LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - The University of Edinburgh

Formal Program Development in Modular Prolog: A Case Study

by

M G Read
E A Kazmierczak

LFCS Report Series ECS-LFCS-92-195

LFCS January 1992

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Formal Program Development in Modular Prolog : A Case Study

M. G. Read
E. A. Kazmierczak*

Department of Computer Science
University of Edinburgh
Edinburgh, EH9 3JZ
eka@uk.ac.ed.lfcs

Abstract

In this paper we present a case study in which we specify and *formally* develop a modular Prolog program from its specification. The modular Prolog which we use is that proposed in [SW87] which is based on the modules system of Standard ML [HMT90, MT90]. We give a specification language for writing specifications of modular Prolog programs and also a methodology, based on that of Extended ML, for the *stepwise* construction of a program from its specification. The case study is intended to examine the Extended ML methodology applied to the formal development of Prolog programs and to assess the feasibility and outline the potential difficulties of this approach.

1 Introduction

In this paper we present a *model* oriented approach to formally developing PROLOG programs. The approach is based upon that of the Extended ML wide-spectrum language [ST89, San89, ST91]. In the Extended ML approach a specification denotes a class of models or possible programs. The final program must be a member of this class. This program is not given in a single implementation step but is *constructed* from the original specification via a number of smaller *program development* steps. At each step some design choices are made, for example, choosing the modular structure of the final program or giving an actual algorithm to realize some aspect of the specification. To guarantee that each step results in a *correct* refinement of the original specification proof obligations are associated with each step and must be *formally* discharged.

This style of program development is independent of the particular logic and programming language used and only depends upon a definition of what it means for a

*This research was supported by SERC grant GR/E 78463

program module to satisfy its specification (written in the specification logic in question). The question that we ask is if this style of program development is suited to other programming languages, for example, PROLOG.

The purpose of this paper is to describe a case study showing just how program development using the Extended ML approach may be put into practice in developing modular PROLOG programs. We begin by describing a modular version of PROLOG and a language for specifying modular PROLOG programs. We then present our case study which is the formal development of a program to test confluence of rewrite rules [HO80, Klo87].

The reason for choosing PROLOG is that it, like Standard ML [HMT90, MT90], has a semantics which can be formalized [Llo84]. This is required for the kind of formal development which is presented here because of the requirement that proof obligations need to be formally discharged. Formally discharging a proof obligation means knowing exactly when a program satisfies (or, is a model of) a specification.

The reason for choosing the term rewriting example is so that the example is sufficiently large to naturally lead to a modular PROLOG program via several program development steps. Consequently it is a good test for the development methodology. It was also chosen because of its relevance to other work on decomposing term rewriting systems into modules, for example, [Les89, Les90].

The remainder of this paper is organized as follows. Section 2 describes the module language and program development methodology in more detail. In section 3 we present the major results about term rewriting which we use and in section 4 our requirements specification and one program development step are presented. We conclude in section 5.

2 Specifying Modular Prolog Programs

2.1 PROLOG with Modules

The target language which we use is PROLOG enhanced with module constructs similar to those of Standard ML [SW87]. The modules system which we use consists of three components: *structures*, *signatures* and *functors*. Structures are program modules and program clauses in a structure are "executable". They consist of a two basic elements: (1) data declarations and (2) predicate and function definitions.

An example of a structure is given in figure 1. The data declaration in figure 1 is given by:

```
fun zero : 0
fun suc  : 1
```

which declares two function constants `zero` with arity 0 and `suc` with arity 1. Predicates may also be declared as follows:

```
pred le : 2
```

```
structure Elements =
  struct
     fun zero : 0
     fun suc  : 1

     le(zero,X).
     le(suc(X),suc(Y)) :- le(X,Y).
  end.
```

Figure 1:

```
structure Sort =
  struct
     structure Elem = Elements

     insert(X,[],[X]).
     insert(X,[Y|Z],[X,Y|Z]) :- Elem/le(X,Y).
     insert(X,[Y|Z],[Y|W]) :- not(Elem/le(X,Y)),
                              insert(X,Z,W).

     sort([],[]).
     sort([X],[X]).
     sort([X|Y],Z) :- sort(Y,W),
                      insert(X,W,Z).
  end.
```

Figure 2:

but this is always used in signatures for the purpose of making some predicates visible while hiding others. Function declarations introduce the language which is available for constructing terms while predicate and function declarations introduce the language for constructing (atomic) formulae. In the structure `Elements` the predicate `le` uses terms built up only from the function constants `zero` and `suc`. Another example is the structure in figure 2 which provides a sorting predicate based upon a substructure `Elem`.

Signatures are interfaces to structures. They specify which components of the structure are externally visible but hide the details of the code. An example is given in figure 3. The structure `Elements` *matches* the signature `ELEM` precisely because it contains a predicate definition `le` and function constants `zero` and `suc` as required by the signature. If the signature `ELEM` in 3 is ascribed to the structure `Elements` then this is written as:

```
structure Elements : ELEM =
  struct
    fun zero : 0
```

```
signature ELEM =
  sig
    fun zero : 0
    fun suc  : 1

    pred le : 2
  end.
```

Figure 3:

```
signature SORT =
  sig
    structure Elem : ELEM
    pred sort : 2
  end.
```

Figure 4:

```
    fun suc : 1

    le(zero,X).
    le(suc(X),suc(Y)) :- le(X,Y).
  end.
```

Signatures may also specify substructures. Any structure matching the signature `SORT` in figure 4 must also contain a substructure `Elem` with the signature `ELEM`. The structure `Sort` of figure 2 matches the signature `SORT` because it contains a predicate `sort` of arity 2 and a substructure `Elem` which will match the signature `ELEM`. Note that the predicate `insert` is hidden by the signature `SORT`.

Functors are parameterized modules. They accept structures as arguments and return structures as results. For example, in figure 2 the substructure `Elem` is assigned to `Elements` but if we wish to have a generic sorting structure which will sort elements from any structure which includes a `le` predicate then we may use a functor as in figure 5.

2.2 Specifying Structures and Functors

In practice PROLOG programs use many extra logical features such as cut. Also the ordering of clauses in a program is important for termination. Here we assume only pure PROLOG with negation and no extra-logical features or cut as the language in which programs will be written.

To specify pure PROLOG programs we make the following additions, along the same lines as Extended ML [San89, ST89], to the language outlined above.

```
functor Sort(X:ELEM):SORT =
  struct
    structure Elem = X

    . . .

   end.
```

Figure 5:

```
signature ELEM =
  sig
    fun zero : 0
    fun suc  : 1

    pred le : 2

    axiom forall x => le(x,x)
    axiom forall x => forall y => forall z =>
          le(x,y) & le(y,z) -> le(x,z)
  end;
```

Figure 6: A signature specifying a pre-order

1. Axioms, written in First Order Predicate logic, are allowed in signatures and structures. This means that the same module constructs are used to structure both PROLOG programs and their specifications. Axioms are written using the following notation: `&` (conjunction), `|` (disjunction), `->` (implication), `not` (negation) and `<->` (equivalence).

2. Signatures must be ascribed to every interface of a structure or functor. This was not necessary in our examples of 2.1 but in specification interfaces provide the description of the behaviour of modules from which they to be developed.

3. Requirements specifications are given by adding a `?` in place of actual structure or functor bodies.

Signatures with axioms specify classes of PROLOG structures, for example, consider the signature in figure 6. The class of structures which will now match the signature `ELEM` will be all those with at least two function constants, `zero` and `suc`, and a predicate `le` which is a pre-order. The structure in figure 2 certainly matches this signature.

As well as just the *flat* specifications like the pre-order in figure 6 specifications may exhibit internal structure. Substructures and *local* or auxiliary predicates and axioms can be used in specifications. The signature in figure 7 uses both the substructure `Elem` and three auxiliary predicates, `member`, `permutation` and `ordered` to specify sorting.

```
signature SORT =
  sig
    structure Elements : ELEM

    pred sort : 2

    local

      pred member : 2

      axiom forall x => not(member(x,[]))
      axiom forall x,y => forall l =>
            member(x,[x|l])
      axiom forall x,y => forall l =>
            member(x,l) -> member(x,[y,l])

      pred permutation : 2

      axiom forall l => forall l' =>
            permutation(l,l') <->
            (forall x =>
             member(x,l) <-> member(x,l'))

      pred ordered : 1

      axiom ordered([])
      axiom forall a => ordered([a])
      axiom forall a => forall b => forall l =>
            Elem/le(a,b) & ordered([b|l]) -> ordered([a,b|l])
    in
      axiom forall l,l' =>
            permutation(l,l') & ordered(l')
            <-> sort(l,l')
    end
  end;
```

Figure 7: A signature with substructures and hidden functions

```
functor Sort(X:ELEM):sig
                       include SORT
                       sharing X = Elements
                     end = ?
```

Figure 8: Specification of a Sorting Functor

A statement of the programming task can now be given as in figure 8. This is the *requirements specification* from which further program development takes place. What is required in this case is a parameterized module which accepts any structure with a predicate `le` and function constants `zero` and `suc` and returns a sorting module for lists of that data.

The phrase `include SORT` includes all the axioms and definitions in the signature `SORT`. The phrase `sharing X = Elements` introduces a *sharing constraint.* Sharing constraints are part of the module language for PROLOG [SW87] and impose the constraint that two structures be built up in exactly the same way. In specifications like the requirements specification of figure 8 a sharing constraint expresses the fact that the result *depends* upon the input[1].

In figure 8 for example, the sharing constraint specifies that the predicate `Elements/le` in `SORT` is the same as the predicate `X/le` in the (actual) parameter. Without this sharing constraint the predicate `Elements/le` need not be the same as that of the parameter `X`, and so the requirements specification would not explicitly require us to sort lists of data from the module `X`.

2.3 Stepwise Development of PROLOG Programs

One proceeds from a requirements specification to a program by a series of development steps. Each development step results in a program which is *correct* (in the sense described below) with respect to the results of the previous development step if all the proof obligations associated with that step are *formally* discharged. We may think of each development step as filling in some detail left open in the previous step, for example, providing a predicate definition for some predicate which hitherto has only been specified using axioms. Once the results of a development step includes no axioms and all the predicates are defined by PROLOG predicate definitions then the development process is complete. If all the proof obligations have been discharged then this final program satisfies the original requirements specification by *construction.*

There are three possible kinds of development steps in the Extended ML methodology which we also use to stepwise refine our modular PROLOG programs [San89].

Functor Decomposition

Intuitively functor decomposition is used to break a task into subtasks. Suppose

[1]This is a form of dependent type. See for example [ST91].

we are given the following specification:

$$functor\ F(X : \Sigma) : \Sigma' = ?$$

The first of the development steps allows us to define the functor F in terms of the composition of a number of other functors, for example, in the simple case of two new functors G and H we have:

$$functor\ F(X : \Sigma) : \Sigma' = G(H(X))$$

where

$$functor\ G(Y : \Sigma_G) : \Sigma'_G = ?$$
$$functor\ H(Z : \Sigma_H) : \Sigma'_H = ?$$

and Σ_H, Σ'_H, Σ_G and Σ'_G are all appropriately defined signatures. The task of finding a solution to F has been broken up into the subtasks of finding solutions to G and H. This decomposition is *correct* if:

1. all structures matching the parameter signature of F also match the parameter signature of H, that is, $\Sigma \models \Sigma_H$[2];
2. all structures matching the result signature of H can be used as an argument for G, that is, $\Sigma'_H \models \Sigma_G$;
3. all structures matching the result signature of G also match the result signature of F, that is, $\Sigma'_G \models \Sigma'$.

The development of the functors H and G may now proceed separately.

Coding

Given a specification of the form:

$$structure\ A : \Sigma = ?$$

or

$$functor\ F(X : \Sigma) : \Sigma' = ?$$

coding is used to replace the ? by an actual structure body to give

$$structure\ A : \Sigma = strexp$$

or in the case of functors

$$functor\ F(X : \Sigma) : \Sigma' = strexp$$

A *coding* development step is *correct* if

$$strexp \models \Sigma$$

in the case of structures and

$$\Sigma \cup strexp \models \Sigma'$$

in the case of functors. A structure body need not be all PROLOG code. Indeed the possibility of fixing only some design details exists since axioms are allowed within structure bodies. An example is the functor in figure 9 which could be one stage in the development of the sorting functor of figure 8.

[2]The notation $\Sigma \models \Sigma_H$ is to be read as Σ "matches" Σ_H and is defined in section 2.4

```
functor Sort(X:ELEM):sig
                       include SORT
                       sharing X = Elements
                     end =
  struct
     structure Elem = X

     pred insert : 2

     axiom insert(x,[],[x])
       and forall x,y => forall l =>
           Elem/le(x,y) -> insert(x,[y|l],[x,y|l])
       and forall x,y,z => forall l =>
           not(Elem/le(x,y)) & insert(x,l,z) -> insert(x,[y|l],[y|z]).

     sort([],[]).
     sort([X],[X]).
     sort([X|Y],Z) :- sort(Y,W),
                      insert(X,W,Z).
  end.
```

Figure 9:

Refinement Refinement is the third kind of development step used to fill in design choices left open by a coding step or by another refinement step. Given a functor of the form:

$$functor\ F(X : \Sigma) : \Sigma' = strexp$$

we can replace *strexp* by $strexp'$ in a refinement step to give:

$$functor\ F(X : \Sigma) : \Sigma' = strexp'$$

A refinement step is *correct* if

$$\Sigma \cup strexp' \models strexp$$

The rules for coding structures are similar.

2.4 Matching

So far we have not explicitly defined *matching* in the sense of a structure matching a signature. For the purposes of "matching" PROLOG programs are considered to be equivalent to their *predicate completions* [Cla78].

Let *strexp* be a structure and *sigexp* be a signature. The rules for matching, written $strexp \models sigexp$, are defined as follows:

1. the set of function constants in *strexp* must be equal to the set of function constants in *sigexp* and the arity of each function constant in *strexp* must be the same as the corresponding function constant in *sigexp*;

2. the predicate symbols of *sigexp* must be a subset of the predicate symbols in *strexp*;

3. the axioms, including the predicate completions of any programs, of *strexp* must entail the axioms of *sigexp*.

The rules for matching signatures $sigexp_1 \models sigexp_2$ are identical.

The restriction on sets of function constants in point 1. above is necessary because quantifiers range over the terms defined by function constants and these need to be identical in both *strexp* and *sigexp*. This is important for proofs involving existential quantifiers.

3 Term Rewriting and the Knuth Bendix Theorem

The subject of our case study is the formal development of a module to test the confluence of sets of rewrite rules and so below we review some of the important definitions of term rewriting.

Let Σ be a (universal algebra) signature, X a set of variables, $T_\Sigma(X)$ the set of all terms which can be constructed using the operator symbols in Σ and the variables in X and σ and ρ be two terms in $T_\Sigma(X)$. A *rewrite rule* [Bun83, HO80, Klo87] is an ordered pair of terms, which we write $\sigma \rightarrow \rho$ (σ "rewrites to" ρ), such that the variables of ρ are a subset of the variables of σ. The following is now taken from [HO80, Klo87].

Definition 1 *Let B be a set of rewrite rules and $\phi : X \rightarrow T_\Sigma(Y)$ be any substitution. The* one step rewriting relation $\rightarrow_B$ *defined by B is inductively defined as follows:*

1. *if $\sigma \rightarrow \rho \in B$ then $\phi(\sigma) \rightarrow_B \phi(\rho)$ where by abuse of notation we assume that ϕ is extended to Σ terms;*

2. *if $\phi_1(x) = t_1$, $\phi_2(x) = t_2$ and $\forall y.\, y \neq x \Rightarrow \phi_1(y) = \phi_2(y) = y$ and $t_1 \rightarrow_B t_2$ then for any term t, $\phi_1(t) \rightarrow_B \phi_2(t)$;*

3. *if $t_1 \rightarrow_B t_2$ then $\phi(t_1) \rightarrow_B \phi(t_2)$.*

The reflexive and transitive closure of $\rightarrow_B$ is written $\rightarrow_B^*$.

Definition 2 $\rightarrow_B^*$ *is* confluent *if*

$$\forall t \in T_\Sigma(X).\quad t \rightarrow_B^* t_1 \wedge t \rightarrow_B^* t_2 \Rightarrow \exists t'.t_1 \rightarrow_B^* t' \wedge t_2 \rightarrow_B^* t'$$

The Newman theorem [New42] is often used to reduce the problem of testing a set of rewrite rules for confluence to the problem of testing the set of rules for *local confluence* and *termination.*

Definition 3 *A set of rewrite rules is* locally confluent *if*

$$\forall t \in T_\Sigma(X). \quad t \rightarrow_B t_1 \wedge t \rightarrow_B t_2 \Rightarrow \exists t'.t_1 \rightarrow_B^* t' \wedge t_2 \rightarrow_B^* t'$$

Definition 4 *A set of rewrite rules is* terminating *if for no term $t \in T_\Sigma(X)$ there exists an infinite string of one step reductions*

$$t \rightarrow_B t_1 \rightarrow_B t_2 \rightarrow_B \cdots$$

Theorem 5 (Newman) *A set of rewrite rules is confluent if and only if it is terminating and locally confluent.*

One procedure for testing local confluence relies on finding *critical pairs* [KB70, HO80, Bun83].

Definition 6 *If $\sigma_1 \rightarrow \rho_1$ and $\sigma_2 \rightarrow \rho_2$ are two rewrite rules such that there exists a unifier ϕ of σ_2 and some subterm μ of σ_1 then a* critical pair *for $\sigma_1 \rightarrow \rho_1$ and $\sigma_2 \rightarrow \rho_2$ is defined to be*

$$\langle \phi(\sigma_1)[\mu \leftarrow \phi(\rho_2)], \phi(\rho_1) \rangle$$

We use the notation $\sigma[\mu \leftarrow \rho]$ to mean the term σ with the subterm μ replaced by ρ. The Knuth-Bendix theorem now gives a means for testing local confluence of set of rewrite rules under the assumption that the rewrite rules are terminating.

Theorem 7 (Knuth-Bendix) *Let* B *be a set of rewrite rules. If for all critical pairs $\langle P, Q \rangle$ in* B *there exists a term R such that $P \rightarrow^* R$ and $Q \rightarrow R^*$ then* B *is locally confluent.*

Testing the termination property is not as straightforward. The basic technique which we assume is based on the concept of a *termination* ordering [Der82, Klo87]. The basic idea is to define an ordering on terms in the language such that if $t \rightarrow t'$ then $t' \leq t$. If for every rewrite rule in a set of rewrite rules the left hand side is greater than the right hand side and the ordering on terms is well founded and satisfies some additional closure properties then the set of rewrite rules is terminating.

4 A Case Study: Testing the Confluence of a Set of Rewrite Rules

We wish to now give a requirements specification for a program that will test a set of rewrite rules for confluence. Rather than starting with the specification of confluence as

$$\forall t, t_1, t_2.t \rightarrow_B^* t_1 \wedge t \rightarrow_B^* t_2 \Rightarrow \exists t'.t_1 \rightarrow_B^* t' \wedge t_2 \rightarrow_B^* t'$$

we begin with the simpler problem of testing for the confluence of a set of rewrite rules by using the Newman theorem 5.

```
functor Confluence(R:REWRITES):
            sig
              include CONFLUENCE
              sharing R = Rewrites
            end = ?
```

Figure 10:

4.1 The Requirements Specification

The requirements specification is given in figure 10 while the signatures for the requirements specification are given in appendix A.

The parameter to this functor is a structure which manipulates Σ terms with the predicates in the signature `REWRITES`. Intuitively the predicates do the following:

isrule(r,rs) succeeds if `r` is a rewrite rule in the set `rs` of rewrite rules;

lhs : 2 returns the left hand side of a rule;

rhs : 2 returns the right hand side of a rule;

subexp(e,e') succeeds if `e` is a subexpression of `e'`;

unify:3 unifies two terms and returns the most general unifier;

replsubexp(e1,e2,e3,e4) replaces the subexpression `e2` in `e1` by `e3` returning `e4`;

apply:3 applies a substitution to a term;

rewrite(e,e',rs) performs a 1 step rewrite of `e` to `e'` using one of the rules in `rs`.

The signature `REWRITES` also contains a number of auxiliary predicates, for example, `member:2`, `modify:4` and `subst:4`.

The result signature is given in appendix A.2. The only predicate definition which is returned by the functor `Confluence` is `confluent` and the substructure `Rewrites`.

The sharing constraints expresses that the substructure `Rewrites` must be the same as the actual parameter and so also states that the axioms and predicates in the result signature are *dependent* on the actual parameter. The programming task is now to provide a functor body which gives a program to implement `confluent`.

4.2 A Program Development Step

We now wish to construct a PROLOG program from the requirements specification in figure 10 by using the methodology outlined in section 2.3. Indeed there are only two choices of program development step to apply to the functor in figure 10, that is, functor decomposition or coding. We choose a functor decomposition as in figure 11

```
functor Confluence(R:REWRITES):
            sig
              include CONFLUENCE
              sharing R = Rewrites
            end =
        Local(Noetherian(R))
```

Figure 11:

```
functor Local(N:NOETHERIAN) :
        sig
          include LOCAL_CONFLUENCE
          sharing N/Rewrites = Rewrites
        end = ?

functor Noetherian(R:REWRITES):
        sig
          include NOETHERIAN
          sharing R = Rewrites
        end = ?
```

Figure 12:

where the requirements specifications for the functors `Local` and `Noetherian` are given in figure 12. The signature `NOETHERIAN` specifies termination in terms of a termination ordering `po`.

This decomposition gives rise to only one non-trivial proof obligation:

$$LOCAL\_CONFLUENCE \models CONFLUENCE$$

What needs to be shown is that the axioms for the predicate `noetherian` in `LOCAL_CONFLUENCE` are sufficient to prove the properties of `noetherian` in `CONFLUENCE`. We have only given an informal argument for the correctness of this decomposition step [Rea] and it remains to give a formal proof of it. At this point we have used only first order logic in our specifications and so formally discharging this proof obligation would require a proof system for first order predicate logic.

Much harder is showing that parts of the PROLOG code satisfy their specification. In this case we would require two additional features in the proof system:

1. reasoning with negation and the order of clauses in a logic program;
2. induction rules for reasoning about data inductively defined by function constants.

A means of meeting the second requirement is to augment a proof system for the first order predicate calculus by the appropriate inductive rules for data types, for example,

for lists we have:

$$\frac{P(nil/x) \quad \forall a \forall l.P(l/x) \Rightarrow P([a|l]/x)}{\forall t.P(t)}$$

for any predicate P.

5 Conclusions and Further Work

In this paper we have presented a language for structuring PROLOG programs, a language for specifying structured PROLOG programs, a methodology for *formally* deriving programs from specifications of modular programs and briefly outlined a case study in the use of the specification language and the program development methodology.

This case study has shown up at least two areas in which further work is required:

1. the area of proof systems for discharging proof obligations and especially for rules dealing with negation and ordering of axioms (see for example [And89] where some work has been done on the latter topic);

2. a formal definition of the specification language is required to establish exactly the class of programs which can be models of a specification.

More generally in the area of discharging proof obligations is the problem of proving theorems in structured specifications [FC89]. A definition of the language would need to specify precisely what program modules denote and what it means for them to satisfy a specification.

The benefits to be gained by adopting this approach are precisely those that can be gained by using a modular approach to program design. Also once a program module has been constructed from its requirements specification it can be re-used. All that is known about a module is given in the interfaces and the details of the code are hidden from the user.

To make the approach feasible and practical for specifying and constructing PROLOG programs from specifications the major hurdle appears to be in discharging proof obligations. With the advent of the proper tools and proof systems this burden ought to be eased and this is one of the goals of research into Extended ML. If this can be done then this approach gives a simple method of constructing PROLOG programs to meet specifications with all the benefits of a modular specification and target language.

Acknowledgements

The authors would like to thank Don Sannella for his valuable comments on earlier drafts of this paper and also Terry Stroup, Steffan Kahrs and James Harland for listening to and commenting on our ideas.

References

[And89] J. H. Andrews. Proof-theoretic characterisations of logic programming. Technical Report ECS-LFCS-89-77, University of Edinburgh, 1989.

[Bun83] A. Bundy. *The computer modelling of mathematical reasoning.* Academic Press, 1983.

[Cla78] K. L. Clark. Negation as failure. In H Gallaire and J Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[Der82] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17:279 – 301, 1982.

[FC89] J. Farrés-Casals. Proving Correctness of Constructor Implementations. In *1989 Symp. on Mathematical Foundations of Computer Science, LNCS 379*, pages 225 – 235. Springer-Verlag, 1989.

[HMT90] R. Harper, R. Milner, and M. Tofte. *The Definition of Standard ML.* MIT Press, 1990.

[HO80] G. P. Huet and D. C. Oppen. Equations and Rewrite Rules. In Ronald V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349 – 405. Academic Press, 1980.

[KB70] D. E. Knuth and P. B. Bendix. Simple problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 203 – 297. Pergammon Press, 1970.

[Klo87] J.W. Klop. Term Rewriting Systems : A Tutorial. *Bulletin of the EATCS*, pages 144 – 192, June 1987.

[Les89] Pierre Lescanne. Implementation of Completion by Transition Rules + Control: ORME. In *TAPSOFT'89, LNCS 351*, pages 262 – 269. Springer - Verlag, 1989.

[Les90] P. Lescanne. Completion Procedures as Transition Rules + Control. In *Algebraic and Logic Programming, Springer LNCS 463*, pages 28 – 41. Springer - Verlag, 1990.

[Llo84] J. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1984.

[MT90] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press, 1990.

[New42] M. H. A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2), April 1942.

[Rea] M. G. Read. Formal Development of Prolog Programs. 4th year project report, University of Edinburgh, May 1991.

[San89] D. Sannella. Formal Program Development in Extended ML for the Working Programmer. Technical Monograph ECS-LFCS-89-102, Laboratory for the Foundations of Computer Science, December 1989.

[ST89] D. Sannella and A. Tarlecki. Toward Formal Development of ML Programs: Foundations and Methodology - Extended Abstract. In *Proceedings of the Colloquium on Current Issues in Programming Languages, LNCS 352*, pages 375 – 389. Springer Verlag, 1989.

[ST91] D. Sannella and A. Tarlecki. A Kernel Specification Formalism with Higher Order Parameterization. In *7th Workshop on Specification of Abstract Data Types*. Lecture Notes in Computer Science, to appear. Springer Verlag, 1991.

[SW87] D. T. Sannella and L. A. Wallen. A Calculus for the Construction of Modular Prolog Programs. In *IEEE 4th Symp. on logic programming*, 1987.

A Signatures for the Requirements Specification

A.1 The Signature REWRITES

```
signature REWRITES =
  sig
    pred isrule:2, lhs:2, rhs:2, subexp:2
    pred unify:3, apply:3, rewrite : 3
    pred replsubexp : 4

    fun var : 1
    fun rule : 2, op : 2

    local
      pred member : 2

      axiom forall x => not(member(x,[]))
      axiom forall x => forall l => member(x,[x|l])
      axiom forall x,y => forall l =>
            member(x,l) -> member(x,[y|l])

      pred modify : 4

      axiom forall e1,e2 => modify([],e1,e2,[])
      axiom forall e,e1,e2 => forall l =>
            e \== e1 -> modify([e|l],e1,e2,[e|l])
      axiom forall e,e1,e2 => forall l =>
            e == e1 -> modify([e|l],e1,e2,[e2|l])

      pred subst : 4

      axiom forall v,x,y =>
            v \== x -> subst(var(x),v,y,var(x))
      axiom forall v,x,t =>
            v == x -> subst(var(v),v,t,t)
      axiom forall v,x,t => forall l,l' =>
            subst(op(x,[]),v,t,op(x,[]))
      axiom forall e,e',v,x,t => forall l,l' =>
            subst(e,v,t,e') &
            subst(op(x,l),v,t,op(x,l')) ->
                  subst(op(x,[e|l]),v,t,op(x,[e'|l']))

      pred FV : 2
```

```
  axiom forall x => FV(var(x),[var(x)])
  axiom forall x => FV(op(x,[]),[])
  axiom forall x => forall t => forall terms =>
        FV(t,l1) & FV(terms,l2) & append(l1,l2,l) ->
        FV(op(x,[t|terms]),l)

  pred dom : 2

  axiom dom([],[])
  axiom forall l,l' => forall v => forall t =>
        dom(l,l') -> dom([(v,t)|l],[v|l'])

  pred disjoint : 2

  axiom forall l,l' => forall x =>
        disjoint(l,l') <->
        member(x,l) -> not(member(x,l'))

in

  axiom forall r,rs =>
        member(r,rs) -> isrule(r,rs)

  axiom forall e => lhs(rule(e,_),e)
  axiom forall e => rhs(rule(_,e),e)

  axiom forall e => subexp(e,e)
  axiom forall e,e' => forall l =>
        subexp(e,e') & member(e',l) ->
               subexp(e,op(_,l))

  axiom forall e,e' => replsubexp(e,e,e',e')
  axiom forall e,e' => forall l,l' =>
        (exists e1,e2 =>
           member(e1,l) &
           replsubexp(e1,e,e',e2) &
           modify(l,e1,e2,l'))
        -> replsubexp(op(x,l),e,e',op(x,l'))

  axiom forall e => apply([],e,e)
  axiom forall e,e' => forall v,t => forall l =>
        forall fv => forall s,s' =>
        FV(t,fv) &
        dom([(v,t)|s],s') &
        disjoint(fv,s') &
        subst(e,v,t,e') &
        apply(s,e',e'')  ->
```

```
                apply([(v,t)|z],e,e'')

    axiom forall e,e',t => forall u =>
          apply(u,e,t) &
          apply(u,e',t) ->
                unify(e,e',u)

    axiom forall e,e' => forall rs =>
          (exists r,lr,rr,rr' => exists e'' =>
           exists phi =>
           lhs(r,lr)
           & subexp(e'',e)
           & apply(phi,lr,e'')
           & rhs(r,rr)
           & apply(phi,rr,rr')
           & replsubexp(e,e'',rr',e'))
             -> rewrite(e,e',rs)
  end
end
```

A.2 The Signature CONFLUENCE

```
signature CONFLUENT =
  sig
    pred confluent:1

    structure Rewrites : REWRITES

    local

      pred normal_form : 2

      axiom forall e => forall rs =>
            normal_form(e,rs) <->
            not(exists t => Rewrites/rewrite(e,t,rs))

      pred critical_pair : 3

      axiom forall p,q,rs =>
            exists theta =>
            exists r1,r2,lhs1,lhs2,rhs1,rhs2 =>
            exists sub1,sub1',lhs1',lhs2' =>
                Rewrites/isrule(r1,rs) &
                Rewrites/isrule(r2,rs) &
                Rewrites/lhs(r1,lhs1) &
                Rewrites/lhs(r2,lhs2) &
                Rewrites/rhs(r1,rhs1) &
```

```
                Rewrites/rhs(r2,rhs2) &
                Rewrites/subexp(sub1,lhs1) &
                Rewrites/unify(sub1,lhs2,theta) &
                Rewrites/apply(theta,lhs1,lhs1') &
                Rewrites/apply(theta,lhs2,lhs2') &
                Rewrites/apply(theta,sub1,sub1') &
                Rewrites/apply(theta,rhs1,p) &
                Rewrites/replsubexp(lhs1',sub1',rhs2',p) ->
                        critical_pair(p,q,rs)

    pred reduces : 3

    axiom forall e => reduces(e,e)
    axiom forall rs => forall e,e' =>
          (exists e'' =>
           exists r   =>
                Rewrites/isrule(r,rs) &
                Rewrites/rewrite(e,e'',r) &
                reduces(e'',e',rs)) ->
                   reduces(e,e',rs)

    pred locally_confluent : 1

    axiom forall rs =>
          (forall p,q =>
           critical_pair(p,q,rs) ->
            (exists e =>
             reduces(p,e,rs) & reduces(q,e,rs)) )
          -> locally_confluent(rs)

    pred noetherian : 1

    axiom forall rs =>
          ( forall e =>
            exists e' =>
              reduces(e,e',rs) & normal_form(e',rs) )
          -> noetherian(rs)

  in

    axiom forall rs  =>
          noetherian(rs) &
          locally_confluent(rs) ->
            confluent(rs)
  end
end
```

B The Signatures for the Program Development Steps

B.1 The Signature LOCAL_CONFLUENCE

```
signature LOCAL_CONFLUENCE =
  sig
    pred locally_confluent : 1, confluent : 1
    pred reduces : 3

    structure Noetherian : NOETHERIAN

    local

      pred critical_pair : 3

      axiom forall p,q,rs =>
            exists theta =>
            exists r1,r2,lhs1,lhs2,rhs1,rhs2 =>
            exists sub1,sub1',lhs1',lhs2' =>
               Rewrites/isrule(r1,rs) &
               Rewrites/isrule(r2,rs) &
               Rewrites/lhs(r1,lhs1) &
               Rewrites/lhs(r2,lhs2) &
               Rewrites/rhs(r1,rhs1) &
               Rewrites/rhs(r2,rhs2) &
               Rewrites/subexp(sub1,lhs1) &
               Rewrites/unify(sub1,lhs2,theta) &
               Rewrites/apply(theta,lhs1,lhs1') &
               Rewrites/apply(theta,lhs2,lhs2') &
               Rewrites/apply(theta,sub1,sub1') &
               Rewrites/apply(theta,rhs1,p) &
               Rewrites/replsubexp(lhs1',sub1',rhs2',p) ->
                        critical_pair(p,q,rs)

      pred reduces : 3

      axiom forall e => reduces(e,e)
      axiom forall rs => forall e,e' =>
            (exists e'' =>
             exists r   =>
                  Rewrites/isrule(r,rs) &
                  Rewrites/rewrite(e,e'',r) &
                  reduces(e'',e',rs)) ->
                     reduces(e,e',rs)

    in
      axiom forall rs =>
```

```
        (forall p,q => critical_pair(p,q,rs) ->
          (exists r => reduces(p,r,rs) & reduces(q,r,rs)) )
         -> locally_confluent(rs)

    axiom forall rs =>
          Noetherian/noetherian(rs) & locally_confluent(rs)
          -> confluent(rs)
  end
end
```

B.2 The Signature NOETHERIAN

```
signature NOETHERIAN =
  sig
    pred noetherian : 1

    structure Rewrites : REWRITES

    local

      pred po : 2

      axiom forall x => po(x,x)

      axiom forall x,y,z =>
            po(x,y) & po(y,z) -> po(x,z)

      axiom exists x => forall y => po(x,y)

      axiom forall e, e' => forall rs =>
            Rewrites/reduces(e,e',rs) <-> po(e,e')

      axiom forall e,e' => forall t,t' =>
            po(e,e') &
            Rewrites/subexp(e,t) &
            Rewrites/subexp(e',t') ->
              po(t,t')

      axiom forall theta => forall t,t' =>
            forall e,e' =>
            po(t,t') &
            Rewrites/apply(theta,t,e) &
            Rewrites/apply(theta,t',e') ->
              po(e,e')
    in

      axiom forall rs => forall r => forall lr,rr =>
```

```
          Rewrites/isrule(r,rs) &
          Rewrites/lhs(r,lr) &
          Rewrites/rhs(r,rr) &
          po(rr,lr)
          -> noetherian(rs)
  end
end
```