In *Proc. Int. Workshop on Programming Language Implementation and Logic Programming*, Orléans, May 1988. Springer LNCS 348, pp. 273-282

Formal Specification of a Prolog Compiler

Michael Hanus
Fachbereich Informatik, Universität Dortmund
D-4600 Dortmund 50, W. Germany
(uucp: michael@unidoi5)

This paper presents an outline of a formal specification of a compiler and a virtual machine for the programming language Prolog. The specification of the compiler can be transformed into an equivalent Prolog program. The specification of the virtual machine is the basis of an implementation of the virtual machine as an interpreter written in a low-level language. Moreover, the specification may be used for correctness proofs of a Prolog system.

1 Introduction

Many compilers for the programming language Prolog are based on the so-called "Warren Abstract Machine" (WAM) presented in [Warren 83]. The WAM is a virtual machine with specific operations for executing logic programs (unify operations, indexing for clauses etc.). The execution of a Prolog program is done in two steps in a WAM-based implementation:

1. Compilation of the program into a sequence of WAM instructions.
2. Execution of the WAM program by an interpreter or compilation of the WAM program into another language, e.g., machine language.

Although there are several formal descriptions of the operational semantics of Prolog and Prolog interpreters [Arbab/Berry 87] [Debray/Mishra 88] [Deransart/Ferrand 87] [Jones/Mycroft 84] [Nilsson 84], there is no formal specification of the translation of Prolog programs into WAM programs and the semantics of the WAM. [Warren 83] is a good foundation for the implementation of the compiler and the WAM. But there are many problems if we go into details, because it is only a proposal for the implementation of pure Prolog (without cut, disjunction and non-logical predicates) based on an informal description. From a practical point of view a formal specification is very useful for an implementation of a Prolog compiler, because it is a precise description of the implementor's tasks and the interfaces between different modules. On the other hand, it is necessary for a correctness proof of the Prolog implementation. [Kursawe 86] has shown that the unify instructions of a WAM program can be derived from the source program by partial evaluation, but his method covers only a small part of the WAM.

In the following we present an outline of a full specification of the translation of Prolog programs into WAM programs and the operational semantics of the WAM. These specifications are the basis of a Prolog system developed at the University of Dortmund (see [ProCom 87]). We assume the reader is familiar with the basic concepts of the WAM (for details, the reader is referred to [Warren 83]).

2 Specification of the Prolog compiler

The compilation of Prolog programs into WAM programs is a complex task [Van Roy 84] and therefore it is desirable to divide the specification into several subtasks. We use the ideas of attribute coupled grammars [Ganzinger/Giegerich 84] and specify the compilation as the composition of translations between abstract syntax trees:

$$AST_1 \longrightarrow AST_2 \dashrightarrow AST_{n\text{-}1} \longrightarrow AST_n$$

AST_1 is the abstract syntax tree corresponding to the given clauses of a predicate and AST_n is the abstract syntax tree corresponding to the translated WAM program. The signatures of the intermediate trees $AST_2, ..., AST_{n-1}$ are enriched with informations that are computed in a preceding translation step and may be used in a subsequent translation step. We give some examples from the compiler specification.

The signature of the abstract syntax of Prolog programs is specified in the following way:

sorts: *Prog*, *Clauses*, *Clause*, *Terms*, *Term*, *String*
operations:

`prog:`	*Clauses*	$\rightarrow$	*Prog*
`csequ:`	*Clause*, *Clauses*	$\rightarrow$	*Clauses*
`empty_csequ:`		$\rightarrow$	*Clauses*
`clause:`	*Term*, *Term*	$\rightarrow$	*Clause*
`tsequ:`	*Term*, *Terms*	$\rightarrow$	*Terms*
`empty_tsequ:`		$\rightarrow$	*Terms*
`var:`	*String*	$\rightarrow$	*Term*
`structure:`	*String*, *Terms*	$\rightarrow$	*Term*

A Prolog program is a list of clauses. A clause is a pair of terms consisting of the head and the body of the clause. A term is either a variable or a structure with a (possibly empty) list of subterms. The sort *String* is assumed to be a basic sort. Note that the subdivision of a clause body into disjunction and conjunction of literals is done in subsequent translation steps. The term corresponding to the clauses

```
append([],L,L).
append([E|R],L,[E|RL]) :- append(R,L,RL).
```

for the predicate `append` is

```
prog([
      clause(append([],var(L),var(L)) , true),
      clause(append([var(E)|var(R)],var(L),[var(E)|var(RL)]) ,
             append(var(R),var(L),var(RL))
     ])
```

(We use the usual Prolog notations for lists of clauses and structured terms [Clocksin/Mellish 81] and omit operators like "`structure`", "`csequ`" etc.)

In the first translation step a term of sort *Prog* will be enriched with the following informations:

- Is there a cut in one clause? This will be used for generating instructions for handling the cut.
- Name and arity of the predicate which is defined by the clauses.
- Number of the clauses.
- List of the first arguments in each clause. This will be used to generate the indexing instructions.

The signature of the abstract syntax tree after the first translation step is the following:

sorts: *Prog*2, *Clauses*, *Clause*, *Terms*, *Term*, *Args*, *Arg*, *String*, *Nat*, *Bool*

operations:

`prog2:`	*Bool*, *String*, *Nat*, *Nat*, *Args*, *Clauses* $\rightarrow$ *Prog*2		
`var_arg:`	*String*	$\rightarrow$	*Arg*
`struct_arg:`	*String*, *Terms*	$\rightarrow$	*Arg*
`asequ:`	*Arg*, *Args*	$\rightarrow$	*Args*
`empty_asequ:`		$\rightarrow$	*Args*
`csequ:`	*Clause*, *Clauses*	$\rightarrow$	*Clauses*
`empty_csequ:`		$\rightarrow$	*Clauses*
`clause:`	*Term*, *Term*	$\rightarrow$	*Clause*
`tsequ:`	*Term*, *Terms*	$\rightarrow$	*Terms*
`empty_tsequ:`		$\rightarrow$	*Terms*
`var:`	*String*	$\rightarrow$	*Term*
`structure:`	*String*, *Terms*	$\rightarrow$	*Term*

If there is a term of the form

`prog2(`C, F, A, N, As, Cs`)`

then the following conditions holds after the first translation step:

- C is true iff the clauses Cs contain a cut.
- F is the name and A the arity of the predicate defined by the clauses Cs.

- N is the number of elements in the clause list Cs.
- As is the list of the first arguments in the heads of Cs.

Hence the term corresponding to the clauses of `append` is translated into the term

```
prog2(false, append, 3, 2,
      [ struct_arg([]) , struct_arg([var(E)|var(R)]) ],
      [
       clause(append([],var(L),var(L)) , true),
       clause(append([var(E)|var(R)],var(L),[var(E)|var(RL)]) ,
              append(var(R),var(L),var(RL))
      ])
```

in the first translation step. In subsequent translation steps program terms are enriched with the following informations:

- Classification of cuts in the clauses.
- Elimination of disjunctions by introducing new predicates.
- Numbering of the literals in each clause.
- Numbering of the arguments in each literal.
- Classification of arguments: Subargument or head argument (this is important for generating unify, get and put instructions).
- Classification of variables (permanent or temporary).

With this information it is possible to generate the WAM code for each clause. The complete code for all clauses consists of the code for each clause and an indexing scheme which filters out a subset of clauses that could match a given procedure call. The creation of this indexing scheme can also be specified by several translation steps. The last translation step of the specification merges the code for the indexing scheme with the code for each clause.

Each translation step is formally specified as a mapping from terms of the preceding signature into terms of the next signature. For the description of these mappings we use no specific formalism as in [Ganzinger/Giegerich 84], but the mappings are inductively defined on the structure of the terms using usual mathematical notations.

This description of the compiler has several advantages:

- All details of the compilation are described. It is clear how to translate a given Prolog program into a WAM program.
- It can be proofed that a WAM program is the translation of a Prolog program.

- The complexity of the compilation is divided into several simple translation steps.
- The description is a good basis for implementing a Prolog compiler. Several authors have shown the advantages of logic programming for compiler writing ([Warren 80] [Ganzinger/Hanus 85], among others) and therefore we have used Prolog as an implementation language of the compiler: The abstract syntax trees are Prolog terms and the implementation of the translation steps is a simple coding of the specification into Prolog. The performance of the compiler is acceptable (it translates approximate 30 clauses per second on a Sun-3).

3 Specification of the Warren abstract machine

We give an operational semantics of the WAM. The WAM is defined as an abstract interpreter for WAM instructions. The interpreter has an initial state and each interpretation of a WAM instruction maps a state into a new state. The state of the WAM interpreter consists of the values of all registers and the contents of the memory (code area, heap, local stack and trail stack). For example, the heap contains all terms created during a computation of a program. Hence we model a particular state of the heap as a function from the domain of heap addresses into the domain of terms.

In detail, the following domains are defined:

Code_address	=	<an infinite denumerable set with a linear ordering, i.e. there exists a successor $succ(e)$ for each element e and there exists a predecessor $pred(e)$ for each element e except for the least element $min(Code\_address)$ >
Heap_address	=	<similar definition as $Code\_address$ >
Local_stack_address	=	<similar definition as $Code\_address$ >
Trail_address	=	<similar definition as $Code\_address$ >

The Y-, X-, and A-registers of the WAM are addressed by natural numbers (the first components are used for disjunctive unions of the register domains):

$$\begin{array}{lcl} Y\_register & = & reg:\{Y\} \ \times \ nr:Nat \\ X\_register & = & reg:\{X\} \ \times \ nr:Nat \\ A\_register & = & reg:\{A\} \ \times \ nr:Nat \end{array}$$

We use cartesian products with a tag for each component. This allows us to identify a component by dot notation: If r is an element of type $X\_register$, then $r.reg$ denotes the first and $r.nr$ the second component of that element.

If we denote by "+" the disjunctive union, we can define the domain of **terms** as

$$
\begin{array}{rcl}
Term & = & tag:\{int\} \times cont:Integer \\
 & & + \; tag:\{atom\} \times cont:(Atom + func:Atom \times ar:Nat) \\
 & & + \; Trail\_element \\
 & & + \; tag:\{var\} \times (adr:Heap\_address + \\
 & & \qquad\qquad eadr:Local\_stack\_address \times ereg:Nat) \\
 & & + \; tag:\{structure\} \times adr:Heap\_address \\
 & & + \; tag:\{list\} \times adr:Heap\_address
\end{array}
$$

(*Atom* is the domain of all Prolog atoms), where a $Trail\_element$ represents an unbound variable:

$$
\begin{array}{rcl}
Trail\_element & = & tag:\{undef\} \times (adr:Heap\_address + \\
 & & \qquad eadr:Local\_stack\_address \times ereg:Nat)
\end{array}
$$

Hence a term is either an integer value, an atom or functor (atoms have arity zero whereas functors have an arity greater than zero), an unbound variable, a variable bound to a term on the heap or local stack, or a reference to a structure or list stored on the heap.

The local stack of the WAM contains two kinds of objects: environments and backtrack points. "An **environment** consists of a vector of value cells for variables occuring in the body of some clause, together with a continuation comprising a pointer into the body of another clause and its associated environment" [Warren 83]. Therefore the domain of environments is defined as

$$Environment = ee:Local\_stack\_address \times ecp:Code\_address \times y:[Nat \to Term]$$

($[A \to B]$ denotes the domain of all functions from domain A into domain B). A **backtrack point** "contains all the information necessary to restore an earlier state of computation in the event of backtracking" [Warren 83]. Thus a backtrack point consists of the values of the relevant registers of the WAM (see below) and a reference to alternative clauses (component *bp*):

$$
\begin{array}{rcll}
Backtrackpoint & = & ba & : [Nat \to Term] \times \\
 & & bce & : Local\_stack\_address \times \\
 & & bcp & : Code\_address \times \\
 & & bb & : Local\_stack\_address \times \\
 & & bp & : Code\_address \times \\
 & & btr & : Trail\_address \times \\
 & & bh & : Heap\_address \times \\
 & & nr & : Nat
\end{array}
$$

For example, if b is an element of *Backtrackpoint*, then $b.ba$ denotes the contents of all A-registers stored in b and $b.ba[0]$ is the contents of the first A-register stored in b. The component nr contains the arity of the predicate in which the backtrack point is created. Now it is easy to define the memory areas of the WAM:

$$
\begin{array}{lcl}
Local\_stack\_element & = & Environment + Backtrackpoint \\
Local\_stack & = & [Local\_stack\_address \rightarrow Local\_stack\_element] \\
Heap & = & [Heap\_address \rightarrow Term] \\
Trail & = & [Trail\_address \rightarrow Trail\_element] \\
Code\_area & = & [Code\_address \rightarrow Instructions]
\end{array}
$$

Instructions is the domain of all instructions with appropiate parameters. We omit the necessary definitions here. A **state of the WAM interpreter** contains the following components:

Name	Type	Comment
p	$Code\_address$	program pointer
cp	$Code\_address$	continutation pointer
e	$Local\_stack\_address$	environment pointer
b	$Local\_stack\_address$	backtrack pointer
t	$Trail\_address$	trail pointer
h	$Heap\_address$	heap pointer
s	$Heap\_address$	structure pointer
rw	$\{read, write\}$	read/write register
$error$	$Boolean$	error register for predefined predicates
$occur\_check$	$Boolean$	occur check register
a	$[Nat \rightarrow Term]$	A-registers
x	$[Nat \rightarrow Term]$	X-registers
ca	$Code\_area$	code area
hp	$Heap$	heap
ls	$Local\_stack$	local stack
trl	$Trail$	trail

There are several restrictions on a well-defined state, e.g., $ls(e)$ must be an element of *Environment*, $ls(b)$ must be an element of *Backtrackpoint* and so on. The state transitions of the WAM must preserve these restrictions. The component *error* of the state is set to true if there is a run-time error in some predefined predicate, e.g., arithmetic errors, file errors etc. If the component *occur_check* is set to true then the unification of two terms is done with the occur check (cf. [Lloyd 87]). In the **initial state** all memory areas are undefined (contains no elements) and the values of cp, e, t, h and s are the least addresses of the address domains. The **terminal state** is defined as the state with $error = true$ or $ca(p)$ is a "stop" instruction.

The main part of the WAM specification is the description of the state transition corresponding to an interpretation of a particular WAM instruction. We omit the full description here but present the specification of three WAM instructions. We denote the modification of a component x of a state by

$$x \;\leftarrow\; <\!newvalue\!>$$

If f is an element of the domain $[A \rightarrow B]$, then $f[x \leftarrow y]$ denotes the element of $[A \rightarrow B]$ which is identical to f but has the value y for the argument x. If x is a component of a state, then $'x$ denotes the old value of x before the interpretation of the instruction. Only changes of the WAM state are listed in the specification of a WAM instruction. Components of the WAM state that are not mentioned in the instruction specification are not changed by that instruction.

Examples: The WAM instruction "execute(x)" calls the last goal in a clause body. The program pointer is set to the code address of the predicate in the last goal:

instruction execute(x)
x: *Code_address*
Transformation:
$$p \leftarrow x$$

The instruction "put_nil(v)" puts the constant representing an empty list into A-register v:

instruction put_nil(v)
v: *A_register*
Transformation:
$$a \leftarrow {}'a[v.nr \leftarrow (atom, nil)]$$

The instruction "put_value(u,v)" puts the value of variable u into A-register v. If u is an A- or X-register, we have direct access to the value of u. If u is a Y-register, the value of u can be found in the actual environment and therefore we need a reference to the actual environment:

instruction put_value(u, v)
u: $(X\_register + A\_register + Y\_register)$
v: *A_register*
Transformation:
$$\begin{array}{l} \text{if } u.reg = X \\ \quad \text{then } a \leftarrow {}'a[v.nr \leftarrow x(u.nr)] \\ \quad \text{else } \ \text{if } u.reg = A \\ \qquad\qquad \text{then } a \leftarrow {}'a[v.nr \leftarrow {}'a(u.nr)] \\ \qquad\qquad \text{else } \ a \leftarrow {}'a[v.nr \leftarrow (ls(e).y)(u.nr)] \end{array}$$

The full specification of the WAM can be found in [ProCom 87].

This specification is the basis of a 68000-based WAM implementation at the University of Dortmund. The specified state is mapped into memory and machine registers and the transformations are mapped into sequences of machine code. The complete system is shown in the following picture.

$$\boxed{\begin{array}{c}\text{Prolog}\\\text{program}\end{array}} \xrightarrow{\text{pass1}} \boxed{\begin{array}{c}\text{WAM}\\\text{program}\end{array}} \xrightarrow{\text{pass2}} \boxed{\begin{array}{c}\text{68000}\\\text{code}\end{array}}$$

"pass1" is the implementation of the Prolog compiler described in chapter 2 and "pass2" translates every WAM instruction into a sequence of machine instructions. Our implementation is a direct realization of the WAM specification without any optimizations. The performance of our system is approximate 40 Klips on a Sun-3 for the naive reverse example, which is comparable with other Prolog implementations.

4 Correctness proofs

There are at least two reasons for the formal specification of a Prolog system:

1. It is the foundation of an implementation. The specification decreases the problems in coordinating several people working on the project.
2. It makes it possible to prove the correctness of the complete system or some translated programs.

The first reason was very important to our project, whereas the second point is a task for the future. Actually, we are working on correctness proofs for particular programs. [Kursawe 86] has shown that it is possible to derive specific WAM instructions from the source program by partial evaluation. We want to refine the interpreters that define the operational semantics of Prolog (cf. [Debray/Mishra 88], [Jones/Mycroft 84]) in a way that makes it possible to compare these interpreters with the operational semantics of the WAM.

5 Conclusions

We have presented an outline of a formal specification of a Prolog compiler and the Warren abstract machine. The specification is a good basis for a direct implementation of a Prolog system with an acceptable performance. Further work includes a more efficient implementation of the specification and correctness proofs for particular programs.

Acknowledgements

The author is grateful to Jörg Süggel, Jörg Petersen and the members of the project group "ProCom" for their work on the Prolog system.

References

[Arbab/Berry 87]
B. Arbab and D.M. Berry. Operational and Denotational Semantics of Prolog. *Journal of Logic Programming (4)*, pp. 309–329, 1987.

[Clocksin/Mellish 81]
W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer, 1981.

[Debray/Mishra 88]
S.K. Debray and P. Mishra. Denotational and Operational Semantics for Prolog. *Journal of Logic Programming (5)*, pp. 61–91, 1988.

[Deransart/Ferrand 87]
P. Deransart and G. Ferrand. An Operational Formal Definition of PROLOG. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 162–172, San Francisco, 1987.

[Ganzinger/Giegerich 84]
H. Ganzinger and R. Giegerich. Attribute Coupled Grammars. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pp. 157–170, Montreal, 1984.

[Ganzinger/Hanus 85]
H. Ganzinger and M. Hanus. Modular Logic Programming of Compilers. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 242–253, Boston, 1985.

[Jones/Mycroft 84]
N.D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 281–288, Atlantic City, 1984.

[Kursawe 86]
P. Kursawe. How to invent a Prolog machine. In *Proc. Third International Conference on Logic Programming (London)*, pp. 134–148. Springer LNCS 225, 1986.

[Lloyd 87]
J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.

[Nilsson 84]
J.F. Nilsson. Formal Vienna-Definition-Method models of Prolog. In J.A. Campbell, editor, *Implementations of Prolog*, pp. 281–308. Ellis Horwood, 1984.

[ProCom 87]
Projektgruppe ProCom. Zwischen- und Abschlußbericht der Projektgruppe ProCom (Prolog Compiler). Univ. Dortmund, 1987.

[Van Roy 84]
P. Van Roy. A Prolog Compiler for the PLM. Report No. UCB/CSD 84/203, Univ. of California, Berkeley, 1984.

[Warren 80]
D.H.D. Warren. Logic Programming and Compiler Writing. *Software - Practice and Experience*, Vol. 10, pp. 97–125, 1980.

[Warren 83]
D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, 1983.