An Extension to ML to Handle Bound Variables in Data Structures: Preliminary Report

Dale Miller
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104–6389 USA
dale@cis.upenn.edu

Abstract

Most conventional programming languages have direct methods for representing first-order terms (say, via concrete datatypes in ML). If it is necessary to represent structures containing bound variables, such as λ-terms, formulas, types, or proofs, these must first be mapped into first-order terms, and then a significant number of auxiliary procedures must be implemented to manage bound variable names, check for free occurrences, do substitution, test for equality modulo alpha-conversion, etc. We shall show how the applicative core of the ML programming language can be enhanced so that λ-terms can be represented more directly and so that the enhanced language, called ML_λ, provides a more elegant method of manipulating bound variables within data structures. In fact, the names of bound variables will not be accessible to the ML_λ programmer. This extension to ML involves the following: introduction of the new type constructor `'a => 'b` for the type of λ-terms formed by abstracting a parameter of type `'a` out of a term of type `'b`; a very restricted and simple form of higher-order pattern matching; a method for extending a given data structure with a new constructor; and, a method for extending function definitions to handle such new constructors. We present several examples of ML_λ programs.

1 Introduction

Recent work in the specification of a wide variety of meta-programming systems — type checkers and inferrers, theorem provers, program manipulation systems, evaluators, and compilers — has revealed several important specification techniques, including one called *higher-order abstract syntax* [16]. This specification technique uses a typed λ-calculus to succinctly capture many complex syntactic notions pertaining to data structures containing notions of bound variables, such as those of free and bound occurrences, scopes of binders, equality up to alphabetic change of bound variable names, and substitutions. Huet and Lang [9] seem to have been the first to recognize the potential of this approach to abstract syntax in an actual implementation. Some of their ideas were later generalized in [11] to a logic programming setting. Higher-order abstract syntax is now central to

[1]This research was supported in part by grants ONR N00014-88-K-0633, NSF CCR-87-05596, and DARPA N00014-85-K-0018. This paper was first presented at the Logical Frameworks Basic Research Action meeting in Nice, France, 7 – 12 May 1990. I am grateful to John Hannan, Frank Pfenning, and several attendees of the Logical Frameworks BRA Workshop for comments on the content of this paper.

several recent specification systems. The Logical Framework (LF) [8], the Calculus of Constructions (CC) [3], and hereditary Harrop formulas [12] are some recent logics that support this new view of syntax. These systems have been used to specify various meta-programming tasks: LF has been used to specify numerous proof systems [1] as well as various aspects of conventional programming language semantics [2]; hereditary Harrop formulas have been implemented in the Isabelle theorem prover [14, 15] and in λProlog, where a wide range of meta-programs have been written [4, 5, 6]; and, CC has been used to specify and compute with several deep mathematical theorems as well as specify and develop various algorithms.

Such a specification technique has been accounted for in practice in essentially two ways. The first can be called the package-style approach. That is, an implementor writes a collection of functions and routines in a given programming language that captures this specification technique. For example, the Mentor system [9] contained an implementation of second-order matching in order to implement template matching for sophisticated program transformations. The Isabelle system is a package of ML programs that can manipulate the higher-order abstract syntax of various object logics.

The second approach to accounting for these specification techniques is to design programming languages in which they are directly incorporated. The programming languages λProlog [13] and Elf [17] are two such programming languages.

In all of these cases — Mentor, Isabelle, λProlog, and Elf — rather strong forms of unification have been used to manipulate typed λ-terms. Thus, at first glance, it would seem difficult to find an extension to the functional programming language ML that could incorporate higher-order abstract syntax since general unification, especially unification that may be undecidable and does not admit most general unifiers, is difficult to integrate directly into a functional setting.

In this paper, we shall attempt to illustrate how ML can be extended so as to allow the direct manipulation of structures with bound variables. The extensions, called ML_λ, will therefore permit direct exploitation of the technique of higher-order abstract syntax. The key idea to making this extension is that a weak form of unification of simply typed λ-terms, described in [10], is adequate for capturing much of the ideas of higher-order abstract syntax and that if unification is restricted to pattern matching, it comprises a simple and natural extension to usual ML pattern matching.

Familiarity with the basic elements of ML is necessary for reading this paper.

2 A new datatype constructor

When referring to ML in this paper, we shall only consider a very small subset of the language, a kind of mini-ML, which contains polymorphism, pattern matching, `let`, datatypes, and recursion. We shall not consider modules, abstract datatypes, references, records, and several other features. This restriction is intended to focus our attention. It is not clear how well the extension described here will interact with the full definition of Standard ML [7] but the main ideas of integrating higher-order abstract syntax with ML can be seen within this restriction.

The first-order syntax for the untyped λ-calculus can be declared using the following ML datatype definition.

```
datatype ltm = app of ltm * ltm | abs of string * ltm | var of string;
```

The λ-term $\lambda x(fxx)$ can be encoded by the term

```
abs("x", app(app(var "f", var "x"), var "x"))
```

Before this definition can be meaningfully used, it is necessary to write several functions for testing for alphabetic variants, changing bound variable names, testing for free or bound occurrences, etc. ML itself does not treat bound variables directly. For example, while the ML term

```
abs("y", app(app(var "f", var "y"), var "y"))
```

denotes the same "abstract" λ-term, a user defined function is necessary in order to establish this fact.

ML$_\lambda$ is the result of extending (mini-)ML with the following items.

1. One new type constructor: `'a => 'b` denotes the type of a λ-term with an abstracted variable of type `'a` over a term of type `'b`. We shall assume that both `'a` and `'b` are equality types. As we shall see below, `'a => 'b` will then also be an equality type. We shall furthermore restrict the type `'a` to be a user defined type. That is, it cannot be a type like `int` or `string` nor can it be a pair or list type. The reason for this restriction is that the type `'a` will be treated as an "open" type, that is, new constants of that type will appear during computations. Some concepts, such as integers and pairs, should be considered closed. This type should not be confused with the type `'a -> 'b`.

2. Two new term constructors:

   (a) `x\t : 'a => 'b` if `x` is an identifier of type `'a` and `t` is a term of type `'b`. Here, identifiers should be taken to be of the same class as the value constructor class Con of [7].

   (b) `t^x : 'b` if `t` is of type `'a => 'b` and `x` is an identifier introduced with \ and is of type `'a`. This symbol will only appear with pattern variables of `=>` type. This infix symbol can be thought of as having the type `('a => 'b) * 'a -> 'b`.

3. One new expression constructors `fun fname tok = exp1  ==> exp2`. This construction is necessary in order to extend the definitions of functions in scopes where new identifiers are introduced via \. The precedence of `==>` is higher than that of function definition.

We describe each of these extensions in turn.

Given the new type constructor above, we shall introduce three datatypes that will be used throughout the rest of this paper.

```
datatype tm = abs of tm => tm | app of tm * tm;

datatype term = a | b | f of term | g of term * term;

datatype form =  p  of term          | q    of term * term
              | and of form * form   | or   of form * form
              | imp of form * form   | not  of form
              | all of term => form  | some of term => form;
```

Here, the type `tm` is the type of untyped λ-terms, while `term` and `form` are the types for first-order terms and first-order formulas, resp. Notice that in each case, where a bound variables is intended, the type constructor `=>` is used. For example, to form a universally quantified expression, the constructor `all` is applied to a ML$_\lambda$ λ-term. The λ-expressions for the S, K, I combinators would be the following ML$_\lambda$ terms

```
abs x\(abs y\ (abs z\(app(app(x,z),app(y,z)))))
abs x\(abs y\x)
abs x\x
```

Similarly, the first-order formula $\forall x(p(x) \supset q(f(x), a)) \wedge \forall y \exists x(q(x, g(y, x)))$ would be the following term

```
and(all x\(imp(p(x), q(f(x),a)), all y\(some x\(q(x,g(y,x)))))
```

3 Equality and pattern matching

There would be no force to this extension if ML$_\lambda$ did not have built into it some equational facts about λ-terms. In particular, these terms will satisfy the equations for α and η-conversion along with the following very weak form of β conversion

`(x\t)~x = t` (β_0).

(This equation is only required when pattern variables of `=>` type are used.) Given this kind of equality theory for ML$_\lambda$ terms, it is not possible to destruct a λ-term by separating its bound variable from its body, since that operation is not invariant under α-conversion. Destructuring can be done, however, by suitably extending the notion of pattern matching.

A pattern variable, say `M` of type `t1 => t2 => ... => tn => t` (`=>` associates to the right) is permitted in a pattern/expression combination if every occurrence of `M` in that combination is of the form `M~x1~...~xm` where `m` is less than or equal to `n` and `x1,...,xm` is a list of distinct `\`-bound variables within the pattern or expression. Consider the following patterns and values for which they are to be matched.

```
(1)  x\y\(f(H~x))                u\v\(f(f(u)))
(2)  x\y\(f(H~x))                u\v\(f(f(v)))
(3)  x\y\(g(H~y~x,(f(L~x))))     u\v\(g(u,f(u)))
(4)  x\y\(g(H~x,L~x))            u\v\(g(g(a,u),g(u,u)))
```

In each of these examples, a pattern variable is written with a capital letter. Solving these patterns over the theory of α, β_o, η is a very simple generalization of first-order pattern matching. The following are the substitutions for solving these match problems.

```
(1)  H == w\(f(w))
(2)  match failure
(3)  H == y\x\x         L == x\x
(4)  H == x\(g(a,x))    L == x\(g(x,x))
```

The match failure for `(2)` arises from the fact that substitution for λ-terms must avoid variable capture. Hence, it is not possible to substitution a term for `H` in line `(2)` so that `y` is captured. This aspect of pattern matching is very useful. For example, the pattern `all x\(and(P,Q~x))` (with pattern variables `P` and `Q`) would match with a term denoting a universally quantifed conjunction in which the first conjunct does not contain a free occurrence of the quantified variable. For a more complete treatment of unification with variables of higher-type restricted as above see [10].

Unification in such a setting is like unification for first-order logic in that unification problems are decidable and most general unifiers exist when unifiers exist.

Given this use of pattern variables, the simplest functions that we can write in ML_λ would be the following:

```
fun vacuousp (x\T) = true
  | vacuousp S     = false;
(* ...   or using wild-cards  ... *)
fun vacuousp (x\_) = true
  | vacuousp _     = false;

exception DISCHARGE;
fun discharge (x\M) = M
  | discharge   _    = raise DISCHARGE
```

The function `vacuousp` has type `('a => 'b) -> bool` and can be used to determine whether or not an abstraction is vacuous. The function `discharge` has type `('a => 'b) -> 'b`. It returns the body of a vacuous abstraction or raises an exception if the abstraction is not vacuous. As the second way of writing `vacuousp` illustrates, the wildcard `_` denotes a pattern variable and that variable behaves similarly to other variables. It is not a "textual" variable; for example, the expression `x\_` does not match any λ-abstraction, it only matches a vacuous one. Use either `x\(_~x)` or simply `_` to match any λ-abstraction.

For another example, consider the following function that determines whether or not its argument is a term that denotes a Church numeral, that is, an untyped λ-term of the form $\lambda x \lambda f. f^n x$ for some $n \geq 0$.

```
fun numeralp (abs x\(abs f\x)) = true
  | numeralp (abs x\(abs f\(app(f,M~x~f)))) = numeralp (abs x\(abs f\(M~x~f)))
  | numeralp _ = false;
```

4 Extending function definitions

As the last example using Church numeral illustrates, while certain simple recursions over λ-terms is possible, more general recursions are not possible. For example, it is not possible to write a function that counts the number of applications `app` in a term of type `tm`. For more flexible recursion, we need the ability to extend datatypes, such as `tm` and `term`, with new constants and to also extend the definition of functions to include these constants. To motivate this, consider the following description of the syntax of simply typed λ-terms. Let Σ be a signature, that is, a set of simply typed constants. A term $ct_1 \cdots t_n$ (where c is neither an application nor abstraction) is a Σ-term of type τ over this signature if Σ contains the constant c at type $\tau_1 \to \cdots \to \tau_n \to \tau$ and for $i = 1, \ldots, n$, t_i is a Σ-term of type τ_i. The syntax rule for λ-abstraction is given by: $\lambda x.t$ is a Σ-term of type $\tau' \to \tau''$ if t is a $\Sigma \cup \{x : \tau'\}$-term of type τ'' (assuming x is not in Σ). That is, passing through an abstraction causes the signature (set of constants) to be increased: a bound variable can be thought of as introducing a "scoped constant."

Given this model of syntax, a functional program that performs recursion on the structure of λ-terms will need to have new constants introduced to stand for bound variables and will need to have procedures for extending functions so that they will behave correctly on those new terms. We illustrate such processing by writing one of the simplest recursive programs, the identity function. The functions `copyterm` and `copyform` defined below are such that they return their arguments unchanged by recursively copying that argument into its output.

```
fun copyterm a = a
  | copyterm b = b
  | copyterm (f X) = f(copyterm X)
  | copyterm (g(X,Y)) = g(copyterm X, copyterm Y);

fun copyform (p X)      = p(copyterm X)
  | copyform (q(X,Y))   = q(copyterm X, copyterm Y)
  | copyform (and(X,Y)) = and(copyform X, copyform Y)
  | copyform (or(X,Y))  = or(copyform X, copyform Y)
  | copyform (imp(X,Y)) = imp(copyform X, copyform Y)
  | copyform (all M)    = all  x\(fun copyterm x = x ==> copyform (M^x))
  | copyform (some M)   = some x\(fun copyterm x = x ==> copyform (M^x))
```

The only new item in these lines is in the clauses for copying `all` and `some`. Here, the `==>` expression construction is used. Evaluating the following expression

```
fun fname tok = exp1  ==> exp2
```

first extends the definition of the `fname` function with the clause that says that the \-identifier `tok` rewrites to the expression `exp1`; second, evaluates the expression `exp2`; and third, discharges the function definition extension once a value is returned.

Consider computing the value `copyform (some u\(all v\(p(u,v))))`. This would yeild the following sequence of expressions.

```
some x\(fun copyterm x = x ==> copyform (u\(all v\(p(u,v)))^x)
some x\(fun copyterm x = x ==> copyform (all v\(p(x,v))))
some x\(fun copyterm x = x ==>
  all y\(fun copyterm y = y ==> copyform (v\(p(x,v))^y)))
some x\(fun copyterm x = x ==>
  all y\(fun copyterm y = y ==> copyform (p(x,y))))
some x\(fun copyterm x = x ==>
  all y\(fun copyterm y = y ==> p(copyterm x,copyterm y)))
some x\(fun copyterm x = x ==> all y\(fun copyterm y = y ==> p(x,y)))
```

The final expression is, of course, more simply `some x\(all y\(p(x,y)))`, which is α-convertible to the initial term.

These copy-functions can be used to implement substitution into formulas. Consider the following ML_λ program.

```
fun substform M T = discharge x\(fun copyterm x = T ==> copyform (M^x))
```

Here `substform` has the type `(term => form) -> term -> form`. Its operation can be describe as follows: introduce a new identifier `x` of type `term`, instruct `copyterm` to copy `x` to `T` and then return the result of `copyform (M~x)`. Since `x` is not copied to itself, `x` cannot appear in the value `copyform (M~x)` since it is not in either the values `M` or `T`.. Thus, `discharge` will always be given a vacuous abstraction and hence will not raise an exception.

Substititution for the untyped λ-terms can be implemented similarly.

```
fun copy (app(S,T)) = app(copy S, copy T)
  | copy (abs M)    = abs(x\(fun copy x = x ==> copy (M~x)));

fun subst M T = discharge x\(fun copy x = T ==> copy (M~x));
```

Notice that in all the cases above, the coding of substitution is particularly simple and natural. The reader should consider writing similar subsitution functions in ML on the first-order syntax for untyped λ-terms and compare them to the above implementations.

In these examples, the substitution functions have been written in curried form. This choice was made simply to illustrate that substitution can be used to carry `=>` to `->`. For example, if `M` is of the type `term => form` then `substform M` is of the type `term -> form`. In otherwords, `M` can be seen as code describing a function from `term` to `form` and it is the `substform` function that translates that code into that actual function.

5 More examples

In this section we present several examples of computing on first-order formulas and on untyped λ-terms.

The following program simply counts the number of applications in an untyped λ-term.

```
fun count (app(T,S)) = 1 + count T + count S
  | count (abs M) = discharge x\(fun count x = 0 ==> count (M~x));
```

Notice that in this function, the base cases for recursion are introduced during the execution of this function.

The following two programs perform call-by-name and call-by-value reductions on untyped λ-terms.

```
fun cbn (abs M) = abs M
  | cbn (app(T,S)) =
     let val (abs M) = cbn T in
         cbn (subst M S)
     end;

fun cbv (abs M) = abs M
  | cbv (app(T,S)) =
     let val (abs M) = cbv T in
        cbv (subst M (cbv S))
     end;
```

The following function `normal` computes the $\beta\eta$-normal form of an untyped λ-terms (when such normal forms exist).

```
fun onepass (app(abs M, T)) = onepass (subst M T)
  | onepass (abs x\(app(T,x))) = onepass T
  | onepass (app(T,S)) = app(onepass T, onepass S)
  | onepass (abs M) = abs x\(fun onepass x = x ==> onepass (M~x));

fun normal T =
  let val S = onepass T in
    if T = S then T else normal S
  end;
```

The first clause of onepass reduces β-redexes and the second clause reduces η-redexes. Notice that the proviso that the abstracted varaible x in an η-reduce is not free in the term T is handled automatically.

The following program computes the negation normal form of first-order formulas by removing implications and by pushing negations using deMorgan's laws until they have atomic scopes.

```
fun nnf (p T) = p T
  | nnf (q(T,S)) = q(T,S)
  | nnf (neg(p T)) = neg(p T)
  | nnf (neg(q(T,S))) = neg(q(t,S))
  | nnf (neg(neg M)) = nnf M
  | nnf (and(M,N)) = and(nnf M, nnf N)
  | nnf ( or(M,N)) =  or(nnf M, nnf N)
  | nnf (imp(M,N)) = or(nnf(neg M),nnf N)
  | nnf (neg(imp(M,N)) = and(nnf M, nnf(neg N))
  | nnf (neg(and(M,N))) = or(nnf(neg M), nnf(neg N))
  | nnf (neg(or(M,N))) = and(nnf(neg M), nnf(neg N))
  | nnf (forall M) = forall x\(nnf(M~x))
  | nnf (exists M) = exists x\(nnf(M~x))
  | nnf (neg(forall M)) = exists x\(nnf(neg(M~x)))
  | nnf (neg(exists M)) = forall x\(nnf(neg(M~x)));
```

Notice that this function does not need to use the ==> construction to extend a function since new constants of type term are handled correctly by the first four clauses.

The following functions compute a prenex normal form of a formula in negation normal form. The merge auxillary function is used to combine the prefixes of two formulas in prenex normal form. If the first argument to merge is true, this merging is assumed to be across a conjunction; if that argument is false, it is assume to be across a disjunction.

```
fun merge (true,  all M,  all N)  = all  x\(merge(true,  M~x, N~x))
  | merge (false, some M, some N) = some x\(merge(false, M~x, N~x))
  | merge (flag, (all M), N)  = all  x\(merge(flag, M~x, N))
  | merge (flag, (some M), N) = some x\(merge(flag, M~x, N))
  | merge (flag, N, (all M))  = all  x\(merge(flag, N, M~x))
  | merge (flag, N, (some M)) = some x\(merge(flag, N, M~x))
  | merge (true, M, N)  = and(M,N)
  | merge (false, M, N) =  or(M,N);
```

```
fun prenex (p T) = p T
  | prenex (q(T,S)) = q(T,S)
  | prenex (not M) = not M
  | prenex (and(P,Q)) = merge(true,  prenex P, prenex Q)
  | prenex (or(P,Q))  = merge(false, prenex P, prenex Q)
  | prenex (all M)  = all  x\(prenex(M~x))
  | prenex (some M) = some x\(prenex(M~x));
```

6 Some problems and possible variations

The design of new programming languages and extensions to old languages is a serious business that should be carefully considered. The informal presentation in this paper is certainly not such a serious study. All that is indicated here is that there might be a dimension in which ML can be extended to address the concerns of handling the data structures containing bound variables. In this section, some problems with ML_λ are mentioned and possible variations of it are considered.

6.1 Problems regarding function definition extension

When a new constant is introduced by the \ construct, it is important to know if all the necessary functions have been extended to correctly handle that new constant. In all the examples in this paper, it was an easy matter to check the calling structure of functions to be sure that suitable definition extensions were actually done. When examples get to be larger, such checks might get to be very difficult. If higher-order programming is also involved in recursions over the structure of λ-terms, then it would be impossible in general to have a static check determine that all functions that might be passed in as values are extended correctly.

Another problem with function definition extensions is that functions can only be extended to handle just the new constant: more general patterns involving that constant are not possible. For example, consider the following two programs. Both seem quite sensible as computations while they make use of a stronger form of function definition extension than is permitted above.

```
fun eq (app(T,S),app(U,V)) = eq (T,U) andalso eq (S,V)
  | eq (abs M, app N) = discharge x\(fun eq (x,x) = true ==> eq (M~x,N~x))
  | eq (_,_) = false

fun subst M t =
  let fun aux x\x = t
        | aux x\(app (M~x, N~x)) = app(aux M, aux N)
        | aux x\(abs (M~x)) = abs y\(fun aux x\y = y ==> aux x\(M~x~y))
   in aux M
  end;
```

The function `eq` of type `(tm * tm) -> bool` determines whether or not its arguments are alphabetic variants. The function definition extension `fun eq (x,x) = true` involves extending `eq` on the value `(x,x)` and not just the identifier `x`. The function `subst` reimplements the previously given function of the same type, except this time it does not use `discharge` and the `copy` function.

Here again, the extension `fun aux x\y = y` is more general than permitted earlier. Notice that in both of these cases, the only constants of type `tm` that are permitted in new patterns are those involving the new constant.

6.2 Weakening pattern matching

In ML, the cost of pattern matching is dominated by the size of the pattern and not the value being matched against. This is not true of ML_λ since pattern matching may need to check if an abstraction is vacuous in a given input and this check will require a possible descent of the entire input. It is possible to modify pattern matching so that this check for vacuous abstraction is not part of the matching process. This can be done by requiring that if a pattern variable is in the scope of `\`-abstracted identifiers, that pattern variable must be applied to all those abstracted variables (in some order). Thus the pattern `all x\(and(P,Q^x))` would not be permitted while the pattern `all x\(and(P^x,Q^x))` would be permitted. The check for vacuous abstraction can be programmed in this weaker language on a per-signature basis (a fully polymorphic `vacuousp` presented earlier is not possible). For example, the following program is of type `(term => term) -> bool`.

```
fun vacuoustermp x\a = true
  | vacuoustermp x\b = true
  | vacuoustermp x\(f(M^x)) = vacuoustermp M
  | vacuoustermp x\(g(M^x, N^x)) = vacuoustermp M andalso vacuoustermp N
  | vacuoustermp x\x = false;
```

There seems to be no compelling reason for making pattern matching simpler in this manner: it seems more elegant to use pattern matching to achieve the same ends as calling the various `vacuousp` functions.

6.3 Internalizing `subst`

The implementation of the various `subst` predicates in this paper was completely determined by the signature of the constants building the data structures into which substitution is done. It is therefor sensible for an implementation of ML_λ to have a generic `subst` of type `(''a => ''b) -> ''a -> ''b` for equality types `''a` and `''b`. Internalizing substitution in this way is very similar to internalizing equality in SML to certain "concrete" datatypes.

6.4 The standard litany

This preliminary report does not address the large number of questions that should be addressed in serious language design. We list and briefly comment on some of these.

1. Type inference. This should be essentially the same as type inference for ML.

2. Run time type errors. These should not be possible. The extended pattern matching process can be done without respect to type information.

3. Match exception. It is very useful to have static checks that can warn a programmer of a function definition that may not have enough cases to deal with all the constructors that can be used to build its arguments. It should be possible to extend the analysis used in ML

to ML_λ, although very little can probable be done statically if higher-order programming is mixed with function definition extension.

4. Semantics of new constructions. Good question. This extension seems very intensional so getting a denotational semantics for it looks hard. On the other hand, this extension does seem to have significant "declarative context" and so should have some of meaningful semantics. Here, semantic notions used to address programs in λProlog might prove useful [12].

7 Conclusion

An extension to ML that would permit rather direct handling of data structures containing internal abstract has been informally proposed. That extension, called ML_λ, provides a way to represent such structures so that the programmer does not have direct access to bound variables names. Other more declarative ways of dealing with bound variables are made available.

References

[1] Avron, A., Honsell, F., and Mason, I (1987), *Using Typed Lambda Calculus to Implement Formal Systems on a Machine.* Technical Report ECS-LFCS-87-31, Laboratory for the Foundations of Computer Science, University of Edinburgh.

[2] Burstall, R. and Honsell, F. (1988), A natural deduction treatment of operational semantics. In *Foundations of Software Technology and Theoretical Computer Science*, pages 250–269, Springer-Verlag LNCS, Vol. 338.

[3] Coquand, T. and Huet, G. (1988), The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[4] Felty, A. and Miller, D. (1988), Specifying Theorem Provers in a Higher-Order Logic Programming Language, Proceedings of the Ninth International Conference on Automated Deduction, Argonne, IL, 23 – 26, eds. E. Lusk and R. Overbeek, Springer-Verlag Lecture Notes in Computer Science, Vol. 310, 61 – 80.

[5] Hannan, J. and Miller, D. (1988), Uses of Higher-Order Unification for Implementing Program Transformers, Fifth International Conference and Symposium on Logic Programming, ed. K. Bowen and R. Kowalski, MIT Press, 942 – 959.

[6] Hannan, J. and Miller, D. (1989), A Meta Language for Functional Programs, Chapter 24 of *Meta-Programming in Logic Programming*, eds. H. Rogers and H. Abramson, MIT Press, 453–476.

[7] Harper, R., Milner, R., and Tofte, M. (1989), *The Definition of Standard ML: Version 3.* Technical Report ECS-LFCS-89-81, Laboratory for the Foundations of Computer Science, University of Edinburgh.

[8] Harper, R., Honsell, F., and Plotkin, G. (1987), A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY.

[9] Huet, G. and Lang, B. (1978), Proving and Applying Program Transformations Expressed with Second-Order Logic, *Acta Informatica* 11, 31 – 55.

[10] Miller, D. (1990), "A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification," in *Extensions of Logic Programming* edited by Peter Schroeder-Heister, Springer-Verlag.

[11] Miller, D. and Nadathur, G. (1987), A Logic Programming Approach to Manipulating Formulas and Programs, Proceedings of the IEEE Fourth Symposium on Logic Programming, IEEE Press, 379 – 388.

[12] Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. (1988), Uniform proofs as a foundation for logic programming. To appear in the Annals of Pure and Applied Logic.

[13] Nadathur, G. and Miller, D. (1988), An Overview of λProlog, Fifth International Conference on Logic Programming, eds. R. Kowlaski and K. Bowen, MIT Press, 810 – 827.

[14] Paulson, L. (1986), Natural Deduction as Higher-Order Resolution, Journal of Logic Programming **3**, 237 – 258.

[15] Pauslon, L. (1989), The Foundation of a Generic Theorem Prover, Journal of Automated Reasoning, Vol. 5, 363 – 397.

[16] Pfenning, F. and Elliott, C. (1988), Higher-Order Abstract Syntax, Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 199 – 208.

[17] Pfenning, F. (1989), Elf: A Language for Logic Definition and Verified Metaprogramming, Fourth Annual Symposium on Logic in Computer Science, Monterey, CA, 313 – 321.