THE VIENNA ABSTRACT MACHINE

ANDREAS KRALL

▷

The goal of Prolog implementations is to achieve high overall efficiency. Many high-speed implementations sacrifice the performance of the compilation built-in predicates for expensive optimizations. The Vienna Abstract Machine (VAM) aims at both, fast compilation and fast execution. Different versions of the VAM are used for different purposes: the VAM$_{2P}$ is well suited for interpretation; the VAM$_{1P}$ has been designed for native code compilation. The VAM$_{2P}$ has been modified to the VAM$_{AI}$, an abstract machine for fast abstract interpretation. This article presents all three versions, explains their implementations and compares them with state of the art Prolog systems.

◁

1. INTRODUCTION

The implementation of Prolog has a long history. Early systems were implemented by the group around Colmerauer [10] 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 [2]. 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 Fernando and Luis Pereira, David Warren developed the DEC-10 Prolog, the first Prolog compiler [33]. This compiler and the portable interpreter C-Prolog spread around the world and contributed to the success of Prolog.

Address correspondence to Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria, `andi@complang.tuwien.ac.at`.

In 1983 David Warren presented the Warren Abstract Machine (WAM) [34] which has been the basis of nearly all later Prolog implementations. The WAM divides the unification process into two steps. In the first step the arguments of the calling goal are created in or copied into argument registers. In the second step the values in the argument registers are unified with the arguments of the head of the called predicate. This concept is as well-suited for intermediate code interpreters as for compilers.

1.1. The Vienna Abstract Machine

The development of the Vienna Abstract Machine (VAM) started in 1985 as an alternative to the WAM. The aim was to eliminate the parameter passing bottleneck of the WAM. The first result of the development was an interpreter [17] which led to the $\mathrm{VAM}_{\mathrm{2P}}$ [23]. Partial evaluation of predicate calls led to the $\mathrm{VAM}_{\mathrm{1P}}$ which is well-suited for machine code compilation [20]. This compiler was enhanced by global analysis and modified to support incremental compilation [21]. Short analysis times were achieved through the $\mathrm{VAM}_{\mathrm{AI}}$, an abstract machine for abstract interpretation [22].

The $\mathrm{VAM}_{\mathrm{2P}}$ (VAM with two instruction pointers) is well-suited for an intermediate code interpreter implemented in C or in assembly language using direct threaded code [4]. The VAM eliminates the WAM register interface by performing the unification of each pair of a goal and a head argument in a single step. The goal instruction pointer refers to the instructions of the calling goal, the head instruction pointer to the instructions of the head of the called clause. An inference step of the $\mathrm{VAM}_{\mathrm{2P}}$ fetches one instruction from the goal and one instruction from the head, combines them and executes the combined instruction. Because information about both, the calling goal and the called head, is available simultaneously, more optimizations than in the WAM are possible. The VAM features cheap backtracking and cut, needs less dereferencing and trailing and has smaller stack sizes.

The $\mathrm{VAM}_{\mathrm{1P}}$ (VAM with one instruction pointer) uses only one instruction pointer and is well-suited for native code compilation. It combines goal and head 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.

A common solution for solving data flow analysis problems is abstract interpretation. To collect information about a program abstract interpretation executes the program over an abstract domain. Current abstract interpretation systems for Prolog were too slow for use in an optimizing Prolog compiler. So the $\mathrm{VAM}_{\mathrm{AI}}$ (VAM for abstract interpretation) has been designed. It is an order of magnitude faster than older abstract interpretation systems.

The article is structured as follows. Section 2 gives the details of the $\mathrm{VAM}_{\mathrm{2P}}$. Section 3 explains the principles of the $\mathrm{VAM}_{\mathrm{1P}}$. Section 4 describes the $\mathrm{VAM}_{\mathrm{AI}}$, a modified version of the $\mathrm{VAM}_{\mathrm{2P}}$ developed for abstract interpretation. Section 5 raises some implementation issues. Section 6 presents an evaluation of the performance of the VAM.

2. THE VAM$_{2P}$

The VAM$_{2P}$ execution model is very similar to the execution model of the classical Prolog interpreter described by Bruynooghe [5]. The main difference is that the unification has been broken up into a larger number of atomic parts, which leads to the definition of an instruction set and gives room for additional optimizations.

2.1. The VAM$_{2P}$ memory model

The VAM$_{2P}$ uses three stacks and one heap (see fig. 2.1): stack frames and choice points are allocated on the environment stack; structures and unbound variables are stored on the copy stack (also called global stack or heap in the VAM); bindings of variables are marked on the trail; the intermediate code of clauses is held in the code area (organized as heap). Machine registers (see fig. 2.2) are the `goalptr` and `headptr` (pointer to the code of the calling goal and the called clause, respectively), the `goalframeptr` and the `headframeptr` (pointer to the stack frame of the clause containing the calling goal and the called clause, respectively), the top of the environment stack (`stackptr`), the top of the copy stack (`copyptr`), the top of the trail (`trailptr`), and the pointer to the last choice point (`choicepntptr`).

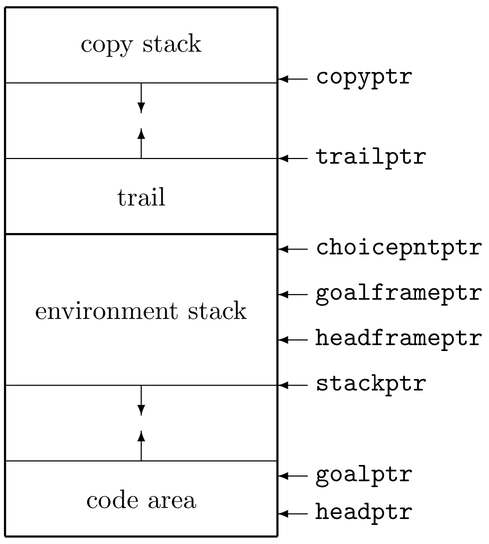

FIGURE 2.1. VAM$_{1P}$ data areas

Values are stored together with a tag in one machine word. The VAM distinguishes between integers, atoms, nil, lists, structures, unbound variables and references. Atoms contain a pointer to a character string. The technique of structure copying as described by Mellish [26] is used for the representation of structures. Lists and structures are represented as tagged pointers to a group of machine words. A list element uses two machine words, one for each argument. Structures use one

register	usage
`stackptr`	top of environment stack
`copyptr`	top of copy stack
`trailptr`	top of trail
`goalptr`	pointer to instructions of calling goal
`headptr`	pointer to instructions of called clause
`goalframeptr`	frame pointer of calling goal
`headframeptr`	frame pointer of called clause
`choicepntptr`	previous choice point

FIGURE 2.2. machine registers

machine word for the functor and one for each argument. The functor is a pointer to a location containing the arity and the atom. Long integers and floating point values are special structures. Unbound variables are represented as self references and are allocated on the copy stack in order to avoid dangling references and the unsafe variables of the WAM. Additionally the test for trailing of variables is simplified.

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, which treats the first subgoal as belonging to the head, in the VAM temporary variables occur only in the head or in one subgoal, counting a group of built-in predicates as a single goal. The built-in predicates following the head are treated as if they belong 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. These registers are stored in the stack frame prior to the call of the first subgoal. To avoid the initialization of variables, the first and further occurrences are distinguished.

2.2. The VAM$_{2P}$ instruction set

Prolog source code is translated to the VAM$_{2P}$ abstract machine code (see fig. 2.3). This translation is simple due to the direct mapping between source code and VAM$_{2P}$ code. The head arguments of a clause are translated to unification instructions. A fact is terminated by the instruction `nogoal`. Each subgoal is translated to the `goal` instruction, unification instructions for each argument and is terminated by the `call` instructions or the `lastcall` instruction if it is the last subgoal. The first subgoal of a clause has an additional operand, that specifies the number of local variables in the head. This number is needed for last-call optimization. Example 2.1 shows the VAM$_{2P}$ code for the `append` procedure.

Example 2.1.

```
append([],           nil
```

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 [N,]P`	subgoal (followed by arguments and call/lastcall)
`nogoal`	termination of a fact
`cut`	cut
`builtin I`	built-in predicate (followed by its arguments)
termination instructions	
`call`	termination of a goal
`lastcall`	termination of last goal

FIGURE 2.3. VAM$_{2P}$ instruction set

```
       L,             fsttmp L
       L              nxttmp L
       ).             nogoal

append([              list
       H|             fsttmp H
       L1],           fstvar L1
       L2,            fstvar L2
       [              list
       H|             nxttmp H
       L3]) :-        fstvar L3
append(               goal 3,append
       L1,            nxtvar L1
       L2,            nxtvar L2
       L3             nxtvar L3
       ).             lastcall
```

The translation of terms into intermediate code is done in two passes using three steps (see fig. 2.4). The first pass scans the terms for variables and collects information about the variables in the var table. The second pass again scans the terms and generates the VAM$_{2P}$ instructions. Between these two passes the variable classes and offsets are determined.

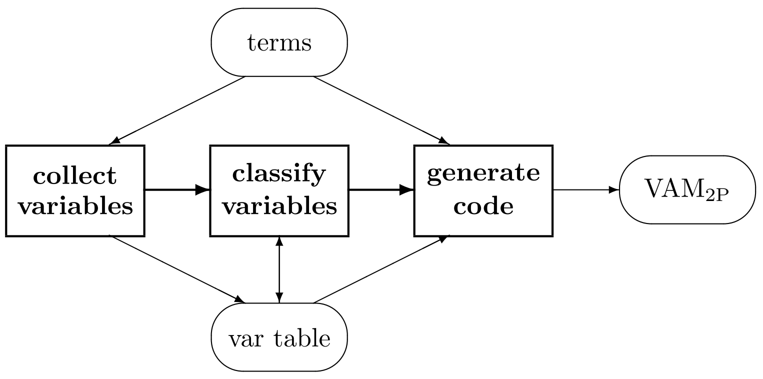

FIGURE 2.4. translator structure

2.3. The VAM$_{2P}$ execution model

A stack frame (see fig. 2.5) is allocated on the environment stack for each called procedure. For each non-deterministic procedure a choice point (see fig. 2.6) is pushed on the environment stack, too. The stack frame contains the variables and the continuation. The continuation is saved in the stack frame prior to the call of the first subgoal. The continuation contains the frame pointer of the calling clause (`goalframeptr'`) and the address of the instruction following the calling goal (`goalptr'`). If the calling goal was the last subgoal, the continuation of the caller is copied to the new stack frame.

`goalptr'`	continuation code pointer
`goalframeptr'`	continuation frame pointer
variable$_0$ ... variable$_n$	local variables

FIGURE 2.5. stack frame

`trailptr'`	copy of top of trail
`copyptr'`	copy of top of copy stack
`headptr'`	alternative clauses
`goalptr'`	restart code pointer
`goalframeptr'`	restart frame pointer
`choicepntptr'`	previous choice point

FIGURE 2.6. choice point

During unification of a goal with the head of a called clause the next goal and head instructions are fetched, the two instructions are combined, and the combined instruction is executed. Goal unification instructions are combined with head unification instructions and resolution instructions with termination instructions. To enable fast decoding, goal and head unification instructions are encoded differently and the instruction combination is performed by adding the instruction codes. Therefore, the sum of each two instruction codes must be unique. The C statement

```
switch(*headptr++ + *goalptr++)
```

implements this instruction fetch and decoding. An assembly language implementation can use direct threaded code for faster execution [4]. Direct threaded code uses the address of the interpretation routine of an instruction as intermediate code. Portability of an assembly language implementation is achieved by macro expansion of a low level virtual machine code into assembly language.

As in the WAM, unification of structures is solved by executing the interpreter in read mode or write mode. In these modes, instructions are fetched using only one of the two instruction pointers. The proper interpreter for the mode is called recursively for each recursive structure. Unification of void variables with structures leads to skipping the argument of the structure. Thus the $\mathrm{VAM}_{2\mathrm{P}}$ interpreter is executed either in *combine*, *read (unify)*, *write (create)* or *skip* mode. Example 2.2 shows a code fragment of the interpreter with the parts for the four different modes (there exist three additional parts for skipping, writing and reading goal structures). A one page interpreter for ground Prolog is contained in [23].

Example 2.2.

```
combine:    switch(*headptr++ + *goalptr++) {
                case (h_const+g_const): ... goto combine;
                case (h_list+g_void):   ... i = 2; goto head_skip;
                case (h_list+g_fstvar): ... i = 2; goto head_write;
                case (h_struct+g_nxtvar):
                    i = arity(headptr); goto head_read;
                ...
            }
            goto combine;

head_skip:  while (--i >= 0)
             switch (*headptr++) {
             case h_const: ...
             ...
             }
            goto combine;

head_write: ...

head_read:  ...
```

2.4. Built-in predicate interface

The simple built-in predicate interface is identical to that of the WAM. The instructions are scanned and the corresponding values are put in the argument array for the call of the built-in predicate. This model simplifies the meta-call of built-in predicates but is not very efficient. To avoid this bottleneck, the arithmetic and some type checking built-in predicates scan the instructions on their own.

2.5. Meta-call

For a meta-call a special `metacall Vn` instruction is generated instead of the `goal` instruction. The operand `Vn` contains the offset of the meta-call variable in the stack frame. A meta-call is terminated by a `call` or `lastcall` instruction. At run time the variable contains an atom or a structure. The functor of a structure refers to code of the corresponding procedure. The `headptr` is set to the code of the first clause and the interpreter continues executing instructions in read mode. For backtracking of meta-calls it is necessary that the instruction combination of `metacall` with head unification and control instructions is unique.

2.6. Last-call optimization

The WAM implements last-call optimzation by copying the variables of the caller's stack frame into argument registers. Then the new stack frame is allocated in the place of the old one. The VAM first allocates the new stack frame and performs the unification between the caller and the callee. Afterwards, if the call of a procedure is deterministic, prior to the call of the first subgoal the stack frame of the called procedure is copied over the stack frame of the calling goal. For this purpose the optional operand `N` of the first `goal` instruction contains the number of variables to be copied. In an assembly language implementation some head variables can be held in registers. Prior to the call of the first subgoal these registers can be stored in the place of the caller's stack frame. A more efficient version can be implemented with the VAM$_{1P}$ (see chapter 3.1).

2.7. Clause indexing

In contrast to the WAM the VAM$_{2P}$ does not translate indexing information into instructions but stores this information in indexing data structures. Since the VAM$_{2P}$ unifies the goal arguments from left to right, only first argument indexing can be supported. The current implementation uses a balanced binary tree for clause selection. The index values are either integers, atoms or the functors of structures. Each clause gets an additional header (see fig. 2.7) which contains a pointer to the

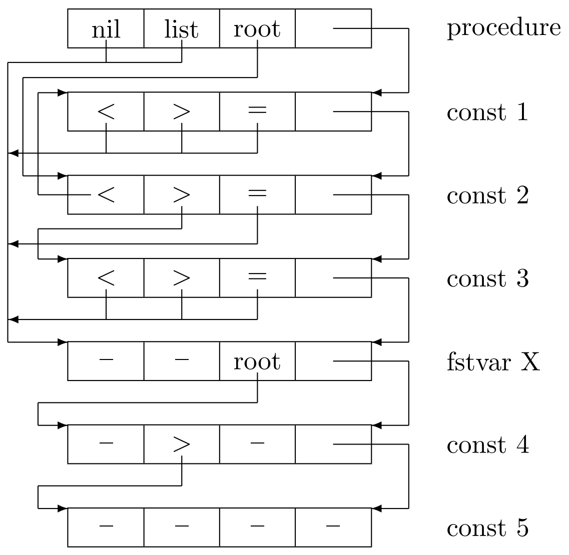

FIGURE 2.7. clause indexing

clauses with smaller indices, a pointer to the clauses with larger indices, a pointer to the clauses with the same index and a pointer for linear connection of the clauses.

For each procedure there exists a header which contains a pointer to the clauses with nil as first argument, a pointer to the clauses with a list as first argument, a pointer to the root of the binary search tree and a pointer to the first clause. If there exists a clause with a variable as first argument, the leaf clauses of the binary search tree point to this clause and the header contains pointers to the nil clauses, the list clauses, the root of the next search tree and the linear connection.

To simplify the implementation of the indexing part of the interpreter a special case of the `goal` instruction is used. The instruction `xgoal [N,]P,Vn` is generated if the first argument of a subgoal is a variable. The operand `Vn` contains the offset of this variable in the stack frame. The choice point contains a flag which indicates if indexing was used for the call of this procedure.

The previously presented indexing scheme does not belong to the definition of the VAM$_{2P}$, but is an implementation model well-suited for this abstract machine. The balanced binary search tree could be replaced by a hash table. Since the number of clauses in a procedure is in general very small, the search tree is faster and better suited for incremental compilation. An indexing scheme as used by Carlson [7] or Demoen [13] for the WAM would be suitable, too.

2.8. Design alternatives

An important design decision is the allocation of unbound variables on the copy stack. This is a prerequisite for a fast last-call optimization and simplifies the check for the trailing of variables. An alternative solution would be to prohibit temporary variables in a subgoal (which eliminates references to the callee's stack frame) and use unsafe variables as the WAM does. The stack usage of both versions is very similar (the VAM only creates 36Therefore, the method to store unbound variables on the copy stack is simpler and faster. Updating the references in the moved stack frame as proposed by Bruynooghe in [5] is too costly.

It is difficult to decide if the choice point should be allocated before or after the variables in the stack frame. If it is allocated before the variables, fast shallow backtracking can be implemented by updating only the pointer to the alternative clauses. If the choice point is allocated after the variables, the space consumed by the choice point can be reclaimed easily when executing a cut. Since cuts occur more frequently than choice point updates [18], the current implementation allocates the choice point after the variables and goes for space instead of speed. Furthermore, the cut does stack trimming and removes the variables in the stack frame which are not used in later subgoals.

The size of the choice point could be reduced further by eliminating the restart code pointer and restart frame pointer by using the continuation instead. This would require an additional argument to the `goal` instruction which contains an offset to the next goal and a `nogoal` instruction after each lastcall. The disadvantage of this solution is that the continuations must be dereferenced if no last-call optimization is applicable.

3. THE VAM$_{1P}$

The VAM$_{1P}$ has been designed for native code compilation. A complete description can be found in [20]. The main difference to the VAM$_{2P}$ is that instruction combination is done during compile time instead of run time. The VAM$_{1P}$ generates specialized code for each call of a procedure. For example

```
a([]).
a([_|L]) :- a(L).

:- a([3]).
```

is translated into VAM$_{1P}$ instructions (represented as unifications in pseudo-code)

```
a2:  :- (a(Lg)=a([])) ; (a(Lg)=a([_|Lh]), goto(a2).

:- (a([3])=a([])) ; (a([3])=a([_|L]), goto(a2).
```

The representation of data, the stacks and the stack frames (see fig. 2.5) are identical to the VAM$_{2P}$. The two instruction pointers `goalptr` and `headptr` are replaced by one instruction pointer called `codeptr`. The choice point (see fig. 3.1) is smaller by one element. The pointer to the alternative clauses points directly to the code of the remaining matching clauses.

`trailptr'`	copy of top of trail
`copyptr'`	copy of top of copy stack
`headptr'`	alternative clauses
`goalframeptr'`	restart frame pointer
`choicepntptr'`	previous choice point

FIGURE 3.1. choice point

3.1. Basic optimizations

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

All constants and functors are combined and evaluated to true or false at compile time. No code is emitted for a true result. Clauses which contain 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 creating structures is split and different code is generated for each path. This makes it possible to refer to each argument of a structure through an offset from the top of the copy stack or from the base pointer of the structure. If a temporary variable is contained in more than one structure, combined unification or copying instructions are generated. The two stream method as described by e.g. Meier [24] would have been slower than the generation of flattened code but would reduce the code size.

All necessary information for clause indexing is computed at compile time. Some alternatives may be 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 $\mathrm{VAM}_{\mathrm{2P}}$ a balanced binary tree is used for clause selection. A version of the $\mathrm{VAM}_{\mathrm{1P}}$ [19] which uses a complete indexing as described by Hickey and Mudambi [16] has been implemented too.

The $\mathrm{VAM}_{\mathrm{1P}}$ implements two versions of last-call optimization. The first variant (called post-optimization) is identical to that of the $\mathrm{VAM}_{\mathrm{2P}}$. If a goal is deterministic at run time, the registers containing the head variables are stored in the caller's 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 determinism of a clause can be detected at compile time, the space used by the caller's stack frame is used immediately by the callee. Therefore, all unifications between variables with the same offset can be eliminated. If not all head variables are held in registers, they have to be read and written in the right order. This optimization, called pre-optimization, can be seen as a generalization of recursion replacement by iteration to every last-call [25].

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 of 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.

3.2. The VAM$_{1P}$ instructions set

The instruction set for the VAM$_{1P}$ is divided into general unification instructions, structure unification instructions, structure creation instructions, control instructions, indexing instructions and optimization instructions. In the following descriptions variables used as arguments are either registers or frame pointers with offset.

The unification instructions (see fig. 3.2) handle the unification of an argument of the calling goal and the related head argument of the called clause. If a structure is combined with the first occurrence of a variable (`fstvar_struct` structure creation is started. The operand `Size` gives the size of the structure including all substructures. The operand `Reg` of the `nxtvar_struct` instruction is the register which holds the base pointer of the structure. The structure unification instructions follow immediately, the operand `CAddr` is the label of the structure creation instructions. Fig. 3.3 gives an example for the code generation of structures.

general unification
`fstvar_const Var,Const`
`nxtvar_const Var,Const`
`fstvar_nil Var`
`nxtvar_nil Var`
`fstvar_list Var,Size`
`nxtvar_list Var,Reg,Size,CAddr`
`fstvar_struct Var,Functor,Size`
`nxtvar_struct Var,Functor,Reg,Size,CAddr`
`fstvar_fstvar Var,Var`
`nxtvar_fstvar Var,Var`
`nxtvar_nxtvar Var,Var`
`fstvar_void Var`

FIGURE 3.2. VAM$_{1P}$ general unification instructions

Structure unification (see fig. 3.4) and structure creation instructions (see fig. 3.5) are equivalent to the unify instructions of the WAM in read and write mode. Due

```
            nxtvar_struct  X,s/2,r0,5,label_2
            unify_const    1,1(r0)
            unify_struct   t/1,r1,2,label_3
            unify_const    2,1(r1)
label_1:    ...
label_2:    create_functor s/2,-5(copyptr)
            create_const   1,-4(copyptr)
            create_struct  -3(copyptr),-2(copyptr)
label_3:    create_functor t/1,-2(copyptr)
            create_const   2,-1(copyptr)
            goto           label_1
```

FIGURE 3.3. flattened structure unification code for X=s(1,t(2))

to the splitting of the paths for unification and creation, the variable's offset from the top of the copy stack or the start of the structure is known at compile time.

structure unification
`unify_const Const,Offset`
`unify_nil Offset`
`unify_list Reg,Size,Offset,CAddr`
`unify_struct Functor,Reg,Size,Offset,CAddr`
`unify_fstvar Var,Offset`
`unify_nxtvar Var,Offset`

FIGURE 3.4. $\mathrm{VAM}_{\mathrm{1P}}$ structure unification instructions

The control instructions (see fig. 3.6) implement stack handling and the transfer of control: `alloc_stackframe` allocates the space for variables in a stack frame; `adjust` performs stack trimming; `store_cont_goto` saves the continuation in the stack frame and jumps to the procedure `ProcAddr`. For the last subgoal the instruction `copy_cont_goto` is used instead. `read_cont_goto` reads the continuation and executes the next subgoal. `store_var` saves the head variables residing in registers in the stackframe. If not all head variables can be held in registers, the instruction `copy_var` copies the variables from the callee's stack frame to the caller's stack frame during last-call optimization. `push_choice_point` creates a choice point with `AltAddr` used as the pointer to the code of the remaining clauses. `cut` deletes the choice points and performs stack trimming.

Depending on the type of the value contained in the variable `Var`, the indexing instruction `index_var` (see fig. 3.7) branches to one of the labels for lists, nil, constants and structures. The compare instruction `cmp_const` implements a binary search tree.

The optimization instructions (see fig. 3.8) support the unification of structures optimized by temporary variable elimination or loop optimization. `create_undef`

structure creation
`create_const Const,Offset`
`create_nil Offset`
`create_list Offset,Offset`
`create_struct Offset,Offset`
`create_functor FunctorOffset`
`create_fstvar Var,Offset`
`create_nxtvar Var,Offset`

FIGURE 3.5. VAM$_{1P}$ structure creation instructions

control
`alloc_stackframe VarCount`
`adjust VarCount`
`store_cont_goto ContAddr,ProcAddr`
`read_cont_goto`
`copy_cont_goto ProcAddr`
`goto Addr`
`store_var Reg,Offset`
`copy_var Offset`
`push_choice_point AltAddr`
`cut VarCount`
`call_bip BipId`
`inline_bip BipId`

FIGURE 3.6. VAM$_{1P}$ control instructions

initializes a cell on the copy stack to unbound. `unify_create` copies a value from the source structure to the destination structure. `unify_unify` implements full unification between two arguments of structures. `create_ref` lets one argument of a structure reference the other. If the cell at `SAddr` contains a structure, `extract_struct` stores the address of the start of the structure in register `Reg`. Otherwise it continues execution at `ContAddr`. `construct_struct` creates a structure. `construct_extract_struct` is the combination of these two instructions.

3.3. Code size

The size of the generated code can become quite large since for each call of a procedure specialized code is generated especially if there exist many calls of a procedure consisting of many clauses. But since many of these calls have the same calling pattern, the calls can share the same generated code. If this is not sufficient, a dummy call must be introduced between the call and the procedure, leading to an interface which resembles that of the WAM.

indexing
`index_var Var,LAddr,NAddr,CAddr,SAddr`
`cmp_const Const,LessAddr,GreaterAddr`

FIGURE 3.7. VAM$_{1P}$ indexing instructions

optimization
`create_undef Offset`
`unify_create Offset,Offset`
`unify_unify Offset,Offset`
`create_ref Offset,Offset`
`extract_list Reg,Laddr,ContAddr`
`construct_list Offset`
`construct_extract_list Reg,LAddr,UAddr`
`extract_struct Functor,Reg,Size,Saddr,ContAddr`
`construct_struct Functor,Size,Offset`
`construct_extract_struct Functor,Reg,Size,SAddr,UAddr`

FIGURE 3.8. VAM$_{1P}$ optimization instructions

4. THE VAM$_{AI}$

During the execution of a Prolog program a great amount of time is spent in useless type tests and dereferencing. This code can be eliminated if more information about variables is available to the compiler. Information about types, modes, trailing, reference chain lengths and aliasing of the variables of a program can be inferred using abstract interpretation.

4.1. Abstract Interpretation

Abstract interpretation is a technique for global data flow analysis of programs. It was introduced by the Cousots [11] for data flow analysis of imperative languages. This work was the basis of much of the recent work in the field of declarative and logic programming [1] [6] [9] [12] [15] [27] [29] [31]. 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 reaches a fixpoint. Termination is assured by limiting the size of the domain. Most of the previously cited systems are meta-interpreters written in Prolog which are very slow.

A practical implementation of abstract interpretation has been done by Tan and Lin [30]. 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, and analysed a set of small benchmark programs in a few milliseconds. This is about 150 times faster than the previous systems.

4.2. A basic execution model of the VAM$_{AI}$

The VAM$_{\mathrm{AI}}$, an abstract machine for abstract interpretation, has been designed following the way of Tan and Lin. It has been developed on the basis of the VAM$_{\mathrm{2P}}$ and benefits from the fast decoding mechanism of this machine. The inferred data flow information is stored directly in the intermediate code of the VAM$_{\mathrm{AI}}$. The VAM was chosen as the basis of an abstract machine for abstract interpretation because it is much better suited than the WAM: The parameter passing of the WAM via registers and storing registers in backtrack points slow down the interpretation. Furthermore, in the WAM some instructions are eliminated so that the relation between argument registers and variables is sometimes difficult to determine. The translation to a VAM$_{\mathrm{2P}}$-like intermediate code is much simpler and faster than WAM code generation. A VAM$_{\mathrm{2P}}$-like interpreter enables the modeling of low level features of the VAM. Furthermore, the VAM$_{\mathrm{2P}}$ intermediate code is needed for the generation of the VAM$_{\mathrm{1P}}$ instructions.

A top-down approach is used for the analysis of the desired information. Different (static) calls to the same clause are handled separately to get more exact types. This is achieved by duplicating the clauses for each call of a procedure. So for each call of a goal there exists an own copy of the intermediate code of the called procedure. To save code size, only the heads of the clauses are copied. The bodies are shared. This duplication of the code gives a more precise analysis for the use in the VAM$_{\mathrm{1P}}$ which generates specialized code for each call and simplifies many parts of the VAM$_{\mathrm{AI}}$.

Recursive calls of a clause are computed until a fixpoint for the gathered information is reached. If there already exists information about a call and the new gathered information is more special than the previously derived one, i.e. the union of the old type and the new type is equal to the old type, a fixpoint has been reached and the interpretation of this call to the clause is terminated.

Abstract interpretation with the VAM$_{\mathrm{AI}}$ is demonstrated by a short example. Fig. 4.1 shows a simple Prolog program part and a simplified view of its code duplication for the representation in the VAM$_{\mathrm{AI}}$ intermediate code.

The procedure B has two clauses, the alternatives B_1 and B_2. The code for the procedures B and C is duplicated because both procedures are called twice in this program. Abstract interpretation starts at the beginning of the program with the clause A_1^1. The information of the variables in the subgoal B^1 are determined by the inferable data flow information from the two clauses B_1^1 and B_2^1. After the information for both clauses has been computed, abstract interpretation is finished because there is no further subgoal for the first clause A_1.

In the conservative scheme it has to be supposed that both B_1^1 and B_2^1 could be reached during program execution. Therefore, the union of the derived data flow information sets for the alternative clauses of procedure B has to be formed. For B_1^1 only information from C_1^1 has to be derived because it is the only subgoal for B_1^1. For

Prolog program:

A_1 :– B^1
B_1 :– C^1
B_2 :– B^2, C^2
C_1 :– *true*

Code representation:

A_1^1 :– B^1
B_1^1 :– C^1
B_2^1 :– B^2, C^2
B_1^2 :– C^1
B_2^2 :– B^2, C^2
C_1^1 :– *true*
C_1^2 :– *true*

FIGURE 4.1. Prolog program part and its representation in VAM$_{\mathrm{AI}}$

B_2^1 there exists a recursive call of B, named B^2. Recursion in abstract interpretation is handled by computing a fixpoint, i.e. the recursive call is interpreted as long as the derived data information changes. After the fixpoint has been reached, computation stops for the recursive call. The data flow information for the recursion is assigned to the clauses B_1^2 and B_2^2. After all inferable information has been computed for a clause, it is stored directly into the intermediate code. The entry pattern and success patterns are stored in the head variables information fields, the variables of a subgoal contain the the success patterns of the calls of subgoals at the left to the current subgoal. The same intermediate code is used efficiently in the next pass of the compiler that generates code.

4.3. The abstract domain

The goal of the VAM$_{\mathrm{AI}}$ is to gather information about mode, type, reference chain length and aliasing of variables. Reference chain lengths of 0, 1 and greater 1 are distinguished. The type of a variable is represented by a set comprised of following simple types:

free	is an unbound variable and contains a reference to all aliased variables
list	is a non empty list (it contains the types of its arguments)
struct	is a term
nil	represents the empty list
atom	is the set of all atoms
integer	is the set of all integer numbers

Possible infinite nesting of compound terms makes the handling of the types *list* and *struct* difficult. To gather useful information about recursive data structures a recursive list type is introduced which contains also the information about the termination type.

To represent the alias information, variables are collected in alias sets. Variables which could possibly be aliased are in the same set. The alias sets are represented as double linked sorted lists. In the intermediate code each set is identified by an unique number. Variables which are always aliased, can be represented by references like in ordinary Prolog interpreters. The intersection of this sets has to be stored in the intermediate code.

Efficient interpretation is achieved by using fixed-sized variable cells, which enables static stack frame size determination and the saving of the domains in intermediate code fields. The set of the domain values is represented as a bit field. Set operations like union or difference can be implemented using logical operations. The computation of the least upper bound of two domains is implemented by a *bitwise or* operation, the abstract unification by a *bitwise and*.

4.4. The VAM$_{AI}$ instruction set

The representation of the arguments of a Prolog term is the same as that in the VAM$_{2P}$ (see fig. 4.2) with the following exceptions:

- Local variables have four additional information fields in their intermediate code: the actual domain of the variable, the reference chain length and two fields for alias information. These information fields replace the extension table of conventional abstract interpretation algorithms. Local variables of the head have split information fields because they store the information at both the entry of the clause and after a successful computation of this clause. This information is used for the handling of recursive calls.
- The argument of a temporary variable contains an offset which references this variable in a global table. The global table contains entries for the domain and reference chain length information or a pointer to a variable.
- The intermediate code `lastcall` has been removed because last-call optimization makes no sense in abstract interpretation. Instead, the intermediate code `nogoal` indicates the end of a clause. When this instruction is executed, the computation continues with the next alternative clause (artificial fail).
- The intermediate code `goal` got an additional argument: a pointer to the end of this goal. This eliminates the distinction between the continuation and the restart code pointer (see fig. 2.6).
- The instruction `const` has been split into `integer` and `atom`.

4.5. The VAM$_{AI}$ execution model

Another significant difference to the VAM$_{2P}$ concerns the data areas: While the VAM$_{2P}$ needs three stacks, in VAM$_{AI}$ a modified environment stack and a trail are

unification instructions	
`int I`	integer
`atom A`	atom
`nil`	empty list
`list`	list (followed by its two arguments)
`struct F`	structure (functor)(followed by its arguments)
`void`	void variable
`fsttmp Xn`	first occurrence of temporary var (offset)
`nxttmp Xn`	further occurrence of temporary var (offset)
`fstvar Vn,D,R,Ai,Ac`	first occurrence of local var (offset, domain, ref. chain length, is aliased, can be aliased)
`nxtvar Vn,D,R,Ai,Ac`	further occurrence of local var (offset, domain, ref. chain length, is aliased, can be aliased)

resolution instructions	
`goal P,O`	subgoal (procedure pointer, end of goal)
`nogoal`	termination of a clause
`cut`	cut
`builtin I`	built-in predicate (built-in number)

termination instructions	
`call`	termination of a goal

FIGURE 4.2. VAM$_{\text{AI}}$ instruction set

sufficient. Fig. 4.3 shows a stack frame for the environment stack of the VAM$_{\text{AI}}$. Note that every stack frame is a choice point because all alternatives for a call are considered to be the result of the computation. Similar to CLP systems the trail is implemented as a value trail. It contains both, the address of the variable and its content.

domain for variable n
⋮
domain for variable 1
goalptr
clauseptr
goalframeptr
trailptr

FIGURE 4.3. structure of the stack frame

The stack frame contains the actual information for all local variables of a clause. The register `goalptr` points to the intermediate code of a goal. It allows to find the continuation after a goal has been computed. Register `clauseptr` points to the head of the next alternative clause for the called procedure, and `goalframeptr` points to the stack frame of the calling procedure.

reference	
domain	ref-len
alias-prev	alias-next
union-domain	union-ref-len
union-prev	union-next

FIGURE 4.4. a local variable on the stack

Fig. 4.4 is a detailed description of the stack entry for a local variable. The fields *reference*, *domain*, *ref-len*, *alias-prev* and *alias-next* hold the information derived for a variable by analysing a single alternative of the current goal. The *union* fields get the union of all previously analysed alternatives.

The *reference* field connects the caller's variables with the callee's variables. Aliased variables are stored in a sorted list. The fields *alias-prev* and *alias-next* connect the variables of this list. The *domain* field contains all actual type information at each state of computation. Its contents may change at each occurrence of the variable in the intermediate code. The *ref-len* field contains the length of the reference chain. After analysing an alternative of a goal, the *union* fields contain the least upper bound of the information of all alternatives analysed so far.

4.6. Handling of recursion

The information in the fields of local variables of a clause head is used for fixpoint computation. These fields hold information for these local variables at both, the entry of the clause and at the successful computation of the clause, i.e. the success pattern. When the interpreter reaches the last instruction of a clause (*nogoal*), the success pattern has to be updated. The success pattern fields of the clause's head variables are replaced with the least upper bound of their actual entries (the old success pattern) and the new variable domains. These new domains can be found on the stack after the computation of the clause.

During abstract unification of goal and head arguments the entry pattern for head variables is stored in the intermediate code of the head if this call is computed the first time. If the intermediate code information already contains entry pattern information, the old information is replaced with the least upper bound of the new and the old information. If the information in the head's intermediate code fields do not change, i.e. the new entry pattern contains more special or equal information than patterns applied previously, there is no sense in a further recomputation of the clause. Instead, information about the clause's actual success pattern is gained from the actual intermediate code fields of the head. This information is then used in the variables occurring in the calling goal, and the interpreter computes the next alternative or the next subgoal of the calling clause if there are no more alternatives to compute. Whenever the success pattern of a clause changes, a flag is set in this clause and all of its calling clauses. The flag marks these clauses for recomputation. Interpretation is iterated until no success pattern changes any more.

4.7. Incremental abstract interpretation

The VAM$_{AI}$ is well suited also for incremental abstract interpretation. Incremental abstract interpretation is similar to recomputation if a success pattern has changed. Incremental abstract interpretation starts local analysis with all callers of the modified procedures and interprets the intermediate code of all dependent procedures. Interpretation is stopped when the derived domains are equal to the original domains (those derived by the previous analysis).

To make incremental abstract interpretation possible, pointers to the callers of each procedure are stored in the VAM$_{AI}$ code. They help in finding the top goal of the whole program. The pointer chain for a procedure is used in reconstructing the contents of the stack prior to the call of this procedure. Now, abstract interpretation can be executed as usual. In general, only a small part of the program is reinterpreted. In the worst case, incremental interpretation has to walk through the whole program.

5. IMPLEMENTATION ISSUES

5.1. A mixed interpreter system

The advantage of the VAM$_{2P}$ is its compact intermediate code size. The advantage of the VAM$_{1P}$ is its fast execution. It is possible to build an interpreter using a combination of these two abstract machines. The idea is to use the VAM$_{1P}$ only at self recursive calls of the last subgoal and the VAM$_{2P}$ otherwise. The VAM$_{1P}$ code can be either translated to machine code or executed by an intermediate code interpreter. This speeds up the often used loops and uses the compact representation for the other parts of a program.

5.2. The incremental compiler

The compilation of a Prolog program is carried out in five passes (see fig. 5.1). In the first pass a clause is read in by the built-in predicate `read` and transformed to term representation. The built-in predicate `assert` comprises the remaining passes. The compiler first translates the term representation into VAM$_{AI}$ intermediate code. Incremental abstract interpretation is executed on this intermediate code and the code is annotated with type, mode, alias and dereferencing information. The VAM$_{AI}$ intermediate code is traversed again, compiled to VAM$_{1P}$ code and expanded on the fly to machine code. The last step is instruction scheduling of the machine code and patching of branch offsets and address constants.

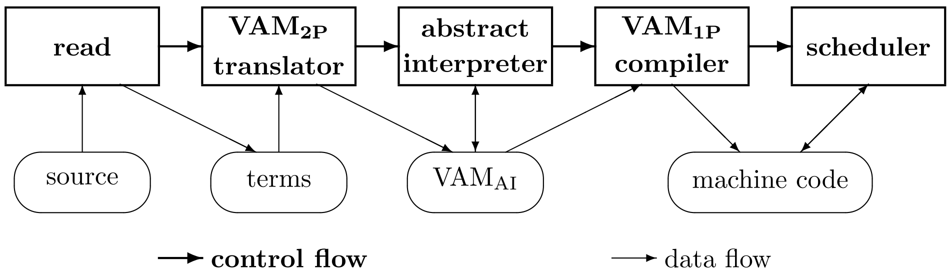

FIGURE 5.1. compiler passes

6. RESULTS

To evaluate the performance of the Vienna Abstract Machine we executed the well-known benchmarks described by Beer in [3]. These benchmarks were executed on a DECStation 5000/200 (25 MHz R3000) with 40 MB Memory. Following systems were benchmarked: a VAM_{2P} intermediate code interpreter written in C; SICStus Prolog [8], a WAM based intermediate code interpreter implemented in C; the VAM_{1P} compiler with global data flow analysis; the Aquarius compiler of Peter Van Roy [29] and the Parma system of Andrew Taylor [32]. The Parma system was not available to us, so the benchmark data reported in Van Roys article [28] are used. Data about the native code compiler of SICStus Prolog is extracted from Haygoods article [14]. Table 6.1 shows that the VAM_{1P} compiler produces faster code than the Aquarius system. Global analysis improved the execution time of the VAM_{1P}) by about 25other built-in predicates are compiled to machine code the compiler should achieve similar speedups on larger programs. The SICStus native code compiler produces code which is two to three times faster than the SICStus emulator for small benchmarks and 1.5 to two times faster for large programs.

	interpreters			compilers		
test	VAM_{2P} ms	VAM_{2P} scaled	SICStus scaled	VAM_{1P} scaled	Aquarius scaled	Parma scaled
det. append	0.25	1	1.1	26.1	19.3	-
naive reverse	4.17	1	1.06	20.0	14.5	25.6
quicksort	6.00	1	1.1	18.1	14.9	28.0
8-queens	65.4	1	1.1	13.5	15.4	-
serialize	3.90	1	0.83	6.84	4.26	16.4
differentiate	1.14	1	0.99	8.14	7.13	14.6
query	41.7	1	0.89	9.70	8.25	13.2
bucket	247	1	0.88	5.24	3.71	-
permutation	2660	1	0.70	6.48	6.96	-

TABLE 6.1. execution time, factor of improvement compared to the VAM_{2P}

In table 6.2 the compile time of the VAM_{1P} compiler is compared to that of the VAM_{2P} and SICStus intermediate code translators. It shows that the optimizing VAM_{1P} compiler is about ten times slower than the VAM_{2P} intermediate code translator but about two times faster than the simple SICStus intermediate code translator. The SICStus native code compiler is a factor of two slower than the SICStus intermediate code translator. The Aquarius compiler is by a factor of about 2000 slower than the VAM_{2P} translator. However, a direct comparison is not objective since the Aquarius compiler has three passes which communicate with the assembler and linker via files.

	VAM_{2P}	VAM_{2P}	VAM_{1P}	SICStus
test	ms	scaled	scaled	scaled
det. append	5.78	1	11.43	21.5
naive reverse	7.31	1	10.5	19.3
quicksort	9.30	1	9.9	23.1
8-queens	9.18	1	11.6	19.7
serialize	11.36	1	11.22	19.2
differentiate	13.71	1	11.41	30.3
query	21.05	1	7.5	13.4
bucket	15.59	1	7.25	12.7
permutation	4.88	1	8.88	18.1

TABLE 6.2. compile time, compared to the VAM_{2P}

The comparison of the VAM_{2P} interpreter with the VAM_{1P} shows that the size of the generated machine code is about a factor of ten larger than the internal representation of the VAM_{2P} (see table 6.3). The annotated VAM_{AI} intermediate code is about three times larger than the simple VAM_{2P} intermediate code. VAM_{2P} intermediate code has about the same size as SICStus intermediate code. SICStus native code is about twice as large as SICStus intermediate code [14].

7. CONCLUSION

We presented the Vienna Abstract Machine, an abstract machine for Prolog. Different versions are used as an interpreter, a compiler and a base for abstract interpretation. All three versions combine short translation times with fast execution.

ACKNOWLEDGEMENT

I thank all the people who contributed to the success of the Vienna Abstract Machine: Professor Manfred Brockhaus initiated this project and made this work possible. Eva Kühn implemented the first VAM_{2P} translator, a memory manager and some of the built-in predicates and had to cope with the frequent changes of

test	VAM_{2P} bytes	VAM_{2P} scaled	VAM_{AI} scaled	VAM_{1P} scaled
det. append	288	1	3.63	9.96
naive reverse	380	1	3.59	11.3
quicksort	764	1	2.65	9.95
8-queens	536	1	2.95	8.25
serialize	1044	1	3.33	15.7
differentiate	1064	1	8.37	28.4
query	2084	1	0.89	3.13
bucket	996	1	1.96	9.75
permutation	296	1	2.77	6.21

TABLE 6.3. code size of intermediate representations

the instruction set in the beginning. Martin Wais developed an Intel x86 emulator and Herbert Pohlai the portable Motorola 86k emulator. Franz Puntigam implemented a memory management system. Ulrich Neumerkel developed the concept of meta-structures and designed a garbage collector. Thomas Berger implemented the prototype VAM_{1P} compiler and abstract interpretation. Dongyang Cheng helped implementing the VAM_{AI} and the incremental compiler. I express my thanks to Thomas Berger, Anton Ertl, Franz Puntigam and the anonymous referees for their comments on earlier drafts of this paper.

REFERENCES

1. Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.
2. 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.
3. Joachim Beer. *Concepts, Design, and Performance Analysis of a Parallel Prolog Machine*. Springer, 1989.
4. James R. Bell. Threaded code. *CACM*, 16(6), June 1973.
5. Maurice Bruynooghe. The memory management of PROLOG implementations. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*. Academic Press, 1982.
6. Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic programming*, 10(1), 1991.
7. Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In *Fourth International Conference on Logic Programming*. MIT Press, 1987.
8. Mats Carlsson and J. Widen. SICStus Prolog user's manual. Research Report R88007C, SICS, 1990.
9. Baudouin Le Charlier and Pascal Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM TOPLAS*, 16(1), 1994.

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

11. 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.

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

13. Bart Demoen, Andre Marien, and Alain Callebaut. Indexing prolog clauses. In *North American Conference on Logic Programming.* MIT Press, 1989.

14. Ralph Clarke Haygood. Native code compilation in SICStus Prolog. In *Eleventh International Conference on Logic Programming.* MIT Press, 1994.

15. Manuel Hermenegildo, Richard Warren, and Saumya K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(2), 1992.

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

17. Andreas Krall. Implementation of a high-speed Prolog interpreter. In *Conf. on Interpreters and Interpretative Techniques*, volume 22(7) of *SIGPLAN.* ACM, 1987.

18. Andreas Krall. An empirical study of the Vienna Abstract Machine. Bericht TR 1851/90/1, Institut für Computersprachen, TU Wien, 1990.

19. Andreas Krall. Clause indexing in VAM and WAM based compilers. In *Second International Workshop on Functional/Logic Programming*, Bericht 9311. Ludwig-Maximilians-Universität München, 1993.

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

21. Andreas Krall and Thomas Berger. Incremental global compilation of Prolog with the Vienna Abstract Machine. In *Twelfth International Conference on Logic Programming*, Tokyo, 1995. MIT Press.

22. Andreas Krall and Thomas Berger. The $\mathrm{VAM}_{\mathrm{AI}}$ - an abstract machine for incremental global dataflow analysis of Prolog. In *ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages.* Science University of Tokyo, 1995.

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

24. Micha Meier. Compilation of compound terms in Prolog. In *North American Conference on Logic Programming*, 1990.

25. Micha Meier. Recursion vs. iteration in Prolog. In *Eighth International Conference on Logic Programming*, Paris, 1991. MIT Press.

26. 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.

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

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

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

30. 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.

31. Andrew Taylor. Removal of dereferencing and trailing in Prolog compilation. In *Sixth International Conference on Logic Programming*, Lisbon, 1989. MIT Press.

32. Andrew Taylor. LIPS on a MIPS. In *Seventh International Conference on Logic Programming*, Jerusalem, 1990. MIT Press.

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

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