June 1988

Use of Higher-Order Unification for Implementing Program Transformers

John Hannon
University of Pennsylvania

Dale Miller
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-46.

Use of Higher-Order Unification for Implementing Program Transformers

Abstract

Source-to-source program transformers belong to the class of meta-programs that manipulate programs as objects. It has previously been argued that a higher-order extension of Prolog, such as λProlog, makes a suitable implementation language for such meta-programs. In this paper, we consider this claim in more detail. In λProlog, object-level programs and program schemata can be represented using simply typed λ-terms and higher-order (functional) variables. Unification of these λ-terms, called higher-order unification, can elegantly describe several important meta-level operations on programs. We detail some properties of higher-order unification that make it suitable for analyzing program structures. We then present (in λProlog) the specification of several simple program transformers together with a more involved partial evaluator. With the depth-first control strategy of λProlog for both clause selection and unifier selection all the above mentioned specifications can be and have been executed and tested.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-46.

USES OF HIGHER-ORDER UNIFICATION FOR IMPLEMENTING PROGRAM TRANSFORMERS

John Hannan
Dale Miller

MS-CIS-88-46
LINC LAB 118

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

June 1988

Appear in the proceedings of the Fifth International Logic Programming Conference, August 1988, Seattle, Washington.

Acknowledgements: This research was supported in part by NSF grants CCR-87-05596, MCS-8219196-CER, IRI84-10413-AO2, DARPA grant NOOO14-85-K-0018, and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

USES OF HIGHER-ORDER UNIFICATION FOR IMPLEMENTING PROGRAM TRANSFORMERS

JOHN HANNAN and DALE MILLER
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Abstract
Source-to-source program transformers belong to the class of meta-programs that manipulate programs as objects. It has previously been argued that a higher-order extension of Prolog, such as λProlog, makes a suitable implementation language for such meta-programs. In this paper, we consider this claim in more detail. In λProlog, object-level programs and program schemata can be represented using simply typed λ-terms and higher-order (functional) variables. Unification of these λ-terms, called higher-order unification, can elegantly describe several important meta-level operations on programs. We detail some properties of higher-order unification that make it suitable for analyzing program structures. We then present (in λProlog) the specification of several simple program transformers together with a more involved partial evaluator. With the depth-first control strategy of λProlog for both clause selection and unifier selection all the above mentioned specifications can be and have been executed and tested.

1 Introduction

Source-to-source program transformations have been the subject of considerable research over the past twenty years. Most program transformation systems are organized around a collection of transformation rules that specify source and target programs together with a set of constraints on these programs. On the surface, such rules have a natural declarative reading: if the input program matches the source template and certain auxiliary constraints are satisfied, then the output program is the result of instantiating the output template. In practice, however, such rules become rather complex and awkward; auxiliary constraints evolve in number and complexity as their functionality often narrows. Many of these auxiliary constraints are needed to handle those syntactic aspects of comparing an input program with a template that are not captured by first-order unification. In particular, many of these constraints need to treat both syntactic conditions, such as "C is a constant" and "X is not free in T," as well as semantic constraints, such as "F is the composition of G with itself" or "functions F and G commute." Often these syntactic and semantic conditions get mixed in ways that obfuscate their separate roles, thereby complicating any reasoning about such systems. In discussions of transformation systems this issue is often skirted by presenting transformations instead of the transformers (the actual code) that implement them. An

informal, often mathematical, high-level language is usually used to present transformations, thereby passing the problems of syntactic and semantic conditions into the informal language; actual transformers, however, are often not presented.

In this paper, we present the actual transformers for several familiar transformations. Our transformers are also presented in a higher-level language, but one that is formalized and can be executed directly. The implementation language is the higher-order logic programming language λProlog [11, 14]. We shall argue that the process of matching a template with a program is considerably enriched if both the program and the template are considered as λ-terms and the match is achieved by higher-order unification. In such a setting, the number of auxiliary predicates can be reduced and their use can largely be limited to mostly semantic aspects of the programs being transformed.

This paper is organized as follows. In Section 2, we describe some high-level concerns of implementing programs that manipulate other programs. In Section 3, we present simply typed λ-terms and various properties of higher-order unification. In Section 4, we demonstrate how a programming language, in particular, a subset of the functional programming language ML, can be represented as λ-terms. Section 5 presents the implementation of several simple program transformers in λProlog and section 6 presents a slightly more elaborate transformation program, a partial evaluator. We conclude in Section 7.

2 Programs as Values

Consider using Lisp to write program transformers for, say functional programs. While Lisp provides a representation of programs via its notation for λ-terms, the only primitive mechanisms for manipulating such terms in Lisp are essentially those for manipulating lists, namely, CAR, CDR, and CONS. Programs and lists are different objects, however, and the complexity of the structure of programs is not captured by simple list manipulation functions. While any transformer can be implemented using lists to represent programs and CAR, CDR, and CONS to decompose and construct programs, the resulting implementation of such transformers is often complex and difficult to understand. Also, in Lisp, the equality operator EQUAL is not sensitive to the usual meaning of λ-terms. For example, if two Lisp terms differ only in their bound variable names, they are not EQUAL. Thus, while Lisp contains a notation for λ-terms, it does not treat them as being their own data type.

One characteristic that distinguishes between programs (especially functional programs) as values and list structures is that equality between λ-terms is typically considered modulo λ-conversion. This notion of equality is a much more complex operation than simple syntactic equality (see Section 3). In particular, using this notion of equality, a λ-term is equal to any alphabetic variant of itself. With respect to this notion of equality, accessing the name of a bound variable in a λ-term is not a meaningful operation since equal terms might return different values. Adhering to this notion of equality disqualifies most conventional methods of analyzing the structure of programs.

Higher-order unification is a mechanism that can be used to probe the structure of programs, respecting congruence classes modulo λ-conversion. Illustrations of how this is achieved are outlined in Section 3. If the only method for manipulating λ-terms is via higher-

order unification then it is impossible to distinguish between two programs which are equal modulo λ-conversion. In particular, it is impossible to access the names of bound variables. In this paper we make use of a higher-order logic programming language, λProlog [14], to implement programs transformations because this language employs higher-order unification in a direct fashion. While this language extends Prolog in several directions, we shall only regard it as an implementation of the theory of higher-order Horn clauses [13].

The use of λ-terms and of higher-order unification to implement program manipulation systems has been proposed by various people. Huet and Lang in [10] employed second-order matching (a decidable subcase of higher-order unification) to express certain restricted, "template" program transformations. Miller and Nadathur in [12] extended their approach by adding to their scheme the flexibility of Horn clause programming and richer forms of unification. In [6] we argued that if the Prolog component of the CENTAUR system [2] were enriched with higher-order features, logic programming could play a stronger role as a specification language for various kinds of interpreters and compilers.

While we are only concerned in this paper with the simply typed λ-calculus, richer and more flexible λ-calculi have been proposed as a suitable representation system for programs. For example, Pfenning and Elliot in [16] have extended the simply typed λ-calculus to include simple product types. They also discuss in depth the role of *higher-order abstract syntax*, *i.e.*, the representation of programs as λ-terms, in the construction of flexible and general program manipulation systems. The LF specification language [7] uses a λ-calculus with a strong typing mechanism to specify various components of proof systems: much of this specification language could profitably be used in the context we are concerned with here. While extensions of higher-order unification to such rich notions of terms and types are important, we do not consider them here.

Similar advantages of the blend of higher-order unification and logic programming have been exploited in systems that manipulate formulas and proofs of logical systems. Felty and Miller in [5] discuss the use of λProlog to specify and implement theorem provers and proofs systems. Here again, λ-terms and higher-order unification are used to represent and manipulate formulas and proofs. The Isabelle theorem prover of Paulson [15] also makes use of these features to implement flexible theorem provers.

3 λ-Terms and Higher-Order Unification

In this section we present particular properties of λ-terms and higher-order unification that are exploited in the program transformers described in Sections 5 and 6.

Throughout the rest of this paper we use the word "term" to mean simply typed λ-term. The simple types are composed of some collection of primitive types and of all functional types built from these primitive types. The collection of primitive types must contain at least one type, written as o, which is used to denote the type of logic programming propositions. Other primitive types can be added by the programmer of this system. We shall assume that there are denumerably many constants and variables at all types. Simply typed λ-terms are built from these using the usual notions of application and abstraction.

We denote by the operation $[N/x]M$ the result of substituting the term N for all free

occurrences of x in M. Bound variables of M may require renaming to avoid capture of free variables in N. A *β-redex* is a formula of the form $(\lambda x\ M)N$ and an *η-redex* is a formula of the form $\lambda x\ (Px)$ where x is not free in P. The equality for λ-terms is the reflexive, symmetric, and transitive closure of the following relation on terms of the same type: P is related to Q if they are either alphabetic variants of each other, or Q results by replacing a β-redex $(\lambda x\ M)N$ in P with $[N/x]M$ or by replacing an η-redex $\lambda x\ (Mx)$ in P with M. This relation between λ-terms is called $\beta\eta$-conversion. A λ-term is in *β-normal form* if it contains no β-redexes, and is in *$\beta\eta$-normal form* if it is in β-normal form and contains no η-redexes. Any simply typed λ-term is $\beta\eta$-convertible to a $\beta\eta$-normal form, which is unique up to alphabetic changes of bound variables. Also, two terms are equal in this theory if and only if they have a common $\beta\eta$-normal form. (Proofs of these results may be found in the literature.) In this paper, whenever we refer to a normal form or λ-conversion we mean $\beta\eta$-normal form and $\beta\eta$-conversion. See [8] or [9] for more details on the λ-calculus.

Let t and s be two terms of the same type possibly with free variables of any type. The problem of deciding if there is a substitution, say σ, such that $\sigma(t)$ λ-converts to $\sigma(s)$, is known as *higher-order unification.* When one of the two terms to be unified is closed we refer to unification as matching. The nature of higher-order unification is described in detail in [9] and in lesser detail in [11]. Although general higher-order unification is undecidable, all the unification problems arising in the examples in this paper are decidable. A simple, depth-first implementation of [9] is all that is necessary to find unifiers. We do not present higher-order unification in detail here since the points we wish to make only require an understanding of some of its high-level properties. These properties can be determined by looking at the behavior of λ-terms under substitution and λ-conversion. In particular, the following definition and properties of terms will be exploited in our subsequent transformers.

Definition 1 (Dependence on an abstraction) *We say that term t is* dependent on its i^{th} abstraction *if a λ-normal form of t is of the form $\lambda x_1 \ldots \lambda x_i.t'$ and x_i is free in t'. Notice that t' may be a function abstraction itself, i.e., it might be of functional type.*

Property 1 (Dependency Invariance) *Let t be a term which is dependent on its i^{th} abstraction. If t λ-converts to s, then s is dependent on its i^{th} abstraction.*

This property concludes that dependence on an abstraction is well-defined. This fact is important since equality between terms is determined by λ-conversion.

Property 2 (Dependency and Substitution) *Let t be a term and σ a substitution. If $\sigma(t)$ is dependent on its i^{th} abstraction, then t is dependent on its i^{th} abstraction.*

This property states that abstraction dependency cannot be introduced by substitution. For example, if $t = \lambda x \lambda y.(f\ x)$ then there is no substitution σ (for the free variable f) such that $\sigma(t)$ is dependent on its second abstraction. If the substitution term for f contains a free occurrence of y, for example $\sigma = \{f \mapsto (g\ y)\}$, the bound occurrence of y is renamed to avoid variable capture. Thus, $\sigma(t)$ converts to $\lambda x \lambda z.(g\ y\ x)$, which is not dependent on its second abstraction.

Note that the converse of Property 2 is not true; dependencies can disappear under higher-order substitution. For example, consider the term $t = \lambda x \lambda y.(f\ x\ y)$ and let $\sigma = \{(f, \lambda x \lambda y.y)\}$. While t is dependent on its first and second abstractions, $\sigma(t)$ is dependent only on its second abstraction.

Property 3 (Nested Dependency) *Let t be a term, let σ be a substitution, and assume that t and $\sigma(t)$ have λ-normal forms $\lambda x_1 \ldots \lambda x_n.t'$ and $\lambda x_1 \ldots \lambda x_n.t''$, respectively. Let i, j be integers such that $1 \leq i, j \leq n$. If every occurrence of x_i in t' is in the scope of an occurrence of x_j in t' then every occurrence of x_i in t'' is in the scope of an occurrence of x_j in t''.*

That is, the "in the scope of" relationship (*i.e.*, if a bound variable x is the head of a subterm then all occurrences of bound variables within this subterm are said to be in the scope of x) between bound variables does not change under substitution.

As an example of how these properties can be related to the unification of two terms we consider the term

$$t_0 = \lambda u \lambda v \lambda w \lambda h.(f\ u\ h(g(v))),$$

which we attempt to unify with each of the closed terms

$$\begin{aligned} t_1 &= \lambda u \lambda v \lambda w \lambda h.((2 * w) + h(3 * v)), \\ t_2 &= \lambda u \lambda v \lambda w \lambda h.((2 * u) + (3 * v)), \\ t_3 &= \lambda u \lambda v \lambda w \lambda h.((2 * u) + h(3 * v)). \end{aligned}$$

Property 2 states that no substitution instance of t_0 will have a dependence on its third argument and, therefore by Property 1, there is no substitution instance of t_0 that is equal to the term t_1. Hence, t_1 and t_0 are not unifiable. Similarly, Property 3 implies that t_0 and t_2 are not unifiable since every occurrence of v in t_0 is in the scope of h, while this is not true of t_2. t_0 does, however, unify with t_3 by the substitution

$$\{f \mapsto \lambda x \lambda y.((2 * x) + y),\ g \mapsto \lambda x.(3 * x)\}.$$

This substitution is the only such unifier for t_0 and t_3.

If we consider terms like t_1, t_2, t_3 as fragments of programs that are to be transformed and terms like t_0 as templates or expression schemata then this kind of unification can provide a useful mechanism for probing the structure of such fragments. For example, an expression fragment of the form $\lambda u \lambda v \lambda w \lambda h.t$ matches the template t_0 only if w is not free in t, all free occurrences of u in t do not occur in the scope of h, and all free occurrences of v in t do occur in the scope of h. Notice also that v may have multiple occurrences in the scope of h. The function (λ-term) g would then denote the abstraction of all such occurrences. Furthermore, the combination $h(g\ v)$ could occur multiple times in t; the function f would need to represent not only the abstraction of all occurrences of u but also all the occurrences of $h(g\ v)$. Thus, the term

$$t_4 = \lambda u \lambda v \lambda w \lambda h.(u + (h(v + v)) * (h(v + v)))$$

unifies with t_0 with the resulting substitution

$$\{f \mapsto \lambda x \lambda y.(x + y * y),\ g \mapsto \lambda x.(x + x)\}.$$

The ability of higher-order unification to provide sophisticated probing and analysis of programs is exploited in the remainder of this paper.

4 A Simple Object Language

Before introducing some simple transformers, we describe the object language that our example transformers will manipulate. For this we use a restricted subset of the functional programming language mini-ML, which is a subset of the language ML not containing exceptions, pattern matching, datatype declarations or modules [3]. We do not present the actual concrete syntax of our language but present only its higher-order abstract syntax [16]. We do this by providing an illustrative example: more details can be found in [6].

To the pure typed λ-calculus we add new typed constants to denote program language constructions. For example, we add constants *car, cdr, cons, null* to denote the familiar operations on lists and the constants $0, 1, 2, \ldots, +, *, \ldots$. We also need to add the constants *truth* and *false* denoting the booleans and the constant *if* denoting the usual conditional. To represent recursion, something not primitive or representable in the simply typed λ-calculus, we add a constant *fix* to denote the "least fixed point" operator (sometimes called the Y combinator). These constants are not interpreted by the logic programming language in which they are embedded. For example, the unifier does not unify the expression x with $(cons\ (car\ x)\ (cdr\ x))$. The intended meaning of these constants arises in our setting only in the way in which they are used in program transformers. The exact set of constants in the object-level programming language does not concern us greatly since all the transformations we consider are either unaffected by the addition of new constants or are simply and modularly extended to handle new constants. The constants *if* and *fix* are of particular interest in the next section.

The append program, written in a functional programming style as

```
app K L = (if (null K) L (cons (car K) (app (cdr K) L))),
```

is coded as the λ-term

$$(fix\ \lambda f \lambda x \lambda y.\ (if\ \ (null\ x)\ \ y\ \ (cons\ (car\ x)\ (f\ (cdr\ x)\ y)))).$$

We replace the name of the function (`app`) by an abstraction (λf) and the fixed point operator. This is a standard representation of recursion and one that we use repeatedly in this paper. We replace the bound variables of the function, namely `K` and `L`, by the abstractions λx and λy. These abstractions, for the function name and the bound variables, are crucial to some of the transformers introduced in the next section.

5 Some Basic Transformers

In this section we present a number of transformers that effect simple transformations. These transformations typically comprise part of the basis of any rule-based program transformation system. Our emphasis in this section is not on producing new transformations but on providing simple implementations of some familiar transformations. Before presenting such

transformations, we briefly describe a few aspects of our meta-language: the higher-order Horn clauses fragment of λProlog.

Terms in these higher-order Horn clauses are simply typed λ-terms in which individual and function variables may occur. These variables can be universally quantified over program clauses. The theory of higher-order Horn clauses provides for quantification of some occurrences of predicate variables [11, 14] but this aspect is not needed here and is ignored. A term constructed by the operation of application is represented by writing two terms in juxtaposition and separated by a space. A term constructed by the operation of abstraction is represented by the infix operator '\' separating the bound variable being abstracted and the term over which it is being abstracted.

In displaying Horn clauses, we use the common convention of not explicitly displaying universal quantifiers. Instead, the variables which are intended to be quantified are written as upper case letters. All other symbols are constants.

Our first example of a higher-order Horn clause is particularly simple. The transformation that unwinds the definition of a recursive program can be represented as an atomic Horn clause containing one universally quantified functional variable:

```
unwind  (fix A)  (A (fix A)).
```

If we use this clause in its forward direction, that is, with its first argument instantiated and second argument unbound, unification computes the appropriate abstraction to bind to `A`. The second argument is then constructed by applying this abstraction to the original argument. The strength of this transformer arises from the substitution, which is implicit in using β-conversion. In particular, any term that unifies with `(fix A)` would cause `A` to be bound to a term of the form $(\lambda f\ t)$. Thus the output value `(A (fix A))` is of the form $(\lambda f\ t)\ (\mathit{fix}\ \lambda f\ t)$, which, in λ-normal form, is $[(\mathit{fix}\ \lambda f\ t)/f]t$. All recursive calls (marked by the bound variable f) are replaced by the code of the recursive program. Notice that t may be of any functional type. If this `unwind` program is "run in reverse", that is, if the first argument is unbound and the second is instantiated, significant higher-order unification is required to decide if the program in its second argument is the result of unwinding.

If a recursive program contains no recursive calls, it can be transformed simply into a non-recursive program. This transformer, called *vacuous-recursion*, is given by the clause

```
vacuous_recursion  (fix F\A)  A.
```

Note how this clause makes explicit use of property 2 to determine that the body of a recursive program does not contain a recursive call. Any λ-term that matches `F\A` cannot depend on its first abstraction and, in this context, cannot contain any recursive calls.

Perhaps the most familiar transformation rule is the *fold/unfold* rule. The *unfold* transformation replaces a call to a function with the function's body, substituting actual parameters for formal ones. The *fold* transformation replaces an instance of a function's body by a call to the function, substituting formal parameters for actual ones. The following clause can be used to implement unfold.

```
unfold  (fix A)  (fix F\(A (A F))).
```

Operationally, we can think of supplying this clause with a closed term for its first argument and applying higher-order unification and λ-conversion to produce the second argument.

As an example of unfolding, consider calling the *unfold* predicate with the append program, that is, the λ-term

```
(fix F\X\Y\(if (null X) Y (cons (car X) (F (cdr X) Y))))
```

as its first argument. Such a call would succeed only if the second argument of the call can unify with the term

```
(fix F\X\Y\(if (null X) Y
               (cons (car X)
                     (if (null (cdr X)) Y
                         (cons (car (cdr X))
                               (F (cdr (cdr X)) Y)))))).
```

Another simple transformation is one which reverses the order of the first two arguments of a recursive program. This transformation is specified by the atomic clause

```
swap_args  (fix B)
           (fix F\Y\X\(B (U\V\(F V U)) X Y)).
```

Note that this particular clause interchanges the first and second arguments of a function with *at least* two arguments (possibly more).

Our fifth transformer in this section, one which we call the *parametric* transformer, employs a more subtle use of higher-order unification. If the first argument of a recursive function is not modified in any recursive call then we can transform the function definition into a "parametric" recursive function of one less argument. This transformer is given by the clause

```
parametric  (fix F\U\(G (F U) U))
            U\(fix R\(G R U)).
```

The first argument is a recursive function (of at least one argument) and the second argument is the new equivalent function definition in which the recursion has been simplified. In any term unifying with the first argument, all recursive calls to `F` must have exactly `U` as its first argument. (Its other arguments can be functions of `U` and for distinct occurrences of `F` these functions need not be the same.) Hence, the first argument to the function defined by this term remains constant. This is a typical structure of many functions in which some argument is not used until the recursion bottoms out (*cf.* the second argument of the append function). In the new function definition the first argument has been "abstracted out" over the fixed point operator, yielding a parametric recursive function of one less argument.

As an example consider the term for the append program with its two arguments swapped (this could have been obtained by applying `swap_args` to the term for the append program

given above):

```
(fix F\Y\X\(if (null X) Y (cons (car X) (F Y (cdr X)))))
```

Supplying this term as the first argument to the `parametric` clause produces the term (as the second argument)

```
Y\(fix F\X\(if (null X) Y (cons (car X) (F (cdr X)))))
```

with the free variable `G` in the definition of `parametric` instantiated to

```
F\Y\X\(if (null X) Y (cons (car X) (F (cdr X)))).
```

Our final example is slightly more elaborate and illustrates the use of higher-order Horn clauses to write a recursive transformer. Assume that we have a function of one argument whose body can contain a nesting of `if` statements. If a condition in one of these `if` statements does not depend on the argument of the function, the value of this condition could be computed and determined to be either `truth` or `false`. In such a case the function could be simplified by replacing the `if` statement with either its first or second branch, depending on the computed value of the boolean condition. This operation can be specified by the following clauses:

```
prune_if  X\(if C (H1 X) (H2 X)) G :-
              eval C truth, prune_if H1 G.
prune_if  X\(if C (H1 X) (H2 X)) G :-
              eval C false, prune_if H2 G.
prune_if  X\(if (C X) (H1 X) (H2 X))
          X\(if (C X) (G1 X) (G2 X)) :-
              prune_if H1 G1, prune_if H2 G2.
prune_if  H H.
```

In the first two clauses note the key use of Property 2, which ensures that `C` is independent of the expression's arguments and so can be evaluated.

This transformer differs from the other ones in this section in several respects. First, it is recursive and makes use of a richer collection of Horn clauses to accomplish this recursion. Second, it works only for functions of exactly one argument. This is a rather serious limitation. A similar pruning operation for functions of two arguments or any fixed number of arguments can be given. Using higher-order Horn clauses, however, we have no natural way to write pruning functions that work for the general case (*i.e.*, functions with an arbitrary number of arguments). A solution to this problem has been proposed by Pfenning and Elliot [16]: introducing product types into the λ-calculus adds the necessary flexibility. Finally, this transformer requires the language-specific predicate `eval` that evaluates closed terms. The code for this predicate is not presented here; its structure is straightforward and is given in [6]. Of course, to make this pruning operation deterministic within the depth-first interpreter for λProlog cuts should be added strategically in the first three clauses in the definition of `prune_if`.

All these transformations can be implemented, of course, using first-order methods (*e.g.*, as in Lisp or Prolog), but consider what such implementations would look like. They certainly would not be the simple "one-liners" afforded by higher-order Horn clauses.

6 Specifying a Partial Evaluator

We now demonstrate how some of these basic transformers can be composed to provide a slightly more elaborate program transformer. We concentrate on the declarative aspects of the transformers, largely ignoring issues of control. As our example we consider the task of partial evaluation, a technique of source-to-source optimizations. The topic of partial evaluation is receiving increased attention lately [1]; an earlier summary by Ershov provides a good introduction to the subject [4]. The partial evaluator we implement is restricted in its domain of operation; it serves only to illustrate how such transformers can be specified. More elaborate partial evaluators can easily be built using these techniques.

Let us now consider the tasks involved in partially evaluating a function with respect to some known input for its first parameter. Assume we are given some function f of, for example, two arguments x and y. For a given value c we want to compute a new function g of one argument such that for all values of y, $f(c,y) = g(y)$. One intent of this operation is that for any value of y computing $g(y)$ should be easier (*e.g.*, faster) than computing $f(c,y)$. Such improvement is possible by "compiling" the information of $x=c$ in f into the definition of g. Thus partial evaluation can be decomposed into two phases: (*i*) substitution of the known value for the corresponding parameter in the function definition, and (*ii*) simplification of the resulting definition.

The substitution in phase (*i*) is easily accomplished by λ-conversion since the formal parameters of functions are represented by λ-abstractions. The simplification in phase (*ii*) entails a careful structural analysis of the result of phase (*i*), which requires a descent through the structure of the function body searching for expressions that can be evaluated or reduced. Typically, subexpressions that can be completely evaluated (*i.e.*, are independent of the remaining formal parameter) are located and replaced with their values. The analysis and transformers of the previous section can be brought together to achieve this task. We decompose phase (*ii*) into two distinct stages: expand and reduce. The expand stage nondeterministically expands (or embellishes) the given expression. In our example this process is limited to unfolding, but additional strategies could be employed. The reduce stage then attempts to evaluate and reduce expressions by, for example, an operation similar to the `prune_if` operation of the previous section.

Figure 1 contains a set of higher-order Horn clauses (in the syntax of λProlog) for a simple partial evaluator. For this example we rely on the default ordering of clauses, with a depth-first search strategy, for control.

The clause for `pe` gives the top-level of the partial evaluator. It has three arguments: `(fix F)`, a recursive function of (at least) two parameters; `C`, a value for the first parameter to `(fix F)`; and `G`, the residual function of one argument. The partial evaluator has three sub-goals: substitution, simplification and elimination. Substitution (phase (*i*)) is given by the single clause `substitute`, and it substitutes the value `C` for `F`'s first parameter (`X`). Note the implicit

```
pe (fix F) C G :-
   substitute F C F1,
   simplify F F1 G1,
   eliminate G1 G.

substitute  F\X\Y\(B F X Y)  C  F\X\Y\(B F C Y).

eliminate  F\X\Y\(B Y)  Y\(B Y).

simplify F F1 G1 :-
   expand F F1 H,  reduce H G1.

expand F F1 F1.
expand F F1 G1 :-
   unfold F F1 H,  expand F H G1.

unfold F  F1  G\(F1 (F G)).


%%% reduce: base cases
reduce F\X\Y\Y G\X\Y\Y.
reduce F\X\Y\A G\X\Y\A.

%%% reduce: pruning if statements
reduce F\X\Y\(if (F0 F X Y) (F1 F X Y) (F2 F X Y)) G :-
   reduce F0 G0\X\Y\truth,  reduce F1 G.

reduce F\X\Y\(if (F0 F X Y) (F1 F X Y) (F2 F X Y)) G :-
   reduce F0 G0\X\Y\false,  reduce F2 G.

reduce F\X\Y\(if (F0 F X Y) (F1 F X Y) (F2 F X Y))
       G\X\Y\(if (G0 G X Y) (G1 G X Y) (G2 G X Y)) :-
   reduce F0 G0,  reduce F1 G1,  reduce F2 G2.

%%% reduce: primitives (cons, car, etc.)
reduce F\X\Y\(cons (F1 F X Y) (F2 F X Y))
       G\X\Y\(cons (G1 G X Y) (G2 G X Y)) :-
   reduce F1 G1,  reduce F2 G2.
```

Figure 1: Part of a Partial Evaluator

use of β-conversion as `B` is a higher-order variable. The clause for `simplify` performs both stages of phase (*ii*): expansion and reduction. Declaratively, `expand` nondeterministically unfolds the function zero or more times. Notice that for the `unfold` clause we use the original function definition `F` for the unfolding and not the function `F1` for which `C` has been substituted. `F1` is a function that ignores its first argument and is *not* a recursive function definition. For this reason, the `unfold` predicate here is different than the one in the previous section. The clause for `eliminate` converts a recursive function of two arguments, in which the recursion is vacuous and the first argument does not occur free, into a non-recursive function of just one argument. This is achieved by applying the vacuous-recursion principle (eliminating the abstraction `F`) and similarly eliminating the abstraction `X`. Other techniques of converting the result of `simplify` into a function of one argument could be used by adding additional clauses for `eliminate`.

Finally, the clauses for `reduce` implement the reduction stage of phase (*ii*). This particular implementation of `reduce` is very simple and limited; more elaborate reductions are studied in the literature on partial evaluation. The arguments of the `reduce` predicate denote fragments of recursive programs of two arguments. Maintaining these abstractions during the recursion of `reduce` enables higher-order unification to handle correctly the restrictions on variables needed at various stages of the reduction. The `reduce` operation is intended to descend through the body of a recursive program of two arguments. While making its descent, it moves through and prunes, when possible, *if* statements. Unlike the `prune_if` program, the recursive descent of `reduce` is intended to move through all the constants of the language. A clause to descend through `cons` is given; others can easily be added. The base cases for this recursion are the first two clauses. Note that the bound variable `X` cannot occur in the body of the program and need not be considered as a base case.

The clauses for pruning `if` statements replace calls to an `eval` predicate with recursive calls to `reduce`. Thus in the full presentation of `reduce` we include clauses that produce (as their second argument) terms of the form `G0\X\Y\truth` and `G0\X\Y\false`. Such clauses would, in effect, evaluate boolean valued expressions built from constructors like `null` and `=`.

If all recursive calls are eventually eliminated (by a series of unfoldings and reductions) then `pe` succeeds. Note that interleaving unfoldings with reductions might provide a more efficient implementation (requiring less backtracking). For the purposes of this paper, however, we are more concerned with the declarative nature of partial evaluation and so maintaining the two distinct stages is beneficial.

While this evaluator is simple it demonstrates some of the basic techniques of partial evaluation and how they can be implemented in λProlog. Consider, for example, using `pe` on the append program presented in the previous section. The goal

```
pe (fix F\X\Y\(if (null X) Y (cons (car X) (F (cdr X) Y))))
   (cons 1 nil) G.
```

succeeds with the sole free variable `G` being instantiated to

```
Y\(cons (car (cons 1 nil)) Y)).
```

This transformation succeeds because the unfolding and pruning strategy worked here to remove all recursive calls. From a procedural perspective the simplification phase entailed one unfolding followed by two reductions (the first being the "false" clause of an `if` expression and the second being the "true" clause of an `if` expression). With a more sophisticated `reduce` predicate, the expression `(car (cons 1 nil))` could have been simplified to just `1`.

7 Conclusion

To provide clear specifications of program transformers, ones that focus more on semantic issues, we re-examined a higher-order logic programming language as a meta-language for manipulating programs as objects. In this logic programming language object-level programs and program schemata are represented using simply typed λ-terms and higher-order variables. With this approach we argued that higher-order unification of terms denoting programs and program schemata provides an elegant method of implicitly specifying the syntactic constraints of transformations. To illustrate this point we first presented several transformers written in higher-order Horn clauses. These transformers have a clear declarative reading and avoid the use of non-logical constructs for describing syntactic constraints. We then illustrated how these simple transformers can be used to specify more sophisticated transformers by presenting a simple partial evaluator. The default depth-first control mechanisms of λProlog provided an adequate experimental implementation of all these specifications. With further research in this area we hope to develop our understanding of richer program transformers, focusing on the areas of specification, implementation (including control issues) and formal properties.

Acknowledgements: We would like to thank Frank Pfenning for useful discussions related to this paper. This work was supported by NSF grant CCR-87-05596, DARPA grant N000-14-85-K-0018 and also by a fellowship from the Corporate Research and Architecture Group, Digital Equipment Corporation, Maynard, MA.

References

[1] D. Bjørner, A. Ershov, and N. Jones, editors. *Proceedings of the IFIP Workshop on Partial Evaluation and Mixed Computation.* IFIP TC-2, 1987.

[2] P. Borras *et. al.* *CENTAUR: the System.* Technical Report 777, INRIA, December 1987.

[3] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: mini-ML. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 13–27, 1986.

[4] A. Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[5] A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In *Proceedings of the Ninth International Conference on Automated Deduction*, 1988.

[6] J. Hannan and D. Miller. Enriching a meta-language with higher-order features. In *Workshop on Meta-Programming in Logic Programming*, Bristol, June 1988.

[7] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, 1987.

[8] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[9] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[10] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order logic. *Acta Informatica*, 11:31–55, 1978.

[11] D. Miller and G. Nadathur. Higher-order logic programming. In *Proceedings of the Third International Logic Programming Conference*, Springer-Verlag, 1986.

[12] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *Proceedings of the IEEE Fourth Symposium on Logic Programming*, IEEE Press, 1987.

[13] G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD thesis, University of Pennsylvania, May 1987.

[14] G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference Symposium on Logic Programming*, MIT Press, 1988.

[15] L. Paulson. *The Foundation of a Generic Theorem Prover.* Technical Report 130, University of Cambridge, Cambridge, England, March 1988.

[16] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 1988.