \Rightarrow [\![(M\ x),\ N : \tau]\!]^+]
\end{aligned}
$$

$$
\begin{aligned}
& \quad \textit{where } subst^l_{i_1\rightarrow\ldots\rightarrow i_l\rightarrow j}\ R\ v_1, \ldots, v_l\ H \quad = \\
& \exists H_1 \ldots \exists H_{l-1}\ (subst_{i_1\rightarrow\ldots\rightarrow i_l\rightarrow j}\ R\ v_1\ H_1 \wedge subst_{i_2\rightarrow\ldots\rightarrow i_l\rightarrow j}\ H_1\ v_2\ H_2 \wedge \\
& \qquad \ldots \wedge subst_{i_l\rightarrow j}\ H_{l-1}\ v_l\ H).
\end{aligned}
$$

For example, $subst_{(i\rightarrow j)\rightarrow t}$ where i, j and t are primitive is equivalent to:

```
subst_(i-j)-t M R N :- pi x\ ( pi H\ pi \u\ pi v\ (
    copyj (x u) H :- subst_i-j R v H, copyi u v) => copyt (M x) N).
```

The intended purpose of the *subst* clauses is that, in $[\![x,\ R : i \rightarrow j]\!]^-$ which equals

```
pi U\ pi V\ (copy (x U) (R V) :- copy U V).
```

(if `i` and `j` are primitive), we can replace `(R V)`, which is non-L_λ, with a new variable `H` and the condition `subst R V H`.

The first thing we need to show is that using these *subst* clauses do not introduce new non-L_λ elements. The following lemma identifies certain types of clauses that are properly in L_λ. First note that all closed terms are trivially L_λ terms.

Lemma 4.3 *(L_λ-stable clauses)*

Under any signature, the following holds:

1. *For all L_λ terms M and N of primitive type, $[\![M,\ N : \tau]\!]^-$ is a L_λ program clause.*
2. *For all L_λ terms M and N, $[\![M,\ N : \tau]\!]^+$ is a L_λ goal clause.*
3. *For all L_λ terms M and N of $\beta\eta$-normal form $(f\ g_1, \ldots, g_n)$ where f is not a free variable, $[\![M,\ N : \tau]\!]^-$ is a L_λ program clause.*
4. *All $subst_{\sigma\rightarrow\tau}$ clauses are L_λ program clauses.*

Proof: 1 holds since primitive-type copy clauses are atomic formulas. 4 follows from 2. 2 and 3 are proved by simultaneous induction on type τ:

The base cases follow from definition.

If $\tau = i \rightarrow j$, then we have

$$[\![M,\ N : \tau]\!]^+ = \forall z([\![z,\ z : i]\!]^- \Rightarrow [\![Mz,\ Nz : j]\!]^+)$$

and

$$[\![M,\ N : \tau]\!]^- = \forall u, v([\![u,\ v : i]\!]^+ \Rightarrow [\![Mu,\ Nv : j]\!]^-).$$

In the first case, since z is a λ-bound variable, by inductive hypotheses on the shorter type, $[\![z,\ z : i]\!]^-$ is a L_λ program clause. Furthermore, since M and N are assumed to be L_λ terms, Mz and Nz are also L_λ terms (this is straightforward to verify). So again by inductive hypothesis, $[\![Mz,\ Nz : j]\!]^+$ is a L_λ goal clause.

In the second case, since u and v are variables, they are trivially L_λ terms. So by inductive hypothesis $[\![z,\ z : i]\!]^+$ is a L_λ goal clause. Furthermore since M and N are assumed to be L_λ terms of normal form $(f\ t_1, \ldots, t_n)$, Mu and Nv are also L_λ terms of this form. So again by inductive hypotheses, $[\![Mu,\ Nv : j]\!]^-$ is a L_λ program clause.

□

The condition on M and N in part 3 can not be relaxed. In particular, the result does *not* hold for arbitrary closed terms. For example,

$$[\![(\lambda u \lambda v.f(uv)),\ g : i \to j \to k]\!]^-$$

is not a valid L_λ program clause (for any term g and constant f) since it equals

$$\forall U_1, U_2 \forall V_1, V_2(copy_i\ U_1\ U_2 \Rightarrow copy_j\ V_1\ V_2 \Rightarrow copy_k\ f(U_1\ V_1)\ (g\ U_2\ V_2)^{\beta\eta})$$

which contains the non-L_λ component $(U_1\ V_1)$. The result more than suffices, however, for showing the crucial property we need here, which is that clauses added to a λProlog program for the purpose of translating it to L_λ are already L_λ clauses. There is no need for recursive translation.

The following theorem establishes the correctness of the $subst_{\sigma\to\tau}$ clauses. The theorem and proof can easily be generalized for $subst^n_{\sigma_1\to\ldots\to\sigma_n\to\tau}$ clauses. Recall that these clauses can be considered as a sequence of single-argument $subst$ clauses. They can also be given a direct definition allowing simultaneous multiple substitutions. We assume the single-argument interpretation of $subst^n_{\sigma\to\tau}$ to achieve a simpler inductive measure in the proof below [3].

Theorem 4.4 *(Validity of subst)*
For $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m\}$, let $K = \{[\![k_i,\ k_i : s_i]\!]^- \mid 0 \leq i \leq m\}$. Let S contain the clauses for $subst_\gamma$ for all subtypes γ of a type $\sigma \to \tau$. For all Σ-terms $M : \sigma \to \tau$, $T : \sigma$ and $N : \tau$ (that do not contain the $subst_\rho$ constants),
$\Sigma; S, K\ \vdash\ subst_{\sigma\to\tau}\ M\ T\ N$ if and only if $(M\ T) =_{\beta\eta} N$.

Proof: by induction on the structure of type σ.

Base Case. For primitive σ, the definition for $subst_{\sigma\to\tau}$ is exactly $\forall x([\![x,\ T : \sigma]\!]^- \Rightarrow [\![Mx,\ N : \tau]\!]^+)$. The result follows by the polarized version of the Substitution Corollary.

Inductive Case. It is valid to assume that $\sigma = i_1 \to \ldots \to i_l \to j$, where j is primitive. $\Sigma; S, K\ \vdash\ subst_{(i_1\to\ldots\to i_l\to j)\to\tau}\ M\ T\ N$ holds if and only if

[3] The definition of $subst$ and $subst^n$ here are different from those given in [37].

$$\Sigma, x:\sigma; S, K,\ (\forall H \forall u_1,\ldots,u_l \forall v_1,\ldots,v_l (copy_j\ (x\ u_1,\ldots,u_l)\ H \Leftarrow$$
$$subst^l_{i_1\to\ldots\to i_l\to j}\ T\ v_1,\ldots,v_l\ H \wedge [\![u_1,\ v_1 : i_1]\!]^+ \wedge\ldots\wedge [\![u_l,\ v_l : i_l]\!]^+))$$
$$\vdash [\![(M\ x),\ N:\tau]\!]^+$$

holds for a new constant x. Now assume $\tau = \tau_1 \to \ldots \to \tau_m \to \gamma$ where γ is primitive. By the definition of polarized copy clauses, $[\![(M\ x),\ N:\tau]\!]^+$ if and only if

$$[\![z_1,\ z_1:\tau_1]\!]^- \Rightarrow \ldots \Rightarrow [\![z_m,\ z_m:\tau_m]\!]^- \Rightarrow\ copy_\gamma\ (M\ x,\ z_1,\ldots,z_m)^{\beta\eta}\ (N\ z_1,\ldots,z_m)^{\beta\eta}$$

holds for new constants $z_1,\ldots,z_m$. That is, we have that (if and only if)

$$\Sigma, x:\sigma, z_1:\tau_1,\ldots,z_m:\tau_m;\ S, (K \cup \{[\![z_1,\ z_1:\tau_1]\!]^-,\ldots,[\![z_m,\ z_m:\tau_m]\!]^-\}),$$
$$(\forall H \forall u_1,\ldots,u_l \forall v_1,\ldots,v_l (copy_j\ (x\ u_1,\ldots,u_l)\ H \Leftarrow$$
$$subst^l_{i_1\to\ldots\to i_l\to j}\ T\ v_1,\ldots,v_l\ H \wedge [\![u_1,\ v_1:i_1]\!]^+ \wedge\ldots\wedge [\![u_l,\ v_l:i_l]\!]^+))$$
$$\vdash\ copy_\gamma\ (M\ x,\ z_1,\ldots,z_m)^{\beta\eta}\ (N\ z_1,\ldots,z_m)^{\beta\eta}$$

holds. Atomic formulas can only be proved by Backchain (i.e, without changing the signature and program). Since x is a new constant, it can not appear free in $(N\ z_1,\ldots,z_m)^{\beta\eta}$, or in M and T. Thus if and when backchaining on the clause

$$(\forall H \forall u_1,\ldots,u_l \forall v_1,\ldots,v_l (copy_j\ (x\ u_1,\ldots,u_l)\ H \Leftarrow$$
$$subst^l_{i_1\to\ldots\to i_l\to j}\ T\ v_1,\ldots,v_l\ H \wedge [\![u_1,\ v_1:i_1]\!]^+ \wedge\ldots\wedge [\![u_l,\ v_l:i_l]\!]^+)),$$

x does not appear in any instantiation of H. Thus by (several applications of the) inductive hypotheses,

$$\Sigma, z_1:\tau_1,\ldots,z_m:\tau_m;\ S, K \cup \{[\![z_1,\ z_1:\tau_1]\!]^-,\ldots,[\![z_m,\ z_m:\tau_m]\!]^-\}$$
$$\vdash\ subst^l_{i_1\to\ldots\to i_l\to j}\ T\ v_1,\ldots,v_l\ H$$
$$\text{if and only if}$$
$$(T\ v_1,\ldots,v_l) =_{\beta\eta} H.$$

That is, we have shown that (if and only if) the following holds:

$$\Sigma, x:\sigma, z_1:\tau_1,\ldots,z_m:\tau_m;\ S, (K \cup \{[\![z_1,\ z_1:\tau_1]\!]^-,\ldots,[\![z_m,\ z_m:\tau_m]\!]^-\}),$$
$$(\forall u_1,\ldots,u_l \forall v_1,\ldots,v_l (copy_j\ (x\ u_1,\ldots,u_l)\ (T\ v_1,\ldots,v_l)) \Leftarrow$$
$$[\![u_1,\ v_1:i_1]\!]^+ \wedge\ldots\wedge [\![u_l,\ v_l:i_l]\!]^+)$$
$$\vdash\ copy_\gamma\ (M\ x,\ z_1,\ldots,z_m)^{\beta\eta}\ (N\ z_1,\ldots,z_m)^{\beta\eta}.$$

But this is equivalent, by discharging constants and clauses using $\Rightarrow$-Right and $\forall$-Right, to

$$\Sigma;\ S, K \vdash\ \forall x([\![x,\ T:\sigma]\!]^- \Rightarrow [\![(M\ x),\ N:\tau]\!]^+).$$

Again by the polarized version of the Substitution Corollary, this holds if and only if $(M\ T) =_{\beta\eta} N$.

$\square$

The crucial role of polarized copy clauses was played in the proof when K was augmented with $\{[\![z_1,\ z_1 : \tau_1]\!]^-, \ldots, [\![z_m,\ z_m : \tau_m]\!]^-\}$. With un-polarized copy clauses, K would have been instead augmented with $\{\ldots, [\![z_k,\ z'_k : \tau_k]\!], \ldots\}$ for eigenconstants z'_k distinct from the z_k's. We therefore can not apply the inductive hypotheses on *subst* at lower types.

The following corollary relates *subst* with copy clauses:

Corollary 4.4.1 *For $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m\}$, let $K = \{[\![k_i,\ k_i : s_i]\!]^- \mid 0 \leq i \leq m\}$. Let S contain the clauses for $subst_\gamma$ for all subtypes γ of a type $\sigma \to \tau$. For all Σ-terms $M : \sigma \to \tau$, $T : \sigma$ and $N : \tau$ (that do not contain the $subst_\rho$ constants),*

$$\Sigma; S, K \ \vdash\ subst_{\sigma\to\tau}\ M\ T\ N.$$

if and only if

$$\Sigma; K \ \vdash\ [\![(M\ T),\ N : \tau]\!]^+$$

if and only if

$$\Sigma; K \ \vdash\ \forall x([\![x,\ T : \sigma]\!]^- \Rightarrow [\![(M\ x),\ N : \tau]\!]^+$$

4.2.2 The Translation Algorithm

The validity of the *subst* clauses means that all λProlog expressions can be translated using *subst* in innermost-first fashion, i.e, starting with the innermost occurrences of non-L_λ terms, until all are eliminated. First of all, in order to ensure that the translation procedure does not interfere with the correctness of the program being translated, we will reserve a special copy predicate $copy_\lambda$, and used copy clauses of the form $[\![M,\ N : \sigma]\!]_\lambda^{\overset{+}{-}}$ to encode substitutions for the sole purpose of translation.

Given a λProlog program Γ and goal clause G, the translation procedure is:

1. For each constant $k : \sigma$ in the signature of the program Γ, add the clause $[\![k,\ k : \sigma]\!]_\lambda^-$ to the program.

2. Replace all positive instances of `pi` in Γ and G with `pi+`.

3. pick a non-L_λ subterm $(X\ T_1, \ldots, T_n)$ that occurs in an atomic formula A in G such that $T_1, \ldots, T_n$ are L_λ terms.

4. Let A' be A with (this occurrence of) $(X\ T_1, \ldots, T_n)$ replaced by $(H y_1, \ldots, y_m)$ where H is a new variable and $y_1, \ldots, y_m$ are all the essentially universal variables (eigenvariables) universally bound within the scope of $\exists X$, and may appear free in $(X\ T_1, \ldots, T_n)$.

5. Perform the following:

   - If A occurs positively in G, replace A in G with

   $$\exists H(\forall y_1 \ldots \forall y_m[[\![y_1,\ y_1 : \rho_1]\!]_\lambda{}^- \Rightarrow \ldots \Rightarrow [\![y_m,\ y_m : \rho_m]\!]_\lambda{}^-$$

   $$\Rightarrow subst^n_{\sigma_1 \to \ldots \to \sigma_n \to \tau}\ X\ T_1, \ldots, T_n\ (H\ y_1, \ldots, y_m)] \wedge A')$$

     where σ_i is the type of T_i, τ the type of $(X\ T_1, \ldots, T_n)$ and $\rho_1, \ldots, \rho_m$ the types of $y_1, \ldots, y_m$, respectively.

   - If A occurs negatively in G, replace A with

   $$\forall H(\forall y_1 \ldots \forall y_m[[\![y_1,\ y_1 : \rho_1]\!]_\lambda{}^- \Rightarrow \ldots \Rightarrow [\![y_m,\ y_m : \rho_m]\!]_\lambda{}^-$$

   $$\Rightarrow subst^n_{\sigma_1 \to \ldots \to \sigma_n \to \tau}\ X\ T_1, \ldots, T_n\ (H\ y_1, \ldots, y_m)] \Rightarrow A')$$

6. Repeat steps 3,4,5 until there are no more non-L_λ terms in the clause.

This procedure dualizes easily for definite clauses, since $\Gamma, D \vdash G$ if and only if $\Gamma \vdash D \Rightarrow G$. Note that the definition of the *subst* clauses themselves followed this translation, though its validity was independently established.

The following result should replace **Proposition 3** of [37]. Note that we must make sure that the constants used in defining the *subst* clauses, as well as the reserved predicate $copy_\lambda$, do not intersect with the signature of the program being translated.

Theorem 4.5 *(Completeness of L_λ)*
For any signature $\Sigma = \{k_1 : s_1, \ldots, k_m : s_m \mid k_i \neq subst_\tau$ or $copy_\lambda$ for any $i, \tau\}$, let

$K = \{[\![k_i,\ k_i : s_i]\!]_\lambda{}^- \mid 0 \leq i \leq m\}$. Let $S = \{subst_{\sigma\rightarrow\tau}$, for all types σ and $\tau\}$. Let P be the clause $\forall^+ G \Leftarrow \forall x([\![x,\ x : \sigma]\!]^- \Rightarrow (G\ x))$[4]. Let Γ be a λProlog program containing only Σ-terms. For Σ-goal clause G, let Γ'' and G'' be the result of applying the above procedure to Γ and G respectively . The following holds:

$$\Sigma; \Gamma \vdash G \text{ if and only if } \Sigma; \Gamma'', S, P, K \vdash G''.$$

Furthermore if G is of the form $\exists x G_0$ then $\Sigma; \Gamma \vdash G_0[t/x]$ if and only if $\Sigma; \Gamma, S, P, K \vdash G_0''[t/x]$ for any term t.

Proof: First, by the L_λ-Stable Clauses lemma, each clause $[\![k,\ k : \sigma]\!]_\lambda{}^-$ added to the program in steps 1 and 2 is a L_λ clause. Also, since we use reserved predicates in defining *subst* and $[\![k,\ k : \sigma]\!]_\lambda{}^-$, these additions do not change the meaning of the original program. The proof for the rest of the translation is by induction on the structure of G. The forward and reverse directions are analogous.

- For $G = A$ an atomic formula with a non-L_λ term $(X\ T)$ (or more generally $(X\ T_1, \ldots, T_n)$), by the algorithm $G'' =$

$$\exists H(\forall y_1 \ldots \forall y_m[[\![y_1,\ y_1 : \rho_1]\!]_\lambda{}^- \Rightarrow \ldots \Rightarrow [\![y_m,\ y_m : \rho_m]\!]_\lambda{}^-$$
$$\Rightarrow subst_{\sigma\rightarrow\tau}\ X\ T\ (H\ y_1, \ldots, y_m)] \wedge A').$$

  But by the Validity of *subst* theorem, $subst_{\sigma\rightarrow\tau}\ X\ T\ (H\ y_1, \ldots, y_m)$ holds if and only if $(X\ T) =_{\beta\eta} (H\ y_1, \ldots, y_m)$. Since uniform provability is certainly invariant under $\beta\eta$-conversion, A' must also hold.

- The cases for $G = G_1 \wedge G_2$ and $G = G_1 \vee G_2$ are trivial.
- For $G = \exists x G_0$, $\Sigma; \Gamma, S, P, K \vdash \exists x G_0$ if and only $\Sigma; \Gamma, S, P, K \vdash G_0[t/x]$ for some term t. The result follows by inductive hypotheses on the goal $G_0[t/x]$.
- For the case $G = D \Rightarrow G_0$, $\Sigma; \Gamma, \vdash G$ if and only if $\Sigma; \Gamma, D \vdash G_0$. By the inductive hypotheses, this implies $\Sigma; \Gamma'', D'' \vdash G_0''$, which means

$$\Sigma; \Gamma'' \vdash D'' \Rightarrow G_0''.$$

  This is the desired result; it is straightforward to verify that $G'' = D'' \Rightarrow G_0''$.

- For $G = \forall x G_0$, by the translation algorithm this will be replaced by $\forall^+ x G_0$. But

$$\Sigma; \Gamma, S, P, K \vdash \forall x([\![x,\ x : \sigma]\!]_\lambda{}^- \Rightarrow (G_0\ x))$$

[4]To be technically correct, there should be one such clause for each type σ.

means for some fresh constant v,

$$\Sigma; \Gamma, S, P, (K, [\![v,\ v : \sigma]\!]_\lambda^-) \vdash (G_0\ v)$$

and the result follows by inductive hypotheses on the goal $(G_0\ v)$, under the augmented K clauses.

□

This completes the translation of λProlog to L_λ, and the project proposed and begun by Miller in [37].

In practice, even before the advent of L_λ, it was found that most uses of higher-order unification in λProlog were in fact in the L_λ fragment. This is understandable since, given its excessively non-deterministic nature, Huet's algorithm is simply impractical as a general programming tool. There are a few occasions where non-L_λ programs are desirable, such as in

```
type map (i -> o) -> (list i) -> o.
map P nil.
map P (X::Xs) :- (P X), map P Xs.
```

which contains the non-L_λ goal `(P X)` in the second clause. Using the translation procedure, however, the second clause can be rewritten as:

```
map P (X::Xs) :- subst_i-o P X H, H, map P Xs.
```

which is equivalent to

```
map P (X::Xs) :- pi z\(copyi z X => copyo (P z) H), H, map P Xs.
```

(Provided the appropriate copy clauses $[\![k_i,\ k_i : \sigma_i]\!]_\lambda^-$ are also added to the program.) The process of substitution has therefore been moved from the underlining logic program interpreter to the logic program itself. The underlining interpreter need only be concerned with L_λ unification, which is little more than first-order unification respecting the scopes of bound variables.

As demonstrated in the previous chapters, the high-level control of substitution is essential for meta-programming. Showing that all of higher-order unification can be so transformed sets the theoretical groundwork for the rest of this chapter. Having demonstrated that L_λ suffices for general higher-order logic programming, we will allow ourselves the luxury of a few non-L_λ (but λProlog) clauses in the remainder of this thesis. We have non-L_λ clauses, even without the *subst* translation, in all real code presented except in the `quicktype` program of section 3.5.3.

4.3 Unification Under a Mixed Prefix

In [39], Miller re-examined the problem of unification as the problem of solving a system of equations under a *quantifier prefix*:

$$\mathcal{Q}(S_1 = T_1 \wedge \ldots, \wedge S_k = T_k),$$

where $\mathcal{Q}$ a list of $\forall$ and $\exists$ quantifiers over variables in the equations (or difference pairs). First-order unification in the context of first-order Horn clause prolog can ignore the quantifier prefix, since it is always a list of existential quantifiers over the free variables. For HoHH formulas, however, this prefix is of greater significance. Because universal quantification can appear in a goal clause, solving λProlog (L_λ) goals will often produce unification problems under some mixed prefix of existential and universal quantifiers, and the unification algorithm must respect these scoping rules when assigning substitutions. Miller showed that every such unification problem can be transformed into an equivalent one with a quantifier prefix of the form

$$\forall x_1 \ldots \forall x_l \exists y_1 \ldots \exists y_m \forall z_1 \ldots \forall z_n \ldots$$

which he calls $\forall\exists\forall$-unification. The initial $\forall$ quantifiers effectively defines the "signature" or global constants which can appear in the solutions to the $\exists$-quantified variables. The final (right-most) sequence of $\forall$-quantified variables can not appear free in any solution for the $\exists$-quantified variables.

4.3.1 Object-Level Higher-Order Unifiability

The copy-clause formulation of object-level unifiability of Chapter 1 is valid for first and higher order terms. Meta-programming unifiability under a mixed prefix is not difficult, since we can use the scoping rules of the higher-order meta-language. That is, we can use meta-level mixed prefixes to simulate object-level mixed prefixes. Assume that the object-level quantifiers are represented by $\forall_o$ and $\exists_o$, both of type $(\sigma \rightarrow bool) \rightarrow bool$ where σ ranges over the types of terms and *bool* is the type of (object-level) unification problems. Let $eq : \sigma \rightarrow \sigma \rightarrow bool$ be the constructor of unification equations. Let τ and τ' range over the types of object-level variables quantified by $\forall_o$ and $\exists_o$. Assuming the visibility of clauses $[\![k,\ k : \sigma]\!]$ for each constant k in the current signature, unifiablity under a mixed prefix can be formulated by:

$$
\begin{array}{l}
unifiable\ (\forall_o\ E) \Leftarrow \\
\quad \forall v([\![v,\ v : \tau]\!] \Rightarrow unifiable\ (E\ v)) \\
unifiable\ (\exists_o\ E) \Leftarrow \\
\quad \forall v(\exists X_v([\![v,\ X_v : \tau']\!] \Rightarrow unfiable\ (E\ v))) \\
unifiable\ (eq\ A\ B) \Leftarrow \exists N([\![A,\ N : \sigma]\!] \wedge [\![B,\ N : \sigma]\!])
\end{array}
$$

The object-level quantifiers $\forall_o$ and $\exists_o$ are simulated by the meta-level quantifiers $\forall$ and $\exists$ since X_v can not be bound to terms with eigenvariables introduced within its scope. This program is valid for all higher-order terms, and is not restricted to the L_λ fragment. Its correctness is by the Unification corollary (and not by the correctness of an algorithm), and it is purely declarative.

Unfortunately, inserting a `setsub Xv v` after the second clause does not lead to an implementation of object-level higher-order unification. This is because arbitrary new variables can be introduced by higher-order unification. These variables are not just renamings of variables in the original domain of unification; in fact, they can be of arbitrary type. This holds even in the L_λ fragment. Inserting `setsub Xv v` in the second clause will in general still leave free variables in the meta-level unifier generated.

4.4 Object-level L_λ Unification: The Algorithmic Approach

We will now give a purely declarative formulation of the L_λ unification algorithm that is similar in spirit to the first-order unification program of section 3.2.1. In this program, the techniques of copy clauses and λ-representation of new variables developed in previous chapters will be employed. This formulation will thus demonstrate two points. It provides an important meta-programming technique allowing object-level unification, which will aide us in formulating other tasks such as Knuth-Bendix style completion. But it is also a demonstration of the techniques already given up to this point, in the same way that type inferencing was such a demonstration.

To facilitate the formulation we first modify slightly the manner in which object-level terms are represented. The L_λ unification algorithm requires that terms be treated at a very explicit level. For example, we need to be able to recognize when the head of an application is a $\forall$-variable or a $\exists$-variable. It is thus easier to adopt a more indirect representation of application at the term level. That is, instead of object-level terms of the form $(f\ x_1, \ldots, x_n)$ for a constant symbol f, we will have terms of the form $(appl\ f\ [x_1, \ldots, x_n])$ using an object level constructor *appl* to replace the meta-level constructor, and a list of arguments. This means that all λ-free structures are effectively first-order structures at the meta-level, though they represent terms of arbitrary order at the object level. Object-level λ-abstraction can be represented in the style used before (but denoted by `lam x\...` instead of `abs x\...`), so the most essential aspect of higher-order abstract syntax is retained. Some important meta-level signature declarations used in this chapter are given below. We use the meta-level type `i` to categorize arbitrary-order individuals; this type generalizes over the type `term` used for first-order terms in the previous chapter.

```
kind i type.
kind up type.
type eq i -> i -> up.
type appl i -> (list i) -> i.
type lam (i -> i) -> i.
```

Using the `appl` and `lam` representation, meta-level types also can no longer be used to represent object-level types. Object-level types and type checking/inferencing must also be re-implemented (in the manner of MLtyper, of course). We will not address object-level types in this section, since that has been done in the foregoing.

Adopting the longer representation means more cumbersome meta-programming and encoding. This problem can be alleviated in various ways, such as with user-definable editor or parsing macros. (These being non-existent however, our actual code may seem somewhat cluttered.)

The formulation is given for terms already in β-normal form, though technically it does not have to be in β_0-normal form.

4.4.1 Raising

The version of L_λ unification we present here will not represent signature constants in the initial $\forall$-prefix. Instead of $\forall\exists\forall$-unification, we will simply have $\exists\forall$-unification. That is, we will attempt to normalize the quantifier prefix of every unification problem into the form

$$\exists y_1 \ldots \exists y_m \forall z_1 \ldots \forall z_n \ldots.$$

This is achieved by a process called *raising:* the unification problem

$$\mathcal{Q}_1 \forall v \exists u \mathcal{Q}_2 E,$$

is transformed to

$$\mathcal{Q}_1 \exists u' \forall v \mathcal{Q}_2 E[(u' \ v)/u].$$

for some new variable u'. If θ is a solution for this new unification problem, then $[(u' \ v)/u] \circ \theta$ is a solution for the original problem. We can also apply multiple steps of this translation at once, moving all $\exists$-quantifiers to the left of all $\forall$-quantifiers while replacing each $\exists$-variable u with $(u' \ v_1, \ldots, v_n)$ where $v_1, \ldots, v_n$ are all the $\forall$-variables originally quantified to the left of where u was quantified. Adopting $\exists\forall$-prefixes simplifies the unification algorithm with an easy constraint: *no $\forall$-variable may appear free in the binding for any $\exists$-variable.*

The program of Figure 4.1 performs raisings to obtain a $\exists\forall$-unification problem. It is written in the style of the unifiability program of the above section. Let `all` and `some` represent object-level $\forall$ and $\exists$-quantifiers (i.e, $\forall_o$ and $\exists_o$). An object-level mixed-prefix unification problem will have the form `(all x\some y\all z\...(eq A B))`.

In the program, the lists `El` and `Ul` are lists of constant symbols representing the $\exists$- and $\forall$-variables respectively. The first two arguments of `raise-aux` accumulates the $\exists$- and $\forall$-variable lists. $\forall$-variables and $\exists$-variable not under the scope of any $\forall$-variables are treated the same way (by the first two clauses of `raise-aux`). The key is the third clause of `raise-aux`, for the case when a $\exists$-variable `x` is quantified under the scope of a positive number of `all`-quantifiers. The fact that `x` is to be raised by is declared by

```
raise S T :- raise-aux nil nil S T.
raise-aux El Ul (all S) Z :-
  pi v\ (copy v v => raise-aux El (v::Ul) (S v) Z).
raise-aux El nil (some S) Z :-
  pi x\ (copy x x => raise-aux (x::El) nil (S x) Z).
raise-aux El (U::Us) (some S) Z :-
  pi x\ (pi r\ (sigma Er\ (copy x (appl r (U::Us)) =>
     (append El (x::r::nil) Er, raise-aux Er (U::Us) (S x) Z)))).
raise-aux El Ul (eq A B) C :- copy (eq A B) C.

abstract-over (E::EQ) UQ S (some N) :-
  pi x\ (copy x E => abstract-over EQ UQ S (N x)).
abstract-over nil (U::UQ) S (all N) :-
  pi v\ (copy v U => abstract-over nil UQ S (N v)).
abstract-over nil nil S N :- copy N S.
```

Figure 4.1: Mixed-Prefix to ∃∀-Unification Transformation

`(copy x (appl r (U::Us))` where `(U::Us` are the ∀-variables and `r` is the new, "raised" ∃-variable. When raising is complete, the `El` list followed by the `Ul` list represents the normalized ∃∀-prefix. If we wish to transform the new unification problem into its original form, quantified by `some` and `all`, we can use the `abstract-over` predicate, which uses the standard technique of quantifying over constant symbols.

4.4.2 The L_λ ∃∀-Unification Program

To represent an unification problem, we use the following meta-level declarations. The type `subst` categorizes object-level higher-order substitutions in the same style as the previous chapters.

```
kind ustate type.
type newuvar (i -> ustate) -> ustate.
type newevar (i -> ustate) -> ustate.
type ss (list i) -> (list i) -> (list up) -> subst -> ustate.
```

An unification problem (or *state*) is represented by an expression (of meta-level type `ustate`) of the following form:

```
newevar x\ newuvar y\ ...  (ss El Ul P R)
```

The constructor `ss` forms a quadruple where `El` and `Ul` are lists of constant symbols representing the ∃- and ∀-variables respectively, `P` is a list of equations to be solved, and

R is a substitution for the ∃-variables. New variables may be introduced by the algorithm in the manner of the raising procedure. Furthermore, these new variables may appear in any component of the ss-quadruple. We thus need newevar and newuvar as abstractions over the entire quadruple, to declare new ∃- and ∀-variables respectively.

The core code for the unification program is given in Figures 4.2, 4.3, and 4.4. This program shares a fundamental feature with the first-order unification program in that distinct constant-symbols that represent object-level variables can be distinguished by an ordering relation vrd. Also like in first-order unification, the order in which the equations or difference pairs are solved does not matter, so we can always try to solve the first one in the list, until we reach an empty list. The top-level L_λ unification predicate of the program is lunify. We start the unification procedure with an initial trivial substitution for every ∃-variable, i.e, with an initial ss-quadruple of the form

```
( ss El Ul P (sub a a (sub b b (sub c c .... ) )
```

where a,b,c... are the ∃-variable in El. Unification succeeds when we achieve a ss-quadruple with a nil difference-pair list.

All terms in the list of equations or difference-pairs are always in one of the five following forms:

1. An application of a term to a list of arguments, of the form (appl A L)
2. A λ-abstraction over a term, of the form (lam A).
3. A ∀-variable, of the general form V such that (uvar V) holds.
4. A ∃-variable, of the general form V such that (evar V) holds.
5. A signature constant (such as zero), of the general form V such that (rigid V) holds.

The predicates non-lam, non-flex, distinct-uvar, and distinct-evar all have the obvious meanings. vrd is used to order variables, with ∃- and ∀-variables ordered in seperate chains (this choice was arbitrary). (copyi A B) is the copy clause on A and B of primitive meta-type i, representing object-level terms of arbitrary type. The following auxillary predicates are also needed for the program (the full code is in the appendix).

```
type r-r, xi, prune    ustate -> ustate -> o.

r-r (ss E U ((eq (appl A nil) (appl A nil))::O) Rho) (ss E U O Rho).
r-r (ss E U ((eq (appl A Ar) (appl A Br))::O) Rho) (ss E U O2 Rho) :-
  pair-up Ar Br PN, append PN O O2.

xi (ss E U ((eq (lam A) (lam B))::O) Rho) (newuvar Z) :-
  pi v\ (uvar v => xi (ss E (v::U) ((eq (A v) (B v))::O) Rho) (Z v)).
xi (ss E U ((eq (lam A) B)::O) Rho) (newuvar Z) :- non-lam B,
  pi v\ (sigma Bv\ (uvar v => (append-app B v Bv,
    xi (ss E (v::U) ((eq (A v) Bv)::O) Rho) (Z v)))).
xi (ss E U ((eq B (lam A))::O) Rho) (newuvar Z) :- non-lam B,
  pi v\ (sigma Bv\ (uvar v => (append-app B v Bv,
    xi (ss E (v::U) ((eq Bv (A v))::O) Rho) (Z v)))).
xi (ss E U ((eq A B)::O) Rho) (ss E U ((eq A B)::O) Rho) :-
  non-lam A, non-lam B.

prune (ss El Ul P S) (newevar T)  :-
  pi h\ (evar h => ((pi Z\ (vrd h Z) :- member Z El) =>
        prune-aux (ss (h::El) Ul P S) (T h))).
prune-aux (ss El Ul ((eq R (appl V Y))::O) Rho) S2 :- evar V, non-flex R,
  prune-aux (ss El Ul ((eq (appl V Y) R)::O) Rho) S2.
prune-aux (ss El Ul ((eq (appl V Y) R)::O) Rho) SS2 :- evar V,
  find-flexhead Y R F,
  prune-aux2 F (ss El Ul ((eq (appl V Y) R)::O) Rho) SS2.
prune-aux2 (lam U) S T :-
  pi l\ ((copyi l l, (pi Z\ (vrd l Z :- uvar Z))) =>
      prune-aux2 (U l) S T).
prune-aux2 (appl U W) (ss (H::El) Ul ((eq (appl V Y) R)::O) Rho) SS2 :-
  list-same W Y Z,
  (copyi H H => form-lam W (appl H Z) NU),
  (copyi U NU => standard-except U
(update-ss (ss (H::El) Ul ((eq (appl V Y) R)::O) Rho) SS1)),
  ssbeta_0 SS1 SS2.
```

Figure 4.2: L_λ Main Transformations, Part 1

```
type fr-ff1, ff2  ustate -> ustate -> o.
type step1        (list i) -> (list i) -> ustate -> ustate -> o.
type lunify, step2, step3   ustate -> ustate -> o.

fr-ff1 (ss El Ul ((eq (appl V L) (appl R M))::O) Rho) SS2 :-
  ((evar V, (rigid R; uvar R));
   (evar R, evar V, distinct-evar R V)),
  ((pi Z\ (copyi Z Z :- evar Z, distinct-evar Z V))
       => form-lam L (appl R M) Sub),
  (copyi V Sub => (standard-except V (update-ss (ss El Ul O Rho) SS1))),
  ssbeta_0 SS1 SS2.
fr-ff1 (ss El Ul ((eq (appl V Y) (appl U W))::O) Rho) SS2 :-
  (evar U, (rigid V; uvar V)),
  fr-ff1 (ss El Ul ((eq (appl U W) (appl V Y))::O) Rho) SS2.

ff2 (ss El Ul E R) (newevar Sn) :-
  pi h\ (evar h => ((pi Z\ (vrd h Z :- member Z El)) =>
ff2-aux (ss (h::El) Ul E R) (Sn h))).
ff2-aux (ss (H::El) Ul ((eq (appl V Y) (appl V W))::O) Rho) SS2 :-
  evar V, same-index Y W Z,
  (copyi H H => form-lam Y (appl H Z) Sub),
  (copyi V Sub => (standard-except V (update-ss (ss (H::El) Ul O Rho) SS1))),
  ssbeta_0 SS1 SS2.
```

Figure 4.3: L_λ Main Transformations, Part 2

```
step1 (E::Es) U S T:-
 ((pi Z\ (vrd E Z :- member Z Es)) => (evar E => step1 Es U S T)).
step1 nil (U::Us) S T :-
 ((pi Z\ (vrd U Z :- member Z Us)) => (uvar U => step1 nil Us S T)).
step1 nil nil (ss E U ((eq A B)::O) R) T :-
  app-norm A C, app-norm B D, step2 (ss E U ((eq C D)::O) R) T.

step2 (ss E U ((eq A B)::O) R) T :-
  (A = (lam X); B = (lam Y)),
  xi (ss E U ((eq A B)::O) R) S2, lunify S2 T.
step2 (ss E U ((eq (appl A Ar) (appl A Br))::O) R) T :-
  (rigid A; uvar A; (evar A, Ar = nil, Br = nil)),
  r-r (ss E U ((eq (appl A Ar) (appl A Br))::O) R) S2, lunify S2 T.
step2 (ss E U ((eq (appl A Ar) (appl B Br))::O) R) T :-
  (evar A; evar B), non-lam A, non-lam B,
  step3 (ss E U ((eq (appl A Ar) (appl B Br))::O) R) T.
step3 (ss E U ((eq (appl A Ar) (appl B Br))::O) R) T :-
  ((evar A, find-flexhead Ar (appl B Br) F);
   (evar B, find-flexhead Br (appl A Ar) G)),
  prune (ss E U ((eq (appl A Ar) (appl B Br))::O) R) S2, lunify S2 T.
step3 (ss E U P R) T :-
  (fr-ff1 (ss E U P R) S3; ff2 (ss E U P R) S3), lunify S3 T.

lunify (newevar S) (newevar T) :- pi e\ (lunify (S e) (T e)).
lunify (newuvar S) (newuvar T) :- pi u\ (lunify (S u) (T u)).
lunify (ss E U nil R) (ss E U nil R).
lunify (ss E U (P::Ps) R) T :- step1 E U (ss E U (P::Ps) R) T.
```

Figure 4.4: L_λ Top-Level Control

- `(list-same A B C)` iff `A` and `B` are lists of distinct ∀-variables and `C` contains the variables that are in both lists (in a particular order). This predicate is definable since ∀-variables can be distinguished by an ordering relation.

- `(list-dif A B C)`: like `list-same`, with `C` containing variables in `B` but not in `A`.

- `(same-index A B C)`: like `list-same`, with `C` containing common elements of `A` and `B` that also appear in the same position in their respective lists. Presumably `A` and `B` are permutations of eachother, but this need not be tested explicitly.

- `(pair-up A B C)`: pairs up corresponding terms in `A` and `B`, forming list of difference pairs (of the form `(eq S T)`).

- `app-norm A (appl A nil)`: if `A` is an individual constant (be it a variable or signature constant). This simply allows more uniform definitions of the transformations.

- `(find-flexhead Vs A F)`: given a list of ∀-variables `Vs` and a term `A`, `F` is a subterm of `A` of the form `(appl G L)` where `G` is a ∃-variable, and `L` contains ∀-variables not in `Vs`. If such a subterm is inside a `lam`-abstraction in `A`, then `lam`-bound variables will be considered as ∀-variables, and `F` would also be a `lam`-abstraction (over the original `lam`-bound variables, of course). For example, consider the case $Vs = \{v_1, v_2\}$, $A = (h\ zero\ (\lambda x.(G\ x\ v_1, \ldots, v_3)))$ where h is an object-level binary function symbol and G is a ∃-variable. `find-flexhead` $Vs\ A\ F$ will imply $F = \lambda x.(G\ x\ v_1, \ldots, v_3)$ (except written in the longer notation of `appl` and `lam`). This rule is used in the `prune` operation.

- `(form-lam Us T LT)` takes a term `T` and `lam`-abstracts over all ∀-variables in list `Us`, forming `LT`, i.e., $LT = (\lambda u_1 \ldots \lambda u_n.T), u_i \in Us$. This predicate is defined using the standard method of copying constants to be abstracted over to eigenvariables.

- `beta_0 A B, subbeta_0 M N, ssbeta_0 S T` defines β_0-normalization on terms, substitutions, and `ss`-quadruples. Previous work ([12, 46]) have shown how full β-reduction can be defined, so little contribution can be made in discussing the details of how this much simpler predicate is specified.

- `(update-sub M N)`: applies an internal substitution (encoded by copy clauses) to the range of the substitution `M`, forming `N`. That is, for each element `(sub X S)` in `M`, solve `(copyi S T)` and put `(sub X T)` in `N`.

- `update-ss (ss El Ul P R) (ss El Ul Q S)`: uses `copyi` to copy each term in the list of difference pairs `P` to the list of difference pairs `Q`. Uses `update-sub` to update substitution `R` to `S`.

In addition, the predicate `standard-except` has the following definition:

```
standard-except V G :-
  (pi X\ (copyi X X :- uvar X; vrd X V; vrd V X)) => G.
```

That is, it solves the arbitrary goal `G` under the assumption `copyi Z Z` for `Z` that are either ∀-variables or ∃-variables distinct form ∃-variable `V`. Presumably, `V` will be copied to something else, something "non-standard".

The L_λ-unification algorithm we implement here is adopted from the version presented in [36]. One notably different algorithm was given by Nipkow [44]. However, Nipkow presented L_λ- unification as a special case of the higher-order unification algorithm. Miller's original algorithm, especially when restricted to ∃∀-prefixes, is more similar to the standard first-order unification algorithm. Other than prior raising, the only significant differences between Miller's algorithm and the version here are that we also perform several "pruning" operations at once, and combine the "rigid-rigid" case and the "flexible-flexible" case where the ∃-variables are distinct.

The algorithm attempts to solve each difference pair (starting with the first one in the list) with the following transformations, which are not unlike the transformations for first-order unification. Each transformation is defined for an entire `ss`-quadruple. First the transformations `r-r` and `xi` are applied repeatedly until they are no longer applicable, or if failure results. This pair of rules corresponds to *term decomposition* in first-order unification, upgraded to deal with λ-headed terms. Then `prune` is applied as many times as possible, followed by either `fr-ff1` or `ff2` which eliminates the difference pair. These three rules correspond to *variable elimination* in first-order unification, upgraded to deal with scoping constraints. We will explain the meaning of each transformation below, along with the code. For a list of (∀) variables $L = v_1, \ldots, v_n$, we write $\lambda L.T$ for $\lambda v_1 \ldots \lambda v_n.T$.

r-r: rigid-rigid This transformation is identical to one found in the first-order unification program. If the difference-pairs list is of the form

```
(eq (appl A L) (appl A M)) :: Ds
```

where `A` is either a signature constant or a $\forall$-variable, then form a new list of difference-pairs from corresponding terms in lists `L` and `M`, and append this list to `Ds`.

xi If either term in the difference-pair is a λ-abstraction, apply η-expansion so that both terms have a λ-prefix of the same length. Now, $\lambda x.A =_{\beta\eta} \lambda x.B$ if and only if **for all** x, $A =_{\beta\eta} B$. Therefore, for each λ-heading in both terms we introduce a new $\forall$-variable into the $\exists\forall$-prefix, and eliminate that λ-heading from the terms. In the code, we perform the η-expansion and the λ-elimination simultaneously. The resulting `ss`-quadruple will be headed by `newuvar` abstractions, and neither term in the first difference pair will begin with a λ. Note that the $\exists\forall$-form of the quantifier prefix is preserved[5]. In the case of typed terms, we would need to require that η-expansions are not applied to terms of primitive type; failure will result if this is needed.

prune After as many `r-r` and `xi` transformations as possible have been performed (without failure), we are ready to eliminate the first difference pair. This pair is of the form `(eq (appl V L) T)` where `V` is an $\exists$-variable (switching the order of the terms if this is not so; this is the function of the fourth clause of `prune-aux`). What we would "like" to do at this point is to form the solution $[(\lambda L.T)/V]$. But before we can apply this variable-elimination, it is necessary to disallow invalid solutions to V by adding constraints to the unification problem. By the $\exists\forall$-form of the prefix, no $\forall$-variable can appear free in the solution for V, and so no $\forall$-variable not in the list L can appear free in T. However, not all such renegade variables in T leads to failure of unification. If a subterm of T of the form $(W\ m_1, \ldots, m_n)$ (`(appl W M)`), where W is a $\exists$-variable, contains such a $\forall$-variable u in the list $m_1, \ldots, m_n$, then unification is still possible by imposing the constraint that W is vacuous for u. This is done (using `update-ss`) by replacing W with a new $\exists$-variable W' and applying

[5]It is important to emphasize that the `newuvar` and `newevar` abstractions do *not* represent the quantifier prefix. They only mark new variables that were created, and can come in any order. The $\exists\forall$-prefix is encoded by the lists `El` and `Ul` in the `ss`-quadruple.

the substitution $[(W' m_1', \ldots, m_{n-1}')/W]$, where $m_1', \ldots, m_{n-1}'$ is $m_1, \ldots, m_n$ with u removed, to all terms of the `ss`-quadruple. In the code, we prune all such u from the subterm $(W \ m_1, \ldots, m_n)$ at once, using `find-flexhead` to find such subterms and `listsame` to form $(W' m_1', \ldots, m_{n-1}')$. Pruning is applied until there are no more flexible-head subterms in T with $\forall$-variables not in L. The following transformations will then perform variable-elimination, depending on the form of T.

ff2: flexible-flexible-same If the difference pair is of the form

```
(eq (appl V L) (appl V M))
```

for an $\exists$-variable V, we can assume that L and M are permutations of eachother (since pruning would have been applied otherwise). However, if there is a $\forall$-variable in `L` that does not appear in the same position in `M`, then `V` must be vacuous for both positions in which the variable appears in `L` and `M`. We therefore use `same-index` to find the variables that appear in the same positions in both lists. These are the only positions a most general solution for `V` can depend on. This step is like the pruning step, and could be combined with it if we choose to do so. We then introduce a new $\exists$-variable `V'`, and use `form-lam` to form the abstraction $\lambda L.(V' \ W)$ where W is the list of variables in the same positions in both lists. This difference pair is then eliminated, and the current `ss`-quadruple is updated by the substitution $[(\lambda L.(V' \ W))/V]$.

fr-ff1: flexible-rigid, flexible-flexible-different If the difference pair is of the form

```
eq (appl V L) (appl R M)
```

such that `V` is a $\exists$-variable, and `R` is either a signature constant, a $\forall$-variable or a $\exists$-variable distinct from `V`, then use `form-lam` to create the solution $[(\lambda L.(R \ M))/V]$. As in the `ff2` case, eliminate this difference pair, and update the `ss`-quadruple with this substitution.

The order and repetition of applying these transformations are controlled by the `step1`, `step2` and `step3` clauses at the end of the code. `step1` imposes and updates the ordering of variables. `step2` applies `r-r` and `xi` repeatedly, and `step3` applies `prune` repeatedly followed by `ff2` or `fr-ff1`.

Occur-Check and Other Failure Conditions:

In the last transformation `fr-ff1` we were not specific about under what conditions should unification fail. These conditions were implemented as an indirect consequence of copy clauses. Specifically, the substitution $[(\lambda L.(R\ M))/V]$ would not be valid if either V appears free in $(R\ M)$ (the occur-check condition), or if a $\forall$-variable not in L appears in $(R\ M)$. These two conditions were guaranteed by the goal in the first clause of `fr-ff1`:

```
((pi Z\ (copyi Z Z :- evar Z, distinct-evar Z V))
       => form-lam L (appl R M) Sub),
```

Let's also give the definition of `form-lam`:

```
form-lam nil Body Newbody :- copyi Newbody Body.
form-lam (Y::Ys) Body (lam Nb) :-
  pi v\ (copyi v Y => form-lam Ys Body (Nb v)).
```

Note that `form-lam` will work correctly only if every symbol S in `Body` that is not in the list of variables to be abstracted over `(Y::Ys)` "copies" to itself. Every signature constant always copies to itself. But copy clauses for variables must be given in the program before `form-lam` is called. By assuming the clause

```
(pi Z\ (copyi Z Z :- evar Z, distinct-evar Z V)
```

in `fr-ff1`, we ensure that every $\exists$-variable *except* `V` copies to itself. Now if `V` occurs free in `(appl R M)`, then the final goal of `form-lam`, `copyi Newbody body`, would fail since there are no copy clauses visible for `V`. This implements occur-check. Similarly, the only copy clauses for $\forall$-variables that are visible when `copyi Newbody Body` is called are the `(copyi v Y)` clauses introduced in the second clause of `form-lam`. Thus if a $\forall$-variable not in the list `(Y::Ys)` appears in the `Body`, then `copyi Newbody Body` would fail. This ensures that no $\forall$-variable appears free in the substitution $[(\lambda L.(R\ M))/V]$ created by `fr-ffr`.

As an example of the program, consider the unification problem

$$\exists u \exists v \forall y [f(\lambda x.f(uxy)) = f(\lambda w.vy)].$$

In our notation, this is represented with the initial `ss`-quadruple

```
S = (ss (u::v::nil) (y::nil)
        ((eq (appl f ((lam x\(appl f ((appl u (x::y::nil))::nil)))::nil))
             (appl f ((lam w\(appl v (y::nil)))::nil)))::nil)
        (sub u u (sub v v emp)))
```

Now solving `(lunify S R)` will yield the `ss`-quadruple

```
R = newuvar z\ newevar e\
      (ss (e :: u :: v :: nil) (z :: y :: nil)
          nil
          (sub u (lam v1\ lam v11\ appl e (v11 :: nil))
           (sub v (lam v1\ appl f (appl e (v1 :: nil) :: nil)) emp))).
```

which means that the most general unifier is

$$[\lambda v_1 \lambda v_2.(ev_2)/u,\ \lambda v_1(f(ev_1))/v]$$

for some new free variable e.

The transformations applied to this unification problem are `r-r`, `xi`, `prune` and `fr-ff1`, with β_0-normalization also carried out after the `prune` transformation.

We stress that for purpose of clarity we have not addressed the problem of object-level types in the formulation above. The program presented is technically for untyped, β-normal terms only. Typed unification can be recovered by making the slight changes indicated. Finally, since λ-unification encompasses first-order unification, we have also given an alternative formulation to the program of section 3.2.1. This alternative has the advantage of being much less signature-dependent. Explicit occur-check is avoided and the use of the `appl` and `lam` notation allows a uniform definition of term-decomposition. Only copy clauses for the rigid constants are need, but these can be generated automatically from type information.

4.5 `setsub` Under A Mixed Prefix

The program of the previous section is complex when compared with the first order case largely because of the use of the indirect representation of terms. Since application and λ-abstraction are already available in the syntax of the meta-language, it would be convenient to use this syntax directly. In this section we discuss how the `setsub` technique for utilizing the meta-level unification algorithm for object-level unification can be adopted for use in the L_λ case. Although new free variables are introduced by L_λ unification, there is still a direct correspondence between these free variables and those in the original domain of unification. In the unification transformations `prune` and `ff2` (and we can also include raising) that introduce a new $\exists$-variable, an unique old variable is replaced. The old variable can no longer appear in the range of the unifier to be constructed. Thus every new variable in the resulting unifier directly corresponds to an original ($\exists$-)variable. The new variable is either a pruned version of the original variable (i.e, it's vacuous in some of its arguments), or a raised version of the original variable (it takes additional arguments). If we consider typed-unification, then the raised variables have longer types, and the lowered variables have shorter types than the original.

Recall that `setsub` worked in the first-order case because variables in the range of the mgu is a subset of the original variables in the domain of unification, up to renaming. If this were not true, as in the higher-order unification algorithm [27, 50], there may be new free logic variables left in the unifier after `setsub`. In the L_λ case, although the variables in the range of the mgu is not strictly a subset of the originals, we can still use the fact that each range-variable is directly related to an original to implement `setsub`. That is, the original variables can be used, up to *renaming with raising and lowering.* To encode the relationship between the new variables and the originals, we will use two constructor symbols: Υ_i and $\Psi_{i_1,\ldots,i_n}$, or in λProlog syntax `(raise I)` and `(lower Is)`. New variables will be represented by either $(\Upsilon_i\ x)$ or $\Psi_{i_1,\ldots,i_n}\ x$ where x is an original domain variable. The index i in Υ_i represents the number of new arguments the new variable takes in addition to x. The list $i_1, \ldots, i_n$ in $\Psi_{i_1,\ldots,i_n}$ represents the n positions which the new variable must be vacuous for.

Let's consider u a functional variable of one argument. That is, let's say it has type $i \rightarrow i$ for a primitive type i. If during (meta-level) unification u is lowered (i.e., pruned) it would be substituted by a term $\lambda x.U'$ where U is a new (logic) variable (of type i). To instantiate $\lambda x.U'$ back to a closed term, we will "setsub" it to $\lambda x.(\Psi_1\ u)$, or `x\(lower u)` (using `lower` for Ψ_1). This closed structure can then be converted into a properly quantified object-level variable of the correct type using the same technique as in `close-type2` of the

ML-QuickTyper.

If raising was not involved, then `setsub` for unifying terms of type $i \rightarrow i$ is simply:

```
setsub X X :- !.
setsub X Y\(lower X) :- !.
setsub X Y.
```

Although now there are two cuts, together they serve the same purpose as the single cut in first-order unification: to distinguish it from the `setsub X Y` case. `setsub X X :- !.` is given priority to the clause for lowering because the non-lowered solution is more general.

But pruning or lowering is not all we must address directly. We can not assume that the meta-level unification algorithm is restricted to the $\exists\forall$ case as we did for object-level unification in the previous section. That is, we can not separate raising from the rest of the unification problem. During raising at the meta-level, when a variable `v:i` is substituted by `(V'y)`, where `V'` is of type `i->i` and `y` is of type `i`, we will need to instantiate it to `(raise v)y` (where `raise` is used for Υ_1). But the universal variable(s) `y` that was raised over is unkown when `setsub` is used on `v`. We must therefore keep track of these variables explicitly. We thus extend `setsub` to a *prefixed* version, `psetsub`, which carries the context $y_1, \ldots, y_m$. These are the variables (constant symbols) that will be raised over when unification is performed at the meta level. `psetsub` for $i \rightarrow i$ unification can be defined as follows. We make the further assumption that only one variable will be raised over.

```
psetsub (V::nil) X ((raise X) V) :- !.
psetsub C X X :- !.
psetsub C X Y\(lower X) :- !.
psetsub C X Y.
```

Once again, the raising case is given priority over other cases because it is more general. We can also replace the series of cuts in the `psetsub` program clauses with a cut after each call to `psetsub`.

If `psetsub` for arbitrary types is available, then object-level *mixed* prefix L_λ unification can be achieved by altering the unifiability program by calling `psetsub` for each $\exists$-variable with all the $\forall$-variables that appeared to the left of where this $\exists$-variable was quantified as the prefix argument for `psetsub`.

To achieve unification of arbitrarily typed terms, we can give the following program template for `psetsub`:

```
psetsub C X Y :- raise-with C X Y, !.
psetsub C X X :- !.
psetsub C X Y :- lower-to X Y,!.
psetsub C X Y.
```

where `raise-with` constructs the proper raised version of `X` and apply it to the variables in `C`, forming `Y`, and `lower-to` lowers a variable to an arbitrarily shorter type. The `lower-to` program must also take care that as the type "gets shorter," it gets lower priority in the list of possible results for `Y`. Fortunately, an upper bound can always be put on the number of variables that can be pruned, especially if the terms are typed. Though typing information is important at the object-level, this technique would find easier formulation in an untyped meta-language. In any case, we do not claim `psetsub` as an existing, general technique in the same way `setsub` for first-order terms is. One way to use it effectively is to implement it as a new construct of the meta-language. The necessary `psetsub` clauses can be generated dynamically given the type of terms it is to be used on. The meaning of this predicate is still the program of the form given above. Adding `psetsub` to the (meta) language this way is akin to adding a negation-as-failure predicate to the language, since the latter predicate can also be simulated using `!`. Such practice is certainly not without precedence.

For now, however, we only claim that this technique is useful in isolated examples. In the next section we demonstrate how object-level L_λ resolution can be implemented using `psetsub`. For illustrational purposes, this is preferable to using the purely declarative implementation of L_λ unification for we can use meta-level application and abstraction directly, instead of resorting to the more cumbersom notation of `appl` and `lam`.

4.5.1 Object-Level L_λ Resolution

Using the prefixed `setsub` method, we can formulate (forward) resolution for L_λ. Particularly interesting is the introduction of mixed quantifier prefixes by resolving a `pi`-goal with another clause. The method for quantification over the resolvent of two L_λ clauses is based on the "lifting" method of Paulson in [45]. For clarity, we do not address the case of resolving with embedded implications ($D \Rightarrow G$ goals), though this can also be formulated by a similar technique in [45]. We assume that `i` is the meta-level type representing all object-level expressions. Most types are omited for efficiency in presentation.

The code can be made simpler if we have access to an untyped meta-language, such as the untyped L_λ. With current versions of typed λProlog available, we must use different symbols and lists for quantification and variables at different types.

```
type all   (i -> i) -> i.
type all2  ((i -> i) -> i) -> i.
type all3  ((i -> i -> i) -> i) -> i.
type psetsub   list i -> i -> i -> o.
type psetsub2  list i -> (i -> i) -> (i -> i) -> o.
type r2        (i -> i) -> (i -> i -> i).

form-app F nil F.
form-app F (A::B) N :- form-app (F A) B N.
form-app2 F nil F.
form-app2 F (A::B) N :- form-app2 (F A) B N.

psetsub (U::Us) X Y :- form-app (raise X) (U::Us) Y, !.
psetsub NU X X :- !.
psetsub NU X Y.
psetsub2 (U::Us) X Y :- form-app2 (r2 X) (U::Us) Y, !.
psetsub2 NU X X :- !.
psetsub2 NU X (y \ (lower X)) :- !.
psetsub2 NU X Y.

mp Fl Sl Fr Sr A (all B) (all C) :- !
   pi x\(mp Fl Sl (x::Fr) Sr A (B x) (C x)).
```

```
mp Fl Sl Fr Sr A (all2 B) (all2 C) :- !
   pi x\(mp Fl Sl Fr (x::Sr) A (B x) (C x)).
mp Fl Sl Fr Sr (all A) B (all C) :- !
   pi x\(mp (x::Fl) Sl Fr Sr (A x) B (C x)).
mp Fl Sl Fr Sr (all2 A) B (all2 C) :- !
   pi x\(mp Fl (x::Sl) Fr Sr (A x) B (C x)).
mp Fl Sl Fr Sr (imp A Ah) (imp B Bh) C :-
   mpp Fl Sl Fr Sr (imp A Ah) (imp B Bh) C.

mpp Fl Sl (F::Fr) Sr A B C :-
  ((copyi F T) => mpp Fl Sl Fr Sr A B C), psetsub nil F T.
mpp Fl Sl nil (S::Sr) A B C :-
  ((pi x \ (pi y \ (copyi (S x) (T y) :- copyi x y))) =>
      mpp Fl Sl nil Sr A B C),
  psetsub2 nil S T.
mpp Fl Sl nil nil A B C :- mpp1 Fl Sl nil A B C.

mpp1 Fl Sl NU (imp A B) (imp (all C) D) (imp (all E) F) :-
 pi x \ (copyi x x =>
    mpp1 Fl Sl (x::NU) (imp A B) (imp (C x) D) (imp (E x) F)).
mpp1 Fl Sl NU (imp A B) (imp C D) (imp E F) :-
  mpp2 Fl Sl NU (imp A B) (imp C D) (imp E F).
mpp2 (F::Fv) Sv NU A B C :-
  (copyi F T => mpp2 Fv Sv NU A2 B C), psetsub NU F T.
mpp2 nil (S::Sv) NU A B C :-
  ((pi x \ (pi y \ (copyi (S x) (T y) :- copyi x y))) =>
mpp2 nil Sv NU A B C),
  psetsub2 NU S T.
mpp2 nil nil NU (imp A B) (imp C D) (imp E F) :-
 copyi B N, copyi C N, copyi A E, copyi D F.

properlower (all2 A) (all2 x \ (all (B x))) :- !,
  pi x \ (pi y \ (
  ((pi a\ (pi b\ (copyi (x a) (x b) :- copyi a b))),
   (copyi (lower x) y)) =>
```

```
properlower (A x) (B x y))).
properlower A B :- copyi A B.

properraise (all A) (all x \ (all2 (B x))) :-
  pi x \ (pi y \ (
   (copyi x x,
   (pi a\ (pi b\ (copyi (raise x a) (y b) :- copyi a b)))) =>
     properraise (A x) (B x y))).
properraise (all2 A) (all2 x \ (all3 (B x))) :- !,
  pi f \ (pi r \ (
   ((pi a\ (pi b\ (copyi (f a) (f b) :- copyi a b))),
    (pi a\ (pi b\ (pi c\ (pi d\
(copyi (r2 f a b) (r c d) :- copyi a c, copyi b d)))))) =>
   properraise (A f) (B f r))).
properraise A B :- copyi A B.
```

The meta-level constructors `all`, `all2` and `all3` represent object-level $\forall$-quantification at first, second and third order types. The symbol `r2` represents a Υ_i constructor. The `mp Fl Sl Fr Sr A B C` clause uses four lists to represent first- and second-order variables in the left (`A`) and right (`B`) rules to be resolved (into `C`). It discharges quantifiers and collects these variables in the list. The left rule has the form `(imp A Ah)`. `Ah` is to be resolved with the `B` in the right rule `(imp B Bh)`, yielding the resolvent `(imp A Bh)` with the unifier applied to it. The `mpp` clauses introduces logic variables for quantified variables in the right rule with copy clauses, and uses `psetsub` to close them at the end. Note that second-order copy clauses are naturally used for the second-order variables, which are also second-order at the *meta*-level. This would not be true using the indirect representation of `appl` and `lam` (the terms would all be of primitive meta-type). If `B`, the "body" of the right rule is a `pi`-goal, then the eigenvariable introduced must be scoped over `pi` variables quantified in the left rule. This amounts to creating a *mixed* prefix of the form $\mathcal{Q}\forall\exists$. The purpose of the `mpp1` predicate is therefore to discharge `pi` variables in `B` and place them in the list required by `psetsub` (and `psetsub2`). The `mpp2` predicate then introduce logic variables for the $\forall$ variables in the left rule, under this new $\forall$ prefix. The rules can then be resolved (in the last `mpp2` clause by unifying the head of the left rule with the "body" of the left rule. The unifier is applied to the body of the left rule and the head of the right rule to form the resolvent. Finally, the raised and lowered structures representing variables are abstracted by `properraise` and `properlower` using the same technique employed in the `close-type` predicate of the section 3.5.3.

An interesting test example is the problem of resolving the clause

```
(all2 m\ (imp (all x\ (p (m x))) (p (f m))))
```

with itself. For clarity, assume `p` has type `i -> i` and `f` has type `(i -> i) -> i`. Solving the query

```
 mp nil nil nil nil
  (all2 m\ (imp (all x\ (p (m x))) (p (f m))))
  (all2 m\ (imp (all x\ (p (m x))) (p (f m)))) R.
```

will produce the unique solution:

```
R = all2 x\ all2 x1\
      imp (all x11\ all y\ p (r2 (X1\ x1 X1) x11 y))
          (p (f y\ f X\ r2 (X1\ x1 X1) y X)).
```

After the raised-variable constructs have been properly renamed with real variables (and vacuous quantifiers removed), the final solution becomes

```
T = all3 x\
    imp (all y\ all y1\ p (x y y1)) (p (f y\ f y1\ x y y1)).
```

The cuts in `mp` and `properraise` are for efficiency only and can be eliminated by making the conditions explicit. We've done this in all other programs. Here, since we already require a `!`, we will allow ourselfs the luxury of these efficiency, or *green* cuts, which are used very commonly in logic programs.

4.6 Finding Higher-Order Critical Pairs

We have yet to address how Knuth-Bendix completion can be implemented. We shall not attempt a complete implementation here, since that would require a terminating rewrite relation and an ordering on higher-order terms. These topics are too much outside of the scope of this thesis. Instead, we give the core operation of finding a critical pair given two rewrite rules. Given the techniques and examples of the foregoing, one can expect such a formulation to be rather straightforward. Indeed it is not difficult compared to the other applications, especially for first-order rewrite systems. We therefore chose to formulate Nipkow's extention, in [44], of the critical pair result to a restricted class of higher-order rewrite systems involving only L_λ terms. We will use the purely declarative formulation of object-level L_λ unification of section 4.4.2.

In addition to requiring object-level L_λ unification, the higher-order case offers another challenge. Nipkow's formulation of subterms and contexts for λ-terms does *not* respect α-equivalence classes. Specifically, subterms of closed, λ-bound terms may contain free variables. For illustration, assume that `f` is an unary constructor and `h` some binary constructor. Given a sample term $T = \lambda x.(h\ \lambda z.z\ \lambda y.(f\ x))$, we would like to consider $(f\ x)$ as a subterm of T. This is because it is valid to apply a rewrite rule, such as $(f\ Z) \longrightarrow (g\ Z)$, to $(f\ x)$, yielding the term $(g\ x)$, which again contains x free. In creating a critical pair, we would need to put $(g\ x)$ *back* into the context for $(f\ x)$ in T. This requires the *recapturing* of the free variable x by λ-abstraction. We also must ensure that it is captured under the original abstraction that bound it in T, namely the outer-most λx, and not the inner one λy.

Specifically, Nipkow formulated λ-terms as trees, much like first order term-trees, where each λ-binding λx represents one node (the x is not a seperate node). Positions in the tree are represented by sequences in the usual manner. Positions in λ-terms, or terms with `app` and `abs` representation, are sequences over $\{1, 2\}$. (With `appl` and `lam` they would be over arbitrary numbers, of course.) $T_{@}p$ represents the subterm at position p in T. $T[S]/_{@}p$ represents T with S replacing the subterm at position p. In this way, subterms with bound variables are extracted and put back by "brute force" - *allowing* bound-variable capture.

Since α-equivalence forms the foundation of higher-order abstract syntax, a direct formulation of Nipkow's procedure is not advisable (though possible through careful meta-programming). We will give an alternative formulation of subterms and contexts that does respect α-equivalence classes. This will be of more general use in meta-programming than a specific method for critical-pair detection. The technique for this formulation is similar to the merging of compatible prefixes used in ML type-inferencing, though it is much simpler.

4.6.1 Preserving α-Equivalence

We want a predicate `subterm` such that `subterm S T C` holds if and only if S has an occurrence in T under the context C. Naturally, we want to use meta-level λ-abstraction to represent the context, so that $(C\ S) = T$ is an invariant. Furthermore, we want a context $\lambda x.(C\ x)$ be such that x appears exactly once in $(C\ x)$. For first order terms, `subterm` is not difficult to define. For example, the subterm clauses for the symbols h and f can be specified as follows:

```
subterm X X (c\c).
subterm A (f B) (c\ (f (D c))) :- subterm A B D.
subterm A (h B C) (c\ (h (D c) C)) :- subterm A B D.
subterm A (h B C) (c\ (h B (D c))) :- subterm A C D.
```

It helps at times in this section to think of terms as trees. Since a context is a λ-abstraction where the bound variable appears exactly once, we can always determine which branch of a term it will appear in. That is, if `A` is a subterm of the left branch of `(h B C)`, then the context for `A` must be vacuous for the right branch. In this way the context holding the subterm is propagated from the top level to various sub-branches of the term, until a trivial context can be constructed (representing a trivial subterm).

Subterms of a (object-level) λ-term must also be λ-terms for two reasons. In addition to preserving α-equivalence, we also do not have a denumerable supply of constants to represent the arbitrary number of free variables that may appear in a subterm. Each bound variable in a given subterm, such as $(f\ x)$ in $\lambda x.(h\ \lambda z.z\ \lambda y.(f\ x))$, must be bound by a λ somewhere on the path from that subterm to the root (the top-level term). Thus in finding a subterm of T, whenever a λ-abstraction is encountered on a sub-branch of T, the subterm must also be λ-abstracted. The full definition of the `subterm` relation is as follows. We use the mathematical notation $\overline{\lambda} x.P$ for object-level λ-abstraction `lam x\(P x)`.

Definition 4.4 *(`subterm` with `lam`-bindings)*

Let op be any meta-level constructor symbol (such as `appl`) that constructs representations *of object-level expressions. The* `subterm` *relation is that implied exclusively by the following clauses:*

$\forall X($`subterm` $X\ X\ \lambda c.c)$.

$\forall A \forall X_1 \ldots \forall X_n \forall D($

`subterm` $A\ \ (op\, X_1 \ldots X_i \ldots X_n)\ \ \lambda c.(op\, X_1 \ldots (D\ c) \ldots X_n)\ \Leftarrow$ `subterm` $A\ X_i\ D)$.

$\forall A \forall B \forall C \forall D($

`subterm` $\overline{\lambda} y.(A\ y)\ \ \overline{\lambda} y.(B\ y)\ \ \lambda c.(\overline{\lambda} x.(D\ x\ c))\ \Leftarrow\ \forall x($`subterm` $(A\ x)\ (B\ x)\ (D\ x)))$.

The second clause is meant to hold for any n-ary symbol op.

In λProlog (L_λ) syntax, `subterm` for the `lam` case is equivalent to

```
pi A\pi B\pi C\pi D\
   subterm (lam A) (lam B) (c\ (lam x\(D x c))) :-
                        pi x\ (subterm (A x) (B x) (D x)).
```

Note that if `op` is a two-ary relation (like `app` or `appl`), we do *not* consider `(op X)` (`(app X)`) as a subterm, since it does not correspond to any object-level expression. Only fully-applied constructors are considered as subterms[6].

Using the `subterm` predicate, $S = \overline{\lambda} x'.\overline{\lambda} y'.(f\ x')$ is a subterm of $T = \overline{\lambda} x.(h\ \overline{\lambda} z.z\ \overline{\lambda} y.(f\ x))$, with respect to the context $C = \lambda c.[\overline{\lambda} x.(h\ \overline{\lambda} z.z\ \overline{\lambda} y.c)]$ [7]. In the first-order case, applying the context to another term involves straightforward β-reduction. Unfortunately, $(C\ \ S) = T$ is *not* an invariant here. To recapture the bound variables under the same scopes, the $\overline{\lambda}$-bindings in the the subterm and in the *path* of the context leading to the "hole" must be *merged.*

In defining `subterm` we had in mind that the term is given and a subterm and context are to be "found." But it also works perfectly the other way, to apply a given context to a subterm, yielding a new term. This obviously holds in the first-order case, but since β-reduction is immediately available, it need not be used in this dual role. For terms with λ-abstractions, however, this becomes critical. In the definition of the `lam` case of `subterm`,

[6]These would be terms of types designating object-level classes. such as `term` or `form`. Terms of types such as `form -> form` usually does not correspond directly to object-level expressions. *Object-level* higher-order terms *can* be considered as subterms, such as `F` in `(appl F X)`.

[7]We could have used an abstraction constructor other than `lam` to represent bound variables in a subterm, but this is unnecessary and combersome for our purposes.

we used the same eigenvariable `x` in finding the sub-context `(D x)` as in the subterm `(A x)`. This was necessary for a correct definition since the sub-context may contain the same `x` free. This also means, however, that the λ-abstractions in the context and subterm are merged. As in the prefix-compatiblity technique of the type-inferencer, the ordering of the `lam`-abstractions in the subterm must respect the ordering of those in the corresponding path of the context. Also similar to the type inferencer, no special "trick" is used to ensure this; it comes directly as a result of using meta-level λ-abstractions, which are *not* invariant under permutations. We formalize all this below.

Definition 4.5 *(Context)*
A context *is a term of the form $\lambda c.D$ where the bound variable c appears free exactly once in (the $\beta\eta$-normal form of) D.*

Lemma 4.6 *(Validity of* `subterm`*)*
For any terms S, T, and C, `subterm S T C` *holds if and only if:*

1. *$S =_\alpha \overline{\lambda} x_1 \ldots \overline{\lambda} x_n.S'$, $(0 \leq n)$, such that*

2. *for some position p, $T_@p = S'$ where*

3. *$x_1, \ldots, x_n$ are all the $\overline{\lambda}$-bound variables that includes $T_@p$ in their scope, and such that if $i < j$ then $\overline{\lambda} x_i$ includes $\overline{\lambda} x_j$ in its scope, and*

4. *C is a context, and $(C\ x) =_\alpha T[x]/_@p$ for an arbitrary new constant x.*

Proof: both the forward and reverse directions are provable by induction on the (β-reduced) structure of the context, and are analogous. We give the proof for the reverse direction below.

- For the base case, $C = \lambda c.c$. By part 4 of the lemma, $(C\ x) = x = T[x]/_@p$ for some arbitrary x. Thus $p = \epsilon$ (empty sequence). Let $S = S' = T_@p(= T)$ (no $\overline{\lambda}$-prefix), then immediately (`subterm` $S\ T\ C$) holds.

- For $C = \lambda c.(op\ X_1 \ldots X_m)$, since this is a context c appears in one position in exactly one X_i in $X_1, \ldots, X_m$. Thus $C = \lambda c.(op\ X_1 \ldots (D\ c) \ldots X_m)$ for some context D. For an arbitrary y,
$$(C\ y) = T[y]/_@p = (op\ X_1 \ldots (D\ y) \ldots X_m).$$

So $p = i.p_2$ for some p_2. Let $T_2 = T_{@}i$, so that $T_2[y]/_{@}p_2 = (D\ y)$. Let $S' = T_{2@}p_2$, and let $S = \overline{\lambda}x_1 \ldots \overline{\lambda}x_n.S'$ where $x_1, \ldots, x_n$ satisfies the requisite conditions on $T_{2@}p_2$. By inductive hypothesis, (`subterm` $S\ T_2\ D$) holds, and therefore by the definition of `subterm`,

$$\texttt{subterm}\ S\ (op\ X_1 \ldots T_2 \ldots X_m)\ \lambda c.(op\ X_1 \ldots (D\ c) \ldots X_m)$$

holds.

- for the $\overline{\lambda}$ case, $C = \lambda c.(\overline{\lambda}x.((D\ x)\ c))$. For an arbitrary y,

  $$(C\ y) = T[y]/_{@}p = \overline{\lambda}x.((D\ x)\ y).$$

  Again, since C is a context, for some arbitrary variable x, $(D\ x)$ is a context and $p = 1.p_2$ for some p_2. Let $(T_2\ x) = T_{@}1$ where x is not free in T_2 (raise $T_{@}1$ over x). Then $(T_2\ x)[y]/_{@}p_2 = ((D\ x)\ y)$. Let $S' = (T_2\ x)_{@}p_2$. If the variables $x, x_2, \ldots, x_n$ satisfies the requisite conditions on $T_{@}p$, then $x_2, \ldots, x_n$ satisfies the conditions on $T_2@p_2$. Let $(S_2\ x) = \overline{\lambda}x_2 \ldots \overline{\lambda}x_n.S'$ where x is not free in S_2. Since x is arbitrary, by the definition of uniform probability and by the inductive hypothesis, ($\forall x.$`subterm` $(S_2\ x)\ (T_2\ x)\ (D\ x)$) holds. This in turn implies that

  $$\texttt{subterm}\ \ \overline{\lambda}x.\overline{\lambda}x_2 \ldots \overline{\lambda}x_n.S'\ \ \overline{\lambda}x.(T_2\ x)\ \ \lambda c.(\overline{\lambda}x.(D\ x\ c))$$

  also holds.

□

With the λ-term representation of subterms and contexts and the `subterm` relation replacing β-reduction for the purpose of "plugging" a term into a context, we can again work with α-equivalence classes. The rest of Nipkow's formulation do respect higher-order abstract syntax.

4.6.2 Rewrite Rules in Abstract Syntax

Rewrite rules with (free) variables actually represent rule-schemas, which can be applied to any instantiation of the variables. We have already discussed why meta-logic variables can not be used to represent schema variables. Naturally, we will again use meta-level λ-abstraction to quantify over these variables. Rules are encoded in the meta-program by clauses of the form

```
rr sample-rule (scheme Z\ (rw (f Z) (g Z))),
```

which represents the rule $(f\ Z) \longrightarrow (g\ Z)$. The constructor `scheme` uses λ-abstraction to quantify over scheme variables. The binary constructor `rw` forms a left-to-right rewrite rule. The symbol `rr` constructs a meta-program clause given representing rewrite rules, labeled by a *name* `sample-rule`. For the meta-level signature for these constructs, please see the Appendix containing the full code.

Nipkow's formulation of critical pairs only works with rewrite rules involving terms of base-type (or fully applied terms), though these terms may contain higher-order variables and λ-bindings. The left-hand side of each rewrite rule must be a L_λ term.

The definition of the critical-pair predicate `cp` is given in Figure 4.5. We will use the declarative L_λ unification program of section 4.4, which means using the `appl` and `lam` notation. The meta-level signature of this program can be found in the appendix. The `subterm` relation for terms constructed by `appl` and `lam` is given in the program. Since in finding critical pairs we need not worry about trivial terms such as variables, we will restrict the base case of `subterm` to only composite terms of the form `(appl A B)`. The use of lists in `appl` expressions only slightly complicate the `subterm` definition. Since Nipkow's formulation only works for rewrite rules of base-type, we also do not consider `(appl F (x::nil))` as a subterm of `(appl F (x::y::nil))`. This is convenient, since otherwise we will have to adopt terms of the form `(app (app F x) y))` in order for `subterm` to work correctly. We will ignore type information in this presentation for simplicity, but a full implementation would also have to check for (object-level) type consistency.

Nipkow's formulation of critical pairs requires only L_λ $\exists\forall$-unification. Raisings are done manually. Critical pairs are given in the form of a new rewrite rule between the conflicting reducts[8]. Free variables in the critical pairs are again abstracted by `scheme`. Recall that this would not be possible if meta-logic variables are used for object-level free variables.

The main clauses of the program are `find-cp`, `cp-aux` and `form-cp`. Given two rules, `find-cp` discharges `scheme`-bindings with new constants representing object-level $\exists$-variables, so they can participate in unification. `scheme`-bindings in different rules are discharged separately (so there is no need for the renaming function of Nipkow's formulation). It then finds a (non-trivial) subterm `S` of the left handside of one rule (`L1`) as a

[8]Technically, the Knuth-Bendix completion procedure only calls for creating a new rewrite rule from the completely reduced reducts, not the critical pair. For illustrational purposes, since we do not have a complete-reduction relation, we simply create a new rule from the critical pair itself. This rule is still valid with respect to the original rewriting system.

```
subterm X X (c\c) :- X = (appl A B).
subterm A (appl B C) (c\ (appl (D c) C)) :- subterm A B D.
subterm A (appl B C) (c\ (appl B (D c))) :- subterm-list A C D.
subterm (lam A) (lam B) (c\ (lam x\(D x c))) :-
  pi x\ (subterm (A x) (B x) (D x)).
subterm-list A (B::L) (x\ ((C x)::L)) :- subterm A B C.
subterm-list A (B::L) (x\ (B::(M x))) :- subterm-list A L M.

cp R1 R2 Cp :- rr R1 Rule1, rr R2 Rule2, find-cp nil Rule1 Rule2 Cp.
find-cp Vs (scheme Rule1) Rule2 (scheme Cp) :-
  pi v\ (evar v => find-cp (v::Vs) (Rule1 v) Rule2 (Cp v)).
find-cp Vs (rw L1 R1) (scheme Rule2) (scheme Cp) :-
  pi v\ (evar v => find-cp (v::Vs) (rw L1 R1) (Rule2 v) (Cp v)).
find-cp Vs (rw L1 R1) (rw L2 R2) Cp :-
  subterm S L1 CL1, cp-aux Vs nil S CL1 R1 L2 R2 Cp.

cp-aux Fv Bv (lam S) CL1 R1 L2 R2 Cp :-
  pi b\ (append Bv (b::nil) (Br b),
         cp-aux Fv (Br b) (S b) CL1 R1 L2 R2 Cp).
cp-aux Fv Bv S CL1 R1 L2 R2 Cp :- non-lam S,
  ((pi Z\ (copyi Z (appl Z Bv) :- evar Z)) => (copyi L2 L2b, copyi R2 R2b)),
  form-trivials Fv TFv,
  lunify (ss Fv Bv ((eq S L2b)::nil) TFv) U,
  ((pi Z\ (copyi Z Z :- evar Z)) => form-lam Bv R2b R2c),
  subterm R2c Cpr CL1,
  form-cp (rw R1 Cpr) U Cp.

form-cp A (newevar U) (scheme Cp) :-
  pi s\ (evar s => form-cp A (U s) (Cp s)).
form-cp A (newuvar U) Cp :- pi x\ (form-cp A (U x) Cp).
form-cp A (ss Es Us nil (sub X Y U)) C :-
  copyi X Y => form-cp A (ss Es Us nil U) C.
form-cp (rw A B) (ss Es Us nil emp) (rw C D) :-
  copyi A A2, copyi B B2, beta_0 A2 C, beta_0 B2 D.
```

Figure 4.5: L_λ Critical-Pair Program

candidate for a reduction conflict. The first clause of `cp-aux` discharges the `lam`-bindings with new $\forall$-variables (`Bv` for Bound variables). To allow the $\exists$-variables in the second rule to be unified with the subterm, which includes new $\forall$ variables, each such $\exists$-variable `z` must be "raised" to `(appl z Bv)`[9]. This is the purpose of the goal

```
(pi Z\ (copyi Z (appl Z Bv) :- evar Z)) =>
   (copyi L2 L2b, copyi R2 R2b),
```

in the second clause of `cp-aux`. The goal `form-trivials Fv TFv` merely forms the trivial substition $[x/x]$ for each $\exists$-variable in `FV`, since this is needed by the initial `ss`-quadruple of the object-level unification program. The (raised) left-hand side of the second rule, `L2` is then unified with the subterm `S`, with unifier `U`.

The procedure implied by Nipkow's formulation would call for "plugging" the raised right-hand side of the second rule, `R2b`, into the context found with the subterm `S`. To do this correctly using `subterm`, `R2b` must be `lam`-abstracted with the $\forall$-variables `Bv`. In the first clause of `cp-aux`, `append` was used instead of `::` (*cons*) to ensure that `Bv` respected the order of `lam` abstractions in the original subterm. We use `form-lam` of the L_λ unification program for this purpose, forming `R2c`. The critical goal `subterm R2c Cpr CL1` does the "plugging," though not by brute force, of `R2c` into the context `CL1`, forming the term `Cpr`.

Finally, the `form-cp` clauses applies the unifier `U` (which comes as a `ss`-quadruple) to the term `Cpr` and the right-hand side of the first rule to form the critical pair. New $\exists$-variables resulting from the unification naturally becomes `scheme`-variables in the new rule representing the critical pair. Since both these new variables and `scheme`-variables are represented by meta-level λ-abstraction, this is straightforward to program. New $\forall$-variables from the unification can not appear free in the unifier, and so they are ignored (by the second clause of `form-cp`). We also took the liberty of β_0-reducing the critical pair, though this was not part of Nipkow's formulation.

4.6.3 Sample Application: Local-Confluence of $\beta\eta$-Reduction

To illustrate the critical pairs program, we will have to adopt the more indirect representation of application and abstraction. This effectively implies three levels of syntax: the

[9] I.e, a unification problem of prefix form $\mathcal{Q}\forall\exists$ is created.

object-level language, the intermediate representation using `appl` and `lam`, and finally the meta-logic itself. To illustrate this indirect style of meta-programming, we will consider the sample *object-theory* of the untyped λ-calculus[10]. Let `ab0` and `ap0` represent object-level abstraction and application respectively. `ab0` forms an abstraction using the intermediate `lam`, which in turn rely on the meta-level `\`. The rules for object-level β and η reductions can be specified by:

```
rr beta
  (scheme F\ (scheme S\ (rw
    (appl ap0 ((appl ab0 (F::nil))::S::nil))  (appl F (S::nil))))).
rr eta
  (scheme S\ (rw (appl ab0 ((lam x\ (appl ap0 (S::x::nil)))::nil)) S)).
```

That is, we meant to say that (`ab0` $\lambda x.($ `ap0` $S\ x))$ rewrite via `eta` to S. Because of the presence of the intermediate representation, it has to be written in the form above. [11]

The query `cp beta eta A` will yield the (lone) critical-pair

```
A = scheme v\ scheme v1\ scheme v2\
      rw (appl ap0 (v2 :: v1 :: nil)) (appl ap0 (v2 :: v1 :: nil)).
```

Without the intermediate representation this would be equivalent to

```
scheme v1\ scheme v2\ rw (ap0 v2 v1) (ap0 v2 v1),
```

This pair is identical, as expected. The query `cp eta beta B` will yield the critical pair

```
B = scheme v\ scheme v1\ scheme v2\ scheme s\
      rw (appl ab0 (s :: nil))
         (appl ab0 ((lam v3\ appl s (v3 :: nil)) :: nil)).
```

Without the intermediate representation this would be equivalent to

```
scheme s\  (rw (ab0 s) (ab0 x\(s x)).
```

This pair also represents a common reduct, once we notice that (in the intermediate representation),

[10] This example was also borrowed from [44].

[11] We stress that translation between these levels of syntactic representations can be made automatic. We are forced to use the full, cumbersome syntax in order to implement the technique in an existing λProlog interpreter.

```
(lam v3\ appl s (v3 :: nil))
```

is equivalent to `s`. It follows that the terminating fragment of the λ-Calculus is indeed confluent with respect to $\beta\eta$-reduction.

Conclusion

In this thesis, we have provided techniques for the formulation of an important class of problems in meta-programming in logic. In this we have succeeded in retaining the higher-order abstract syntax of object-level expressions. These techniques revolve around the explicit treatment of object-level free variables. Substitutions are formulated in logic programming with copy clauses. Copy clauses are then used to formulate object-level equality, matching and unifiability. The formulation of most-general unifiers, however, can not be as straightforward without using a `!`. In the first-order case, the `setsub` technique, if used only with closed expressions, can lead to simpler programs. We can always avoid the extra-logicality of `setsub` with the algorithmic approach to object-level unification. Fortunately, both the first-order and L_λ higher-order unification algorithms are suitable for HoHH logic-programming specification because they can be given as systems of modular transformations. Preserving higher-order abstract syntax in the so-called "ground representation" requires dynamically-generated object-level free variables to be represented as meta-level λ-bound variables. This technique, which may seem unnatural at first, gives rise to a better style of meta-programming. In particular, since object-level quantification is also represented in the meta-logic by λ-abstraction, generalizing new free variables into quantified variables is almost immediate. In formulating polymorphic type inferencing (and higher-order critical pairs), we also discovered the technique of prefix compatibility as a means to preserve the meaning of free variables under λ-binding.

Concerning the meta-logical treatment of higher-order systems, we extended object-level unification to the L_λ case and have shown its applicability in finding critical pairs with λ-bindings. We have also completed the embedding of higher-order unification in logic programming, up to the point that the non-determinism of higher-order unification is moved to the level of proof-search. In L_λ programming there is no need to be concerned with how the underlining interpreter will behave with regard to unification. We have therefore shown in this thesis that general meta-programming is feasible within the simple meta-logic of L_λ. While we do not deny the usefulness of more powerful languages, this frugality entails more universal applicability.

Related Work of the Author

In addition to the Wolog technique, the Prolog meta-interpreter, meta-partial evaluator, polymorphic type inferencer (both versions), and critical-pair completion, the author has also experimented with other areas requiring the basic techniques given here. Among these is the logic programming re-formulation of Boyer-Moore style inductive theorem proving and the automatic testing of permutability of inference rules in sequent calculi. Except for the Wolog section at the end of Chapter 2, however, the applications we addressed in detail did not include an important class of meta-programming problems: those concerning reasoning with formal proof systems. It has been shown (see [5, 34]) that type-theoretic frameworks can be useful in developing theorem proving enviornments that extract useful proof terms. The LF type-theoretic logical framework supports the judgments-as-types principle while lending itself suitable for direct operationalization in a logic programming-like manner [46, 47]. In the precursor of this thesis, [32], copy clauses were extended to the LF dependent type system. The theorems and proofs of Chapter 2 were extended to dependent types. Copy clauses became four-place predicates mutually defined on three levels (terms, types and kinds). This work was built upon similar efforts by Felty [13]. Formulating a notion of substitution that preserves the relationship between proof (as term) and formula (as type) can clearly be useful. It was hoped that these copy clauses can lead to program transformers and theorem-proving by analogy techniques. To incorporate that work into the context of this thesis, however, requires at least some notion of object-level unification for dependently-typed terms. Although this should be possible, it has yet to be explored. Futhermore, using dependent-typed copy clauses in L_λ, although possible, may put undue strain on the logic programmer. Here more powerful meta-language features would become useful.

Bibliography

[1] Andrew Appel and Zhong Shao. Smartest Recompilation. In *Tenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1993. Longer version as Princeton University Technical Report CS-TR-395-92.

[2] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.

[3] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[5] Theirry Coquand and Gérard Huet. *Constructions: A Higher Order Proof System for Mechanizing Mathematics*, volume 203 of *EUROCAL85, Springer-Verlag LNCS*, pages 151–184. Springer-Verlag, Linz, 1985.

[6] S. Costantini and G. A. Lanzarone. A metalogic programming language. In *Sixth International Conference on Logic Programming*, pages 218–233. MIT Press, 1990.

[7] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 207–212, 1982.

[8] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. In Alberto Maria Segre, editor, *Sixth International*

Workshop on Machine Learning, pages 447–449, San Mateo, California, June 1989. Morgan Kaufmann Publishers. Expanded version available as Technical Report CMU-CS-89-160, Carnegie Mellon University, Pittsburgh.

[9] Scott Dietzen and Frank Pfenning. A declarative alternative to assert in logic programming. In *Proceedings of the 1991 International Logic Programming Symposium*, pages 372–386. MIT Press, 1991.

[10] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In *International conference on typed lambda calculi and applications*, pages 139–145, Utrecht, The Netherlands, 1993. Springer-Verlag LNCS 664.

[11] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[12] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, University of Pennsylvania, August 1989.

[13] Amy Felty. Encoding dependent types in an intuitionistic logic. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks.* Cambridge University Press, 1991.

[14] Amy Felty. Higher-order conditional rewriting in the L_λ logic programming language. In Evelina Lamma and Paola Mello, editors, *Third International Workshop on Extensions to Logic Programming: Preprints of the Proceedings*, February 1992.

[15] Amy Felty and Doug Howe. Tactic theorem proving with refinement-tree proofs and metavariables. In *Twelfth International Conference on Automated Deduction.* Springer-Verlag LNCS, June 1994.

[16] Gerhard Gentzen. Investigations into logical deductions, 1935. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[17] Mike Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, July 1985.

[18] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[19] John J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs.* PhD thesis, University of Pennsylvania, August 1990.

[20] F. Harmelen and A. Bundy. Explanation-based generalization = partial evaluation. *Artificial Intelligence*, 36:401–412, 1988.

[21] Robert Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[22] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[23] P. M. Hill and J. G. Gallagher. Meta-programming in logic programming. Technical Report Report 94.22, University of Leeds, hill@scs.leeds.ac.uk, August 1994. To appear in Vol. 5 of the *Handbook of Logic in Artificial Intelligence and Logic Programming,* Oxford University Press.

[24] Pat Hill and John Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[25] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press, 1986.

[26] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, May 1994. Available as University of Pennsylvania Technical Reports MS-CIS-92-28 or LINC LAB 269.

[27] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[28] S. Kedar-Cabelli and L.T. McCarty. Explanation-based generalization as resolution theorem proving. In *Proceedings of the 4th International Machine Learning Workshop*, pages 383–389. Morgan Kaufmann, 1987.

[29] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[30] J. Komorowski. An introduction to partial deduction. In *Third International Workshop on Meta-Programming in Logic*, pages 49–69, Uppsala, Sweden, June 1992. MIT Press.

[31] R. Kowalski. Predicate logic as a programming language. *Information Processing 74*, pages 569–574, 1974.

[32] C. Liang. Dissertation proposal: Substitutions as logic programming specifications. September 1994. Unpublished Manuscript, available by anonymous ftp from ftp.cis.upenn.edu/pub/papers/liang/proposal.ps.

[33] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Lanuages and Systems*, 4(2):258–282, April 1982.

[34] Per Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory Lecture Notes. Bibliopolis, Napoli, 1984.

[35] Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[36] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[37] Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.

[38] Dale Miller. Abstract syntax and logic programming. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in Lecture Notes in Artificial Intelligence, pages 322–337. Springer-Verlag, 1992.

[39] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, pages 321–358, 1992.

[40] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, pages 272–281, Paris, July 1994.

[41] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[42] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: a unifying view. *Machine Learning, 1*, pages 47–80, 1986.

[43] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[44] Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 342–349. IEEE, July 1991.

[45] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, September 1989.

[46] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[47] Frank Pfenning. Structural cut elimination. In *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166. IEEE Computer Society Press, 1995.

[48] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[49] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[50] Wayne Snyder and Jean H. Gallier. Higher order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1-2):101–140, 1989.

[51] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge MA, 1986.

[52] D. van Dalen. *Logic and Structure*. Springer Verlag Universitext, 1980.

[53] P. J. Voda. The logical reconstruction of cuts in one solution operators. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*. MIT Press, 1989.

Appendix A

Complete Program Codes

This Appendix contains the complete program codes referred to in the thesis. Code for the object-level unification program of section 3.2.1, the Prolog Meta-interpreter of section 3.4, the `quicktyper` program of section 3.5.3 and the L_λ resolution code of section 4.5.1 are already complete as given in the thesis. Nevertheless we will still include the code for the object-level unification program since it includes some important declarations, and is part of the MLtyper program. λProlog codes are stored into *modules.* Comment lines are marked by `%`. The only internal data structure of the λProlog interpreter used are lists and integers in section 3.5.3. Only the built-in predicates used are `member` and `append` and integer addition. All these structures and operations can be given explicit definitions in L_λ as well.

The Purely Declarative Object-Level Unification Program, Section 3.2.1

```
module mltunify.
import lists.
% Standard First-Order Unification Algorithm Without !
% By Chuck Liang, January 1993; slightly revised October 1994.
% Customized for purpose of type-inferencing July 1995, add-trivials added.

kind poly type.
kind substitution type.
kind difference type.
```

```
type emp substitution.
type sub poly -> poly -> substitution -> substitution.
type unify1 (list poly) -> (list poly) -> poly -> poly -> substitution -> o.
type unify (list poly) -> poly -> poly -> substitution -> o.
type sub_apply poly -> substitution -> poly -> o.
type diff poly -> poly -> difference.
type rigid poly -> o.
type vrd poly -> poly -> o.
type var poly -> o.
type distinct-vars poly -> poly -> o.
type occur-check poly -> poly -> o.
type transform (list difference) -> (list difference) -> substitution
                        -> substitution -> o.
type copypoly poly -> poly -> o.
type unify0 (list difference) -> substitution -> substitution -> o.
type diff-subst substitution -> (list difference) -> (list
difference) -> o.
type sub-subst substitution -> substitution -> substitution -> o.
type add-trivials (list poly) -> substitution -> substitution -> o.
type in-domain poly -> substitution -> o.
type not-indomain poly -> substitution -> o.
type integer poly.
type arr poly -> poly -> poly.

copypoly integer integer.
copypoly (arr A B) (arr C D) :- copypoly A C, copypoly B D.
rigid integer.
rigid (arr A B).

transform ((diff X X)::S) S Sub Sub :- var X.
transform ((diff integer integer)::S) S Sub Sub.
transform ((diff (arr A B) (arr C D))::S)
          ((diff A C)::(diff B D)::S) Sub Sub.
transform ((diff X T)::S) S2 Sub (sub X T Sub2) :-
  var X,
```

```
  occur-check X T,
  diff-subst (sub X T emp) S S2,
  sub-subst (sub X T emp) Sub Sub2.

distinct-vars X Y :- var X, var Y, (vrd X Y; vrd Y X).

occur-check X integer.
occur-check X (arr A B) :- occur-check X A, occur-check X B.
occur-check X Y :- distinct-vars X Y.

unify1 O (V::Vs) A B Sub :-
  (var V, (pi Z\ (vrd V Z :- member Z Vs))) => unify1 O Vs A B Sub.
unify1 O nil A B Sub :-
  unify0 ((diff A B)::nil) emp Sub1,
  add-trivials O Sub1 Sub.

unify0 nil Sub Sub.
unify0 ((diff A B)::R) Sub Sub2 :-
  var B, rigid A, unify0 ((diff B A)::R) Sub Sub2.
unify0 (D::Ds) Sub Sub3 :-
  transform (D::Ds) S Sub Sub2,
  unify0 S Sub2 Sub3.

unify Vs A B U :- unify1 Vs Vs A B U.

add-trivials nil S S.
add-trivials (V::Vs) S S2 :- in-domain V S, add-trivials Vs S S2.
add-trivials (V::Vs) S (sub V V S2) :-
  not-indomain V S, add-trivials Vs S S2.

in-domain V (sub V X Y).
in-domain V (sub U X Y) :- in-domain V Y.
not-indomain V emp.
not-indomain V (sub U X Y) :- (vrd V U; vrd U V), not-indomain V Y.
```

```
sub_apply A emp A.
sub_apply A (sub V T Ss) B :-
  ((copypoly V T, (pi Y\ (copypoly Y Y :- distinct-vars Y V))) =>
    copypoly A C), sub_apply C Ss B.

diff-subst Sub nil nil.
diff-subst Sub ((diff A B)::R) ((diff C D)::R2) :-
  sub_apply A Sub C, sub_apply B Sub D,
  diff-subst Sub R R2.

sub-subst Sub emp emp.
sub-subst Sub (sub A B R) (sub C D R2) :-
  sub_apply A Sub C, sub_apply B Sub D,
  sub-subst Sub R R2.
```

The Prolog Meta-Partial Evaluator

This program is given in place of the Prolog Meta-Interpreter.

```
module parteval.
import lists.

kind term type.
kind form type.
type copyterm term -> term -> o.
type copyform   form -> form -> o.

type truth           form.
type neg             form -> form -> form.
type and      form -> form -> form.
type or      form -> form -> form.
type imp      form -> form -> form.
type all      (term -> form) -> form.
type exists      (term -> form) -> form.
type w term.
type v term.
```

```
type u term.
type adj term -> term -> form.
type path term -> term -> form.
type prog form -> o.
type atom form -> o.

copyform (adj A B) (adj C D) :- copyterm A C, copyterm B D.
copyform (path A B) (path C D) :- copyterm A C, copyterm B D.
copyform (all A) (all B) :-
pi x\ (pi y\ (copyterm x y => copyform (A x) (B y))).
copyform (exists A) (exists B) :-
pi x\ (pi y\ (copyterm x y => copyform (A x) (B y))).
copyform (and A B) (and C D) :- copyform A C, copyform B D.
copyform (imp A B) (imp C D) :- copyform A C, copyform B D.
copyform truth truth.

prog (and (adj a b) (and (adj b c) (and (all x\(path x x))
          (all x\(all y\(all z\(imp (and (adj x y) (path y z))
                                    (path x z)))))))).
atom (adj X Y).
atom (path X Y).

kind subst                type.
type emp                  subst.
kind sfpair type.
type pe subst -> (list form) -> sfpair.
type sub                  term -> term -> subst -> subst.
type newvar              (term -> sfpair) -> sfpair.
type apply_sub            form -> subst -> form -> o.
type apply_sub_term       term -> subst -> term -> o.
type list-app (list form) -> subst -> (list form) -> o.
type compose-pe    sfpair -> sfpair -> sfpair -> o.
type compose_subs    subst -> subst -> subst -> o.
type setsub    term -> term -> o.
type form-trivials (list term) -> subst -> o.
```

```
setsub X Y :- !.
setsub X X.

apply_sub M emp N         :- copyform M N.
apply_sub M (sub V VT Ss) N :- copyterm V VT => apply_sub M Ss N.
apply_sub_term M emp N    :- copyterm M N.
apply_sub_term M (sub V VT Ss) N :-
  copyterm V VT => apply_sub_term M Ss N.
list-app nil U nil.
list-app (X::Xs) U (Y::Ys) :- apply_sub X U Y, list-app Xs U Ys.

compose-pe (pe M G) (newvar N) (newvar U) :-
  pi z\ (compose-pe (pe M G) (N z) (U z)).
compose-pe (pe M G) (pe N H) (pe U K) :-
  compose_subs M N U, list-app G N G2, append G2 H K.
compose_subs M emp M.
compose_subs emp (sub V VT Ns) emp.
compose_subs (sub V VT Ms) (sub X Y Z) (sub V UT Us) :-
  apply_sub_term VT (sub X Y Z) UT, compose_subs Ms (sub X Y Z) Us.

form-trivials nil emp.
form-trivials (X::Xs) (sub X X Z) :- form-trivials Xs Z.

type partial      form -> list term -> form -> sfpair -> o.
type partial_aux  form -> list term -> sfpair -> form -> sfpair -> o.
type backchain   form -> list term -> form -> form -> sfpair -> o.
type resolve     list term -> form -> form -> form -> subst -> o.
type operational form -> o.
type non-operational form -> o.

operational (adj X Y).
non-operational (path X Y).

partial Cs Vs A (pe TV (A::nil)) :-
```

```
  atom A, operational A, form-trivials Vs TV, backchain Cs Vs Cs A U.
partial Cs Vs A U :-
  atom A, non-operational A, backchain Cs Vs Cs A U.
partial Cs Vs (and A B) U :- partial Cs Vs A M, partial_aux Cs Vs M B U.
partial_aux Cs Vs (newvar M) B (newvar U) :-
  pi z\(partial_aux Cs (z::Vs) (M z) B (U z)).
partial_aux Cs Vs (pe M Gs) B U :-
  apply_sub B M B2, partial Cs Vs B2 N, compose-pe (pe M Gs) N U.
backchain Cs Vs (and C D) A U :-
  backchain Cs Vs C A U; backchain Cs Vs D A U.
backchain Cs Vs (all D) A (newvar U) :-
  pi z\(backchain Cs (z::Vs) (D z) A (U z)).
backchain Cs Vs D A (pe M nil) :- atom D,
  resolve Vs (imp truth D) A N M.
backchain Cs Vs (imp G D) A U :- resolve Vs (imp G D) A NewG M,
  partial Cs Vs NewG N, compose-pe (pe M nil) N U.

resolve (V::Vs) (imp G D) A NewG (sub V XV Us) :-
  (copyterm V XV => resolve Vs (imp G D) A NewG Us), setsub V XV.
resolve nil (imp G D) A NewG emp :-
  copyform D N, copyform A N, copyform G NewG.

%  prog P, partial P (w::nil) (path a w) U.

type abstract-over (list term) -> form -> form -> o.
type ebg-partial form -> (list term) -> form -> form -> o.
type form-clause (list term) -> form -> sfpair -> form -> o.

abstract-over (V::Vs) C (all NewC) :-
  pi z\ (copyterm V z => abstract-over Vs C (NewC z)).
abstract-over nil C NewC :- copyform C NewC.

ebg-partial Cs Vs G (and N Cs) :-
  partial Cs Vs G U, form-clause Vs G U N.
form-clause Vs G (newvar U) (all N) :-
```

```
  pi x\ (form-clause Vs G (U x) (N x)).
form-clause Vs G (pe U nil) G3 :-
  apply_sub G U G2, abstract-over Vs G2 G3.
form-clause Vs G (pe U (F::nil)) G3 :-
  apply_sub G U G2, abstract-over Vs (imp F G2) G3.
form-clause Vs G (pe U (X::Y::Z)) F :-
  form-clause Vs G (pe U ((and X Y)::Z)) F.
```

The MLTyper Program, Section 3.5.1

```
module mlt.
import mltunify lists.

% ML Type inferencer, with abs, app and let.  Chuck liang, 7/95, revised
% from old version of 4/94.  This is a purely declarative L-lambda program.

kind tm type.
type abs (tm -> tm) -> tm.
type app tm -> tm -> tm.
type let (tm -> tm) -> tm -> tm.
type zero tm.
type succ tm.
type op tm.
type op2 tm.
type op3 tm.
type op4 tm. type op5 tm.
type all      (poly -> poly) -> poly.
type newvar (poly -> substitution) -> substitution.
type typeunify (list poly) -> poly -> poly -> substitution -> o.
type merge-sub substitution -> substitution -> substitution -> o.
type append-sub substitution -> substitution -> substitution -> o.
type merge-type poly -> poly -> poly -> o.
type typesub poly -> substitution -> poly -> o.
type apply_sub       poly -> substitution -> poly -> o.
type update-sub substitution -> substitution -> substitution -> o.
```

```
type fill-out substitution -> poly -> substitution -> poly -> o.
type monotype poly -> o.
type polytype (list poly) -> substitution -> tm -> poly -> o.
type resolve-for poly -> (list poly) -> substitution -> substitution -> o.
type rsub2 (list poly) -> poly -> substitution -> substitution -> o.
type rsub3 (list poly) -> (list poly) -> poly -> substitution -> substitution -> o.
type collect-for poly -> substitution -> (list poly) -> substitution -> o.
type resolve-sub (list poly) -> (list poly) -> substitution -> substitution -> o.
type membrest poly -> (list poly) -> (list poly) -> o.

monotype integer.
monotype (arr A B) :- monotype A, monotype B.

polytype A emp zero integer.
polytype A emp succ (arr integer integer).
polytype A emp op (all x\ (arr x x)).
polytype A emp op2 (all x\ (all y\ (arr x (arr x x)))).
polytype A emp op3 (all x\ (arr (arr x x) (arr x x))).
polytype A emp op4 (all x\ (all y\ (arr x (arr y y)))).
polytype A emp op5 (all x\ (all y\ (arr (arr x x) (arr y y)))).

polytype NV (newvar Mgu) (app M N) (all Ty) :-
pi tv\ (sigma Tn\ (sigma Mg\ (sigma Mg2\ (monotype tv => (
  polytype NV Mt1 M Ar1, fill-out Mt1 Ar1 Mt Artype,
  polytype NV Nt1 N T1,    fill-out Nt1 T1 Nt T,
  merge-type T tv Tn,
  typeunify (tv::NV) Tn Artype Mg,
  append-sub Nt Mt MN, merge-sub MN Mg Mg2,
  resolve-sub (tv::NV) (tv::NV) Mg2 (Mgu tv),
  typesub tv (Mgu tv) (Ty tv)))))).

polytype NV (newvar Mgu) (abs M) (all Typ) :-
  pi x\ (pi a\ (sigma A2\ (sigma B\ (sigma Mg\ (sigma Ty\ (monotype a => (
    ((pi Any\ (polytype Any emp x a))
=> polytype (a::NV) Mg (M x) B),
```

```
    resolve-sub (a::NV) (a::NV) Mg (Mgu a),
    merge-type a B Ty,
    typesub Ty (Mgu a) (Typ a)))))))).

polytype NV M (let R T) C :-
  polytype NV Mt T Ty,
  pi x\ ((pi A\ (polytype A Mt x Ty)) => polytype NV Mr (R x) B),
  append-sub Mr Mt Mrt, resolve-sub NV NV Mrt M,
  typesub B M C.


merge-type (all A) (all B) (all C) :-
  pi v\ (monotype v => merge-type (A v) (B v) (C v)).
merge-type (all A) B (all C) :- monotype B,
  pi u\ (monotype u => merge-type (A u) B (C u)).
merge-type A (all B) (all C) :- monotype A,
  pi u\ (monotype u => merge-type A (B u) (C u)).
merge-type A B (arr A B) :- monotype A, monotype B.

merge-sub emp L L.
merge-sub (newvar M) (newvar N) (newvar L) :-
  pi m\ (merge-sub (M m) (N m) (L m)).
merge-sub (newvar M) N (newvar L) :- (N = emp; N = (sub A B C)),
  pi x\ (copypoly x x => merge-sub (M x) N (L x)).
merge-sub (sub A B C) (newvar N) (newvar L) :-
  pi x\ (merge-sub (sub A B C) (N x) (L x)).
merge-sub (sub X Y Ss) A (sub X Y B) :- merge-sub Ss A B.

append-sub (newvar M) N (newvar L) :-
  pi x\ (copypoly x x => append-sub (M x) N (L x)).
append-sub (sub A B C) (newvar N) (newvar L) :-
  pi z\ (append-sub (sub A B C) (N z) (L z)).
append-sub emp L L.
append-sub (sub X Y Ss) A (sub X Y B) :- append-sub Ss A B.
```

```
resolve-for A Vl M N :- (M = emp; M = (sub X Y Z)),
  ((pi U\ (vrd A U :- member U Vl; rigid U)) => rsub2 Vl A M N).
rsub2 Vl A M (sub A T P) :- collect-for A M AL N, rsub3 (A::Vl) AL T N P.
rsub3 (A::Vl) nil A N N.
rsub3 (A::Vl) (X::nil) X N M :-
  unify Vl X X U, update-sub N (sub A X U) M.
rsub3 Vl (X::Y::R) Z N M :-
  unify Vl X Y U, apply_sub X U X2,
  update-sub N U P, rsub3 Vl (X2::R) Z P M.
collect-for A emp nil emp.
collect-for A (sub A A R) Tr R2 :- collect-for A R Tr R2.
collect-for A (sub A T R) (T::Tr) R2 :- vrd A T, collect-for A R Tr R2.
collect-for A (sub B T R) Tr (sub B T R2) :- (vrd A B; vrd B A),
   collect-for A R Tr R2.

resolve-sub Vs Vl (newvar A) (newvar B) :-
  pi v\ (append Vs (v::nil) (Vt v),
         append Vl (v::nil) (Vm v),
         resolve-sub (Vt v) (Vm v) (A v) (B v)).
resolve-sub nil Vl A A :- (A = emp; A = (sub X Y Z)).
resolve-sub (V::Vs) Vl A C :- (A = emp; A = (sub X Y Z)),
  membrest V Vl Vn,
  resolve-for V Vn A B, resolve-sub Vs Vl B C.

membrest X (X::Xs) Xs.
membrest X (Y::Xs) (Y::Ys) :- membrest X Xs Ys.

typeunify L (all A) B (newvar Mgu) :-
  pi t \ (monotype t => typeunify (t::L) (A t) B (Mgu t)).
typeunify L A (all B) (newvar Mgu) :- monotype A,
  pi t \ (monotype t => typeunify (t::L) A (B t) (Mgu t)).
typeunify L A B Mgu :- monotype A, monotype B,  unify L A B Mgu.

typesub (all A) (newvar U) (all B) :-
  pi v\ (monotype v => typesub (A v) (U v) (B v)).
```

```
typesub (all A) S (all B) :- (S = emp; S = (sub X Y Z)),
  pi x\ (monotype x => (copypoly x x => typesub (A x) S (B x))).
typesub A (newvar S) (all B) :- monotype A,
  pi x\ (typesub A (S x) (B x)).
typesub A S B :- monotype A, (S = emp; S = (sub X Y Z)), apply_sub A S B.

apply_sub A (sub X Y Z) B :- copypoly X Y => apply_sub A Z B.
apply_sub A emp B :- copypoly A B.

update-sub emp U emp.
update-sub (sub X Y Z) U (sub X Y2 Z2) :-
  apply_sub Y U Y2, update-sub Z U Z2.

fill-out N A N A :- monotype A, (N = emp; N = (sub X Y Z)).
fill-out (newvar N) (all A) (newvar M) (all B) :-
  pi u\ (pi v\ (monotype v => fill-out (N u) (A v) (M u) (B v))).
fill-out (newvar N) A (newvar N) (all B) :- monotype A,
  pi u\ (pi v\ (monotype v => fill-out (N u) A (N u) (B v))).
fill-out N (all A) (newvar M) (all A) :- (N = emp; N = (sub X Y Z)),
  pi u\ (pi v\ (monotype v => fill-out N (A v) (M u) (A v))).

type infer-type tm -> poly -> o.
type remvac poly -> poly -> o.
type appear-in poly -> poly -> o.

remvac (all x\A) B :- remvac A B.
remvac (all A) (all B) :-
  pi a\ (monotype a => (appear-in a (A a), remvac (A a) (B a))).
remvac A A :- monotype A.
appear-in A (all B) :- pi x\ (appear-in A (B x)).
appear-in A A :- monotype A.
appear-in A (arr B C) :- appear-in A B; appear-in A C.

infer-type N T :- polytype nil M N S, remvac S T.
```

```
%=============Some Examples Tested:============

% polytype nil M (abs x\ (app op zero)) T.
% polytype nil M (abs f\ (app f zero)) T.
% polytype nil M (abs f\ (abs x\ (app f x))) T.
% polytype nil M (abs x\ (abs f\ (app f x))) T.
% polytype nil M (abs f\ (abs g\ (abs x\ (app g (app f x))))) T.
% polytype nil M (abs x\ (abs f\ (app (app f x) (app (app f x) x)))) T.
% polytype nil M (abs f\ (abs x\ (abs y\ (app (app f (app op x))
%       (app op y))))) T.
% polytype nil M (abs y\ (let (x\ (abs z\x)) (app succ y))) T.
% polytype nil M (let (x\ (app x x)) (abs y\y)) T.
% polytype nil M (let (x\ (let (z\ (app z (app (app x x) (app z z))))
%       (abs u\u))) (abs y\y)) T.
% polytype nil M (let (f\ (app (app (abs d1\ (abs (d2 \ zero)))
%       (app f zero)) (app f (abs x \ x)))) (abs x \ x)) T.
% polytype nil M (abs x\ (let (y\x) (app x zero))) T.
% polytype nil M (abs x\ (app op (app op x))) T.
% polytype nil M (abs x\ (let (y\ (app (abs c\c) y)) (app (abs c\c) x))) T.
% polytype nil M (abs x\ (let (u\ (let (v\v) (app succ u))) (app op x))) T.
% polytype nil M (let (u\ (let (v\v) (app op u))) (abs x\x)) T.

% Untypable examples:
% polytype nil M (app (abs x\ (app x x)) (abs x\ (app x x))) T.
% polytype nil M (abs x\ (abs y\ (app (app y x) (app succ y)))) T.
% polytype nil M (abs f\ (let (x\ (app f x)) (app op f))) T.
% polytype nil M (abs x\ (let y\ (app y y)) (app op2 x)) T.
% polytype nil M (abs x\ (let (y\ (app x y)) (app op3 x))) T.
% polytype nil M (abs x\ (let (y\ (app y y)) (app op3 x))) T.
% polytype nil M (abs x\ (let (v\ (app v v)) (abs z\ (app x z)))) T.
% polytype nil M (let (u\ (let (v\v) (app succ u))) (abs x\x)) T.
```

The L_λ $\exists\forall$-Unificaton Program, Section 4.4.2

```
module llunify.

import listops lists.


% Object-level Llambda unification the hard way

kind i type.

kind up type.  %unification problem

kind ustate type.

kind subst type.

type emp subst.

type sub i -> i -> subst -> subst.

type eq i -> i -> up.

type newuvar (i -> ustate) -> ustate.

type newevar (i -> ustate) -> ustate.

type ss (list i) -> (list i) -> (list up) -> subst -> ustate.

type uvar i -> o.

type evar i -> o.

type rigid i -> o.

type vrd i -> i -> o.

type lam (i -> i) -> i.

type appl i -> (list i) -> i.

type copyi  i -> i -> o.

type find-flexhead (list i) -> i -> i -> o.

type update-ss ustate -> ustate -> o.

type updatesub subst -> subst -> o.

type copyeq up -> up -> o.

type standard-except i -> o -> o.

type beta_0 i -> i -> o.

type eqbeta_0 up -> up -> o.

type subbeta_0 subst -> subst -> o.

type ssbeta_0 ustate -> ustate -> o.

type find-uvarhead (list i) -> (list i) -> i -> o.

type list-same (list i) -> (list i) -> (list i) -> o.

type list-dif (list i) -> (list i) -> (list i) -> o.
```

```
type delete1 i -> (list i) -> (list i) -> o.
type same-index (list i) -> (list i) -> (list i) -> o.
type pair-up (list i) -> (list i) -> (list up) -> o.
type xi ustate -> ustate -> o.
type r-r ustate -> ustate -> o.
type fr-ff1 ustate -> ustate -> o.
type ff2 ustate -> ustate -> o.
type prune ustate -> ustate -> o.
type distinct-uvar i -> i -> o.
type distinct-evar i -> i -> o.
type non-lam i -> o.
type append-app i -> i -> i -> o.
type form-lam (list i) -> i -> i -> o.
type non-flex i -> o.

non-lam X :- rigid X; (X = (appl A B)); evar X; uvar X.
non-flex A :- uvar A; rigid A.
non-flex (appl A B) :- non-flex A.
non-flex (lam A) :- pi x\ (uvar x => non-flex (A x)).

update-ss (ss E U L R) (ss E U M P) :-  map3 copyeq L M, updatesub R P.
updatesub emp emp.
updatesub (sub X T R) (sub X S P) :-  copyi T S, updatesub R P.
copyeq (eq A B) (eq C D) :- copyi A C, copyi B D.
copyi (appl A L) (appl B M) :- copyi A B, map3 copyi L M.
copyi (lam A) (lam B) :- pi x\ (pi y\ (copyi x y => copyi (A x) (B y))).

standard-except V G :-
  (pi X\ (copyi X X :- uvar X; vrd X V; vrd V X)) => G.

pair-up nil nil nil.
pair-up (A::As) (B::Bs) ((eq A B)::C) :- pair-up As Bs C.
list-dif nil B B.
list-dif (X::Xs) B C :- delete1 X B B2, list-dif Xs B2 C.
delete1 X nil nil.
```

```
delete1 X (X::Xs) Xs.
delete1 X (Y::Xs) (Y::Ys) :- distinct-uvar X Y, delete1 X Xs Ys.
list-same (X::A) B (X::C) :- member X B, list-same A B C.
list-same (X::A) B C :- map2 (distinct-uvar X) B, list-same A B C.
list-same nil B nil.
same-index (X::A) (X::B) (X::C) :- same-index A B C.
same-index (X::A) (Y::B) C :- distinct-uvar X Y, same-index A B C.
same-index nil nil nil.
distinct-uvar X Y :- vrd X Y; vrd Y X.
distinct-evar X Y :- vrd X Y; vrd Y X.
append-app (appl A B) X (appl A C) :- append B (X::nil) C.
append-app A X (appl A (X::nil)) :- rigid A; evar A; uvar A; A = (lam Y).
form-lam nil Body Newbody :- copyi Newbody Body.
form-lam (Y::Ys) Body (lam Nb) :-
  pi v\ (copyi v Y => form-lam Ys Body (Nb v)).

find-flexhead Vs (appl U W) (appl U W) :- evar U, list-dif Vs W (H::T).
find-flexhead Vs (appl U W) R :- (rigid U; uvar U),
  ormap2 (find-flexhead Vs) W R.
find-flexhead Vs (lam A) (lam R) :-
  pi v\ (uvar v =>
   ((pi Z\ (vrd v Z :- member Z Vs)) => find-flexhead Vs (A v) (R v))).

type norm-app i -> i -> o.
beta_0 (lam A) (lam B) :- pi v\ (uvar v => beta_0 (A v) (B v)).
beta_0 A A :- uvar A; evar A; rigid A.
beta_0 (appl (appl A B) C) E :- norm-app (appl (appl A B) C) D, beta_0 D E.
beta_0 (appl (lam A) (Y::Ys)) Z :- beta_0 (appl (A Y) Ys) Z.
beta_0 (appl A nil) B :- (A = (lam C); uvar A; evar A; rigid A), beta_0 A B.
beta_0 (appl U (V::Vs)) (appl U (W::Ws)) :- (uvar U; evar U; rigid U),
  map3 beta_0 (V::Vs) (W::Ws).
norm-app (appl (appl A L) M) Z :-
  append L M N, norm-app (appl A N) Z.
norm-app (appl A L) (appl A L) :- uvar A; evar A; rigid A; A = (lam B).
eqbeta_0 (eq A B) (eq C D) :- beta_0 A C, beta_0 B D.
```

```
subbeta_0 emp emp.
subbeta_0 (sub X S R) (sub X T U) :- beta_0 S T, subbeta_0 R U.
ssbeta_0 (ss E U L R) (ss E U M P) :- map3 eqbeta_0 L M, subbeta_0 R P.
ssbeta_0 (newuvar S) (newuvar T) :- pi v\ (uvar v => ssbeta_0 (S v) (T v)).
ssbeta_0 (newevar S) (newevar T) :- pi x\ (evar x => ssbeta_0 (S x) (T x)).

type prune-aux ustate -> ustate -> o.
type prune-aux2 i -> ustate -> ustate -> o.
prune (ss El Ul P S) (newevar T)  :-
  pi h\ (evar h => ((pi Z\ (vrd h Z) :- member Z El) =>
        prune-aux (ss (h::El) Ul P S) (T h))).
prune-aux (ss El Ul ((eq R (appl V Y))::O) Rho) S2 :- evar V, non-flex R,
  prune-aux (ss El Ul ((eq (appl V Y) R)::O) Rho) S2.
prune-aux (ss El Ul ((eq (appl V Y) R)::O) Rho) SS2 :- evar V,
  find-flexhead Y R F,
  prune-aux2 F (ss El Ul ((eq (appl V Y) R)::O) Rho) SS2.
prune-aux2 (lam U) S T :-
  pi l\ ((copyi l l, (pi Z\ (vrd l Z :- uvar Z))) =>
      prune-aux2 (U l) S T).
prune-aux2 (appl U W) (ss (H::El) Ul ((eq (appl V Y) R)::O) Rho) SS2 :-
  list-same W Y Z,
  (copyi H H => form-lam W (appl H Z) NU),
  (copyi U NU => standard-except U
(update-ss (ss (H::El) Ul ((eq (appl V Y) R)::O) Rho) SS1)),
  ssbeta_0 SS1 SS2.

r-r (ss E U ((eq (appl A nil) (appl A nil))::O) Rho) (ss E U O Rho).
r-r (ss E U ((eq (appl A Ar) (appl A Br))::O) Rho) (ss E U O2 Rho) :-
  pair-up Ar Br PN, append PN O O2.

xi (ss E U ((eq (lam A) (lam B))::O) Rho) (newuvar Z) :-
  pi v\ (uvar v => xi (ss E (v::U) ((eq (A v) (B v))::O) Rho) (Z v)).
xi (ss E U ((eq (lam A) B)::O) Rho) (newuvar Z) :- non-lam B,
  pi v\ (sigma Bv\ (uvar v => (append-app B v Bv,
    xi (ss E (v::U) ((eq (A v) Bv)::O) Rho) (Z v)))).
```

```
xi (ss E U ((eq B (lam A))::O) Rho) (newuvar Z) :- non-lam B,
  pi v\ (sigma Bv\ (uvar v => (append-app B v Bv,
    xi (ss E (v::U) ((eq Bv (A v))::O) Rho) (Z v))))).
xi (ss E U ((eq A B)::O) Rho) (ss E U ((eq A B)::O) Rho) :-
  non-lam A, non-lam B.

fr-ff1 (ss El Ul ((eq (appl V L) (appl R M))::O) Rho) SS2 :-
  ((evar V, (rigid R; uvar R));
   (evar R, evar V, distinct-evar R V)),
  ((pi Z\ (copyi Z Z :- evar Z, distinct-evar Z V))
       => form-lam L (appl R M) Sub),
  (copyi V Sub => (standard-except V (update-ss (ss El Ul O Rho) SS1))),
  ssbeta_0 SS1 SS2.
fr-ff1 (ss El Ul ((eq (appl V Y) (appl U W))::O) Rho) SS2 :-
  evar U, (rigid V; uvar V),
  fr-ff1 (ss El Ul ((eq (appl U W) (appl V Y))::O) Rho) SS2.

type ff2-aux ustate -> ustate -> o.
ff2 (ss El Ul E R) (newevar Sn) :-
  pi h\ (evar h => ((pi Z\ (vrd h Z :- member Z El)) =>
ff2-aux (ss (h::El) Ul E R) (Sn h))).
ff2-aux (ss (H::El) Ul ((eq (appl V Y) (appl V W))::O) Rho) SS2 :-
  evar V, same-index Y W Z,
  (copyi H H => form-lam Y (appl H Z) Sub),
  (copyi V Sub => (standard-except V (update-ss (ss (H::El) Ul O Rho) SS1))),
  ssbeta_0 SS1 SS2.

type lunify ustate -> ustate -> o.
type step1 (list i) -> (list i) -> ustate -> ustate -> o.
type step2 ustate -> ustate -> o.
type step3 ustate -> ustate -> o.
type app-norm i -> i -> o.

lunify (newevar S) (newevar T) :- pi e\ (lunify (S e) (T e)).
lunify (newuvar S) (newuvar T) :- pi u\ (lunify (S u) (T u)).
```

```
lunify (ss E U nil R) (ss E U nil R).
lunify (ss E U (P::Ps) R) T :-
  step1 E U (ss E U (P::Ps) R) T.

step1 (E::Es) U S T:-
 ((pi Z\ (vrd E Z :- member Z Es)) => (evar E => step1 Es U S T)).
step1 nil (U::Us) S T :-
 ((pi Z\ (vrd U Z :- member Z Us)) => (uvar U => step1 nil Us S T)).
step1 nil nil (ss E U ((eq A B)::O) R) T :-
  app-norm A C, app-norm B D, step2 (ss E U ((eq C D)::O) R) T.
step2 (ss E U ((eq A B)::O) R) T :-
  (A = (lam X); B = (lam Y)),
  xi (ss E U ((eq A B)::O) R) S2, lunify S2 T.
step2 (ss E U ((eq (appl A Ar) (appl A Br))::O) R) T :-
  (rigid A; uvar A; (evar A, Ar = nil, Br = nil)),
  r-r (ss E U ((eq (appl A Ar) (appl A Br))::O) R) S2, lunify S2 T.
step2 (ss E U ((eq (appl A Ar) (appl B Br))::O) R) T :-
  (evar A; evar B), non-lam A, non-lam B,
  step3 (ss E U ((eq (appl A Ar) (appl B Br))::O) R) T.
step3 (ss E U ((eq (appl A Ar) (appl B Br))::O) R) T :-
  ((evar A, find-flexhead Ar (appl B Br) F);
   (evar B, find-flexhead Br (appl A Ar) G)),
  prune (ss E U ((eq (appl A Ar) (appl B Br))::O) R) S2, lunify S2 T.
step3 (ss E U P R) T :-
  (fr-ff1 (ss E U P R) S3; ff2 (ss E U P R) S3), lunify S3 T.

app-norm (appl A B) (appl A B).
app-norm (lam A) (lam A).
app-norm A (appl A nil) :- rigid A; evar A; uvar A.
```

The Higher Order Critical Pairs Program, Section 4.6

The code includes the example of Local Confluence for $\beta\eta$-reduction given in Nipkow's paper:

```
module hor.
```

```
import llunify lists.

% Example from T. Nipkow's paper.

type abO i. type apO i.
kind rule type.
type rw i -> i -> rule.  %only primitive type rules are allowed%
type scheme (i -> rule) -> rule.
kind name type.
type rr name -> rule -> o.
type beta name. type eta name.

% appl and lam are meta-level application and abstraction.
% apO and abO are object-level application and abstraction.

copyi abO abO.
copyi apO apO.
rigid abO.
rigid apO.

rr beta
  (scheme F\ (scheme S\ (rw
    (appl apO ((appl abO (F::nil))::S::nil))  (appl F (S::nil))))).
rr eta
  (scheme S\ (rw (appl abO ((lam x\ (appl apO (S::x::nil)))::nil)) S)).

type subterm i -> i -> (i -> i) -> o.
type subterm-list i -> (list i) -> (i -> (list i)) -> o.
type absorb i -> i -> (i -> i) -> o.

subterm X X (c\c) :- X = (appl A B).
subterm A (appl B C) (c\ (appl (D c) C)) :- subterm A B D.
subterm A (appl B C) (c\ (appl B (D c))) :- subterm-list A C D.
subterm (lam A) (lam B) (c\ (lam x\(D x c))) :-
  pi x\ (subterm (A x) (B x) (D x)).
```

```
subterm-list A (B::L) (x\ ((C x)::L)) :- subterm A B C.
subterm-list A (B::L) (x\ (B::(M x))) :- subterm-list A L M.

absorb S T C :- (pi X\ (subterm X X (c\c))) => subterm S T C.

type cp name -> name -> rule -> o.
type find-cp (list i) -> rule -> rule -> rule -> o.
type cp-aux (list i) -> (list i) -> i -> (i -> i) -> i -> i -> i -> rule -> o.
type form-cp rule -> ustate -> rule -> o.
type form-trivials (list i) -> subst -> o.

cp R1 R2 Cp :- rr R1 Rule1, rr R2 Rule2, find-cp nil Rule1 Rule2 Cp.
find-cp Vs (scheme Rule1) Rule2 (scheme Cp) :-
  pi v\ (evar v => find-cp (v::Vs) (Rule1 v) Rule2 (Cp v)).
find-cp Vs (rw L1 R1) (scheme Rule2) (scheme Cp) :-
  pi v\ (evar v => find-cp (v::Vs) (rw L1 R1) (Rule2 v) (Cp v)).
find-cp Vs (rw L1 R1) (rw L2 R2) Cp :-
  subterm S L1 CL1,
%nl, writesans "subterm found in L1: ", write S,
  cp-aux Vs nil S CL1 R1 L2 R2 Cp.

cp-aux Fv Bv (lam S) CL1 R1 L2 R2 Cp :-
  pi b\ (append Bv (b::nil) (Br b),
         cp-aux Fv (Br b) (S b) CL1 R1 L2 R2 Cp).
cp-aux Fv Bv S CL1 R1 L2 R2 Cp :- non-lam S,
  ((pi Z\ (copyi Z (appl Z Bv) :- evar Z)) =>
           (copyi L2 L2b, copyi R2 R2b)),
%nl, writesans "raised rule2: ", write (rw L2b R2b),
  form-trivials Fv TFv,
  lunify (ss Fv Bv ((eq S L2b)::nil) TFv) U,
%nl, writesans "unifier of subterm and L2: ", write U,
  ((pi Z\ (copyi Z Z :- evar Z)) => form-lam Bv R2b R2c),
  subterm R2c Cpr CL1,
%nl, writesans "pair before substitution:", write (rw R1 Cpr),
  form-cp (rw R1 Cpr) U Cp.
```

```
form-cp A (newevar U) (scheme Cp) :-
  pi s\ (evar s => form-cp A (U s) (Cp s)).
form-cp A (newuvar U) Cp :- pi x\ (form-cp A (U x) Cp).
form-cp A (ss Es Us nil (sub X Y U)) C :-
  copyi X Y => form-cp A (ss Es Us nil U) C.
form-cp (rw A B) (ss Es Us nil emp) (rw C D) :-
  copyi A A2, copyi B B2, beta_0 A2 C, beta_0 B2 D.

form-trivials nil emp.
form-trivials (X::Xs) (sub X X Ys) :- form-trivials Xs Ys.
```