Implementation Techniques for Prolog

Andreas Krall
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien
andi@mips.complang.tuwien.ac.at

Abstract

This paper is a short survey about currently used implementation techniques for Prolog. It gives an introduction to unification and resolution in Prolog and presents the memory model and a basic execution model. These models are expanded to the Vienna Abstract Machine (VAM) with its two versions, the VAM_{2P} and the VAM_{1P}, and the most famous abstract machine, the Warren Abstract Machine (WAM). The continuation passing style model of Prolog, binary Prolog, leads to the BinWAM. Abstract interpretation can be applied to gather information about a program. This information is used in the generation of very specialized machine code and in optimizations like clause indexing and instruction scheduling on each kind of abstract machine.

1 Introduction

The implementation of Prolog has a long history [Col93]. Early systems were implemented by the group around Colmerauer in Marseille. The first system was an interpreter written in Algol by Phillip Roussel in 1972. With this experience a more efficient and usable system was developed by Gérard Battani, Henry Meloni and René Bazzoli [BM73]. It was a structure sharing interpreter and had essentially the same built-in predicates as modern Prolog systems. This system was reasonably efficient and convinced others of the usefulness of Prolog. Together with Fernande and Luis Pereira David Warren developed the DEC-10 Prolog, the first Prolog compiler [War77]. This compiler and the portable interpreter C-Prolog spread around the world and contributed to the success of Prolog. Further developments are described in [Roy94] and partly in this paper.

Section 2 presents a basic execution model for Prolog. This model helps to understand the Warren Abstract Machine described in section 3 and the Vienna Abstract Machine described in section 4. Section 5 gives on overview of optimizations.

2 A basic execution model

2.1 Introduction

The two basic parts of a Prolog interpreter are the unification part and the resolution part. The resolution is quite simple. It just implements a simplified SLD-resolution mechanism that searches the clauses top-down and evaluates the goals from left to right. This strategy immediately leads to the backtracking implementation and the usual layout of the data areas and stacks. Resolution handles stack frame allocation, calling of procedures, and backtracking.

Unification in Prolog is defined as follows:

- two constants unify if they are equal
- two structures unify if their functors (name and arity) are equal and all arguments unify
- two unbound variables unify and they are bound together
- an unbound variable and a constant or structure unify and the constant or structure is bound to the variable

This definition of unification determines the data representation. A thorough analysis of the recursive unification algorithm pays off because the interpreter spends most of the time in this part.

2.2 The representation of data

Since Prolog is not statically typed, the type and value of a variable can in general be determined only at run time. Therefore, a variable cell is divided into a value part and a tag part which determines the kind of the value. Fig. 1 shows a tagged value cell.

Basic data objects in Prolog are constants (atom and integer), structures and unbound variables. Since unbound variables can be bound together, there are

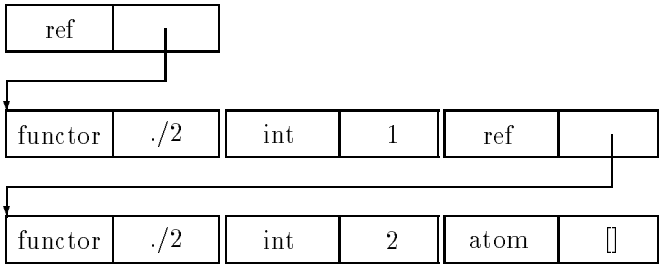

Figure 2: representation of X = 1.2.[]

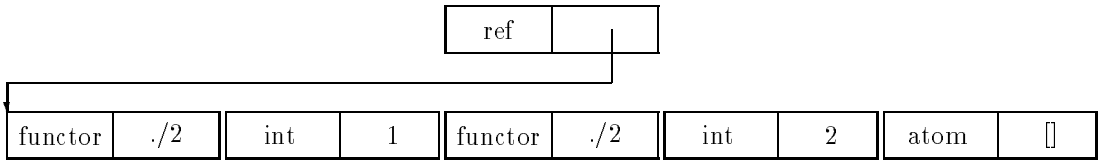

Figure 3: compacted representation of X = 1.2.[]

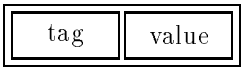

Figure 1: a tagged value cell

references between variables which are represented by pointers. To access a variable it can be necessary to follow the chain of references which is called dereferencing. Following tagged cells are needed in a Prolog system:

atom	unique identifier of character string
integer	integer number
reference	pointer to another tagged cell
unbound	(self-reference pointer)
functor	name and arity of a structure followed by a tagged cell for each argument

Most Prolog implementations do not use a separate tag for unbound variables. They represent unbound variables by a self reference. This can eliminate a tag check during unification of an unbound variable with another variable. The comparison can be replaced by an assignment. Since structures need more than one cell the variable cell contains a reference to the functor cell (see fig. 2). If the last cell of a structure is again a structure and the second structure is allocated directly after the first structure, the reference can be omitted (see fig. 3). This compact allocation can be obtained either at the first allocation or on garbage collection.

Another solution is a special reference tag for structures. The advantage of this method is that the type of a value cell can be determined without a memory access. Many implementations distinguish further between the empty list (nil) and other atoms, and between lists and other structures in order to allow more efficient implementations of lists. Big numbers and floating point numbers are represented as structures.

The tag field can be represented in different ways. It can be an additional memory cell of the standard word size, or it can be a small part of a memory cell. If the tag consists of some bits, the tag is either fixed-sized or variable-sized and uses the most or least significant part of the word.

Useful tag representations try to minimize the tag extraction and insertion overhead. An example is the use of zeroes in the least significant part of the word as an integer tag. Addition and subtraction can so be done without tag manipulation. Another example is to have the stack pointer displaced by the list tag, so that the allocation of list cells is free. A comprehensive study of tag representations can be found in [SH87].

Problems arise if a variable occurring inside a structure should be bound to this structure. In theorem provers in such a case the unification should fail. This test for occurrence of the variable in a structure, called occur check, is expensive. It is omitted in many unification algorithms employed by Prolog systems. If such a structure is assigned to a variable, a recursive structure is created. A simple unification algorithm would enter an infinite loop unifying two infinite structures. There exist linear time unification algorithms for infinite structures [Jaf84], but many Prolog systems do without it and create infinite structures, but cannot unify or print them.

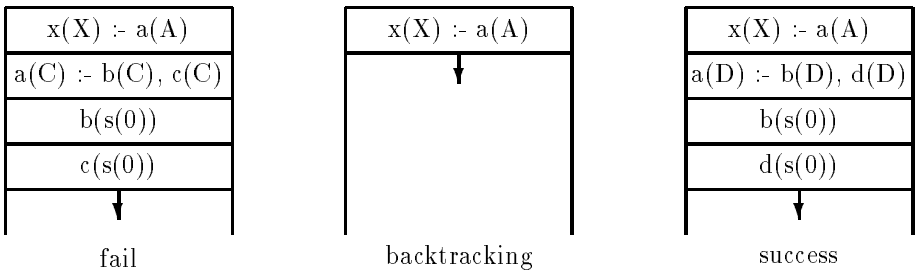

Figure 4: stacks

2.3 The data areas

Variables in a Prolog clause are stored in a stack frame similar to variables in a conventional programming language. The SLD-resolution was chosen as the resolution scheme for Prolog because of its simple stack implementation and efficient memory use. An early description of the memory management of Prolog can be found in [Bru82].

The clause

```
a(C) :- b(C), c(C).
```

can be represented by the tree in fig. 5.

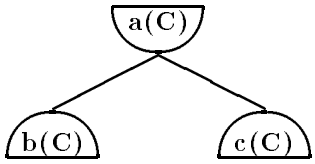

Figure 5: clause

Subtrees can be combined to a complete proof tree, also called AND-OR tree. As an example, take the following short Prolog program:

```
x(X) :- x(X).
a(C) :- b(C), c(C).
a(D) :- b(D), d(D).
b(s(0)).
c(s(0)).
d(s(0)).
```

The AND-OR tree is shown in fig. 6. The thick lines belong to the AND-tree of the last solution, the thin lines belong to the AND-tree of the first solution. The AND-tree represents the calls of the different goals of a clause. The OR-tree represents the alternative solutions.

The AND-OR-tree can be represented in linearized form by a stack (see fig. 4). Since we are only interested in one solution at a time, only an AND-tree

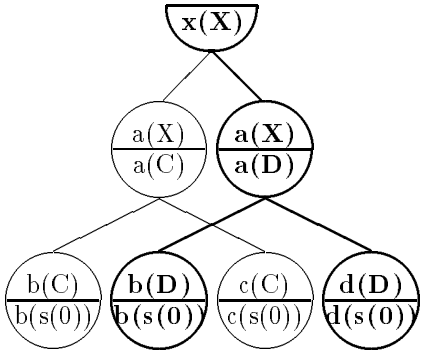

Figure 6: proof tree

is stored in the stack. The OR-tree corresponds to different contents of the stack between backtracking. Fig. 4 represents the AND-OR-tree at three different moments. The left part of the figure shows the first solution, the middle shows the stack after backtracking and the right part shows the second solution.

The cells of structures are allocated on a stack. In fig. 4 the cells for the structure `s(0)` would be allocated after the stack frame for `b(s(0))`. When the stack frame for `c(s(0))` is allocated, the stack frame for `b(s(0))` can be discarded if there are no references into the discarded stack frame and if there are no structure cells on the stack. In order to allow memory reuse the stack is divided into two parts. The environment (or local) stack holds the stack frames and the copy stack (global stack or heap) holds structure cells. The dangling reference problem can be solved if references within the environment stack are directed towards the bottom of the stack or to the heap.

In order to facilitate the removal of stack frames, there is a distinction between deterministic and indeterministic stack frames. A stack frame is deterministic if no alternative clauses are left for this procedure. An indeterministic stack frame is called choice point.

During unification variables in a stack frame may become bound. On backtracking they should be reset

to unbound. An additional stack, the trail, solves this problem. During unification the addresses of bound variables are pushed onto the trail. On backtracking these addresses are popped from the trail and the variables are reset to unbound. It is only necessary to trail the addresses of variables which are closer to the bottom of the stack than the last choice point. Testing this condition is called trail check.

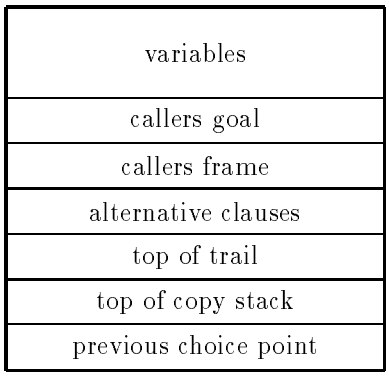

Figure 7: stack frame with choice point

Fig. 7 shows a stack frame with a choice point. A deterministic stack frame contains the cells for the variables, a pointer to the caller of this clause, comparable to the return address in a conventional stack frame, and a pointer to the stack frame of the caller. These two pointers are usually called continuation. A choice point additionally contains a pointer to the next alternative clause, a pointer to the top of trail and a pointer to the top of copy stack at the time the choice point was created, and a pointer to the previous choice point.

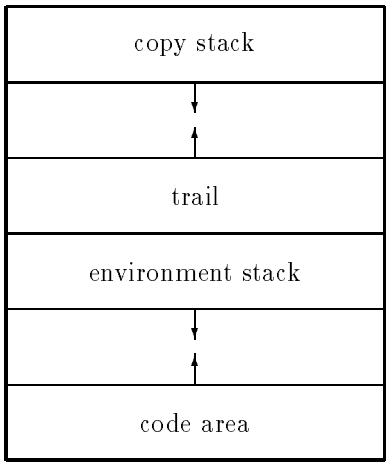

Figure 8: data areas

Fig. 8 shows the stacks and data areas in a Prolog system. The check for pointer directions is simplified if copy and environment stack grow in the same direction, and the copy stack grows towards the environment stack. The code area is needed to store the program and string representations of the atoms.

To enable fast unification, only unique identifiers of atoms are stored in variables. A hash table or search tree is constructed over these strings to enable fast searching when only the string representation is known. The same concept is applied to functors (name and arity of structures).

2.4 Simple Optimizations

2.4.1 The Representation of Terms

In the previous sections we have used a representation of structures known as structure copying. This technique was introduced by Maurice Bruynooghe [Bru82] and Christopher Mellish [Mel82]. Structure copying is now the standard implementation method because it is faster than the previously used structure sharing [BM72]. In general, structure copying also consumes less memory than structure sharing [Mel82].

Structure sharing is based on the assumption that a large part of a structure is constant and contains only few variables. A structure is here divided into the constant, part called skeleton, and a variable part, called environment. The skeleton contains the constants and the offsets into the environment, the environment contains the variables. The skeleton is stored in the code area, the environment in the global stack. A structure is represented by two pointers, one to the skeleton and one to the environment. Therefore, a variable cell has to hold a tag and two pointers. On modern machine architectures this means that a cell needs two words and spends much time in decoding skeletons. Thus only the first Prolog systems [BM73] and David Warrens first Prolog compiler [War77] used structure sharing. But in conjunction with binary Prolog (see section 3.3) structure sharing can gain in interest again.

2.4.2 Interpreters and Compilers

We did not yet address the problem of how to represent programs. A simple solution is to directly use the term representation of the clauses. The interpreter then has two instructions, the unification which operates on a whole goal and the head of the matching clause, and the resolution which pushes whole clauses onto the stack and does the backtracking. This simple model is called clause or goal stacking model. Using structure sharing for the goal level of the term leads to the classical interpreter model with two term pointers and two environment (frame) pointers.

Unification in general consists of assignments, conditional assignments and comparisons. So it is quite natural to break the unification up into its atomic parts. The program is analysed and instructions specialized for the argument types of the goals are generated. The resolution can be divided into stack allocation, clause indexing and calling instructions. The program is represented as a sequence of such instructions which can be either executed by an interpreter or compiled to machine code. Such an instruction set definition together with the memory model is called an abstract machine. Several abstract machines were defined, in this paper only the common Warren Abstract Machine (WAM) and the Vienna Abstract Machine (VAM) are dealt with.

2.4.3 Variable Classification

In the simple execution model presented above it is assumed that during allocation of a stack frame all variable cells are initialized to unbound. Furthermore, for every variable occurring in a clause a cell is allocated.

Variables occurring only once in a clause, called void variables, can be bound only by a single instruction. The value bound to this variable will never be used. So it is not necessary to reserve space for such variables. Another case are variables which occur only within one subgoal. It is not necessary to reserve the space over different goals. Space for these temporary variables is not reserved in the stack frame but in an additional fixed area. To avoid dangling pointers, references must always point from temporary variables to the environment or copy stack.

The initialization of the stack frame and of temporary variables can be eliminated if the first occurrence and further occurrences of a variable are distinguished. The improvement comes not only from the elimination of some initializations but also from the elimination of a complex unification for the first occurrence.

2.4.4 Clause Indexing

Indexing of Prolog clauses is an optimization whose aim is to reduce the number of clauses to be tried and to avoid the creation of choice points if possible. The results are better execution times and memory consumption.

The most trivial optimization done by every Prolog system is to try only the clauses of a procedure instead of all clauses of a program during the the search for a unifying clause. First argument indexing is more complicated: Only clauses which unify with the goal in the first argument are selected. For this purpose an indexing structure is built over the clauses which differentiates the clauses depending on their first arguments. This indexing structure is either a hash table or a search tree. The search tree has the advantage that it easily handles variables in the head of the clauses and allows dynamic clause insertion. Sophisticated clause indexing schemes are presented in section 5.2.

2.4.5 Last-call Optimization

In section 2.3 we noticed that stack frames can be discarded after the subtree has been proved and no alternatives are left. This check is simple. The stack frame has to be the top-most frame. There can be no choice point left on the stack allocated later. A deterministic stack frame can be discarded not only after the call of the last subgoal, but also before this call. The general solution is to copy the stack frame of the called clause over the stack frame of the clause with the last call after the unification of the variables has been done (see fig. 9).

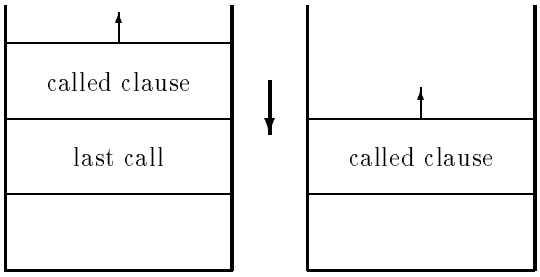

Figure 9: general last-call optimization

This frame moving is complicated by the fact that there could be references to the moved stack frame and references to unbound variables in the discarded stack frame. Therefore, the variables have to be checked and updated prior to the moving of the stack frame. Instead of updating the references, the variables can be globalized. That means that they are allocated on the global (copy) stack. The overhead of moving the stack frame can be avoided by copying the discarded stack frame to registers. The new stack frame then is directly created at the place of the discarded frame (WAM). An other solution is to create the new stack frame in registers and copy the registers to the place of the discarded frame (VAM).

Last-call optimization can be generalized for every call. A deterministic stack frame can be moved over this part of a stack frame which is not used at later calls. For that purpose the variables have to be ordered on their last occurrence. A simple stack trimming without the overhead of generalized last-call optimization can be achieved by discarding only variables which have their last occurrence before the call. Last-call optimization can reduce an infinite memory con-

sumption to a finite one. So it has to be implemented in every Prolog system. Specialized implementations also reduce the run time because unifications can be eliminated if variables occupy the same location.

2.4.6 Garbage Collection

In Prolog unreferenced data (garbage) can be produced both in the code area and in the copy stack. But different kinds of garbage collection algorithms can be applied to these data areas. At least the copy stack needs a compacting collector which preserves the order of the cells. An algorithm which uses pointer reversal has the best space-time complexity. When the copy stack becomes compacted the trail must be updated too. Some Prolog garbage collectors collect only part of the stack due to wrong interpretations of uninitialized variables. Unused data in the code area is easily detected by the retract procedure. If the code is not moved, no updates of the environment stack are necessary.

3 The Warren Abstract Machine

Six years after the development of his successful compiler for the DEC-10 David Warren presented a new abstract Prolog instruction set [War83]. This New Prolog Engine has become very popular under the name Warren Abstract Machine (WAM). It has been the basis of nearly all Prolog systems developed after the year 1983. The aim of the WAM was to serve as a simple and efficient implementation model for byte code interpreters as well as machine code generating compilers. So the first implementation was a structure copying byte code emulator.

3.1 The Original Warren Abstract Machine

The WAM is closer to the execution model of imperative languages than all other implementation models. The main idea is the division of the unification into two parts, the copying of the arguments of the calling goal into argument registers and the unification of the argument registers with the arguments of the head of the called clause. This is very similar to the parameter passing in imperative languages like C. The first parameters are passed via registers. If the registers are exhausted, the stack can be used for additional parameters. The partitioning of the unification reduces the number of instruction pointers to one and the number of frame pointers to one, if all parameters can be kept in registers.

This parameter passing is mirrored in the instruction set. `put` instructions copy the arguments of the goal into the registers, `get` instructions unify the registers with the arguments of the head. `unify` instructions handle the unification of structure arguments. They can be executed in two modes. In write mode a new structure is created, in read mode the structure arguments are unified with the arguments of the head. *procedural* instructions manage the stack and execute procedure calls. *indexing* instructions build the indexing structure. The data areas are identical to the previously presented simple model (see fig. 10), but the choice point is quite different. The original WAM added a push down list used as a stack for the recursive unification procedure. But in a byte code emulator this push down list is hidden in the run time stack of the implementation language. In a machine code generating compiler the environment or the copy stack can be used for this purpose.

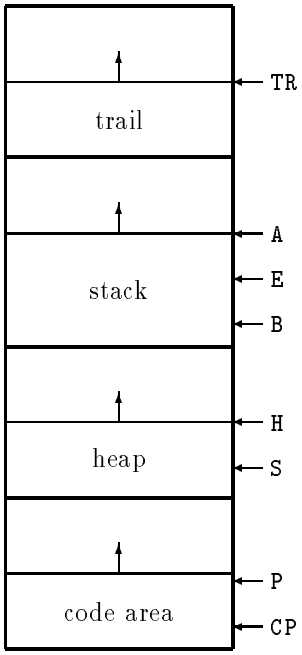

Figure 10: data areas of the WAM

Since all variables in the stack frame are copied into the argument registers before calling a procedure, last-call optimization is simplified. The stack frame of the called procedure can be created directly at the place of the stack frame of a deterministic caller. To avoid the overhead of recreating the argument registers on backtracking using `put` instructions, the argument registers are saved in the choice point. This permits last-call optimization also in these cases where the called procedure has alternative clauses. Furthermore, this leads to a relaxed definition of temporary variables. The head, the first subgoal and all builtin predicates between head and first subgoal count as one subgoal

for the classification of temporary variables. Unfortunately, the problem of dangling references is not solved. Therefore, there are special versions of `put` instructions which check if the last occurrence of a variable in the last subgoal has a reference to the discarded stack frame. Such variables are called unsafe variables and are allocated on the copy stack.

After this introduction we can present the machine registers of the WAM:

`P`	program counter
`CP`	continuation program counter
`E`	current environment pointer
`B`	most recent choice point
`A`	top of stack (not strictly necessary)
`TR`	top of trail
`H`	top of heap
`S`	structure pointer
`A1,A2,...`	argument registers
`X1,X2,...`	temporary registers

The continuation program counter is a register which caches the pointer to the continuation goal. It can be compared with the return address in an imperative language. Holding this value in a register speeds up the execution of the leaf procedures. The environment pointer is comparable to the frame pointer in an imperative language. The original WAM contained a `HB` register (heap backtrack point) which caches the top of heap corresponding to the most recent choice point. It is used to check if a variable has to be trailed. In general it is faster to take this value directly from the choice point than to update this register at every choice point creation and deallocation. The structure pointer `S` is used during the unification of the arguments of structures. Also named different, argument registers and temporary registers share the same pool of registers. Register allocation tries to use the registers in such an order that the number of instructions can be reduced.

The environment contains the local variables and the continuation code pointer `CP'` and a pointer to the previous environment `E'`. The choice point is shown in fig. 11. `B'`, `H'`, `TR'`, `CP'`, `E'` and the `Ai'` are copies of the values of the machine registers before the creation of the choice point. The value `BP` of the retry pointer is supplied by the instruction which creates the choice point and points to the code of the next alternative clause.

Fig. 12 shows the complete WAM instruction set. `Vn` describes either temporary or local variables. `Ri` designates the argument registers. `C` is a constant (integer or atom) in its internal representation and `F` is the functor of a structure which contains the name and the arity of the structure.

`B'`	previous choice point
`H'`	top of heap
`TR'`	top of trail
`BP`	retry program pointer
`CP'`	continuation program pointer
`E'`	environment pointer
`A1'`	argument registers
`...`	
`An'`	

Figure 11: choice point in the WAM

3.2 Optimizing the basic WAM

In an interpreter the execution mode of unify instruction is hidden in the state of the interpreter. There are just two instruction decoding loops, one for the read mode and one for the write mode. In a machine code generating compiler the mode has to become explicit. The simple solution of a flag register, which is checked in every instruction, is not very efficient. The first step is to divide the unify instructions in write and read instructions. The optimal solution, which splits all paths for read and write mode, has exponential code size. Linear space is consumed if the mode flag is only tested once per structure. This scheme can be improved if write mode is propagated down a nested structure and read mode is propagated up. A more detailed description and further references can be found in [Roy94].

In the WAM it is very common that unbound variables are bound to a value shortly after their initialization. This happens e.g. if a variable has its first occurrence in the subgoal which calls a procedure with a constant argument. The variable has to be created in memory and needs to be dereferenced and trailed before being bound. Beer [Bee88] recognized that this is time consuming and additionally would require an occur check if implemented. He developed the idea of an uninitialized variable.

An uninitialized variable is defined to be an unbound variable that is unaliased, that means it is not shared with another variable. Such a variable gets a special reference tag. Creation of an uninitialized variable is simpler, it does not have to be dereferenced or trailed. Binding reduces to a single store operation. It is necessary to keep track of such variables at run time. If they remain uninitialized after the execution of the subgoal they have been created, they must be initialized to unbound.

3.3 Binary Prolog

The key idea of binary Prolog is the transformation of clauses to binary clauses using a continuation passing style. BinProlog, an efficient emulator for binary Prolog has been developed by Paul Tarau [Tar91][Tar92]. The implementation is based on the WAM which can be greatly simplified in that case.

In binary Prolog a clause has at most one subgoal. A clause can be transformed to a binary clause by representing the call of subgoals explicitely using continuations [App92]. For that purpose the first subgoal is given an additional argument containing the success continuation. The success continuation is the list of subgoals to be executed if the first subgoal is executed successfully. The head is given an additional argument which passes on the continuation. A fact is transformed to a clause, whose subgoal executes a meta-call of the continuation. For example, the following clauses

```
nrev([],[]).
nrev([H|T],R) :-
    nrev(T,L), append(L,[H],R).
```

are transformed into

```
nrev([],[],Cont) :- call(Cont).
nrev([H|T],R,Cont) :-
    nrev(T,L,append(L,[H],R,Cont)).
```

Compiling binary Prolog to the WAM it appears that the environment stack is superfluous since all variables are temporary. Therefore, all instruction dealing with local variables or managing the stack can be eliminated. So a small and efficient interpreter can be implemented. But this simplification has a big problem. The continuation, which contains also the variables previously contained in the stack frame, is stored on the copy stack. This means that there is no last-call optimization. So for a working BinWAM an efficient garbage collector is crucial. In some sense the BinWAM can be seen as mixture of a clause stacking model with the WAM.

4 The Vienna Abstract Machine

4.1 Introduction

The VAM has been developed at the TU Wien as an alternative to the WAM. The WAM divides the unification process into two steps. During the first step the arguments of the calling goal are copied into argument registers and during the second step the values in the argument registers are unified with the arguments of the head of the called predicate. The VAM eliminates the register interface by unifying goal and head arguments in one step. The VAM can be seen as a partial evaluation of the call. There are two variants of the VAM, the VAM_{1P} and the VAM_{2P}.

A complete description of the VAM_{2P} can be found in [KN90]. Here we give a short introduction to the VAM_{2P} which helps to understand the VAM_{1P} and the compilation method. The VAM_{2P} (VAM with two instruction pointers) is well suited for an intermediate code interpreter implemented in C or in assembly language using direct threaded code [Bel73]. The goal instruction pointer points to the instructions of the calling goal, the head instruction pointer points to the instructions of the head of the called clause. During an inference the VAM_{2P} fetches one instruction from the goal, one instruction from the head, combines them and executes the combined instruction. Because information about the calling goal and the called head is available at the same time, more optimizations than in the WAM are possible. The VAM features cheap backtracking, needs less dereferencing and trailing, has smaller stack sizes and implements a faster cut.

The VAM_{1P} (VAM with one instruction pointer) uses one instruction pointer and is well suited for native code compilation. It combines instructions at compile time and supports additional optimizations like instruction elimination, resolving temporary variables during compile time, extended clause indexing, fast last-call optimization, and loop optimization.

4.2 The VAM_{2P}

Like the WAM, the VAM_{2P} uses three stacks. Stack frames and choice points are allocated on the environment stack, structures and unbound variables are stored on the copy stack, and bindings of variables are marked on the trail. The intermediate code of the clauses is held in the code area. The machine registers are the goalptr and headptr (pointer to the code of the calling goal and of the called clause respectively), the goalframeptr and the headframeptr (frame pointer of the clause containing the calling goal and of the called clause respectively), the top of the environment stack, the top of the copy stack, the top of the trail, and the pointer to the last choice point.

Values are stored together with a tag in one machine word. We distinguish integers, atoms, nil, lists, structures, unbound variables and references. Unbound variables are allocated on the copy stack to avoid dangling references and the unsafe variables of the WAM. Furthermore it simplifies the check for the trailing of bindings. Structure copying is used for the representation of structures.

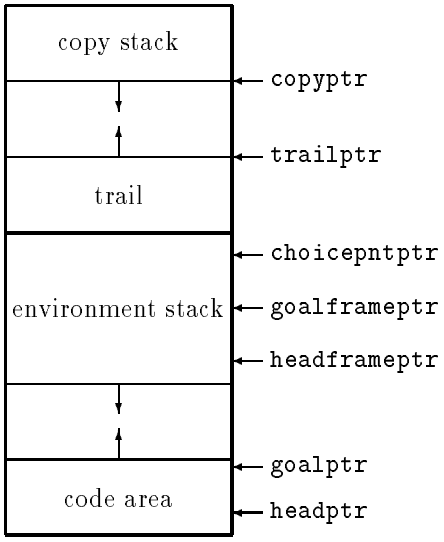

Figure 13: VAM data areas

variables	local variables
goalptr'	continuation code pointer
goalframeptr'	continuation frame pointer

Figure 15: stack frame

trailptr'	copy of top of trail
copyptr'	copy of top of copy stack
headptr'	alternative clauses
goalptr'	restart code pointer (VAM_{2P})
goalframeptr'	restart frame pointer
choicepntptr'	previous choice point

Figure 16: choice point

Variables are classified into void, temporary and local variables. Void variables occur only once in a clause and need neither storage nor unification instructions. Different to the WAM, temporary variables occur only in the head or in one subgoal, counting a group of builtin predicates as one goal. The builtin predicates following the head are treated as belonging to the head. Temporary variables need storage only during one inference and can be held in registers. All other variables are local and are allocated on the environment stack. During an inference the variables of the head are held in registers. Prior to the call of the first subgoal the registers are stored in the stack frame. To avoid initialisation of variables we distinguish between their first occurrence and further occurrences.

The clauses are translated to the VAM_{2P} abstract machine code (see fig. 14). This translation is simple due to the direct mapping between source code and VAM_{2P} code. During run time a goal and a head instruction are fetched and the two instructions are combined. Unification instructions are combined with unification instructions and resolution instructions are combined with termination instructions. A different encoding is used for goal unification instructions and head unification instructions. To enable fast encoding the instruction combination is solved by adding the instruction codes and, therefore, the sum of two instruction codes must be unique.

4.3 The VAM_{1P}

The VAM_{1P} has been designed for native code compilation. A complete description can be found in [KB92]. The main difference to the VAM_{2P} is that instruction combination is done during compile time instead of run time. The representation of data, the stacks and stack frames (see fig. 15) are identical to the VAM_{2P}. The VAM_{1P} has one machine register less than the VAM_{2P}. The two instruction pointers goalptr and headptr are replaced by one instruction pointer called codeptr. Therefore, the choice point (see fig. 16) is also smaller by one element since there is only one instruction pointer. The pointer to the alternative clauses now directly points to the code of the remaining matching clauses.

Due to instruction combination during compile time it is possible to eliminate instructions, to eliminate all temporary variables and to use an extended clause indexing, a fast last-call optimization and loop optimization. In WAM based compilers abstract interpretation is used to derive information about mode, type and reference chain length. Some of this information is locally available in the VAM_{1P} due to the availability of the information of the calling goal.

All constants and functors are combined and evaluated to true or false. For a true result no code is emitted. All clauses which have an argument evaluated to false are removed from the list of alternatives. In general no code is emitted for a combination with a void variable. In a combination of a void variable with the first occurrence of a local variable the next occurrence of this variable is treated as the first occurrence.

Temporary variables are eliminated completely. The unification partner of the first occurrence of a temporary variable is unified directly with the unification partners of the further occurrences of the temporary variable. If the unification partners are constants, no code is emitted at all. Flattened code is generated for structures. The paths for unifying and copying structures is split and different code is generated for each path. This makes it possible to reference each argument of a structure as offset from the top of the copy stack or as offset from the base pointer of the struc-

ture. If a temporary variable is contained in more than one structure, combined unification or copying instructions are generated.

All necessary information for clause indexing is computed during compile time. Some alternatives are eliminated because of failing constant combinations. The remaining alternatives are indexed on the argument that contains the most constants or structures. For compatibility reasons with the VAM_{2P} a balanced binary tree is used for clause selection.

The VAM_{1P} implements two versions of last-call optimization. The first variant (we call it post-optimization) is identical to that of the VAM_{2P}. If the determinacy of a clause can be determined during run time, the registers containing the head variables are stored in the callers stack frame. Head variables which reside in the stack frame due to the lack of registers are copied from the head (callee's) stack frame to the goal (caller's) stack frame.

If the determinacy of a clause can be detected during compile time, the caller's and the callee's stack frames are equal. Now all unifications between variables with the same offset can be eliminated. If not all head variables are held in registers reading and writing variables must be done in the right order. We call this variant of last-call optimization pre-optimization.

Loop optimization is done for a determinate recursive call of the last and only subgoal. The restriction to a single subgoal is due to the use of registers for value passing and possible aliasing of variables. Unification between two structures is performed by unifying the arguments directly. The code for the unification of a variable and a structure is split into unification code and copy code.

5 Optimizations

5.1 Abstract Interpretation

Information about types, modes, trailing, reference chain length and aliasing of variables of a program can be inferred using abstract interpretation. Abstract interpretation is a technique of describing and implementing global flow analysis of programs. It was introduced by [CC77] for dataflow analysis of imperative languages. This work was the basis of much of the recent work in the field of logic programming [AH87] [Bru91] [Deb92] [Mel85] [RD92] [Tay89]. Abstract interpretation executes programs over an abstract domain. Recursion is handled by computing fixpoints. To guarantee the termination and completeness of the execution a suitable choice of the abstract domain is necessary. Completeness is achieved by iterating the interpretation until the computed information change. Termination is assured by bounding the size of the domain. The previous cited systems all are meta-interpreters written in Prolog and very slow.

A practical implementation of abstract interpretation has been done by Tan and Lin [TL92]. They modified a WAM emulator implemented in C to execute the abstract operations on the abstract domain. They used this abstract emulator to infer mode, type and alias information. They analysed a set of small benchmark programs in few milliseconds which is about 150 times faster than the previous systems.

5.2 Sophisticated Clause Indexing

The standard indexing method used in WAM-based Prolog systems can create two choice points. Therefore, this method has been called two-level indexing. Carlson [Car87] introduced one-level indexing by delaying the creation of a choice point as long as possible. By discriminating first on the type of the first argument and when appropriate on its principal functor, the set of potentially matching clauses is filtered out. A choice point is then needed only for non singleton sets. In the worst case the number of indexing instructions can be quadratic to the number of clauses. The VAM_{2P} uses pointers instead of indexing instructions to avoid two-level indexing and to enable assert and retract [Kra88]. A similar strategy is used in [DMC89].

The use of field encoded and superimposed code words for clause indexing was proposed by Wise and Powers [WP84] and was refined by Colomb [CJ86] [Col91]. The method is based on content addressable memory (CAM). The CAM consists of an array of bit columns. Logical operations on columns and lines of the CAM can be computed in one cycle. The results of operations can be held in result columns or lines. The idea is to hold hash values for the arguments of a clause in the CAM. The encoding scheme is based on m–in–n coding which sets m bits in a word of size n to 1. Field encoding uses n/2–in–n coding and gives each argument some bits of a line. Superimposed coding uses m–in–n coding, where n is the size of a whole line and m so small that m times number of arguments is n/2. Variables are either represented by a special column or by hash values with all bits set to 1 or 0. The CAM is fast, but too special and expensive to be used in general purpose computer systems.

In [KS88] Kliger and Shapiro describe an algorithm for the compilation of an FCP(—,:,?) procedure into a control-flow decision tree that analyses the possible data states in a procedure call. This tree is translated to a header for the corresponding machine code of the predicate. At run time the generated instructions control the flow which finally reaches the jump instruction pointing to the correct clause. Redundant tests in a

process reduction attempt are eliminated and the candidate clause is found efficiently. The decision tree may need program space exponential in the number of clauses and argument positions. Consequently in [KS90] they choose decision graphs rather than decision trees to encode the possible traces of each predicate.

Hickey and Mudambi [HM89] were the first who applied decision trees as an indexing structure to Prolog. They compile a program as a whole and apply mode inference to determine which arguments are bound. The decision tree is compiled into switching instructions which can be combined with unification instructions and primitive tests. So equivalent unifications which occur in different clauses are evaluated only once. Reusing the result of such a unification requires a consistent register use. A complete indexing scheme generating algorithm is presented which takes into account effects of aliasing and gives a consistent register use. They also show that the size of the switching tree is exponential in the worst case and that finding an optimal switching tree is NP-complete. For cases where the size of the switching tree is a problem they also present a quadratic indexing algorithm. In general the size is no problem and the speedup is a factor of two.

Palmer and Naish [PN91] and Hans [Han92] also noticed the potential exponential size of decision trees. They compute the decision tree for each argument separately and store the set of applicable clauses for each argument. At run time the arguments are evaluated and the intersection of the applicable clause sets of each argument is computed. The disadvantage of this method is the high run time overhead. Furthermore the size of the clause sets is quadratic to the number of clauses, whereas decision trees are rarely exponential with respect to the number of arguments.

5.3 Stack Checking

Since a Prolog system has many stacks, the run time checking of stack overflow can be very time consuming. There are two methods to reduce this overhead. The more effective one uses the memory management unit of the processor to perform the stack check. A write protected page of memory is allocated between the stacks. Catching the trap of the operating system can be applied to promote a more meaningful error message to the user. A problem with this scheme occurs in combination with garbage collection. The trap can occur at a point in the program where the internal state of the system is unclear so that it is difficult to start garbage collection.

The second idea is to reduce the scattered overflow checks to one check per call. It is possible to compute at compile time the maximum number of cells allocated during a single call on the copy and the environment stack. If these stacks grow into one another (possible only if no references are on the environment stack) both stacks can be tested with a single overflow check. The maximum use of the trail during a call can not be determined at compile time.

5.4 Instruction Scheduling

Modern processors can issue instructions while preceding instructions are not yet finished and can issue several instructions in each cycle. It can happen that an instruction has to wait for the results of another instruction. Instruction scheduling tries to reorder instructions so that they can be executed in the shortest possible time.

The simplest instruction schedulers work on basic blocks. The most common technique is list scheduling [War90]. It is a heuristic method which yields nearly optimal results. It encompasses a class of algorithms that schedule operations one at a time from a list of operations to be scheduled, using priorization to resolve conflicts. If there is a conflict between instructions for a processor resource, this conflict is resolved in favour of the instruction which lies on the longest executing path to the end of the basic block. A problem with basic block scheduling is that in Prolog basic blocks are small due to tag checks and dereferencing. So instruction scheduling relies on global program analysis to eliminate conditional instructions and increase basic block sizes. Just as important is alias analysis. Loads and stores can be moved around freely only if they do not address the same memory location.

A technique called trace scheduling can be applied to schedule the instructions for a complete inference [Fis81]. A trace is a possible path through a section of code. In general it would be the path from the entry of a call to the exit of a call. Trace scheduling uses list scheduling, starting with the most frequent path and continuing with less frequent paths. During scheduling it can happen that an instruction has to be moved over a branch or join. In this case compensation code has to be inserted on the other path. In Prolog the less frequent path is often the branch to the backtracking code. In such cases it is often not necessary to compensate the moved instruction.

Acknowledgement

We express our thanks to Alexander Forst, Franz Puntigam and Jian Wang for their comments on earlier drafts of this paper.

References

[AH87] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[Bee88] Joachim Beer. The occur-check problem revisited. *Journal of Logic programming*, 5(3), 1988.

[Bel73] James R. Bell. Threaded code. *CACM*, 16(6), June 1973.

[BM72] Roger S. Boyer and Jay S. Moore. The sharing of structure in theorem proving programs. In Melzer B. and Michie D., editors, *Machine Intelligence 7*. Edinburgh University Press, New York, 1972.

[BM73] Gérard Battani and Henry Meloni. Interpréteur du language PROLOG. Dea report, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université de Aix-Marseille II, 1973.

[Bru82] Maurice Bruynooghe. The memory management of PROLOG implementations. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*. Academic Press, 1982.

[Bru91] Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic programming*, 10(1), 1991.

[Car87] Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In *Fourth International Conference on Logic Programming*, 1987.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth Symp. Priciples of Programming Languages*. ACM, 1977.

[CJ86] Robert M. Colomb and Jayasooriah. A clause indexing system for prolog based on superimposed coding. *Australian Computer Journal*, 18(1), 1986.

[Col91] Robert M. Colomb. Enhancing unification in Prolog through clause indexing. *Journal of Logic programming*, 10(1), 1991.

[Col93] Alain Colmerauer. The birth of Prolog. In *The Second ACM-SIGPLAN History of Programming Languages Conference*, SIGPLAN Notices, pages 37–52. ACM, March 1993.

[Deb92] Saumya Debray. A simple code improvement scheme for Prolog. *Journal of Logic Programming*, 13(1), 1992.

[DMC89] Bart Demoen, Andre Marien, and Alain Callebaut. Indexing prolog clauses. In *North American Conference on Logic Programming*, 1989.

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30, 1981.

[Han92] Werner Hans. A complete indexing scheme for WAM-based abstract machines. In *PLILP'92*, LNCS 631. Springer, 1992.

[HM89] Timothy Hickey and Shyam Mudambi. Global compilation of Prolog. *Journal of Logic Programming*, 7(3), 1989.

[Jaf84] Joxan Jaffar. Efficient unification over infinite terms. *New Generation Computing*, 2, 1984.

[KB92] Andreas Krall and Thomas Berger. Fast Prolog with a VAM_{1P} based Prolog compiler. In *PLILP'92*, LNCS. Springer 631, 1992.

[KN90] Andreas Krall and Ulrich Neumerkel. The Vienna abstract machine. In *PLILP'90*, LNCS. Springer, 1990.

[Kra88] Andreas Krall. *Analyse und Implementierung von Prologsystemen*. PhD thesis, TU Wien, 1988.

[KS88] Shmuel Kliger and Ehud Shapiro. A decision tree compilation algorithm for FCP(—,:,?). In *Fifth International Conference and Symposium on Logic Programming*, Seattle, 1988.

[KS90] Shmuel Kliger and Ehud Shapiro. From decision trees to decision graphs. In *North American Conference on Logic Programming*, 1990.

[Mel82] Christopher S. Mellish. An alternative to structure sharing in the implementation of a Prolog interpreter. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*. Academic Press, 1982.

[Mel85] Christopher S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1), 1985.

[PN91] Doug Palmer and Lee Naish. NUA-Prolog: An extension to the WAM for parallel Andorra. In *Eighth International Conference on Logic Programming*, 1991.

[RD92] Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1), 1992.

[Roy94] Peter Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic programming*, 19/20, 1994.

[SH87] Peter Steenkiste and John Hennessy. Tags and type checking in LISP: Hardware and software approaches. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM/IEEE, October 1987.

[Tar91] Paul Tarau. A compiler and a simplified abstract machine for the execution of binary metaprograms. In *Eighth International Conference on Logic Programming*, 1991.

[Tar92] Paul Tarau. WAM-optimizations in BinProlog: Towards a realistic continuation passing Prolog engine. Technical report, Université de Moncton, Canada, 1992.

[Tay89] Andrew Taylor. Removal of dereferencing and trailing in Prolog compilation. In *Sixth International Conference on Logic Programming*, Lisbon, 1989.

[TL92] Jichang Tan and I-Peng Lin. Compiling dataflow analysis of logic programs. In *Conference on Programming Language Design and Implementation*, volume 27(7) of *SIGPLAN*. ACM, 1992.

[War77] David H.D. Warren. *Applied Logic–Its Use and Implementation as a Programming Tool*. DAI Research Reports 39 & 40, University of Edingburgh, 1977.

[War83] David H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

[War90] Henry S. Warren. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1), 1990.

[WP84] Michael J. Wise and David M.W. Powers. Indexing Prolog clauses via superimposed code words and field encoded words. In *International Symposium on Logic Programming*, 1984.

goal argument register loading instructions	
`put_variable Vn,Ri`	create a new variable, put reference into `Vn` and `Ri`
`put_value Vn,Ri`	move the content of `Vn` to `Ri`
`put_unsafe_value Vn,Ri`	move the content of `Vn` to `Ri` and globalize
`put_constant C,Ri`	move the constant `C` to `Ri`
`put_nil Ri`	move the constant nil to `Ri`
`put_structure F,Ri`	create functor `F`, put structure pointer into `Ri`
`put_list Ri`	put a list pointer into `Ri`
head argument register unifying instructions	
`get_variable Vn,Ri`	move the content of `Ri` to `Vn`
`get_value Vn,Ri`	unify `Ri` with `Vn`
`get_constant C,Ri`	unify `Ri` with the constant `C`
`get_nil Ri`	unify `Ri` with the constant nil
`get_structure F,Ri`	unify `Ri` with the functor `F`
`get_list Ri`	unify `Ri` with a list pointer
structure argument unifying instructions	
`unify_variable Vn`	move next structure argument to `Vn`
`unify_value Vn`	unify `Vn` with next structure argument
`unify_constant C`	unify the constant `C` with next structure argument
`unify_nil Ri`	unify the constant nil with next structure argument
`unify_void N`	skip next `N` structure arguments
procedural instructions	
`call P,N`	call procedure `P`, trim environment size to `N`
`execute P`	call procedure `P` (last subgoal)
`proceed`	return (last instruction of fact)
`allocate`	create an environment
`deallocate`	remove an enviroment
indexing and backtracking instructions	
`switch_on_term V,C,L,S`	four-way jump depending on the type of `A1`
`switch_on_constant N,T`	hashed jump (table `T` with size `N`) on constant in `A1`
`switch_on_structure N,T`	hashed jump (table `T` with size `N`) on structure in `A1`
`try_me_else L`	create choice point to `L`, then fall through
`retry_me_else L`	change retry address to `L`, then fall through
`trust_me_else_fail`	remove choice point, then fall through
`try L`	create choice point, then jump to `L`
`retry L`	change retry address, then jump to `L`
`trust L`	remove choice point, then jump to `L`

Figure 12: WAM instruction set

unification instructions	
`const C`	integer or atom
`nil`	empty list
`list`	list (followed by its arguments)
`struct F`	structure (followed by its arguments)
`void`	void variable
`fsttmp Xn`	first occurrence of temporary variable
`nxttmp Xn`	subsequent occurrence of temporary variable
`fxtvar Vn`	first occurrence of local variable
`nxtvar Vn`	subsequent occurrence of local variable

resolution instructions	
`goal P`	subgoal (followed by arguments and call/lastcall)
`nogoal`	termination of a fact
`cut`	cut
`builtin I`	builtin predicate (followed by its arguments)

termination instructions	
`call`	termination of a goal
`lastcall`	termination of last goal

Figure 14: $\mathrm{VAM}_{2\mathrm{P}}$ instruction set