A Semi-Functional Implementation of a Higher-Order Logic Programming Language

Conal Elliott Frank Pfenning

Draft of February 1990

1 Introduction

In this chapter we develop an interpreter of a higher-order constraint logic programming language in Standard ML (SML). The logic programming language is closely related to λProlog [25], though the type system supported by our implementation is more general, for example by allowing explicit abstraction over types. The implementation is closely modeled after eLP, an implementation of λProlog in the Ergo Support System [8, 20] and may be considered as a rational reconstruction and explanation of the eLP implementation.

This is not a tutorial on λProlog (we present no λProlog programs at all), but for someone familiar with ML this should serve as a high-level operational semantics of a variant of the λProlog language. Prior knowledge of ML is assumed, but not at a very deep or sophisticated level (see [15] for an SML tutorial). We try to emphasize programming techniques as well as the gradual development of the interpreter in its full generality from a very simple starting point. For someone considering experimentation with variations on logic programming languages, this chapter should provide enough detail and techniques for the rapid implementation of a modified interpreter of related languages. Our approach is to write a true interpreter, and not to embed Prolog in ML the way Prolog is embedded in Scheme in [9] and [17]. The primary difference is that we separate carefully the name space of predicates of the logic programming language from the name space of functions in ML.

We do not address the use of the SML module system, nor do we discuss a number of features of λProlog such as its module system, input/output,

and other built-in special predicates. Also omitted are the front end of the interpreter (parsing, unparsing, type inference) and, due to space constraints, we limit ourselves to a sketch of higher-order unification. We hope to write a companion paper which concentrates primarily on a development of higher-order unification and type reconstruction within the framework laid out in this chapter.

One might also miss a discussion of compilation, which is not very well understood in this context and is the subject of current research [24].

We begin with an interpreter for propositional Horn logic, which introduces the central technique of the success continuation, due to Carlsson [2]. We then move on to first-order Horn logic, which is very much in the tradition of Prolog. In Section 4 we generalize this to include embedded implication and universal quantification (see [1, 11, 21] for the motivation for these constructs) which complicates primarily unification. In Section 5 we introduce side-effects and assignment in a controlled way to increase the efficiency of unification. This is refined in the next section where we address non-logical control constructs such as if-then-else and cut and introduce the trail. Section 7 sketches a more efficient clausal representation of programs which hitherto were simply formulas and hints at indexing. In Section 8 we generalize the underlying language of terms from first-order terms to typed λ-terms, at which point unification no longer generates most general unifiers and constraints enter the interpreter. Finally we discuss how to make the transition from terms to goals to allow true higher-order logic programming. Throughout this chapter we remark on the differences between the interpreter developed here and our Common Lisp implementation.

2 Propositional Horn Logic

We begin the development with a very simple propositional logic amenable to an interpretation as a programming language: propositional Horn logic. Our presentation is non-standard in that we do not require the formulas to be in *clausal form*: throughout our development we view this as a normal form, which must be justified by an appropriate metatheorem. A more efficient clausal representation for formulas will be introduced in Section 7.

2.1 The Language of Goals and Programs

The definition of propositional Horn logic is by induction in the form of a BNF grammar. G denotes the legal *goal* formulas (which are also the legal queries) and D the legal *program* formulas.[1] We sometimes refer to these classes as D-formulas and G-formulas, respectively, and collectively as *formulas*.

$$\begin{array}{rcl} G & ::= & A \mid \top \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \\ D & ::= & A \mid \top \mid D_1 \wedge D_2 \mid G \rightarrow D \end{array}$$

The letter A generally stands for atomic formulas; here this means propositional constants, $\top$ stands for truth, $\wedge$ stands for conjunction, $\vee$ for disjunction, and $\rightarrow$ for implication. In logic programming it is often more conspicuous to use $\leftarrow$, where $D \leftarrow G$ can be read as "D if G" and is a purely notational variant of $G \rightarrow D$.

The following figure shows how this would be translated into a `datatype` definition in SML (comments are enclosed in `(* *)`).

```
datatype gform  =                       (* Goal formula *)
     Gtrue                              (* Truth *)
  |  Gand of gform * gform              (* Conjunction *)
  |  Gor of gform * gform               (* Disjunction *)
  |  Gatom of string                    (* Atomic G formula *)

and dform =                             (* Program formula *)
     Dtrue                              (* Truth *)
  |  Dand of dform * dform              (* Conjunction *)
  |  Dimplies of gform * dform          (* Implication *)
  |  Datom of string                    (* Atomic D formula *)
```

This defines *constructors* such as `Dand`, which, when applied to two D-formulas, yields a D-formula. For example, the program $p \wedge (p \rightarrow q)$ would be represented as

```
Dand(Datom("p"),Dimplies(Gatom("p"),Datom("q")))
```

A query, such as whether q is true, would be represented as

```
Gatom("q")
```

[1] D is derived from *definite* as in *definite clauses*, though our definition is broader.

2.2 A First Interpreter

The next step is to give goals and programs an operational interpretation. For this simple logic, this is straightforward, though the precise definition of "upon backtracking" is deferred to the actual interpreter in ML. First comes the reduction of a goal to its subgoals, then the analysis of whether an atomic goal follows from the program.

1. Given goal $\top$, succeed.
2. Given goal $G_1 \wedge G_2$, attempt to solve G_1 and, if it succeeds, attempt to solve G_2.
3. Given goal $G_1 \vee G_2$, attempt to solve G_1. If this succeeds, succeed. If this fails, attempt to solve G_2.
4. Given an atomic goal A, look through the program for ways to establish A following the control structure below.

If we assumed the program to be in clausal form, we would enumerate the clauses of the form $A \leftarrow G$ and attempt to solve G for each such clause. It is easy to see that the following program analysis will behave this way on the special case of clausal form programs. We assume we are given a program D, and atomic goal A. We also have an "accumulated subgoal" which is initialized to $\top$.

1. $D = D_1 \wedge D_2$. Attempt to infer A from D_1. If this fails, attempt to infer A from D_2.
2. $D = G \rightarrow D_1$. Attempt to infer A from D_1, but conjoin G to the subgoal that remains to be solved.
3. $D = A$. Attempt to solve the accumulated subgoal.
4. $D = B$ for atomic B distinct from A. In this case D is not helpful in the attempt to derive A and we backtrack.

Thus our program consists of two mutually recursive functions. `solve` analyzes a composite goal, and `match_atom` analyzes the program with respect to an atomic goal. The fundamental idea of the formulation as a functional program is that of a *success continuation* due to Carlsson [2]. The obvious arguments to `solve` are the current goal and program. The

non-obvious argument is the success-continuation `sc`. `sc` is a function (of no arguments) that is to be called when the current goal succeeds. Backtracking is achieved simply by returning from the current function with an uninteresting value (we have chosen `() : unit`). The function `match_atom` calls a local recursive function `rec_match`, which accumulates subgoals as outlined above.

```
fun solve (Gtrue) prog sc = sc ()
  | solve (Gand(g1,g2)) prog sc =
      solve g1 prog (fn () => solve g2 prog sc)
  | solve (Gor(g1,g2)) prog sc =
      ( solve g1 prog sc ; solve g2 prog sc )
  | solve (Gatom(goal_const)) prog sc =
     match_atom goal_const prog sc

and match_atom goal_const prog sc =
     let fun rec_match (Dtrue) subgoal = ()
           | rec_match (Dand(d1,d2)) subgoal =
               ( rec_match d1 subgoal ; rec_match d2 subgoal )
           | rec_match (Dimplies(g,d)) subgoal =
               rec_match d (Gand(subgoal,g))
           | rec_match (Datom(prog_const)) subgoal =
               if prog_const = goal_const
                  then solve subgoal prog sc
                  else ()
      in rec_match prog (Gtrue) end
```

Let us inspect this compact program line-by-line.

1. If the current goal is $\top$, we succeed by invoking the success continuation.

2. If the current goal is $G_1 \wedge G_2$, we attempt to solve G_1, but also build a success continuation that will eventually solve G_2, which is necessary in order for the conjunction to succeed. `(fn () => ...)` is the SML way of constructing a function of no arguments.

3. If the current goal is $G_1 \vee G_2$, we attempt to solve G_1 with the same success continuation. If this should fail and thus return, we attempt to solve G_2. Semicolon is the SML sequencing operator. Note that

in a purely functional setting without any side-effects, it would make no sense to try the left and right subgoals in succession: if solving the left subgoal succeeds, it must produce some record of this. In the framework of success continuations, this is achieved through the *initial success continuation*, which could be a function such as `(fn () => print "Goal succeeded!")` (see Section 2.3).

4. If the current goal is atomic we look through the program to find D-formulas that might help us prove the goal.

Next we consider the program analysis. We call `rec_match` with a current subgoal `Gtrue`, which will always succeed and the whole program `prog` as the current D-formula. Here are the cases for `rec_match`.

1. If the program is $\top$, any atomic goal (which excludes $\top$) will fail. We return to indicate failure.

2. If the program is a conjunction we attempt to use the left conjunct and then the right conjunct to derive the atomic goal. This is dual to the case of a disjunctive goal.

3. If the program is an implication $G \rightarrow D$, we conjoin G onto the subgoal that would have to be solved if D matched the atomic goal, and continue by attempting to use D to derive the atomic goal.

4. If the program is atomic and equal to the atomic goal, we attempt to solve the accumulated subgoal, otherwise we backtrack by returning.

There are some obvious inefficiencies in this control structure. Some of these will be addressed in later sections.

2.3 The Initial Success Continuation

From the exposition above we can see that

```
val solve : gform -> dform -> (unit -> unit) -> unit}
```

and `solve goal prog sc` may be read as "solve `goal` in program `prog` and call `sc` if successful, otherwise return." This is not quite accurate, since if `sc` *returns*, it will be called again for every way of proving `goal` the interpreter can find.

For example, let `val psc = (fn () => print "Success!  ")`. Then `solve` p $(p \wedge p)$ `psc` will print `Success!  ` twice. On the other hand, due to the incompleteness of depth-first search, `solve` p $((p \to p) \wedge p)$ `psc` will get into an infinite loop, while `solve` p $(p \wedge (p \to p))$ `psc` will print an infinite stream of `Success!`'s.

At first this might seem like a serious limitation of this implementation technique. However, using exceptions we can prevent the initial success continuation from returning. For example, the following top-level interface would stop after the first solution is found.

```
fun one_solve goal prog =
    let exception Success
     in ( solve goal prog (fn () => raise Success) ;
          print "no " )
        handle Success => print "yes "
    end
```

It is also easy to add a query of the user that checks if more solutions are desired or not. The eLP implementation [8] plays even more tricks with the initial success continuation: it presents the first solution, but then works ahead without waiting for instructions as to whether additional solutions are required. If an externally visible side-effect is just about to be executed, it suspends. This has the advantage that we can often return to the top-level without user input if the query has only one solution.

3 First-Order Horn Logic

The interpreter from the previous section can be generalized and improved in several different directions. Before we introduce some improvements, we continue with a few generalizations. The most important and obvious step is that from a propositional to a first-order logic. This requires the introduction of *unification* and *substitution*. Interestingly, the basic structure of the interpreter stays intact: success continuations can be generalized to deal with unification and substitutions. This first version of an interpreter for first-order Horn logic requires no side-effects except for the presentation of solutions.

The definition of first-order Horn logic is again somewhat non-standard in that we do not require a clausal form.

$$
\begin{array}{rcl}
G & ::= & A \mid \top \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x\ G \\
D & ::= & A \mid \top \mid D_1 \wedge D_2 \mid G \rightarrow D \mid \forall x\ D
\end{array}
$$

We add two new cases for the quantifiers to the datatypes `gform` and `dform`, but we also have to change the definition of atomic formulas, since they now may consist of a predicate constant applied to a number of terms built from constants, function constants, and variables. Partly for reasons of simplicity and partly in preparation for a higher-order language, we do not distinguish between constants, function constants, and predicate constants. Thus atomic formulas are considered to be terms, which also means that not every `gform` or `dform` represents a well-formed formula of first-order Horn logic. However, this property can be checked statically and we thus consider it a problem for an appropriate front-end for our interpreter that is beyond the scope of this chapter. Before we go into detail in the representation of terms, here is the changed definition of formulas.

```
datatype gform  =                          (* Goal formula *)
     Gtrue                                 (* Truth *)
  |  Gand of gform * gform                 (* Conjunction *)
  |  Gor of gform * gform                  (* Disjunction *)
  |  Gatom of term                         (* Atomic G formula *)
  |  Gexists of varbind * gform            (* Existential *)

and dform =                                (* Program formula *)
     Dtrue                                 (* Truth *)
  |  Dand of dform * dform                 (* Conjunction *)
  |  Dimplies of gform * dform             (* Implication *)
  |  Datom of term                         (* Atomic D formula *)
  |  Dall of varbind * dform               (* Universal *)
```

3.1 Terms and Substitution

Our inductive definition of terms has three base cases: `Bvar` for *bound variables*, `Evar` for *logic variables*,[2] and `Const` for constants, including predicate and function constants. These variables and constants can be combined via

[2]The name is derived from *existential variables*, since logic variables may be viewed as variables that are existentially quantified in the meta-theory.

application using the `Appl` constructor, giving the representation the flavor of a *curried form*, which will aid us in the transition to the higher-order language later on, but also leads to more compact code here.

```
datatype term =
     Bvar of string              (* Bound Variables *)
  |  Evar of string * int        (* Logic Variables , Stamped *)
  |  Const of string             (* Constants *)
  |  Appl of term * term         (* Applications *)
and varbind = Varbind of string (* Variable binders *)
```

`Varbind`'s are used to bind variables at quantifiers, but seem to belong to the term language rather than the formula language are thus implemented as a separate type rather than merely by strings. Later, in Section 8 a `Varbind` will also contain the bound variable's type.

Under this representation the Prolog clause $append(nil, K, K)$ has an explicit quantifier on K and is represented as[3]

```
Dall(Varbind("K"),
  Datom(Appl(Appl(Appl(Const("append"),Const("nil")),Bvar("K")),Bvar("K"))))
```

Logic variables must be generated many times in such a way as not to conflict with previous logic variables of the same name. For example, every time a clause in Prolog is used, its free variables must be instantiated with fresh logic variables. In our setting, the quantification on the variables is explicit, since this approach lends itself more easily to later generalizations. Nonetheless, we must be able to generate new unique logic variables when instantiating universally quantified D-formulas (programs). We do this by attaching to each logic variable an integer stamp which makes it unique. We use the function `new_evar` to generate new logic variables from given variable names. In order to properly explain substitution and later unification, we will need to introduce some terminology. We say an occurrence of a `Bvar` is *loose* in a term or formula if it is not in the scope of a binding operator binding the same name. We say a term or formula is *tight* if it contains no loose `Bvar`'s, and it is *closed* if it is tight and also contains no `Evar`'s. Here are the types of some of the lower-level functions implemented below.

[3]Note that we use the conventional Prolog uncurried notation in the concrete syntax of examples for this first-order term language.

```
val new_evar : varbind -> term
val shadow : varbind -> varbind -> bool
val subst : term -> varbind -> term -> term
```

`shadow` determines if one variable binding shadows another and is used to correctly substitute in formulas such as $\forall x\ (P(x) \wedge \forall x\ Q(x))$. `subst` s x t substitutes the tight term s for all loose occurrences of x in t. The implementations are straightforward as renaming can be avoided, since t is required to contain no loose `Bvar`'s.

```
(* Externally invisible counter to create unique variables *)
local val varcount = ref 0
 in fun new_evar (Varbind(vname)) =
     ( varcount := !varcount + 1;
       Evar(vname,!varcount) )
end  (* local val varcount *)

fun shadow (Varbind(vname1)) (Varbind(vname2)) = (vname1 = vname2)

fun subst s (Varbind(vname)) t =
    let fun sb (t as Bvar(bvname)) = if vname = bvname then s else t
          | sb (Appl(t1,t2)) = Appl(sb t1,sb t2)
          | sb t = t            (* Evar , Const *)
     in sb t end
```

There are also functions that substitute in G-formulas and D-formulas. These are mutually recursive,[4] but each function changes only a subset of the arguments in the recursion. The following illustrates a general implementation technique for such a recursion structure.

```
local fun formsubst t x =
  let fun gsb (Gtrue) = Gtrue
        | gsb (Gand(g1,g2)) = Gand(gsb g1, gsb g2)
        | gsb (Gor(g1,g2)) = Gor(gsb g1, gsb g2)
        | gsb (Gexists(y,g)) =
           Gexists(y, if shadow x y then g else gsb g)
        | gsb (Gatom(s)) = Gatom(subst t vbd s)
```

[4] Actually, in this version D-formulas may not occur in G-formulas, but this will change later on.

```
      and dsb (Dtrue) = Dtrue
        | dsb (Dand(d1,d2)) = Dand(dsb d1, dsb d2)
        | dsb (Dimplies(g,d)) = Dimplies(gsb g, dsb d)
        | dsb (Datom(s)) = Datom(subst t vbd s)
        | dsb (Dall(y,d)) =
           Dall(y, if shadow x y then d else dsb d)
   in (gsb , dsb) end
in
  fun gsubst t x g =
         let val (gsb ,  _ ) = formsubst t x in gsb g end
  and dsubst t x d =
         let val ( _  , dsb) = formsubst t x in dsb d end
end
```

3.2 Unification

The basic new data structure we need is that of a *substitution*, which maps logic variables to terms. First, a section of the signature.

```
type substitution = (term * term) list
val unify : term -> term -> (substitution -> unit) -> substitution -> unit
```

The structure of `unify` is again based on the idea of a success continuation, except that we now need to communicate some information to the success continuation, namely the substitution that arises from unifying two terms. Thus `unify` *s t sc subst* unifies *s* and *t* under the substitution *subst* and applies *sc* to the resulting new substitution. This means that substitutions arising from unification are never explicitly applied, but when an `Evar` is encountered we need to see if it has been instantiated to a term by previous unifications. If unification fails, `unify` simply returns. `unify` requires the auxiliary function `lookup`, which returns the substitution term for a logical variable in a substitution, or the token `NONE`, if no such term exists.[5]

First, we show a version that implements unsound unification as used in Prolog. Omitting the occurs-check as done here may be justified by efficiency arguments, but has the undesirable side-effect that X and $f(X)$ are unifiable (where X is the variable).

[5] The frequently used type `'a option` is not part of the definition of SML but defined as `datatype 'a option = NONE | SOME of 'a`.

```
(* val lookup : term -> substitution -> term option *)
fun lookup (Evar(_,stamp)) subst =
  let fun lk nil = NONE
        | lk ((Evar(_,tstamp),t)::tail) =
            if stamp = tstamp then SOME(t) else lk tail
   in lk subst end

fun unify (s as Evar _) t sc subst = unify_evar s t sc subst
  | unify s (t as Evar _) sc subst = unify_evar t s sc subst
  | unify (Const(cname1)) (Const(cname2)) sc subst =
      if cname1 = cname2 then (sc subst) else ()
  | unify (Appl(s1,s2)) (Appl(t1,t2)) sc subst =
      unify s1 t1 (fn newsubst => unify s2 t2 sc newsubst) subst
  | unify _ _ sc subst = ()
and unify_evar e t sc subst =
      case (lookup e subst)
         of NONE => sc ((e,t)::subst)     (* Instantiate e to t, succeed *)
          | SOME(s0) => unify s0 t sc subst    (* e is instantiated to s0 *)
```

Adding the occurs-check requires a few auxiliary functions and a modification of the definition of `unify_evar`. The definitions of `occurs_in` and `same_evar` are straightforward and omitted here. The occurs-check is only called once we know that we are not trying to unify a variable with itself. The definition of `unify` remains unchanged and the new definition of `unify_evar` is

```
unify_evar e t sc subst =
      case (lookup e subst)
         of NONE => if same_evar e t subst
                       then sc subst                  (* e = e *)
                       else if occurs_in e t subst
                               then ()                (* Occurs check fails *)
                               else sc ((e,t)::subst) (* Bind e to t *)
          | SOME(s0) => unify s0 t sc subst
```

The obvious inefficiency in the structure of this function is that substitutions must be built up, and that it may be very costly to continue to look up possible substitutions terms for `Evar`'s. This can be corrected using destructive substitutions (see Section 5).

3.3 The Interpreter

The generalized version of the function `solve` now takes one additional argument (the current substitution for `Evar`'s), and the success continuation also expects to be passed a substitution.

```
fun solve (Gtrue) prog sc subst = sc subst
  | solve (Gand(g1,g2)) prog sc subst =
      solve g1 prog (fn newsubst => solve g2 prog sc newsubst) subst
  | solve (Gor(g1,g2)) prog sc subst =
      ( solve g1 prog sc subst ; solve g2 prog sc subst )
  | solve (Gatom(t)) prog sc subst =
      match_atom t prog sc subst
  | solve (Gexists(x,g)) prog sc subst =
      solve (gsubst (new_evar x) x g) prog sc subst

and match_atom t prog sc subst =
     let fun rec_match (Dtrue) subgoal = ()
           | rec_match (Dand(d1,d2)) subgoal =
               ( rec_match d1 subgoal ; rec_match d2 subgoal )
           | rec_match (Dimplies(g,d)) subgoal =
               rec_match d (Gand(subgoal,g))
           | rec_match (Datom(s)) subgoal =
               unify s t (fn newsubst => solve subgoal prog sc newsubst) subst
           | rec_match (Dall(x,d)) subgoal =
               rec_match (dsubst (new_evar x) d) subgoal
      in rec_match prog (Gtrue) end
```

Again, the question arises how we call this interpreter at the top-level. The initial success continuation will have to be slightly more complicated than before since we would like to present a substitution for the logic variables in the query. To this end we have functions

```
val project_substitution : term list -> substitution -> substitution
val print_substitution : substitution -> unit
```

`project_substitution` *evars subst* takes a list of `Evar`'s and determines their substitution terms in *subst*. This includes looking up of all the logic variables in the substitution terms that were instantiated during the unification. `print_substitution` *subst* just presents the substitution in a human

readable format. For the sake of brevity we will not show the implementation of these straightforward functions. It will also be the responsibility of the front end to ensure that all free uppercase identifiers in the original query are converted into new logic variables, and that the final substitution is projected onto these variables and then printed.

4 Hereditary Harrop Logic

We now further generalize from the first-order Horn logic to allow hereditary Harrop formulas as goals. This means that a goal can be an implication (called *embedded implication*) or a universally quantified formula (*embedded universal quantification*). For some general motivation for these constructs refer to [1, 12, 11, 21, 25]. The mutually recursive definitions of the classes of goals and programs now become

$$\begin{array}{lcl} G & ::= & A \mid \top \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x\ G \mid D \rightarrow G \mid \forall x\ G \\ D & ::= & A \mid \top \mid D_1 \wedge D_2 \mid G \rightarrow D \mid \forall x\ D \end{array}$$

The definition of `gform` is changed by adding two new cases.

```
|  Gimplies of dform * gform          (* Embedded Implication *)
|  Gall of varbind * gform            (* Embedded Universal *)
```

The operational interpretation of these new constructs follows the intuitionistic reading of implication and universal quantification.

- Given goal $D \rightarrow G$, assume D into the program and then attempt to solve G. The additional assumption is in effect only while solving G. D is added "to the beginning" of the program, which means that the most recently assumed formula is considered first when we are trying to solve an atomic goal.
- Given goal $\forall x\ G$, create a new parameter a and attempt to solve $[a/x]G$. "New" means that a is not allowed to occur in the current program or G.

As examples, consider the goals $p \rightarrow p$ (which clearly succeeds) and $\exists x\ (P(x) \rightarrow (P(1) \wedge P(2)))$, which fails (due to the intuitionistic reading of $\exists$) where the classically equivalent $(\forall x\ P(x)) \rightarrow (P(1) \wedge P(2))$ succeeds. Quantifier dependence now also becomes an issue, as one can see from the goals $\exists x\ \forall y\ (P(x) \rightarrow P(y))$ (which fails) and $\forall y\ \exists x\ (P(x) \rightarrow P(y))$ (which succeeds).

4.1 Embedded Implication

Embedded implication can be added trivially to the interpreter as we have developed it so far, since the program is an explicit parameter to the `solve` function. We just add a new case to the definition of `solve`:

```
  | solve (Gimplies(d,g)) prog sc subst =
      solve g (Dand(d,prog)) sc subst
```

4.2 Embedded Universal Quantification

Embedded universal quantifiers require much more pervasive changes, since the dependence of existential and universal quantifiers on each other now must be taken into account. In theorem provers this is typically addressed by a one-time Skolemization pass during the preprocessing stage. Here this does not seem possible (since the logic is essentially intuitionistic). Moreover, we can take advantage of special properties of hereditary Harrop formulas to obtain a more efficient implementation.

First of all, we need to update the definition of the datatype of `term` to include the case that the term is a *parameter*. In our implementation we call these parameters `Uvar`'s, thinking of them as universally quantified at the meta-level. Note that in unification they act essentially like constants, except for certain quantifier dependence considerations. The way we implement quantifier dependence is for every `Evar` to explicitly contain a list of parameters on which it may depend.

```
datatype term =
     Bvar of string              (* Bound Variables *)
  |  Evar of string * int * term list
                                 (* Logic Variables , Stamped , Depends on *)
  |  Uvar of string * int        (* Parameters , Stamped *)
  |  Const of string             (* Constants *)
  |  Appl of term * term         (* Applications *)
and varbind = Varbind of string (* Variable binders *)
```

Consider, for example, the goal $\forall x\ \exists y\ G$. First we introduce a new parameter a for x and solve $\exists y\ [a/x]G$. Then we introduce a new logic variable Y and solve $G'' = [Y/y][a/x]G$. We are free to instantiate Y with terms which contain a, that is, Y may depend on a. If, on the other hand,

our goal is $\exists y\ \forall x\ G$, we first introduce a logic variable Y for y and then a parameter a for x. Note that here Y may *not* contain occurrences of a!

In the interpreter this is implemented by adding a new argument to `solve`, namely the list of parameters (`Uvar`'s) which have been introduced so far and thus may occur in the substitution term for any logic variable (`Evar`) which is introduced subsequently. There are some additional minor interface changes. For example, the function `new_evar` must now be passed the list of `Uvar`'s on which the new `Evar` is allowed to depend on. Since the new parameter is passed along unchanged in all cases except the embedded universal quantifier, we only show this case in `solve`.

```
| solve (Gall(x,g)) prog uvars sc subst =
    let val a = new_uvar x
     in solve (gsubst a x g) prog (a::uvars) sc subst end
```

There are some further bookkeeping changes (for example, in substitution), but the crucial change now is in the unifier. More specifically, we have to extend the occurs-check to account for dependency. When the prospective substitution term `t` contains a `Uvar` on which the `Evar s` is not allowed to depend, we have to fail. However, this is not quite sufficient. Consider the problem of unifying X with $f(Y)$, where X and Y are logic variables, and X is allowed to depend only on parameter a, but Y is allowed to depend on parameters a and b. If we merely bind X to $f(Y)$, Y might later be instantiated to a term containing b, thus unwittingly violating the condition that the substitution term for X not depend on b. Thus we also need to restrict further instantiations of Y not to depend on b.

In general, all `Evar`'s Y embedded in a substitution term for an `Evar` X can depend only on the intersection of the parameters legal for X and Y. This is implemented by instantiating Y with a new `Evar` Y' whose `Uvar` list is thus restricted. We don't need to implement this in full generality due to a metatheorem: when we have to consider the lists of `Uvar`'s from two `Evar`'s during the execution of a logic program one of the lists will be an initial segment of the other.[6]

Rather than using two passes, we combine the occurs-check with the restriction of `Evar`'s. Since restriction of `Evar`'s is an instantiation process, the extended occurs-check may need to change the substitution and thus is programmed using success continuations, just as `unify` itself.

[6] This also gives rise to the even more efficient implementation used in eLP, where whole lists of parameters are represented by their upper bound (a single integer).

```
fun init_seg uvars1 uvars2 = length uvars1 <= length uvars2

fun extended_occurs_check (Evar(_,stamp1,uvars1)) t sc subst =
  let fun eoc (e as Evar(x,stamp2,uvars2)) sc subst =
            (case (lookup e subst)
               of NONE => if (stamp1 = stamp2)
                             then ()   (* fail *)
                             else if init_seg uvars2 uvars1
                                     then sc subst
                                     else sc ((e,new_evar (Varbind(x)) uvars1)
                                              ::subst)
                | SOME t0 => eoc t0 sc subst)
        | eoc (Appl(t1,t2)) sc subst =
            eoc t1 (fn newsubst => eoc t2 sc newsubst) subst
        | eoc (Uvar(_,stamp2)) sc subst =
            if exists (fn (Uvar(_,stamp1)) => stamp1 = stamp2
                        | s => raise subtype("eoc",s,"is not a Uvar"))
                      uvars1
               then sc subst
               else ()
        | eoc _ sc subst = sc subst
      in eoc t sc subst end
  | extended_occurs_check s _ _ _ =
      raise subtype("extended_occurs_check",s,"is not an Evar")
```

One more detail here is the use of a function `subtype` which generates an exception from a function name, term, and error message. The intent is that these exceptions signal an internal error, called `subtype`, because being a term, but not an `Evar` or `Uvar` constitutes a form of subtype violation (though subtypes are not supported in SML).

The extended occurs-check is called from `unify_evar`. The only subtlety here is perhaps that we have to postpone the substitution until the extended occurs-check has succeeded. Note also that we have to previously check if we are unifying an `Evar` with itself and succeed without changing the substitution—otherwise the occurs-check would fail for this case.

```
and unify_evar e t sc subst =
       case (lookup e subst)
          of NONE => if same_evar e t subst
```

```
                    then sc subst
                    else extended_occurs_check e t
                           (fn newsubst => sc ((e,t)::newsubst))
                           subst
      | SOME s0 => unify s0 t sc subst
```

5 Destructive Substitution

Of the code presented so far, the two most important optimizations will be (a) introduction of destructive substitution in order to avoid repeated lookup of the substitution terms for logic variables, and (b) the conversion of the program into clausal form in order to have ready access to the part of the program relevant to a particular predicate symbol. In this section we will deal with the first issue.

Up to now we were using almost exclusively purely functional code. Destructive substitutions will violate this principle, but in a relatively disciplined way. This means that whenever the unification algorithm instantiates a logic variable to a term it has to make provisions to undo this instantiation upon backtracking. Since backtracking is indicated simply by returning rather than calling the success continuation, this is easy to implement.[7]

Before plunging into the code, a brief word about assignment in SML. Traditionally in imperative languages we assign to variables. In SML, we assign to *references*. References are distinguished by their type (which will be `'a ref` for some type `'a`) and are created by applications of the function `ref` to a value. `ref` is also a constructor so that we can access the value stored in a location using match expressions as for usual function definitions. Thus the dereferencing operation `!` can be defined explicitly as `fun ! (ref v) = v`.

In order to implement the idea of destructive substitutions, we give `Evar`'s an additional slot that could either hold the term to which the `Evar` was instantiated, or a token indicating that the `Evar` is not instantiated. We must be able to assign to this slot, and it will thus be a reference to an optional term. Here is the updated definition of the datatype of terms.

```
datatype term =
```

[7] In Section 6 we will be forced to abandon this assumption, and will thus require a more sophisticated implementation of instantiation and uninstantiation of variables.

```
    Bvar of string          (* Bound Variables *)
  | Evar of string * int * term list * (term option) ref
                      (* Logic Variables , Stamped , Depends on , Inst'd to *)
  | Uvar of string * int    (* Parameters , Stamped *)
  | Const of string         (* Constants *)
  | Appl of term * term     (* Applications *)
```

When a new `Evar` is created, it is uninstantiated and thus contains a reference to `NONE`. Now most operations will have to *dereference* `Evar`'s if they are instantiated to a term. This is illustrated, for example, in the second clause in the definition of `unify_evar` below.[8]

The most profound changes in `unify` are that (a) it no longer requires a substitution as an argument since it instantiates variables destructively, and (b) it needs to take action to uninstantiate variables upon failure of the success continuation. Instantiation is accomplished by an assignment to the reference in the value slot of an `Evar`, uninstantiation assigns `NONE`. The instantiation is performed when the extended occurs-check succeeds and thus is passed in the success continuation to `extended_occurs_check`. The function for the extended occurs-check must also be changed in an analogous fashion.

```
fun unify (s as Evar _) t sc = unify_evar s t sc
  | unify s (t as Evar _) sc = unify_evar t s sc
  | unify (Const(cname1)) (Const(cname2)) sc =
       if cname1 = cname2 then sc () else ()
  | unify (Uvar(_,stamp1)) (Uvar(_,stamp2)) sc subst =
       if stamp1 = stamp2 then (sc subst) else ()
  | unify (Appl(s1,s2)) (Appl(t1,t2)) sc =
       unify s1 t1 (fn () => unify s2 t2 sc)
  | unify _ _ sc = ()
and unify_evar (e as Evar(_,_,_,(vslot as (ref NONE)))) t sc =
       if same_evar e t
          then sc ()
          else extended_occurs_check e t
                  (fn () => ( vslot := SOME t ; sc () ; vslot := NONE ; () ))
```

[8] An important optimization arises from the invariant that an `Evar` is never instantiated to a term with a loose `Bvar`: In the operation of substituting for a `Bvar` in a term, if the term is an instantiated `Evar`, we can simply return it without dereferencing and further traversal.

```
| unify_evar (Evar(_,_,_,ref (SOME s0))) t sc = unify s0 t sc
| unify_evar s _ _ = raise subtype("unify_evar",s,"is not an Evar")
```

The definition of G-formulas and D-formulas and the functions for substituting into formulas does not change from the previous section. The interpreter undergoes only a very minor change: since explicit substitutions are no longer required, `solve` and the success continuation both require one argument less than before.

6 Control Primitives and Trailing

So far, the only mechanism available for search control in our logic programming language has been clause ordering. There are many programs where this is insufficient and there are a number of ways one can address this deficiency. The most common construct is *cut* (`!`, in concrete Prolog syntax), though we will discuss this only briefly in Section 6.2. Our focus will be on what is commonly called *if-then-else* and written as `G1 -> G2 | G3` . In our abstract syntax we have a corresponding `guard` constructor. `guard` is general enough to allow a direct definition of the constructs `once` and `not` in a higher-order language (see Section 9), and most programs using cut can easily be transformed into programs using `guard`. We show that the `guard` control primitive can be implemented using SML exceptions without disturbing the general structure and organization of the interpreter. However, the use of exceptions relies on a non-local exit from a success continuation, which requires a different implementation of the uninstantiation of variables on backtracking. This alternative implementation technique for backtracking is referred to as *trailing*.

6.1 The `guard` Control Construct

The operational reading of $G_1 \rightarrow G_2 \mid G_3$ (not to be confused with implication) is

- Solve the guard G_1. If this succeeds, solve G_2 (with the new substitution). On backtracking, *do not* reconsider the choices made while solving G_1, but simply fail the overall goal $G_1 \rightarrow G_2 \mid G_3$.
- If solving G_1 fails, solve G_3 and fail the overall goal $G_1 \rightarrow G_2 \mid G_3$ on backtracking.

In the interpreter, we first augment the definition of the `gform` data type and modify the substitution function to include the obvious additional case. A first, as it turns out, incorrect attempt at the additional case for the interpreter (from Section 5) would be

```
| solve (Guard(g1,g2,g3)) prog sc =
    let exception Guard_success in
        ( solve g1 prog (fn () => raise Guard_success) ;
          solve g3 prog sc )
        handle Guard_success => solve g2 prog sc
    end
```

If we succeed in solving the guard `g1`, raising the exception `Guard_success` will transfer control back to the handler for the exception, bypassing all the choice points, and then solve `g2`. Choice points are established, for example, when solving a disjunction, or when descending through a conjunction when analyzing the program in `match_atom`. One can now see what is wrong with this attempt: the code in the unifier which uninstantiates variables on backtracking (the assignment of `NONE` below)

```
extended_occurs_check e t
   (fn () => ( vslot := SOME t ; sc () ; vslot := NONE ; () ))
```

will not be executed when solving `g2` eventually backtracks, because the call `sc ()` exited with an exception, bypassing the second assignment.

The solution to this problem is move the responsibility for uninstantiating variables from the unifier to the choice points. Thus, in the case of a goal $G_1 \vee G_2$, for example, we have to keep track of all the variables which may have been instantiated during an attempt to solve G_1 and uninstantiate them before attempting to solve G_2. Keeping track of these variables is the purpose of the *trail*.

A SML variable `global_trail` contains a reference to a trail. When a variable is instantiated, it is added to the trail (which is accessed as a stack). At a choice point, when the first alternative backtracks, we uninstantiate all variables which have been pushed onto the trail and simultaneously unwind the trail (that is, pop the stack). This is bundled up into a few functions and the datatype `trail`. The function `trail` is used by the interpreter at choice points: it remembers the global stack, evaluates its argument (by applying it to the unit element) and then uninstantiates all the "new" variables it finds

on the stack. The function `instantiate_evar` is used by the unifier in order to instantiate logic variables and simultaneously push them onto the trail. The variable `global_trail` is local to a context with these functions, which guarantees that no other functions can obtain access to it and change the value of the location it refers to. In order to be able to properly unwind the trail, we must have a reference (= pointer) to a trail which we can compare with the result of unwinding it. Since the only way to compare for pointer equality is by comparing references, the tail of a trail must be implemented as a reference to a trail even though is never modified.

```
datatype trail =
     consTrail of term * (trail ref)
  |  nilTrail

local val global_trail = ref (ref nilTrail)
in

fun unwind_trail shorter_trail longer_trail =
    if longer_trail = shorter_trail
       then (global_trail := shorter_trail; ())
       else (case !longer_trail of
                  (consTrail (Evar(_,_,_,vslot),rest_trail)) =>
                     (vslot := NONE; unwind_trail shorter_trail rest_trail)
                | _ => raise Subtype("unwind_trail: Ill-formed trail."))

fun trail func =
      let val old_trail = !global_trail in
          ( func () ; unwind_trail old_trail (!global_trail) ; () )
      end

fun instantiate_evar (s as Evar(_,_,_,vslot)) t =
       ( vslot := SOME t;
         global_trail := ref (consTrail(s,!global_trail)) )
  | instantiate_evar s _ = raise subtype("instantiate_evar",s,"is not an Evar")

end  (* local val global_trail *)
```

The main function of the unifier does not change, but `unify_evar` changes, since it no longer has the responsibility of uninstantiating variables upon backtracking. Thus the call to the extended occurs-check now looks like

```
 extended_occurs_check e t (fn () => ( instantiate_evar e t ; sc () ))
```

The interpreter makes use of the functional `trail` where it establishes choice points. This happens in exactly three cases: `Gor`, `Guard`, `Dand`. The argument to `trail` is protected by a vacuous abstraction in order to prohibit premature evaluation—just as success continuations.

```
fun solve (Gtrue) prog uvars sc = sc ()
  | solve (Gand(g1,g2)) prog uvars sc =
      solve g1 prog uvars (fn () => solve g2 prog uvars sc)
  | solve (Gor(g1,g2)) prog uvars sc =
      ( trail (fn () => solve g1 prog uvars sc) ;
        solve g2 prog uvars sc )
  | solve (Gatom(t)) prog uvars sc =
      match_atom t prog uvars sc
  | solve (Gexists(x,g)) prog uvars sc =
      solve (gsubst (new_evar x uvars) x g) prog uvars sc
  | solve (Gimplies(d,g)) prog uvars sc =
      solve g (Dand(d,prog)) uvars sc
  | solve (Gall(x,g)) prog uvars sc =
      let val a = new_uvar x
       in solve (gsubst a x g) prog (a::uvars) sc end
  | solve (Guard(g1,g2,g3)) prog uvars sc =
      let exception Guard_success
       in ( trail (fn () => solve g1 prog uvars (fn () => raise Guard_success)) ;
            solve g3 prog uvars sc )
          handle Guard_success => solve g2 prog uvars sc
      end

and match_atom t prog uvars sc =
     let fun rec_match (Dtrue) subgoal = ()
           | rec_match (Dand(d1,d2)) subgoal =
               ( trail (fn () => rec_match d1 subgoal) ;
                 rec_match d2 subgoal )
           | rec_match (Dimplies(g,d)) subgoal =
               rec_match d (Gand(subgoal,g))
           | rec_match (Datom(s)) subgoal =
               unify s t (fn () => solve subgoal prog uvars sc)
           | rec_match (Dall(x,d)) subgoal =
               rec_match (dsubst (new_evar x uvars) x d) subgoal
```

```
      in rec_match prog (Gtrue) end
```

6.2 Cut

Instead of the `guard` construct, we can use cut (written as `!`) as a non-logical control primitive. The operational reading of cut is

- When encountering cut as a goal, succeed. When the interpreter backtracks to this point, do not simply backtrack further, but jump past all the choice points which have been created since the immediate atomic supergoal of the cut.

The reference to the "immediate atomic supergoal" requires the addition of another argument to `solve` and `match_atom`. This additional argument is an exception which, when raised, will transfer control back to the immediate atomic supergoal. This additional argument `ctag` is merely passed along in most cases in the interpreter, so we show only the critical changes to the previous incarnation of `solve`.

```
fun solve (Gatom(t)) prog uvars ctag sc =
      let exception new_ctag
       in (match_atom t prog uvars new_ctag sc)
          handle new_ctag => ()
      end
  | solve (Gcut) prog uvars ctag sc =
      ( sc () ; raise ctag )
  ...
and match_atom t prog uvars ctag sc =
     let fun rec_match (Datom(s)) subgoal =
               unify s t (fn () => solve subgoal prog uvars ctag sc)
           ...
      in rec_match prog (Gtrue) end
```

In addition to the initial success continuation (see Section 2.3) we now also need to create an initial exception to pass to `solve`. This is easily accomplished by

```
fun top_solve goal free_vars prog =
```

```
let exception top_ctag
 in (trail (fn () => solve goal prog nil top_ctag
                          (fn () => print_substitution free_vars)))
    handle top_ctag => ()
end
```

where `free_vars` is the list of variables (`Evar`'s) free in `goal`. Trailing is necessary so that `top_solve` does not have a side-effect on the variables among `free_vars` which are instantiated during the call to `solve`. Now we can also see how a Prolog-like top-level can be implemented: the initial success continuation could present the substitution and then require user input. If the user types a semi-colon "`;`" it returns, and otherwise is raises the exception `top_ctag`.

7 Clausal Form

One of the problems with the interpreter so far is the inefficiency of the program analysis. We would like to restrict the search for potentially applicable assumptions as much as possible. Here, the clausal form theorem for Horn logic (and hereditary Harrop logic) is helpful: any legal D-formula is equivalent to one in *clausal form*. The clausal form is defined by

$$\begin{array}{rcl} D & ::= & \top \mid C \mid C \wedge D \\ C & ::= & G \to A \mid \forall x\, C \end{array}$$

where C is a clause, A (referred to as the *clause head*) stands for an atomic formula, and G (referred to as the *clause body*) stands for a G-formula as before. Atomic formulas have the form $P(t_1, \ldots, t_n)$ for a predicate symbol P and terms $t_1, \ldots, t_n$. We call P the *head* of A and, more generally, f the *head* of a term of the form $f(t_1, \ldots, t_n)$, including the cases where $n = 0$. We refer to the head of the clause head of a clause C as the *head predicate* of C. Given an atomic goal A with head P, the interpreter, that is, `match_atom`, can only succeed in applying a clause if its head predicate is also P.

Thus we can represent an arbitrary program as a list of clauses, and store with each clause its head predicate, for direct comparison with the head of an atomic goal. In a first step, the program is searched clause by clause for one with a matching head predicate. A straightforward optimization that we do not discuss here further, stores a list of clauses relevant to each head predicate in a hash table indexed by the head predicate. This can be carried

even further by "indexing" on the head function symbol of one or more of the predicate arguments.

Recall that in our implementation atomic formulas are represented by terms, since this simplifies the code. We now add the definition of the datatype of `clause`:

```
datatype clause = Clause of head * varbind list * term * gform
```

`head` is a new type exported in the implementation of terms: it is the type of legal heads. Up to and including this section, a head can be only a constant and can be implemented simply as its name. The `varbind list` is the list of the universally quantified variables in the clause, `term` is the clause head, and `gform` is the clause body. Together with this we have a function that converts an arbitrary formula into clausal form. `clausify` carries three accumulator arguments: the body, the universally quantified variables, and a list of clauses. Thus `clausify` D `(Gtrue) nil nil` will convert a D-formula D into clausal form.

```
fun gand_opt (Gtrue,g) = g
  | gand_opt (g,Gtrue) = g
  | gand_opt (g1,g2) = Gand (g1,g2)

fun clausify Dtrue _ _ rest = rest
  | clausify (Dand(d1,d2)) body vars rest =
       clausify d1 body vars (clausify d2 body vars rest)
  | clausify (Dall(x,d)) body vars rest =
    if exists (fn y => shadow x y) vars
       then let val (new_x,sb) = rename_sb x
             in clausify (dapply_sb sb d) body (new_x::vars) rest end
       else clausify d body (x::vars) rest
  | clausify (Dimplies(g,d)) body vars rest =
       clausify d (gand_opt (body,g)) vars rest
  | clausify (Datom(t)) body vars rest =
       Clause(head t,vars,t,body) :: rest
```

The function `gand_opt` eliminates some `Gtrue` subgoals. Bound variables may have to be renamed during the conversion to clausal form (consider, for example, the clausal form of $\forall x\ (P\,x \rightarrow \forall x\ (Q\,x \rightarrow R\,x))$). `rename_sb` returns the new variable name and also a renaming substitution. This notion

of substitution is different from the one discussed in Section 3: here we substitute for `Bvar`'s rather than for `Evar`'s. The new version of `match_atom` below takes a list of clauses, instead of a D-formula. It uses a function `new_evar_sb`, which takes a list of bound variables and returns a substitution that, when applied, substitutes new `Evar`'s for all the `Bvar`'s. Note that the body of a clause is not copied (that is, substituted into) until the unification of the atomic goal with the clause head has succeeded.

```
match_atom t clauses uvars sc =
    let val t_head = head t
        fun rec_match nil = ()
          | rec_match ((clause as Clause(s_head,vars,s,gbody))::rest) =
              if head_equal s_head t_head
                 then let val nesb = new_evar_sb vars uvars in
                       ( trail (fn () =>
                           unify (apply_sb nesb s) t (fn () =>
                            solve (gapply_sb nesb gbody) clauses uvars sc)) ;
                         rec_match rest )
                      end
                 else rec_match rest
     in rec_match clauses end
```

We could store the head of atoms in the atomic formula, to avoid the call to `head`. Along similar lines, we could statically convert D-formulas which appear on the left-hand sides of embedded implications into clauses rather than convert them at assumption time. This is an important optimization, but it requires substitution functions into the clausal representation which we would like to avoid in the presentation. Thus the case for embedded implication in `solve` looks as follows:

```
| solve (Gimplies(d,g)) prog uvars sc =
   solve g (clausify d (Gtrue) nil prog) uvars sc
```

Before, functions such as `gsubst` substituted for a single (bound) variable in order to achieve clause copying. Calls to this are now replaced with calls to `gapply_sb`, which achieves the more efficient simultaneous substitution. Checking of variable name conflicts is still avoided, except in a rare case during conversion of programs to clausal form. Changes to the corresponding substitution functions on formulas are straightforward, though with an

interesting twist. When descending through a quantified formula, we augment the substitution by adding a pair substituting the bound variable for itself. Noting that this still cannot introduce "capturing," we rewrite the substitution as follows.

```
type sb = (varbind * term) list

exception Loose_Bvar of term

(* val lookup_vbind : string -> sb -> term *)
fun lookup_vbind vname sb =
    let fun lk ((Varbind(xname),t)::rest) =
               if vname = xname then t else lk rest
          | lk nil = raise Loose_Bvar(Bvar(vname))
     in lk sb end

(* val apply_sb : sb -> term -> term *)
fun apply_sb sb s =
    let fun asb (Bvar(vname)) = lookup_vbind vname sb
          | asb (Appl(s1,s2)) = Appl((asb s1),(asb s2))
          | asb t = t           (* Evar , Uvar , Const *)
     in asb s end
```

Remember that substitution due to unification is done destructively and not by the function above.

8 Higher-order Terms

So far we have been working with a first-order, untyped term language. We will now make the transition to a higher-order, typed term language. This necessitates handling constraints in the interpreter (a simple change) and a major change in unification, which may now branch and is no longer guaranteed to terminate.

8.1 The Interpreter

The modules defining propositions and the interpreter need to change very little. The primary change is that we have to introduce *constraints*, since

the higher-order unification algorithm generates constraints, that is, sets of equations which are known to be satisfiable. Though somewhat more general through our use of types, implication, and explicit quantification, our language now becomes a constraint logic programming language in the sense of Jaffar and Lassez [19]. The way constraints are handled in the interpreter is reminiscent of the way we handled substitutions in Section 3.3 before the introduction of destructive instantiation of variables: where the success continuation previously expected a substitution `subst`, it now expects a constraint `con` as an argument.

```
fun solve (Gtrue) clauses uvars con sc = sc con
  | solve (Gand(g1,g2)) clauses uvars con sc =
      solve g1 clauses uvars con (fn newcon => solve g2 clauses uvars newcon sc)
  | solve (Gor(g1,g2)) clauses uvars con sc =
      ( trail (fn () => solve g1 clauses uvars con sc) ;
        solve g2 clauses uvars con sc )
  | solve (Gatom(M)) clauses uvars con sc =
      match_atom M clauses uvars con sc
  | solve (Gexists(x,g)) clauses uvars con sc =
      solve (gapply_sb (term_sb x (new_evar x uvars)) g) clauses uvars con sc
  | solve (Gimplies(d,g)) prog uvars con sc =
      solve g (clausify d (Gtrue) nil prog) uvars con sc
  | solve (Gall(x,g)) prog uvars con sc =
      let val a = new_uvar x
       in solve (gapply_sb (term_sb x a) g) prog (a::uvars) con sc end
  | solve (Guard(g1,g2,g3)) clauses uvars con sc =
      let exception Guard_success of constraint
       in ( trail (fn () =>
                solve g1 clauses uvars con (fn newcon =>
                   raise Guard_success(newcon))) ;
            solve g3 clauses uvars con sc )
          handle Guard_success(newcon) => solve g2 clauses uvars newcon sc
      end

and match_atom M clauses uvars con sc =
     let val M_head = head M
         fun rec_match nil = ()
           | rec_match ((clause as Clause(N_head,vars,N,gbody))::rest) =
               if head_equal N_head M_head
                  then let val nesb = new_evar_sb vars uvars
```

```
                        in ( trail (fn () =>
                                unify (apply_sb nesb N) M con (fn newcon =>
                                   solve (gapply_sb nesb gbody) clauses
                                         uvars newcon sc)) ;
                             rec_match rest )
                       end
                 else rec_match rest
      in rec_match clauses end
```

8.2 Representing Higher-Order Terms

For convenience, we use a single representation type `term` for both terms and types in our calculus. If we were only interested in implementing a logic programming language over the simply typed λ-calculus without polymorphism this would be unnecessarily complicated, but we are interested in including dependent types (in the form of LF [14]) and polymorphism, at which point it is convenient to have to write only one function each for substitution and unification, rather than two (one for unifying terms and one for unifying types, for example). The algorithm we outline below will be complete only for certain fragments of the full calculus, but we can now implement various subcalculi merely by changing the type checking phase and the set of predeclared constants in the front end. This basic approach, though different in various details, is taken in the Calculus of Constructions [4]. The representation of terms is perhaps more direct, but less efficient than deBruijn indices [5] which are used in eLP and almost all other modern implementations of λ-calculi which require access to the internal structure of λ-terms. Nonetheless, we were quite surprised how little of the code depends on the choice of the representation of bound variables.

```
datatype term =
     Bvar of string              (* Bound Variables *)
  |  Evar of varbind * int * term list * (term option) ref
                         (* Logic Variables , Stamped , Depends on , Inst'd to *)
  |  Uvar of varbind * int       (* Parameters , Stamped *)
  |  Const of string             (* Constants *)
  |  Appl of term * term         (* Applications *)
  |  Abst of varbind * term      (* Abstractions *)
and varbind = Varbind of string * term  (* Variable binders , Type *)
```

In the implementation of the term language and the type checker, we have two constants `type` and `pi`. And, yes, `type` is a type, though this could be avoided by introducing universes (see [16]) without any changes to the code of the unifier. As is customary, we use $A \to B$ as an abbreviation for $\Pi x : A.\, B$ if x does not occur free in B. Also, however, $\Pi x : A.\, B$ is an abbreviation for the application $\mathtt{pi}\, A\, (\lambda x : A.\, B)$. In our formulation, then, the constant `pi` has type $\Pi A{:}\mathtt{type}.\, ((A \to \mathtt{type}) \to \mathtt{type})$.

As an example consider a predicate constant `eq` of type $\Pi A{:}\mathtt{type}.\, A \to A \to \mathtt{o}$ (where `o` is the type of formulas as indicated in Section 9). The single clause $\mathtt{eq}\, A\, M\, M.$ correctly models equality, that is, a goal of the form $\mathtt{eq}\, A\, M\, N$ will succeed if M and N are unifiable. The fact that unification now has to branch can be seen by considering the goal $\mathtt{eq}\, \mathtt{int}\, (F\, \mathtt{1}\, \mathtt{1})\, \mathtt{1}$ which has three solutions for the functional logic variable F, namely $\lambda x{:}\mathtt{int}.\, \lambda y{:}\mathtt{int}.\, x$, $\lambda x{:}\mathtt{int}.\, \lambda y{:}\mathtt{int}.\, y$, and $\lambda x{:}\mathtt{int}.\, \lambda y{:}\mathtt{int}.\, \mathtt{1}$.

The functions supporting substitution are extended in the obvious way. In particular, we now have to substitute inside `Varbind`'s, since they contain terms which may contain free variables. The type of a constant is accessible in a *signature* which maps names of constants to their types, implemented, for example, as a list of pairs of strings and types.

The unification procedure we use is based on the one in [6, 7] for higher-order unification with dependent types, which itself is an extension of Huet's procedure for (higher-order) unification in the simply typed λ-calculus [18]. This procedure is most easily understood in terms of a collection of "transformations," some on terms, some on pairs of terms being unified, and some on sets of such pairs.

With the right control structure (such as iterative deepening) the unifier would be complete for the LF fragment of our calculus, but logic variables ranging over types destroy this completeness. The unifier detects if there is a possibility for incompleteness on a particular execution and can give a warning in such a case, if desired.

8.3 Rewriting

Because we are using transformations as a fundamental structuring device in the implementation of unification, we adopted a very elegant technique from Paulson's higher-order implementation of rewriting [27], which itself was patterned after the tactics and tacticals in LCF [13]. Because we are not

worried about having our implementation *prove* the correctness of applications of its transformations, we can use a somewhat simpler implementation than in [27]. The basic kind of object we deal with we call a *rewriter*, which is simply a *partial* function from some type to itself—"partial" because it may fail to apply (which is communicated by a raised exception) as well as fail to terminate. Thus we have simply

```
type 'a rewriter = 'a -> 'a
```

The exception `Fail` may be raised in case a rewriter fails to apply and takes string as an argument which is intended but not required to give some indication of reason for the failure of the rewriter to apply. Here are the general rewriting primitives we found useful.

```
exception Fail of string

fun rew_and rew1 rew2 = rew2 o rew1

fun rew_or rew1 rew2 x = (rew1 x) handle Fail _ => (rew2 x)

fun rew_id x = x

fun rew_try rew = rew_or rew rew_id

fun rew_repeat rew x = rew_try (rew_and rew (rew_repeat rew)) x

(* rew_first: 'a rewriter -> 'a list rewriter *)
fun rew_first rew nil = raise Fail("rew_first")
  | rew_first rew (x :: l) = (rew x) :: l

(* rew_rest: 'a list rewriter -> 'a list rewriter *)
fun rew_rest list_rew nil = nil
  | rew_rest list_rew (x :: l) = x :: (list_rew l)
```

8.4 Unification

The basic structure of the unifier involves maintaining a collection of pairs of terms to be unified simultaneously. Traditionally such pairs are called *disagreement pairs* and such a collection is called a *disagreement set* (though represented and used as a list).

```
datatype dpair = Dpair of term * term
type dset = dpair list
```

During unification, we transform terms, disagreement pairs, and disagreement sets, as described below. However, because some unification problems have more than just a single most general unifier, we will also have one branching step. This will be implemented using success continuations, thus meshing nicely with the interpreter.[9]

8.4.1 Normalization

One kind of rewriting we will need to do during unification is "weak head normalization" of terms. This just means normalizing enough to determine the top-level structure of the β-normal form of the term.[10] In mathematical notation, a weak head reduction step reduces a term of the form $(\lambda x{:}A.\, M)\, N_1 \ldots N_n$ to the term $([N_1/x]M)\, N_2 \ldots N_n$.[11] Thus the rewriter below fails, if the given term does not have the form of the left-hand side of this rewriting rule (modulo dereferencing of instantiated `Evar`'s).

```
fun head_reduce_term (Appl(M,N)) =
     (let fun hrt (Abst(xofA,M0)) = apply_sb (term_sb xofA N) M0
            | hrt (Evar(_,_,_,ref(SOME M0))) = hrt M0
            | hrt _ = Appl(head_reduce_term M,N)
       in hrt M end)
  | head_reduce_term (Evar(_,_,_,ref(SOME M0))) = head_reduce_term M0
  | head_reduce_term _ = raise Fail("head_reduce_term")
```

The conventions for naming ML variables in the code for the unifier are as follows: we use `M` and `N` for terms and `A`, `B`, and `C` for types.[12] Furthermore, we use `xofA` and `yofB` for `Varbind`'s (a pair consisting of a variable name and its type) and `Gamma` for contexts (lists of `Varbind`'s).

Next, we raise this from a term rewriter to a disagreement pair rewriter that tries to head reduce the left member of the pair and, if this fails, tries

[9] Success continuations can be used for rewriting as well, but they do not seem as appropriate in this context with only a very simple form of nondeterminism (succeed with the rewritten term or fail).

[10] More specifically, "weak" here refers to not doing any normalization inside of abstractions.

[11] Application associates to the left, so with parentheses the redex would be $(\ldots((\lambda x{:}A.\, M)\, N_1) \ldots N_n)$.

[12] "Types" are simply terms used in the capacity of types.

to reduce the right member. The functional `rew_dpair` does this kind of raising:

```
fun rew_dpair rew =
      rew_or (fn (Dpair(M,N)) => Dpair(rew M, N))
             (fn (Dpair(M,N)) => Dpair(M, rew N))

val head_reduce_dpair = rew_dpair head_reduce_term
```

8.4.2 Extensionality

The next transformation involves a disagreement pair made up of one or two abstraction terms. This can be justified by an extensionality principle or the η-rule. For example, consider unifying $\lambda x{:}A.\,M$ and $\lambda y{:}B.\,N$. In this case we introduce a new parameter a and reduce the problem to unifying $[a/x]M$ and $[a/y]N$. There are two similar cases in which only one of terms being unified is an abstraction where we simply form an application.

```
fun abst_reduce_dpair (Dpair(Evar(_,_,_,ref(SOME M0)),N)) =
      abst_reduce_dpair (Dpair(M0,N))
  | abst_reduce_dpair (Dpair(M,Evar(_,_,_,ref(SOME N0)))) =
      abst_reduce_dpair (Dpair(M,N0))
  | abst_reduce_dpair (Dpair(Abst(xofA,M0), Abst(yofA,N0))) =
      let val a = new_uvar xofA
       in Dpair(apply_sb (term_sb xofA a) M0,
                apply_sb (term_sb yofA a) N0)
      end
  | abst_reduce_dpair (Dpair(Abst(xofA,M0), N)) =
      let val a = new_uvar xofA
       in Dpair(apply_sb (term_sb xofA a) M0, Appl(N,a))
      end
  | abst_reduce_dpair (Dpair(M,Abst(yofA,N0))) =
      let val a = new_uvar yofA
       in Dpair(Appl(M,a),apply_sb (term_sb yofA a) N0)
      end
  | abst_reduce_dpair _ = raise Fail("abst_reduce_dpair")
```

8.4.3 Rigid pairs

We now focus on the case of unifying two terms, neither of which is subject to weak head reduction and neither of which is an abstraction. Of these, the simplest case is when each of the two terms is "rigid", *i.e.*, its top-level structure does not change under substitution. This is the case when the head of a term is either a constant or a `Uvar`. The treatment in this case is to compare heads. If they are the same, we replace the disagreement pair with the corresponding pairs of arguments. Otherwise, we conclude that the two terms are non-unifiable, and hence the disagreement set containing them is non-unifiable as well.

To distinguish non-unifiability from non-applicability of a rewriter, we introduce a new exception with `exception Nonunifiable`. Rather than extracting the heads and lists of arguments first, in the implementation below, we accumulate disagreement pairs matching up the corresponding arguments while descending to the heads. When we get to a head, we either succeed or fail.

```
(* rigid_rigid : dpair -> dset -> dset,
   Adds result of rigid-rigid decomposition to the given dset.
   Fails if the dpair is not rigid-rigid.
   Can raise the exception Nonunifiable. *)

fun rigid_rigid (Dpair(Evar(_,_,_,ref (SOME M0)), N)) dset =
      rigid_rigid (Dpair(M0,N)) dset
  | rigid_rigid (Dpair(M, Evar(_,_,_,ref (SOME N0)))) dset =
      rigid_rigid (Dpair(M,N0)) dset
  | rigid_rigid (Dpair(Appl(M1,N1),Appl(M2,N2))) dset =
      (* Note the "head-recursion" *)
      rigid_rigid (Dpair(M1,M2)) (Dpair(N1,N2)::dset)
  | rigid_rigid (Dpair(Const(name1),Const(name2))) dset =
      if name1 = name2 then dset else raise Nonunifiable
  | rigid_rigid (Dpair(Uvar(_,stamp1),Uvar(_,stamp2))) dset =
      if stamp1 = stamp2 then dset else raise Nonunifiable
    (* Otherwise, either not rigid-rigid or unification fails. *)
  | rigid_rigid (Dpair(M,N)) _ =
      if (is_rigid M) andalso (is_rigid N)
         then raise Nonunifiable
         else raise Fail("rigid_rigid")
```

It is a simple matter to make this function into a disagreement set rewriter that attempts to apply `rigid_rigid` to the first disagreement pair in a disagreement set:

```
(* rigid_rigid_rew : dset rewriter *)
fun rigid_rigid_rew nil = raise Fail("rigid_rigid_rew")
  | rigid_rigid_rew (dp :: rest) = rigid_rigid dp rest
```

8.4.4 SIMPL

We can now assemble the previous rewriters into a main component of the unification algorithm, corresponding to Huet's "SIMPL" phase. One step of the SIMPL phase is accomplished by the following disagreement set rewriter, which tries first head reduction, then (if that fails to apply) extensionality, and finally rigid-rigid decomposition.

```
val SIMPL_rew =
     rew_or (rew_or (rew_first head_reduce_dpair)
                    (rew_first abst_reduce_dpair))
            rigid_rigid_rew
```

The complete SIMPL phase repeats `SIMPL_rew` as long as it applies, transforming the first disagreement pair, and then recursively works on the remaining disagreement pairs.

```
fun SIMPL ds =
      rew_and (rew_repeat SIMPL_rew)
              (* if SIMPL_rew fails, (rew_repeat SIMPL_rew) succeeds and
                 either we have run out of disagreement pairs and are done,
                 or the first is no longer rigid-rigid and we need to go on. *)
              (rew_rest SIMPL)
              ds
```

8.4.5 MATCH

The SIMPL phase leads either to non-unifiability or to a disagreement set made up completely of disagreement pairs relating two flexible (non-rigid) terms or one rigid and one flexible term. As in Huet's algorithm, we defer

treating flexible-flexible disagreement pairs, as their treatment leads to an intractable explosion of the search space, or, if a particular solution is chosen, to an overcommitment which must be avoided in this setting of constraint logic programming. At least in the LF sub-calculus, these disagreement sets are always unifiable when they arise [6].

The "MATCH" phase examines a flexible-rigid disagreement pair, instantiates the logic variable at the head of the flexible term, and calls the success continuation on an augmented disagreement set. Upon backtracking, further instantiations may be tried. Once all possibilities are exhausted MATCH returns. The interested reader can consult [6] for an explanation and justification of these instantiations, as well as completeness proofs. Actually, completeness can in general only be guaranteed for a subcalculus where logic variables do not occur at the head of types. As logic variables ranging over types are extremely useful in practice (they provide for polymorphism), and the algorithm will often be complete even in this case, enforcing this restriction statically is counter-productive. Instead, we give a run-time warning if the unification procedure might be incomplete.

A full discussion of MATCH is beyond the scope of this chapter; we merely show some fragments of the code illustrating its control structure.

```
(* MATCH : dpair -> dset -> (dset -> unit) *)

fun MATCH (Dpair(flex,rigid)) ds sc =

 let val F as Evar(Varbind(Fname,Ftype),_,ok_uvars,_) = flex_term_head flex
     ...
     (* Try a substitution given a term and its type *)
     (* Given M and A such that  |- M : A
        instantiate F to M and constrain  Ftype == A
        (not necessary in the simply typed lambda-calculus) *)
     fun try_subst_term M A =
            ( instantiate_evar F M ;
              sc (Dpair(Ftype, A) :: ds) )

     ...
  in
     (* project_from and imitate enumerate the possible substitution terms for
        F, calling try_subst_term on each; expecting it to return upon
        backtracking. *)
```

```
    ( trail (fn () => imitate ()) ; project_from Gamma_w )
 end
```

There is no distinction in λProlog or our hypothetical language between choice points established during unification and choice points established during clause selection. Thus trailing must be done at choice points during unification, as in the interpreter.

8.4.6 Putting it together

Now we combine the various pieces into the function `unify_dset`. It expects a disagreement set and a success continuation, which is to be called on a possibly remaining constraint, if the disagreement set is unifiable. If not, we simply return to signify failure and initiate backtracking. Constraints are nothing but disagreement sets with only flexible-flexible pairs.

```
type constraint = dset

(* unify_dset : dset -> (constraint -> unit) -> unit *)
fun unify_dset ds sc =
      (* First SIMPL.  This may raise exception Nonunifiable handled below. *)
      let val ds' = SIMPL ds
       in (* ds' will have only flex-rigid, rigid-flex, or flex-flex pairs.
             Select a flex-rigid or rigid-flex and call MATCH.
             If there is none, we succeed with the remaining constraints.
             MATCH returns upon backtracking. *)
          case (find_flex_rigid ds')
            of SOME(dp) => MATCH dp ds' (fn match_ds => unify_dset match_ds sc)
             | NONE => sc ds'  (* Success: only flex-flex left *)
      end
      handle Nonunifiable => ()  (* Failure in SIMPL *)

(* unify : term -> term -> constraint -> (constraint -> unit) -> unit *)
fun unify M N dset sc = unify_dset (Dpair(M,N) :: dset) sc
```

9 Higher-Order Logic

The language of higher-order terms introduced so far has not changed the underlying logic of our logic programming language: it is still a first-order

logic, though over a very rich domain. This is sufficient for many application programs (see, for example, [10]), but there are instances where it is very elegant and natural to allow true higher-order programming. By *higher-order programming* we mean the ability of programs to construct other programs (to be assumed) and other goals (to be invoked). In Prolog some semblance of such a facility is provided through the `call` primitive. Here we have a higher-order term language, and, following Church [3], we introduce a distinguished type of propositions (`o`) and constants representing the logical quantifiers and connectives. For example, logical conjunction is represented by a constant `and` of type $\texttt{o} \to \texttt{o} \to \texttt{o}$, and the existential quantifier is represented by a constant `exists` of type $\Pi A{:}\texttt{type}.\,((A \to \texttt{o}) \to \texttt{o})$.

Two simple examples of higher-order programs in the sense given above are clauses defining `once` and `not` (negation-as-failure), given the more general `guard` construct we adopted in Section 6. $\bot$ is simply a goal which always fails.

$\texttt{once}(G) \leftarrow (G \to \top \mid \bot).$
$\texttt{not}(G) \leftarrow (G \to \bot \mid \top).$

Here G is a variable of type `o` and `once` and `not` are constants of type $\texttt{o} \to \texttt{o}$. Note that the argument G will be passed as a term, but has to be converted to a goal before it can be invoked.

If we implemented strictly higher-order hereditary Harrop formulas (see [22]), some of the work below would not be necessary, but in practice it is important not to restrict *statically* to D-formulas and G-formulas whose predicate symbol is fixed and known at the time where terms are translated into propositions. Instead we introduce a new case in the definitions of the datatypes `gform` and `dform`, namely `Gflex of term` and `Dflex of term`, respectively. They convey that we do not yet know whether this term will be atomic, a conjunction, *etc.*, since it begins with a predicate variable which might be instantiated by unification *before* the current G-formula or D-formula is needed by the interpreter.

The functions `term_to_gform` and `term_to_dform` (not shown here) translate a term to a formula. They proceed by converting the term to head normal form and then deciding from the constant at the head of the term if it is a conjunction, disjunction, *etc.* If it matches none of the logical constants, it is either atomic (if the head is rigid) or "flex" if the head is an `Evar` or a `Bvar`. The quantifiers are represented as constants applied to abstractions.

All that is required to complete the interpreter is to modify the clausifi-

cation function to allow `Dflex` formulas and the interpreter to allow `Gflex` formulas. In either case, the argument is again converted to a proposition. If the formula remains flexible, an error results. Recall that flexible formulas may become rigid through instantiation during unification. Consider, for example, $\exists x(x = \top \wedge x)$. At translation time, x is flexible, but by the time the goal x is encountered, it has been instantiated to $\top$, which should simply succeed. We thus add the following case to solve:

```
  | solve (Gflex(M)) clauses uvars con sc =
      (case term_to_gform M
         of Gflex(M') => raise error("solve: Goal " ^ term_makestring(M')
                                     ^ " with variable head predicate.")
          | g => solve g clauses uvars con sc)
```

and a corresponding case to clausify

```
 | clausify (Dflex(M)) body vars rest =
      (case term_to_dform M
         of Dflex(M') => raise error("clausify: Program " ^ term_makestring(M')
                                     ^ " with variable head predicate.")
          | d => clausify d body vars rest)
```

10 Conclusion

The interpreter we have developed is relatively close to eLP, our Common Lisp implementation of λProlog in the Ergo Support System [8, 20]. Most of the differences have already been mentioned: clauses are indexed and stored in a global hashtable, bound variables are represented by deBruijn indices and parameters and logic variables are time-stamped for comparison and unification of terms that have not been Skolemized. There are also the front-end, that is, parsing, unparsing, and type reconstruction, the implementation of λProlog's module system, and "special" (non-logical) predicates which we ignored in this presentation, some of which are by no means trivial. Unification also differs: the one given here supports a much richer λ-calculus, but does not implement a number of important optimizations (see [23]) used in eLP. Finally, there are a number of design mistakes which are still part of the current eLP implementation which we chose not to expose here.

We expect that the next set of significant improvements in the implementation techniques for λProlog and related languages will come from a more economical representation of λ-terms [26] and the development of compilation technology [24].

We conclude with the remark that the complete Standard ML code for all versions of the interpreter discussed here including a modest front end are available via `ftp` over the Internet.[13]

References

[1] Anthony J. Bonner, L. Thorne McCarty, and Kumar Vadaparty. Expressing database queries with intuitionistic logic. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 831–850, Cambridge, Massachusetts, 1989. MIT Press.

[2] Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2(4):347–359, 1984.

[3] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[5] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[6] Conal Elliott. *Extensions and Applications of Higher-order Unification.* PhD thesis, Carnegie Mellon University, May 1990. Available as Technical Report CMU–CS–90–134, Carnegie Mellon University, Pittsburgh.

[7] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, pages 121–136. Springer-Verlag LNCS 355, April 1989.

[8] Conal Elliott and Frank Pfenning. eLP: A Common Lisp implementation of λProlog in the Ergo Support System. Available via ftp over the

[13] Please send mail to the second author at `fp@cs.cmu.edu` for more information.

Internet, October 1989. Send mail to elp-request@cs.cmu.edu on the Internet for further information.

[9] Matthias Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana University, Bloomington, Indiana, October 1985.

[10] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989.

[11] D. M. Gabbay. N-prolog: an extension of Prolog with hypothetical implications II. *Journal of Logic Programming*, 2(4):251–283, 1985.

[12] D. M. Gabbay and U. Reyle. N-prolog: an extension of Prolog with hypothetical implications I. *Journal of Logic Programming*, 1(4):319–355, 1985.

[13] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[14] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. Submitted. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987, January 1989.

[15] Robert Harper, Robin Milner, Kevin Mitchell, Nick Rothwell, and Don Sannella. Functional programming in Standard ML. Notes to a five day course given at the University of Edinburgh, April 1988.

[16] Robert Harper and Robert Pollack. Type checking, universe polymorphism, and typical ambiguity in the Calculus of Constructions. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 241–256. Springer-Verlag LNCS 352, March 1989.

[17] Christopher T. Haynes. Logic continuations. *Journal of Logic Programming*, 4(2):157–176, June 1987.

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

[19] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich*, pages 111–119. ACM, January 1987.

[20] Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. The Ergo Support System: An integrated set of tools for prototyping integrated environments. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 25–34. ACM Press, November 1988. Also available as Ergo Report 88–054, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[21] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):57–77, January 1989.

[22] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Journal of Pure and Applied Logic*, 1988. To appear. Available as Ergo Report 88–055, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[23] Dale A. Miller. Unification under mixed prefixes. Unpublished manuscript, 1987.

[24] Gopalan Nadathur and Bharat Jayaraman. Towards a WAM model for lambda Prolog. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 1180–1198. MIT Press, October 1989.

[25] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[26] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 Conference on Lisp and Functional Programming*. ACM Press, June 1990. To appear.

[27] Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programing*, 3:119–149, 1983.