

A Family of Program Derivations for Higher-Order Unification¹

Conal Elliott and Frank Pfenning

November 1987

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We derive a family of algorithms around the theme of higher-order unification. The main derivation starts with a concise, abstract specification and ends with a version of Huet's algorithm for higher-order unification. Next we derive some very useful high-level optimizations. Finally, we describe how to extend the algorithm to give a complete treatment of products and an incomplete but very useful treatment of polymorphism, yielding a new and significantly more powerful algorithm (which we have implemented).

¹This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

Contents

1	Introduction	2
1.1	Motivation	2
1.2	Derivational Methodology	2
1.3	The Significance of Higher-Order Unification	3
1.4	Related Work	4
2	Preliminaries	4
2.1	The Traditional Approach	4
2.2	Our Approach	5
3	Formal Specifications	6
3.1	The Unification problems	6
4	Conversion Rules	7
5	Some Useful Properties of Convertibility	7
6	The Main derivation	8
6.1	Abstractions	8
6.2	λ -free Head Normal Terms	9
6.3	The Rigid-rigid Case	9
6.4	Disagreement Sets	10
6.5	Rigid-rigid Case Continued	11
6.6	The Rigid-flexible Case	11
6.7	The Flexible-flexible Case	12
7	The Other Specifications	12
7.1	Unifiability	12
7.2	Minimal Complete Set of Unifiers	13
7.3	Matching	14
8	Improvements to the Algorithms	14
8.1	Matching Trees	14
8.2	Matching Dags	15
8.3	Matching Graphs	16

1. Introduction	2
9 Extensions to the Algorithm	16
9.1 Products	16
9.2 Conversion Rules	19
9.3 Normal Forms	19
9.4 Algorithm Modifications	20
10 Polymorphism	20

1 Introduction

1.1 Motivation

There were two different motivations for the work reported in this paper.

Firstly, we recognized the importance and utility of higher-order abstract syntax (generalizing Huet and Lang’s idea in [12]) as the central, language independent data structure for the representation of programs and other syntactic objects in a program development system [23]. Application of a transformation or inference rule then requires higher-order matching and substitution instead of the usual first-order matching and substitution. Besides program transformation systems, higher-order unification has applications in logic programming [18], natural deduction [22], and automated theorem proving [1,13].

Secondly, we wanted to synthesize the algorithm from an abstract and concise specification. We hoped this would provide a deeper understanding of the algorithm and facilitate correctness-preserving improvements and extensions. We believe our results justify our approach.

After setting the stage with a brief introduction to our theory of unification, we give a specification for the set of unifiers of two terms in the typed lambda calculus. From this specification we derive a set of recursive equations describing the set of unifiers. These equations are then specialized in the context of several different specifications, yielding related, but different algorithms. The specifications include unifiability, finding complete sets of unifiers, and higher-order matching. Our algorithms are close to Huet’s [11], but we derive improvements that lower its complexity for many commonly occurring second- and third-order matching problems from exponential to quadratic or even linear time.

It seems difficult to view the derivations as a sequence of *program* transformations, since none of the intermediate stages has an algorithmic interpretation. This led us to view the derivation as proving facts about elements of a semantic domain, arriving at a set of statements that can be recognized as describing the *meaning* of a concrete functional program. This approach can be used to reinterpret the well-known program transformation method of Burstall and Darlington [3], providing new justification and generalization of their work.

1.2 Derivational Methodology

By the “derivational” approach to programming, we mean a methodology that captures the decisions and inferences made in constructing programs from specifications. Program derivation differs from program verification in that, in a derivation, the reasoning is explicitly part of the synthesis of the program, while in verification, the reasoning comes after the program is constructed.

Besides the benefit of ensuring correctness, program derivation is valuable as a way of recording and communicating the conceptual development of a program (this is its “derivation”). This leads to greater understanding of why a program works, and thus facilitates improvement of the program (in terms of efficiency and/or functionality) and modification of the program to meet altered specifications. These considerations are especially important with complex algorithms, such as higher-order unification.

In the program transformation methodology [3,20,27], one begins with an initial program, designed for clarity rather than efficiency, and transforms it, step by step, into a more efficient program, in which relationships between program parts are made explicit and exploited. See [3] for several good examples of this.

A related methodology is programming by proving theorems [2,8,14]. Here, one starts out with a statement of the form “ $\forall x \exists y R(x, y)$ ”, and constructs a proof in a suitable *constructive* logic. Because of the constructiveness of the logic, it is possible to *extract* from the proof a functional program that, given an x , computes a y such that $R(x, y)$. Similar in spirit is the approach of “propositions as types” [17,4], in which very rich type systems are used, and the problem of constructing a program to meet a given specification is viewed as finding a term of a given type.

Our methodology does not fall into either of these categories, but contains elements of both. We begin with an abstract specification, *i.e.* a description of a mathematical object. We want to construct a program whose *meaning* satisfies the specification. Sometimes this is too difficult or even impossible (because of undecidability), and we settle for a program whose meaning *approximates* (is perhaps less defined than) an object satisfying the specification.

To be more concrete, suppose we have a specification $\phi(f)$ of a function f in our semantic domain. Our derivations consist of assuming $\phi(f)$ and performing some deductions whose conclusion is of a special form, saying that f is (or approximates) the meaning of some program P . We find the method of denotational semantics (see *e.g.* [28]) used in the context of higher-order abstract syntax [23] most suitable for defining meanings of programs and making the transition from meanings to programs in this context. In this paper, we leave out this final step, as it obscures rather than clarifies our results.

In the higher-order unification derivations, we begin with axiomatic descriptions of the problems and the α , β and η conversion. We first “generalize” our function f by finding an f' and H such that $f = Hf'$. Then, by mainly equational reasoning, we then deduce an equation of the form $f' = Gf'$, for some G . It then follows that f is approximated by $H(YG)$ (where YG is the least fixed point of G). For these G , it is then easy to construct a functional program whose meaning is $H(YG)$.

The feel of the derivations is similar to the “unfold/fold” derivations in [3]. In fact, the unfold/fold approach can be looked at as transforming equational specifications, rather than as transforming programs. The reason that this system of transformation rules may lose termination can then be explained as in the previous paragraph.

1.3 The Significance of Higher-Order Unification

One important paradigm in traditional program transformation systems is first-order matching and substitution. The left-hand side of a rule with pattern variables is matched against a program or program fragment. The resulting substitution is then applied to the right-hand side. Complications arise when the language under consideration contains binding constructs. Restrictions like not-free-in and assumptions about renaming of bound variables are commonplace. A new approach was proposed by Huet and Lang [12] in which the usual trees (first-order terms) representing the abstract syntax of programs were replaced by higher-order terms in a typed lambda calculus. This framework allows much more concise

program transformation rules with easier correctness proofs. They used second-order matching (which is decidable) and substitution to apply program transformations. Although second-order matching can be very prolific, practical experience has underscored its utility and practicality.

We found that many natural transformation rules actually required patterns with third-order variables, thus necessitating higher-order matching. Plotkin has recently shown that third- and higher-order matching is decidable, no one has proposed a practical algorithm. In the ERGO project, we expect that higher-order abstract syntax (in the sense of [12]) will be at the heart of our program transformation system, and that higher-order matching and substitution will be the main engine carrying out program and specification transformations.

A choice similar to higher-order abstract syntax was made by Paulson [22] to implement logical deduction rather than program transformation. Higher-order unification has also been successfully applied in automated theorem proving [1] and logic programming [18].

1.4 Related Work

First-order unification algorithms have often been presented in a way that may be characterized as “informal program developments”. Some of the steps could be expressed as correctness-preserving program transformations, others are obtained “by analogy”. Examples are in [16,15,21,5].

There have also been formal syntheses of first-order unification algorithms. Manna and Waldinger [14] develop a theory of substitutions in a constructive first-order logic to prove a statement that most general unifiers exist. From the proof they extract a program in a functional language. Eriksson [7] synthesizes an algorithm in a first-order natural deduction calculus.

There is little similarity between our derivation and the first-order syntheses. This is partly due to the non-deterministic nature of the higher-order unification algorithm. Other reasons are our different methodology, and the great simplification achieved by the integration of terms and substitutions.

Related in a different way is recent work by Gallier and Snyder [9] who present an algorithm for equational unification as a set of rules for transforming sets of equations, and give a brief exposition of how their method may apply to higher-order unification. In comparison, we derive facts relating solutions of unification problems to solutions of other unification problems. We could very naturally describe our algorithms as sets of rules justified by these facts.

2 Preliminaries

Our presentation of the unification problem is somewhat unusual. We first briefly present the traditional approach, and then ours.

2.1 The Traditional Approach

In a conventional presentation, *e.g.* [26,11], an instance of the unification problem is given by two terms containing free variables. A solution, or “unifier”, is in the form of a substitution, assigning terms to the

free variables to make the two terms equivalent (in some sense). For instance, take the two terms¹

$$h(Fx) \quad \text{and} \quad F(hy),$$

where F and h have type $\alpha \rightarrow \alpha$ and x and y have type α , for some base type α .

There are infinitely many unifiers of the form

$$\{ h \leftarrow \lambda z . F^n z, x \leftarrow y \}$$

for each $n \geq 0$. There are other solutions, but this collection forms a *complete set of unifiers* in that any other unifier is a *specialization* of one of these. One makes this precise by introducing a notion of composition of substitutions and, from that, a (preorder) relation between substitutions [12,11].

2.2 Our Approach

The main difference in our approach is that we explicitly abstract over what are usually considered to be free variables. This applies to terms being unified and to substitutions. We can then use a natural notion of composition of abstractions, \circ , defined in terms of β -reduction, to carry out the substitutions. For instance, the terms in the example above would be presented as

$$\lambda hxy . h(Fx) \quad \text{and} \quad \lambda hxy . F(hy).$$

Since these abstractions have three arguments, the applicable substitutions will be a triple of abstractions. For convenience, we will often write an abstraction over a sequence to mean a sequence of abstractions, *e.g.*

$$\lambda x . \langle a, b, c \rangle \quad \text{means} \quad \langle \lambda x . a, \lambda x . b, \lambda x . c \rangle$$

Unifiers for the example problem analogous to the ones shown above are

$$\lambda y . \langle \lambda z . F^n z, y, y \rangle.$$

This integration of terms with substitutions yields great simplification.

Composition of two substitutions is the natural extension, which we will also write as \circ . The specialization preordering is then

$$s' \leq s \quad \Leftrightarrow \quad \exists \hat{s} . s' \stackrel{\lambda}{=} s \circ \hat{s}$$

We are working on extending the derivation to handle products with “pattern” binding as in ML [10] (see section 9.1). This is almost complete and will yield a full integration of terms with substitutions, making composition be exactly the standard notion.

¹We use fonts to distinguish between *variables* and Constants.

3 Formal Specifications

3.1 The Unification problems

In this section, we formally specify the problems, relative to the notion of λ -convertibility, which is defined in section 4.

First, a notion of “substitution type” (the set of possible substitutions for trying to unify abstractions of type $\alpha_0 \rightarrow \dots \rightarrow \alpha_m \rightarrow \beta$ for some type β):

$$\Sigma^{\langle \alpha_1, \dots, \alpha_m \rangle} \triangleq \{ \lambda x_1, \dots, x_r . \langle Z_1, \dots, Z_m \rangle \mid Z_i : \alpha_i \ 1 \leq i \leq m \}$$

We first define the set of unifiers of two abstractions f and g (of type $\alpha_1 \rightarrow \dots \rightarrow \alpha_m \rightarrow \alpha_0$, where $\bar{\alpha} = \langle \alpha_1, \dots, \alpha_m \rangle$):

$$\mathcal{U}_{\bar{\alpha}}(f, g) = \{ s \in \Sigma^{\bar{\alpha}} \mid f \circ s \stackrel{\lambda}{=} g \circ s \}$$

This simple specification is mainly for organizational use. We would not want to implement it as is because along with the unifiers we want, we would also get all of their (infinitely many) instances. Rather, we want a subset that can generate (by taking instances) all the unifiers. Thus, the property of being a *complete* subset of a set of substitutions.

$$\text{complete}_{\bar{\alpha}}(\hat{S}, S) \Leftrightarrow S = \{ s \in \Sigma^{\bar{\alpha}} \mid \exists \hat{s} \in \hat{S} . s \leq \hat{s} \}$$

This says that S consists of all the instances of elements of \hat{S} . (Not all S have complete subsets, only those closed under specialization. This will be true in our use of *complete*.)

Finally, the property of being a *minimal* set of substitutions (no substitution is an instance of any other substitution):

$$\text{minimal}(\hat{S}) \triangleq \forall s, s' \in \hat{S} . s \leq s' \Rightarrow s = s'$$

Our family of unification problems is described as follows:

$$\begin{aligned} \mathcal{U}_{\bar{\alpha}}(f, g) &= \{ s \in \Sigma^{\bar{\alpha}} \mid f \circ s \stackrel{\lambda}{=} g \circ s \} \\ \text{unifiable}_{\bar{\alpha}}(f, g) &\Leftrightarrow \mathcal{U}_{\bar{\alpha}}(f, g) \neq \{ \} \\ \text{MCSU}_{\bar{\alpha}}(f, g, \hat{S}) &\triangleq \text{complete}_{\bar{\alpha}}(\hat{S}, \mathcal{U}_{\bar{\alpha}}(f, g)) \wedge \text{minimal}(\hat{S}) \end{aligned}$$

The analogous matching problems are most simply specified as restrictions of the unification problems to the case that one of the terms, say g , is a constant function.

4 Conversion Rules

The most important conversion rule is the β rule, for dealing with terms of the form $(\lambda z . M)Z$. Usually this is expressed in terms of performing a substitution of Z for z in M (under the right conditions or performing renaming to avoid capturing). However, it is more convenient for our purposes to specify it incrementally and separately for each kind of term M .

$$\begin{aligned}
 (\lambda z . v)Z &= Z \\
 (\lambda z . u)Z &= u && \text{if } z \neq u \\
 (\lambda z . c)Z &= c \\
 (\lambda z . MN)Z &= ((\lambda z . M)Z)((\lambda z . N)Z) \\
 (\lambda z . \lambda v . M)Z &= \lambda v . M \\
 (\lambda z . \lambda u . M)Z &= \lambda u . (\lambda z . M)Z && \text{if } z \neq u \wedge u \text{ not free in } Z
 \end{aligned}$$

Next, η -conversion:

$$\lambda x . Mx = M \quad \text{if } x \text{ not free in } M$$

Finally, α -conversion is just an η -conversion followed by β -conversion:

$$\lambda y . M = \lambda x . (\lambda y . M)x \quad \text{if } x \text{ not free in } \lambda y . M$$

5 Some Useful Properties of Convertibility

We make vital use of some facts about convertibility and normal forms. Here we list them with some comments and refer to them as they are used in the derivation. The first involves convertibility of abstractions:

$$\lambda x . M \underset{\lambda}{=} \lambda x . M' \Leftrightarrow M \underset{\lambda}{=} M'$$

Next, let a, a' be “atoms”, *i.e.* variables or constants. Then for any n, n', h_1, \dots, h_n , and h'_1, \dots, h'_n ,

$$\begin{aligned}
 a h_1 \cdots h_n &\underset{\lambda}{=} a' h'_1 \cdots h'_n \\
 \Leftrightarrow a = a' \wedge n = n' \wedge \bigwedge_{1 \leq j \leq n} h_j &\underset{\lambda}{=} h'_j
 \end{aligned}$$

Terms in this form are called “ λ -free head normal”.

Finally, every term is uniquely convertible to one in “long η head normal form”:

$$\lambda z_1 . \cdots \lambda z_n . a h_1 \cdots h_n$$

where the z_i are distinct, $a h_1 \cdots h_n$ is head normal, and not of function type (because otherwise it could be η -expanded.) For convenience, we will often write these terms as $\lambda z_1, \dots, z_m . a(h_1, \dots, h_n)$ or even $\lambda \bar{z} . a(\bar{h})$.

We will often write a substitution $\lambda x_1, \dots, x_r . \langle Z_1, \dots, Z_m \rangle$ as $\lambda \bar{x} . \bar{Z}$. Similarly, we will write a type $\alpha_1 \rightarrow \cdots \rightarrow \alpha_m \rightarrow \alpha_0$ as $\alpha_1, \dots, \alpha_m \rightarrow \alpha_0$ or even $\bar{\alpha} \rightarrow \alpha_0$ where α_0 is not a function type.

6 The Main derivation

Our approach in deriving the higher-order unification algorithms is to expand the problem specification in the context of a few special kinds of terms, proving a collection of useful facts relating the set of unifiers of one problem to the sets of unifiers of other problems. Then, to extend these results to all possible terms, we use two facts, (a) all terms can be λ -converted to one in long η head normal form, and (b) λ -conversion leaves the set of unifiers unchanged.

Because of space limitations, we give only enough details of the derivations to give a feel for what is involved. More details may be found in [6].

Algorithms that manipulate λ -calculus expressions have one point in common: distracting details involving renaming bound variables to avoid capture. All of these naming details can be derived with certainty using the specification of β -conversion. However, for simplicity of this exposition, we will gloss over this issue, giving informal notes to assist a would-be implementor.

6.1 Abstractions

Consider the unification problem $f \circ s \stackrel{\lambda}{=} g \circ s$ when the bodies of f and g are abstractions (the “body” is what follows the binding of the variables being substituted for):

$$\begin{aligned}
 & (\lambda \bar{z} . \lambda y . M) \circ (\lambda \bar{x} . \bar{Z}) \stackrel{\lambda}{=} (\lambda \bar{z} . \lambda y . M') \circ (\lambda \bar{x} . \bar{Z}) \\
 \Leftrightarrow & \lambda \bar{x} . (\lambda \bar{z} . \lambda y . M) \bar{Z} \stackrel{\lambda}{=} \lambda \bar{x} . (\lambda \bar{z} . \lambda y . M') \bar{Z} \\
 \Leftrightarrow & \lambda \bar{x} . \lambda y . (\lambda \bar{z} . M) \bar{Z} \stackrel{\lambda}{=} \lambda \bar{x} . \lambda y . (\lambda \bar{z} . M') \bar{Z} \\
 \Leftrightarrow & \lambda y . (\lambda \bar{z} . M) \bar{Z} \stackrel{\lambda}{=} \lambda y . (\lambda \bar{z} . M') \bar{Z} \\
 \Leftrightarrow & (\lambda \bar{z} . M) \bar{Z} \stackrel{\lambda}{=} (\lambda \bar{z} . M') \bar{Z} \\
 \Leftrightarrow & \lambda \bar{x} . (\lambda \bar{z} . M) \bar{Z} \stackrel{\lambda}{=} \lambda \bar{x} . (\lambda \bar{z} . M') \bar{Z} \\
 \Leftrightarrow & (\lambda \bar{z} . M) \circ (\lambda \bar{x} . \bar{Z}) \stackrel{\lambda}{=} (\lambda \bar{z} . M') \circ (\lambda \bar{x} . \bar{Z})
 \end{aligned}$$

Notes: Being careful with variables names, we would require that y not be among the \bar{z} or \bar{x} . Also, we might need to perform α -conversion to make this fact applicable, *i.e.* to get the same bound variable, y , on both sides. In an efficient implementation we would probably want to do this conversion lazily.

This is typical of the reductions we use. The algorithmic interpretation is

$\lambda \bar{z} . \lambda y' . M''$, α -convert the second one to $\lambda \bar{z} . \lambda y . M'$ and unify the (smaller) terms $\lambda \bar{z} . M$ and $\lambda \bar{z} . M'$.

6.2 λ -free Head Normal Terms

Now consider $f \circ s$ when the body of f is λ -free head normal:

$$\begin{aligned} (\lambda \bar{z} . a(h_1, \dots, h_n)) \circ (\lambda \bar{x} . \bar{Z}) &=_{\lambda} \lambda \bar{x} . (\lambda \bar{z} . a(h_1, \dots, h_n)) \bar{Z} \\ &=_{\lambda} \lambda \bar{x} . ((\lambda \bar{z} . a) \bar{Z})(g_1, \dots, g_n) \end{aligned}$$

where $g_j = (\lambda \bar{z} . h_j) \bar{Z}$. There are two important cases here:

- If $a = \bar{z}_{[i]}$ (the i th variable in \bar{z}) for $1 \leq i \leq m$. Then the term is called *flexible* and $(\lambda \bar{z} . a) \bar{Z} =_{\lambda} \bar{Z}_{[i]}$. In this case,

$$(\lambda \bar{z} . a(h_1, \dots, h_n)) \bar{Z} =_{\lambda} \bar{Z}_{[i]}(g_1, \dots, g_n)$$

so we have to perform more β -reductions to get λ -free head normal form.

- Otherwise, the term is called *rigid* and $(\lambda \bar{z} . a) \bar{Z} =_{\lambda} a$. In this case,

$$(\lambda \bar{z} . a(h_1, \dots, h_n)) \bar{Z} =_{\lambda} a(g_1, \dots, g_n)$$

which is in λ -free head normal form.

These two cases give rise to four main cases in the algorithm, called *rigid-rigid*, *rigid-flexible*, *flexible-rigid*, and *flexible-flexible*. These are examined in the next few sections.

6.3 The Rigid-rigid Case

Consider the case of unifying two rigid terms. Take a and a' not in \bar{z} . Then by the previous section,

$$\begin{aligned} (\lambda \bar{z} . a(h_1, \dots, h_n)) \circ (\lambda \bar{x} . \bar{Z}) &=_{\lambda} (\lambda \bar{z} . a'(h'_1, \dots, h'_{n'})) \circ (\lambda \bar{x} . \bar{Z}) \\ \Leftrightarrow \lambda \bar{x} . a(g_1, \dots, g_n) &=_{\lambda} \lambda \bar{x} . a'(g'_1, \dots, g'_{n'}) \\ \Leftrightarrow a(g_1, \dots, g_n) &=_{\lambda} a'(g'_1, \dots, g'_{n'}) \end{aligned}$$

where $g_j = (\lambda \bar{z} . h_j) \bar{Z}$ and $g'_j = (\lambda \bar{z} . h'_j) \bar{Z}$. Since these are in λ -free head normal form we have

$$\Leftrightarrow a = a' \wedge n = n' \wedge \bigwedge_{1 \leq j \leq n} (\lambda \bar{z} . h_j) \bar{Z} =_{\lambda} (\lambda \bar{z} . h'_j) \bar{Z}$$

Then putting back the $\lambda \bar{x}$. we have

$$\begin{aligned} \Leftrightarrow a = a' \wedge n = n' \wedge \bigwedge_{1 \leq j \leq n} \lambda \bar{x} . (\lambda \bar{z} . h_j) \bar{Z} &=_{\lambda} \lambda \bar{x} . (\lambda \bar{z} . h'_j) \bar{Z} \\ \Leftrightarrow a = a' \wedge n = n' \wedge \bigwedge_{1 \leq j \leq n} (\lambda \bar{z} . h_j) \circ (\lambda \bar{x} . \bar{Z}) &=_{\lambda} (\lambda \bar{z} . h'_j) \circ (\lambda \bar{x} . \bar{Z}) \end{aligned}$$

or, replacing $\lambda \bar{z} . \bar{Z}$ by s ,

$$\begin{aligned} & (\lambda \bar{z} . a(h_1, \dots, h_n)) \circ s = (\lambda \bar{z} . a'(h'_1, \dots, h'_{n'})) \circ s \\ \Leftrightarrow & a = a' \wedge n = n' \wedge \bigwedge_{1 \leq j \leq n} (\lambda \bar{z} . h_j) \circ s = (\lambda \bar{z} . h'_j) \circ s \end{aligned}$$

In fact, the condition $n = n'$ can be eliminated because the types of a and a' determine the values of n and n' .

In summary, given two rigid terms, we can show that either they are nonunifiable (if they have different heads) or that the set of unifiers is equal to the set of substitutions that simultaneously unify a finite collection of pairs of terms.

6.4 Disagreement Sets

The previous section suggests that in order to obtain a recursive formulation of the problem, we will need to generalize the specification of unification to handle “disagreement sets”, *i.e.* collections of pairs of terms to be simultaneously unified. The generalized specification is

$$\mathcal{U}'_{\bar{\alpha}}(D) \triangleq \{ s \in \Sigma^{\bar{\alpha}} \mid \bigwedge_{\langle f, g \rangle \in D} f \circ s =_{\lambda} g \circ s \}$$

We can see our problem to be a specialization:

$$\mathcal{U}'_{\bar{\alpha}}(f, g) = \mathcal{U}'_{\bar{\alpha}}(\{ \langle f, g \rangle \})$$

If D is empty, the specification simplifies to

$$\mathcal{U}'_{\bar{\alpha}}(D) = \mathcal{U}'_{\bar{\alpha}}(\{ \}) = \Sigma^{\bar{\alpha}}$$

Note: it is easy to come up with a minimal complete set of unifiers, namely the singleton $\{ I_{\bar{\alpha}} \}$, where $I_{\bar{\alpha}}$ is the identity substitution in $\Sigma^{\bar{\alpha}}$, *i.e.*

$$\lambda z_1, \dots, z_m . \langle z_1, \dots, z_m \rangle$$

When D is nonempty, the general strategy is to focus one unification problem and use it to transform the whole problem to another one of hopefully simpler structure. Thus we try to fill in the following:

$$\langle f_0, g_0 \rangle \in D \Rightarrow \mathcal{U}'_{\bar{\alpha}}(D) = \dots$$

6.5 Rigid-rigid Case Continued

First, let D' be such that²

$$D' \cup \{ \langle \lambda \bar{z} . a(\bar{h}), \lambda \bar{z} . a'(\bar{h}') \rangle \} = D.$$

Let $D'' = \{ \langle \lambda \bar{z} . h_1, \lambda \bar{z} . h'_1 \rangle, \dots, \langle \lambda \bar{z} . h_n, \lambda \bar{z} . h'_n \rangle \}$. Then we have

$$\begin{aligned} \mathcal{U}'_{\bar{\alpha}}(D) &= \mathcal{U}'_{\bar{\alpha}}(\langle \lambda \bar{z} . a(\bar{h}), \lambda \bar{z} . a'(\bar{h}') \rangle) \\ &= \{ s \in \Sigma^{\bar{\alpha}} \mid (\lambda \bar{z} . a(\bar{h})) \circ s \stackrel{\lambda}{=} (\lambda \bar{z} . a'(\bar{h}')) \circ s \wedge \bigwedge_{(f,g) \in D'} f \circ s \stackrel{\lambda}{=} g \circ s \} \\ &= \{ s \in \Sigma^{\bar{\alpha}} \mid a = a' \wedge \bigwedge_{(f,g) \in D''} f \circ s \stackrel{\lambda}{=} g \circ s \wedge \bigwedge_{(f,g) \in D'} f \circ s \stackrel{\lambda}{=} g \circ s \} \\ &= \{ s \in \Sigma^{\bar{\alpha}} \mid a = a' \wedge \bigwedge_{(f,g) \in D'' \cup D'} f \circ s \stackrel{\lambda}{=} g \circ s \} \\ &= \begin{cases} \mathcal{U}'_{\bar{\alpha}}(D'' \cup D') & \text{if } a = a' \\ \{ \} & \text{otherwise} \end{cases} \end{aligned}$$

6.6 The Rigid-flexible Case

In the previous case, we replaced the chosen disagreement pair with some number of other disagreement pairs to get a new disagreement set with exactly the same set of unifiers. Our strategy for this case is different. Here we deduce a useful constraint on the possible unifiers of the chosen disagreement pair and hence on the unifiers of the whole disagreement set. We then use this constraint to generate a finite collection of new disagreement sets and show how to use the unifiers of the new disagreement sets to construct the unifiers of the old disagreement set.

The constraint takes the form that any unifier must be an instance of one a finite set \tilde{S} of very simple substitutions. That is,

$$f_0 \circ s \stackrel{\lambda}{=} g_0 \circ s \Rightarrow \exists \tilde{s} \in \tilde{S} . \exists s' \in \Sigma^{\bar{\alpha}_i} . s \stackrel{\lambda}{=} \tilde{s} \circ s'$$

These are the “imitations and projections” described in [11].

Now, since we have shown this constraint to follow from the conjunction, we may add it to the conjunction:

$$\begin{aligned} \mathcal{U}'_{\bar{\alpha}}(D) &= \{ s \in \Sigma^{\bar{\alpha}} \mid \bigwedge_{(f,g) \in D} f \circ s \stackrel{\lambda}{=} g \circ s \} \\ &= \{ s \in \Sigma^{\bar{\alpha}} \mid \bigwedge_{(f,g) \in D} f \circ s \stackrel{\lambda}{=} g \circ s \wedge (\exists \tilde{s} \in \tilde{S} . \exists s' \in \Sigma^{\bar{\alpha}_i} . s \stackrel{\lambda}{=} \tilde{s} \circ s') \} \end{aligned}$$

²Note that this D' might or might not contain $\langle \lambda \bar{z} . a(\bar{h}), \lambda \bar{z} . a'(\bar{h}') \rangle$. This gives us freedom in our eventual implementation of disagreement-sets. For instance, they could really be sets, and we would use set subtraction to get D' , or they could be multisets, and we would only remove one copy of $\langle \lambda \bar{z} . a(\bar{h}), \lambda \bar{z} . a'(\bar{h}') \rangle$.

Widening the scope of the quantifiers on \bar{s} and s' , and replacing s by $\bar{s} \circ s'$,

$$= \{ s \in \Sigma^{\bar{\alpha}} \mid \exists \bar{s} \in \bar{S} . \exists s' . s \equiv \bar{s} \circ s' \wedge \bigwedge_{\langle f, g \rangle \in D} f \circ (\bar{s} \circ s') =_{\lambda} g \circ (\bar{s} \circ s') \}$$

Now using associativity of composition

$$\begin{aligned} &= \{ s \in \Sigma^{\bar{\alpha}} \mid \exists \bar{s} \in \bar{S} . \exists s' . s \equiv \bar{s} \circ s' \wedge \bigwedge_{\langle f, g \rangle \in D} (f \circ \bar{s}) \circ s' =_{\lambda} (g \circ \bar{s}) \circ s' \} \\ &= \{ s \in \Sigma^{\bar{\alpha}} \mid \exists \bar{s} \in \bar{S} . \exists s' . s \equiv \bar{s} \circ s' \wedge \bigwedge_{\langle f', g' \rangle \in D \circ \bar{s}} f' \circ s' =_{\lambda} g' \circ s' \} \\ &= \bigcup_{\bar{s} \in \bar{S}} \{ s \in \Sigma^{\bar{\alpha}} \mid \exists s' . s \equiv \bar{s} \circ s' \wedge \bigwedge_{\langle f', g' \rangle \in D \circ \bar{s}} f' \circ s' =_{\lambda} g' \circ s' \} \\ &= \bigcup_{\bar{s} \in \bar{S}} (\bar{s} \circ \mathcal{U}'_{\bar{\alpha}\bar{s}}(D \circ \bar{s})) \end{aligned}$$

where the notation $\bar{s} \circ S$ means the set of all $\bar{s} \circ s$ for $s \in S$, and $D \circ \bar{s}$ means the set of all $\langle f \circ \bar{s}, g \circ \bar{s} \rangle$ for $\langle f, g \rangle \in D$.

In summary, we used the chosen disagreement pair to derive a constraint on possible unifiers. Rephrasing this constraint in terms of composition, we constructed a finite collection of new disagreement-sets that are partial instantiations of the old one, and whose unifiers lead, by composition, to unifiers of the old unification problem.

For the flexible-rigid case, we exploit the symmetry of unification, and handle this case by reflecting the problem and appealing to the rigid-flexible case as follows:

$$\mathcal{U}'_{\bar{\alpha}}(D' \cup \{ \langle f_0, g_0 \rangle \}) = \mathcal{U}'_{\bar{\alpha}}(D' \cup \{ \langle g_0, f_0 \rangle \})$$

6.7 The Flexible-flexible Case

The flexible-flexible case, which is the easiest case in first-order unification, turns out to be the most difficult case to give a complete treatment of in higher-order context. We treat it differently in the various other specifications, as described in the next section.

7 The Other Specifications

In this section, we briefly describe how to specialize the results of previous few sections about the function \mathcal{U} to implement the other specifications.

7.1 Unifiability

Recall the definition of unifiability:

$$\text{unifiable}_{\bar{\alpha}}(f, g) \Leftrightarrow \mathcal{U}_{\bar{\alpha}}(f, g) \neq \{ \}$$

To give an indication of how to use the previous results in this context, here is the rigid-flexible case:

$$\begin{aligned}
\text{unifiable}'_{\bar{\sigma}}(D) &\Leftrightarrow \mathcal{U}'_{\bar{\sigma}}(D) \neq \{ \} \\
&\Leftrightarrow \left(\bigcup_{\bar{s} \in \bar{\mathcal{S}}} \bar{s} \circ \mathcal{U}'_{\bar{\sigma}_i}(D \circ \bar{s}) \right) \neq \{ \} \\
&\Leftrightarrow \bigvee_{\bar{s} \in \bar{\mathcal{S}}} \bar{s} \circ \mathcal{U}'_{\bar{\sigma}_i}(D \circ \bar{s}) \neq \{ \} \\
&\Leftrightarrow \bigvee_{\bar{s} \in \bar{\mathcal{S}}} \mathcal{U}'_{\bar{\sigma}_i}(D \circ \bar{s}) \neq \{ \} \\
&\Leftrightarrow \bigvee_{\bar{s} \in \bar{\mathcal{S}}} \text{unifiable}'_{\bar{\sigma}_i}(D \circ \bar{s})
\end{aligned}$$

Notice how the computation of composing each \bar{s} with unifiers of $D \circ \bar{s}$ is eliminated. In fact, since these compositions are exactly how unifiers are built up, we end up determining whether the set of unifiers is nonempty without constructing any. Also, by using about properties of disjunction, we derive an algorithm that terminates as soon as it has determined that there is a unifier. This is an example of a “continuation-based transformation” [29].

To handle the flexible-flexible case, Huet made the crucial observation that one can always save these pairs until all the others are eliminated, at which time the resulting disagreement set is always unifiable. For this to be useful though, we have to assume that we have constants at all base types.

7.2 Minimal Complete Set of Unifiers

Here the specification is relational rather than functional:

$$\text{MCSU}_{\bar{\alpha}}(f, g, \hat{\mathcal{S}}) \triangleq \text{complete}_{\bar{\alpha}}(\hat{\mathcal{S}}, \mathcal{U}_{\bar{\alpha}}(f, g)) \wedge \text{minimal}(\hat{\mathcal{S}})$$

The trickiest part is proving minimality. We give a summary of our results. Here, assume that a and a' are not in \bar{z}

- MCSU in terms of the generalized function MCSU' :

$$\text{MCSU}_{\bar{\alpha}}(f, g, \hat{\mathcal{S}}) \Leftrightarrow \text{MCSU}'_{\bar{\alpha}}(\{ \langle f, g \rangle \}, \hat{\mathcal{S}})$$

- The empty disagreement set case:

$$\text{MCSU}'_{\bar{\alpha}}(\{ \}, \{ I_{\bar{\alpha}} \})$$

- Abstractions:

$$\begin{aligned}
\text{MCSU}'_{\bar{\alpha}}(D' \cup \{ \langle \lambda \bar{z}. (\lambda y. A), \lambda \bar{z}. (\lambda y. A') \rangle \}, \hat{\mathcal{S}}) &\Leftrightarrow \\
\text{MCSU}'_{\bar{\alpha}}(D' \cup \{ \langle \lambda \bar{z}. A, \lambda \bar{z}. A' \rangle \}, \hat{\mathcal{S}}) &
\end{aligned}$$

- Rigid-rigid:

$$\text{MCSU}'_{\bar{\alpha}}(D' \cup \{ \langle \lambda \bar{z} . a(h_1, \dots, h_n), \lambda \bar{z} . a(h'_1, \dots, h'_n) \rangle \}, \hat{S}) \Leftarrow \\ \text{MCSU}'_{\bar{\alpha}}(D' \cup \{ \langle \lambda \bar{z} . h_1, \lambda \bar{z} . h'_1 \rangle, \dots, \langle \lambda \bar{z} . h_n, \lambda \bar{z} . h'_n \rangle \}, \hat{S})$$

$$\text{MCSU}'_{\bar{\alpha}}(D, \{ \}) \Leftarrow \\ \langle \lambda \bar{z} . a(\bar{h}), \lambda \bar{z} . a'(\bar{h}') \rangle \in D \wedge a \neq a'$$

- Rigid-flexible:

$$\text{MCSU}'_{\bar{\alpha}}(D, \bigcup_{\bar{s} \in \hat{S}} \bar{s} \circ \hat{S}_{\bar{s}}) \Leftarrow \\ \langle \lambda \bar{z} . a(\bar{h}), \lambda \bar{z} . \bar{z}_{[1]}(\bar{h}') \rangle \in D \wedge \bar{S} = \text{fr-subst}(i, a, \bar{\alpha}) \wedge \bigwedge_{\bar{s} \in \hat{S}} \text{MCSU}'_{\bar{\alpha}_{\bar{s}}}(D \circ \bar{s}, \hat{S}_{\bar{s}})$$

- Flexible-rigid:

$$\text{MCSU}'_{\bar{\alpha}}(D' \cup \{ \langle \lambda \bar{z} . \bar{z}_{[1]}(\bar{h}'), \lambda \bar{z} . a(\bar{h}) \rangle \}, \hat{S}) \Leftarrow \\ \text{MCSU}'_{\bar{\alpha}}(D' \cup \{ \langle \lambda \bar{z} . a(\bar{h}), \lambda \bar{z} . \bar{z}_{[1]}(\bar{h}') \rangle \}, \hat{S})$$

If we know the set of constants whose type has as its eventual range a given base type, we can use a technique very similar to the rigid-flexible case to give a full treatment to the flexible-flexible case.

7.3 Matching

The matching specifications are straightforward specializations of the unification specifications. We can make certain simplification, *e.g.* not instantiating the constant term in the rigid-flexible case to form the new disagreement sets. Also, the flexible-flexible case does not arise.

8 Improvements to the Algorithms

In this section, we develop some high-level improvements to the algorithms derived in the previous few sections. The first improvement allows for a complete algorithm (if a unifier exists it will be found). The second is a way to “factor” the state of a unification problem to prevent redundant computation.

8.1 Matching Trees

The algorithms suggested by our derivation perform a depth-first search of an implicit or-tree in their search for unifiers, where the branching nodes are from the rigid-flexible case. The first improvement is to make this tree explicit in order to gain control over the order in which it is traversed. We represent these trees as a set of triples $(\hat{s}, \bar{\alpha}, D)$ where the $\bar{\alpha}, D$ correspond to the nodes and the \hat{s} correspond to the paths from the root. The generalized specification is

$$\mathcal{U}_{\text{trcc}}(\mathcal{T}) = \bigcup_{\langle \hat{s}, \bar{\alpha}, D \rangle \in \mathcal{T}} \hat{s} \circ \mathcal{U}'_{\bar{\alpha}}(D)$$

The interesting case is rigid-flexible:

$$\begin{aligned}
& \mathcal{U}_{\text{tree}}(\mathcal{T}' \cup \{ \langle \hat{s}, \bar{\alpha}, D \rangle \}) \\
&= \mathcal{U}_{\text{tree}}(\mathcal{T}') \cup \hat{s} \circ \mathcal{U}'_{\bar{\alpha}}(D) \\
&= \mathcal{U}_{\text{tree}}(\mathcal{T}') \cup \hat{s} \circ \bigcup_{\bar{s} \in \hat{S}} \mathcal{U}'_{\bar{\alpha}\bar{s}}(D \circ \bar{s}) \\
&= \mathcal{U}_{\text{tree}}(\mathcal{T}') \cup \bigcup_{\bar{s} \in \hat{S}} (\hat{s} \circ \bar{s}) \circ \mathcal{U}'_{\bar{\alpha}\bar{s}}(D \circ \bar{s}) \\
&= \mathcal{U}_{\text{tree}}(\mathcal{T}' \cup \{ \langle \hat{s} \circ \bar{s}, \bar{\alpha}\bar{s}, D \circ \bar{s} \rangle \mid \bar{s} \in \hat{S} \})
\end{aligned}$$

This reformulation lets us choose which disagreement-set to work on. For instance, we can use breadth-first search to get a complete algorithm.

8.2 Matching Dags

Considering the specification of $\mathcal{U}_{\text{tree}}$, we notice that some triples might share the same $\bar{\alpha}, D$, causing unnecessary recomputation of $\mathcal{U}'_{\bar{\alpha}}(D)$. We can “factor out” common $\bar{\alpha}, D$ to save work, using triples $\langle \hat{s}, \bar{\alpha}, D \rangle$, where \hat{S} is a set of substitutions:

$$\mathcal{U}_{\text{dag}}(\mathcal{D}) = \bigcup_{\langle \hat{s}, \bar{\alpha}, D \rangle \in \mathcal{D}} \hat{s} \circ \mathcal{U}'_{\bar{\alpha}}(D)$$

where we write a composition of two sets of substitutions to mean the set of all compositions of elements of the first with elements of the second.

Carrying along the tree analogy from the previous section, interpreting the set \hat{S} as a collection of paths to the nodes $\bar{\alpha}, D$ we call these sets of triples “matching dags”. The relation to the tree unifier specification is

$$\mathcal{U}_{\text{dag}}(\mathcal{D}) = \mathcal{U}_{\text{tree}}\left(\bigcup_{\langle \hat{s}, \bar{\alpha}, D \rangle \in \mathcal{D}} \{ \langle \hat{s}, \bar{\alpha}, D \rangle \mid \hat{s} \in \hat{S} \} \right)$$

Note that the amount of factoring is not at all constrained by this specification. Even though it is in general expensive to check for the existence of identical disagreement sets, there is a simple sufficient condition for factorization. Briefly, when a disagreement set can be partitioned into subsets that do not share free variables, these subsets can be solved independently and their solutions can be combined easily to form the set of all solutions to the original set. In many cases the exponentially large matching tree is reduced to a matching dag linear or quadratic in the size of the original disagreement set.

We illustrate this using the example of a common the second-order unification problem, namely $f\bar{a} = T$ where T is a constant term with n occurrences of \bar{a} . For simplicity of presentation and implementation, the prefix of free variables has been collected outside the disagreement pair. This change is not formally presented in our derivation overview. In our notation we have the initial disagreement set

$$\lambda f . \{ \langle f\bar{a}, T \rangle \}.$$

In such a case there will be 2^n possible matches. For example:

$$\lambda f . \{(fa, gaa)\}$$

has solutions $\langle \lambda x . gxx \rangle, \langle \lambda x . gax \rangle, \langle \lambda x . gxa \rangle, \langle \lambda x . gaa \rangle$.

With the use of matching dags these can be represented and computed in a way that is linear in the size of the constant term rather than exponential as a matching tree would have to be. Figure 1 shows an example of a matching dag for the problem $fa = g(gaa)a$.

Note that there are 2^n different paths from the root to the success node, which means that there are as many unifying substitutions. Also note that there are only $2 * n + c$ different arcs, which means that the second-order unification algorithm will run in linear time in this simple but frequent case of matching a second-order variable applied to a constant against a constant term. In the more general case of matching $fT_1 = T_2$ the matching dag including the failure nodes will have size $|T_1| * |T_2|$, still a significant improvement over the exponential size of a matching tree.

8.3 Matching Graphs

Huet's heuristic improvements (subsumption and rigid-path check), which lead to more frequent termination, can be given a new interpretation in our framework. Integrating subsumption generalizes from a matching dag to a possibly cyclic matching graph. An idea of Gallier and Snyder [9] provides a simple sufficient condition for subsumption. This is subject of current research and the preliminary results are encouraging.

9 Extensions to the Algorithm

9.1 Products

We have extended the basic higher-order unification algorithms to handle a typed λ -calculus enriched with products. Our motivation came from using typed λ -calculus to encode the syntax of programming languages and higher-order unification to mechanically apply program transformation rules, as suggested in [12], and more recently in [23,19]. Some language constructs are most naturally represented with tuples, *e.g.* multiple argument function definitions and parallel let binding. Another example, representing and transforming functional programs is discussed at the end of section 10

The usual way of adding products to introduce a type $\sigma \times \tau$ for types σ and τ and terms $\text{fst } M$, $\text{snd } M$, and $\langle M, N \rangle$ for terms M and N . We prefer the notational convenience of languages like ML [10], in which one may write *patterns* (also known as "varstructs") built up from variables and pairing, rather than simply variables following a lambda.

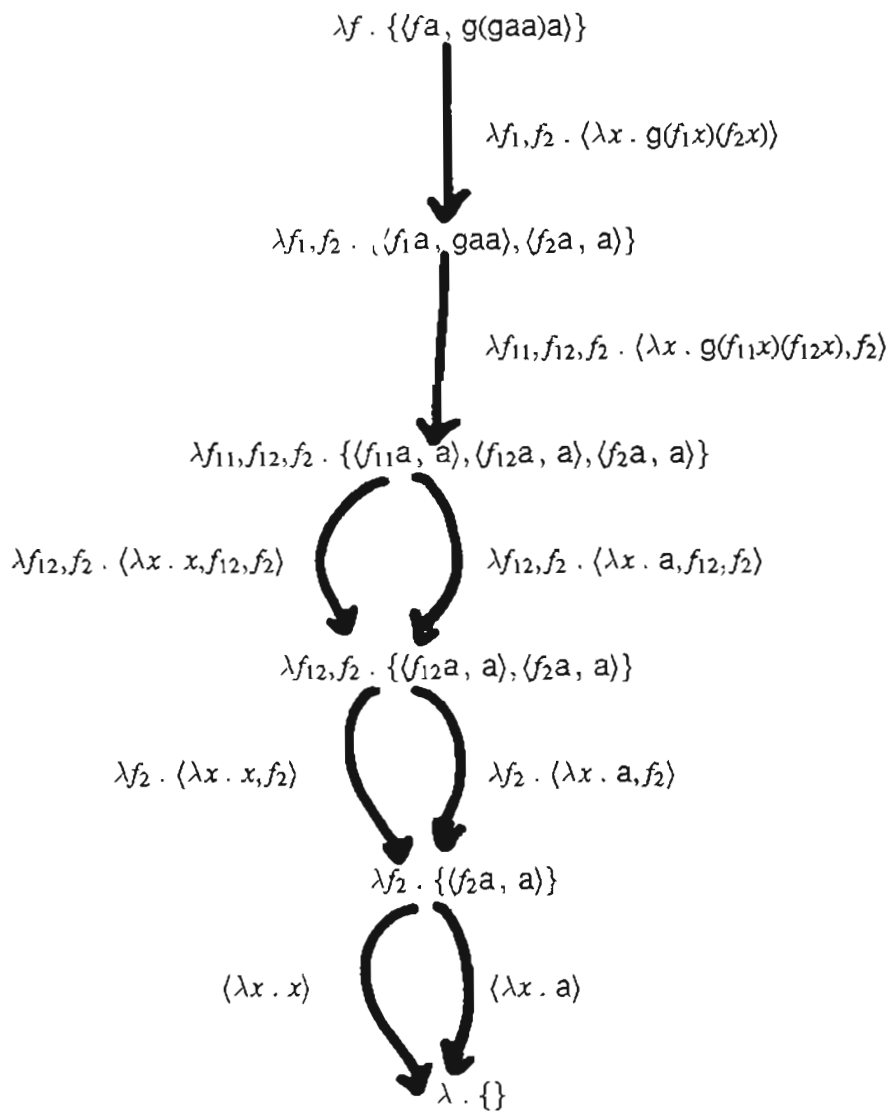


Figure 1: Example of a matching dag

Here is our extended language. First expressions:

$$\begin{aligned}
 E & ::= V_T && \text{typed variables} \\
 & | C_T && \text{typed constants} \\
 & | EE && \text{applications} \\
 & | \lambda P . E && \text{abstractions} \\
 & | \langle E, E \rangle && \text{pairs}
 \end{aligned}$$

Next types,

$$\begin{aligned}
 T & ::= Bt && \text{base types} \\
 & | T \rightarrow T && \text{function types} \\
 & | T \times T && \text{product}
 \end{aligned}$$

Finally, patterns

$$\begin{aligned}
 P & ::= V && \text{variable} \\
 & | \langle P, P \rangle^{T, \dots, T} && \text{pairing patterns (the } P\text{'s must have distinct variables)}
 \end{aligned}$$

As an example, here is a function that takes a pair and returns its reversal. Its type is $\sigma \times \tau \rightarrow \tau \times \sigma$

$$\lambda \langle u_\sigma, v_\tau \rangle . \langle v, u \rangle$$

This is certainly clearer than $\lambda p_{\sigma \times \tau} . \langle \text{snd } p, \text{fst } p \rangle$, especially for complex expressions.

Now suppose we want a function that takes a function that produces pairs and returns a function that produces the reversal of these pairs. One way would be

$$\lambda f_{\alpha \rightarrow \sigma \times \tau} . \lambda x_\alpha . (\lambda \langle u_\sigma, v_\tau \rangle . \langle v, u \rangle)(f x)$$

Instead, we generalize the tuple patterns to what we call “pairing” patterns. The pattern $\langle f, g \rangle^{\alpha, \dots, \gamma}$ will “destructure” a function of type $\alpha \rightarrow \dots \rightarrow \gamma \rightarrow \sigma \times \tau$ by binding f to an object of type $\alpha \rightarrow \dots \rightarrow \gamma \rightarrow \sigma$ and g to an object of type $\alpha \rightarrow \dots \rightarrow \gamma \rightarrow \tau$. (Actually f and g may be patterns themselves performing further destructuring.) This is similar to “target tupling”, but generalized to any number of (curried) arguments. We may also use this notation for terms, but it seems less useful than in patterns, so we prefer to have only simple pairs in the language proper. The interpretation is

$$\langle M, N \rangle^{\alpha, \dots, \gamma} \triangleq \lambda x_\alpha . \dots \lambda z_\gamma . \langle M x \dots z, N x \dots z \rangle$$

where x, y, z are not free in the M or N . Now we may state our previous example as

$$\lambda \langle g, h \rangle^\alpha . \langle h, g \rangle^\alpha$$

or in the language proper

$$\lambda \langle g, h \rangle^\alpha . \lambda x_\alpha . \langle h x, g x \rangle$$

In what follows, we will use the term “pairing type” to refer to types $\alpha \rightarrow \dots \rightarrow \gamma \rightarrow \sigma \times \tau$ and “pairing constant” or “pairing variable” to refer to a constant or variable having pairing type.

9.2 Conversion Rules

First we add a rule of pair formation

$$M_{\sigma \times \tau} = \langle (\lambda \langle u_\sigma, v_\tau \rangle . u)M, (\lambda \langle u_\sigma, v_\tau \rangle . v)M \rangle$$

In addition to the β -conversion rules from section 4, we have one more for pairing patterns:

$$(\lambda \langle p, q \rangle^{\alpha, \dots, \gamma} . A) \langle M, N \rangle^{\alpha, \dots, \gamma} = (\lambda p . \lambda q . A)MN$$

or, in the language proper

$$\begin{aligned} & (\lambda \langle p, q \rangle^{\alpha, \dots, \gamma} . A) (\lambda x_\alpha . \dots \lambda z_\gamma . \langle M, N \rangle) \\ &= (\lambda p . \lambda q . A) (\lambda x_\alpha . \dots \lambda z_\gamma . M) (\lambda x_\alpha . \dots \lambda z_\gamma . N) \end{aligned}$$

Also, we generalize the η rule to all patterns:

$$\lambda p . Mp = M \quad \text{if no } v \text{ in } p \text{ is free in } M$$

(It is a slight abuse of notation to use the pattern p as a term here. We will always mean the obvious term associated with the pattern.)

This gives us a correspondingly generalized derived α rule:

$$\lambda p . M = \lambda q . (\lambda p . M)q$$

9.3 Normal Forms

The generalized α -conversion rule turns out to be crucial to normal forms and hence to our algorithm. The key idea is to introduce structure wherever possible by “renaming”, or more appropriately, “restructuring”, variables of pairing type into pairings of variables of simpler type. For instance, we would perform the following conversion

$$\begin{aligned} (\lambda p_{\sigma \times \tau} . \dots p \dots) &= \lambda \langle x_\alpha, ybe \rangle . (\lambda p_{\sigma \times \tau} . \dots p \dots) \langle x, y \rangle \\ &= \lambda \langle x, y \rangle . \dots \langle x, y \rangle \dots \end{aligned}$$

For now, let us assume that our terms contain no pairing constants or free variables. (The latter assumption is reasonable in the context of unification, since we explicitly abstract over what would be the free variables.) Then by a translation of the results of [24] into our language, we have the following normal form:

$$\lambda p_1 . \dots \lambda p_m . a h_1 \dots h_n \quad \text{or} \quad \lambda p_1 . \dots \lambda p_m . \langle h_1, \dots, h_n \rangle$$

where

- All variables bound in the p_i are distinct and not of pairing type.
- The head, a , is a non pairing variable or constant.

To handle the case of pairing constants and free variables, we introduce the notation of a “locator”, which is a term $\lambda p . v$ in which none of the variables in p has pairing type, and v is in p . Then our normal form is just like the above except that a is either

- A variable not of pairing type, or
- of the form Lc for a locator L and constant c .

9.4 Algorithm Modifications

Somewhat surprisingly, there are only a few modifications required to our higher-order unification algorithms derived in the previous sections.

- There is one more case, the *tuple-tuple* case, which is handled similarly to the rigid-rigid case, *i.e.* replace the tuple-tuple disagreement pair by the set of pairs of corresponding elements. Note that the first alternative of our normal form is constrained not to have pairing type, so there are no “rigid-tuple”, “flexible-tuple”, etc. cases.
- In the rigid-rigid case, we again require equality of heads, but now this means comparing locators as well as constants.
- There are more possible “projections” [11] because unification variables (conventionally, the free variables) may take arguments of pairing type.

10 Polymorphism

We can also give an incomplete but very useful treatment of polymorphism similar to that done by λ Prolog [18]. Here we mean “implicit”, or “global”, polymorphism as in ML [10], rather than the more powerful “explicit” polymorphism of the second-order polymorphic λ -calculus [25].

The idea is very simple: For two terms to be unifiable, they must have the same type. Thus before unifying any disagreement pair, first perform first-order unification on their types. If the type unification fails, the term unification must fail. Otherwise, use the resulting most general type unifier to instantiate the types of all of the disagreement pairs. The place we can get into trouble is the flexible-rigid case. If the type of the flexible head is not yet determined, we cannot know the possible projection substitutions. In our experience, this case seems to be rare.

Polymorphism is especially useful in conjunction with products because it lets us express general transformation and inference rules that apply to constructs of any arity. For instance, we have proved and mechanically applied an extremely simple and general rule of “unfolding” [3] that can simultaneously unfold any subset of a collection of recursive functions at any subset of their call sites. It is simply

$$\forall g . (\text{Rec } \lambda f . g f f) = (\text{Rec } \lambda f . g (g f f) f)$$

Here, f is a tuple of functions each of which will likely take a tuple as its argument. Unification with a concrete program will cause g to be instantiated to a function of two arguments, the first of which occurs where unfolding will take place. In practice we would often want to specialize this rule (by substitution) so as not to become overwhelmed by the nondeterminism.

References

- [1] Peter B. Andrews, Dale Miller, Eve Cohen, and Frank Pfenning. Automating higher-order logic. *Contemporary Mathematics*, 29:169–192, August 1984.
- [2] Joseph Bates and Robert Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.
- [3] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.
- [4] Thierry Coquand and Gérard Huet. Constructions: a higher order proof system for mechanizing mathematics. In *EUROCAL85*, Springer-Verlag LNCS 203, 1985.
- [5] Jacques Corbin and Michel Bidoit. A rehabilitation of Robinson's unification algorithm. In R.E.A. Mason, editor, *Information Processing*, pages 909–914, Elsevier Science Publishers, 1983.
- [6] Conal Eliou and Frank Pfenning. *A Family of Program Derivations for First- and Higher Order Unification*. Technical Report, Carnegie Mellon University, Pittsburgh, 1987 (in preparation).
- [7] Lars-Henrik Eriksson. Synthesis of a unification algorithm in a logic programming calculus. *Journal of Logic Programming*, 1:3–18, 1984.
- [8] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [9] Jean Gallier and Wayne Snyder. A general complete E-unification procedure. In Pierre Lescanne, editor, *Rewriting Techniques and Applications, Bordeaux, France*, pages 216–227, Springer-Verlag, LNCS 256, Berlin, May 1987.
- [10] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Volume 78 of *Lecture Notes in Computer Science*, Springer Verlag, 1979.
- [11] Gérard Huet. A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, 1:27–57, 1975.
- [12] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
- [13] D. C. Jensen and T. Pietrzykowski. Mechanizing ω -order type theory through unification. *Theoretical Computer Science*, 3:123–171, 1976.
- [14] Zohar Manna and Richard Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.
- [15] Alberto Martelli and Ugo Montanari. Theorem proving with structure sharing and efficient unification. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, page 543 ff, IJCAI, Boston, 1977.
- [16] Alberto Martelli and Ugo Montanari. *Unification in Linear Time and Space: A Structured Presentation*. Internal Report B76-16, Ist. di Elaborazione delle Informazioni, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.

- [17] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175, North-Holland, 1980.
- [18] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Programming*, Springer Verlag, July 1986.
- [19] Dale A. Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *Symposium on Logic Programming, San Francisco*, IEEE, September 1987.
- [20] B. Möller. *A survey of the project CIP: Computer-aided, intuition-guided programming*. Technical Report TUM-18406, Institut für Informatik der TU München, Munich, West Germany, 1984.
- [21] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.
- [22] Lawrence Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
- [23] Frank Pfenning and Conal Elliott. *Definition and Applications of Higher-Order Abstract Syntax*. Technical Report, Carnegie Mellon University, Pittsburgh, 1987 (in preparation).
- [24] Garrel Pottinger. The Church-Rosser theorem for the typed λ -calculus with surjective pairing. *Notre Dame Journal of Formal Logic*, 22(3):264–268, July 1981.
- [25] J.C. Reynolds. *Towards a Theory of Type Structure*, pages 408–425. Volume 19 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1974.
- [26] J. A. Robinson. Computational logic: the unification computation. *Machine Intelligence*, 6:63–72, 1971.
- [27] William L. Scherlis. Program improvement by internal specialization. In *Eighth Symposium on Principles of Programming Languages*, pages 41–49, ACM, ACM, January 1981.
- [28] R.D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19:437–453, 1976.
- [29] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the Association for Computing Machinery*, 27(1):164–180, January 1980.