Incremental Global Compilation of Prolog with the Vienna Abstract Machine

Andreas Krall and Thomas Berger
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien, Austria
{andi,tb}@mips.complang.tuwien.ac.at

Abstract

The Vienna Abstract Machine (VAM) is an abstract machine which has been designed to eliminate some weaknesses of the Warren Abstract Machine (WAM). Different versions of the VAM are used for different purposes. The VAM_{2P} is well suited for interpretation, the VAM_{1P} is aimed for native code generation. The VAM_{2P} has been modified to the VAM_{AI}, a concept suited for abstract interpretation. Analysis with the VAM_{AI} is so fast that it is feasible to support both, global analysis and database updates with `assert` and `retract`. We present an incremental compiler based on the VAM_{1P} and the VAM_{AI}. A preliminary evaluation of our compiler shows that the generated code competes with the best existing compilers whereas the compile time is comparable to that of simple bytecode translators.

1 Introduction

The development of the Vienna Abstract Machine (VAM) started in 1985 as an alternative to the Warren Abstract Machine (WAM) [18]. The aim was to eliminate the parameter passing bottleneck of the WAM. The development started with an interpreter [8] which led to the VAM_{2P} [10]. Partial evaluation of predicate calls led to the VAM_{1P}, which is well suited for machine code compilation [9]. The first compiler was a prototype implemented in Prolog without any global analysis. This compiler was enhanced by global analysis and modified to support incremental compilation. This prototype implementation was quite slow. So we designed the VAM_{AI}, an abstract machine for abstract interpretation, and implemented the whole incremental compiler in the programming language C.

The paper is structured as follows. Section 2 introduces the Vienna Abstract Machine with its two versions VAM_{2P} and VAM_{1P}. Section 3 describes the incremental compiler and gives a detailed introduction to the VAM_{AI}. Section 4 presents some results about the efficiency of the compiler.

2 The Vienna Abstract Machine

2.1 Introduction

The WAM divides the unification process into two steps. During the first step the arguments of the calling goal are copied into argument registers. 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 a single step.

A complete description of the VAM_{2P} can be found in [10]. In this section we give only a short introduction 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 [3]. 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 VAM_{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 at the same time, more optimizations than in the WAM are possible. The VAM features cheap backtracking, needs less dereferencing and trailing and has smaller stack sizes.

The VAM_{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.

2.2 The VAM_{2P}

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. 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 the called clause, respectively), the `goalframeptr` and the `headframeptr` (frame pointer 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`).

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 and to simplify the test for the trailing of bindings. Structure copying is used for the representation of structures.

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 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. For free variables an additional cell is allocated on the copy 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 initialization of variables we distinguish between their first and further occurrences.

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 1: VAM_{2P} instruction set

The Prolog source code is translated to the VAM_{2P} abstract machine code (see fig. 1). This translation is simple due to the direct mapping between source code and VAM_{2P} code. During runtime a goal and a head instruction 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 encoding the instruction combination is computed by adding the instruction codes and, therefore, the sum of each two instruction codes must be unique. The C statement

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

implements this instruction fetch and decoding.

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

Figure 2: 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 3: choice point

2.3 The VAM_{1P}

The VAM_{1P} has been designed for native code compilation. A complete description can be found in [9]. 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. 2) 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) is also smaller by one element. The pointer to the alternative clauses points directly to the code of the remaining matching clauses.

Due to instruction combination during compile time it is possible to eliminate unnecessary instructions, to eliminate 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 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 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 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 structure. 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 a goal is deterministic at 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 determinism of a clause can be detected during 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. We call this variant of last-call optimization pre-optimization. This optimization can be seen as a generalization of recursion replacement by iteration to every last-call as compared to the optimization of [11].

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.

A problem of the VAM_{1P} can be the size of the generated code. Since for each call of a procedure spezialised code is generated the code size can

become large if there exist many calls of a procedure with many clauses. But since many of this calls have the same calling pattern, these 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.

3 The Incremental Compiler

The compilation of a Prolog program is carried out in five passes (see fig. 4). 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.

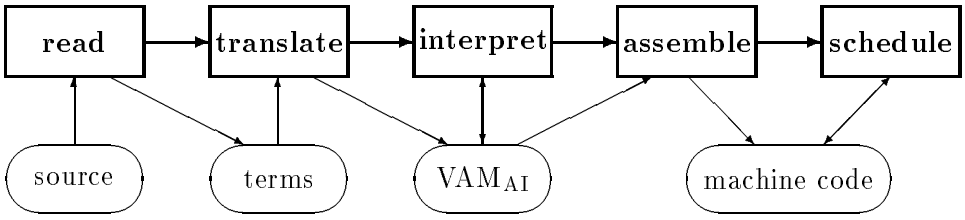

Figure 4: compiler passes

3.1 The VAM_{AI}

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 for global flow analysis of programs. It was introduced by [6] for dataflow analysis of imperative languages. This work was the basis of much of the recent work in the field of declarative and logic programming [1] [4] [7] [12] [14] [16]. 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 and very slow.

A practical implementation of abstract interpretation has been done by Tan and Lin [15]. 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 a few milliseconds which is about 150 times faster than the previous systems.

We followed the way of Tan and Lin and designed an abstract machine for abstract interpretation, the VAM_{AI}. It has been developed on the basis of the VAM_{2P} and benefits from the fast decoding mechanism of this machine. The inferred dataflow information is stored directly in the intermediate code of the VAM_{AI}. We choose the VAM as the basis for 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 a backtrack point slows 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_{2P}-like intermediate code is much simpler and faster than WAM code generation. A VAM_{2P}-like interpreter enabled us to model low level features of the VAM. Furthermore, the VAM_{2P} intermediate code is needed for the generation of the VAM_{1P} instructions. Efficient interpretation is achieved by using fixed-sized variable cells and by representing the domains as bit fields.

Our goal is to gather information about mode, type, reference chain length and aliasing of variables. We distinguish between 0, 1 and greater 1 reference chain lengths. The type of a variable is represented by a set comprised of following simple types:

free	describes 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 we introduced a recursive list type which contains also the information about the termination type.

We use a top-down approach for the analysis of the desired information. Different calls to the same clause are handled separately to get the exact types. The gathered information is a description of the living variables.

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, i.e. the union of the old type and the new type is equal to the old type, a fixpoint has been reached and interpretation of this call to the clause is stopped.

Abstract interpretation with the VAM_{AI} is demonstrated with the following short example. Fig. 5 shows a simple Prolog program part, and a simplified view of its code duplication for the representation in the VAM_{AI} intermediate code.

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 5: Prolog program part and its representation in VAM_{AI}

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. This code duplication leads to more exact types for the variables, because the dataflow information input might be different (more or less exact) for different calls of the same procedure in a program (in the implementation the code duplication is done only for the heads, the bodys of the clauses are shared). 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 inferrable dataflow 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 dataflow 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 B_2^1 there exists a recursive call for B, named B^2 in the example. 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 is reached, computation stops for the recursive call. The dataflow information for the recursion is assigned to the clauses B_1^2 and B_2^2. After all inferrable information is computed for a clause, it is stored directly into the intermediate code. So the same intermediate code is used efficiently in the next pass of the compiler, the code generation.

The representation for the arguments of a Prolog term is the same for VAM_{AI} (see fig. 6) and VAM_{2P} 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
- 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 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, that is the instruction following the call. This eliminates the distinction between the continuation and the restart code pointer (see fig. 3).
- The instruction `const` has been split into `integer` and `atom`.

Another significant difference between the two abstract machines concerns the data areas, i.e. the stacks. While the VAM_{2P} needs three stacks, in VAM_{AI} a modified environment stack and a trail are sufficient. Fig. 7 shows a stack frame for the environment stack from the VAM_{AI}. Note that every stackframe is a choicepoint because all alternatives for a call are considered for 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.

The stack frame contains the actual information for all the local variables of a clause. The `goalptr` points to the intermediate code of a goal (it is used

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 6: VAM_{AI} instruction set

to find the continuation after a goal has been computed), the `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.

Fig. 8 is a detailed description of the stack entry for a local variable. The fields *reference*, *domain*, *ref-len*, *alias-prev* and *alias-next* are used to store the information derived for a variable 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 *alias-prev* and the *alias-next* field are used to connect the variables of this list. The *domain* field contains all actual type information at any 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 union of the informations of all alternatives analysed so far.

The information in the variable fields is also used for fixpoint computation. Abstract interpretation of a clause is stopped if for all variables of a

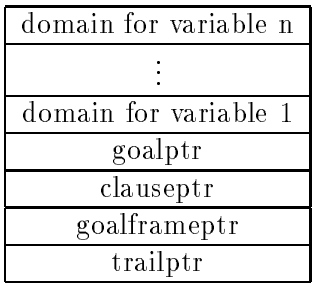

Figure 7: structure of the stack frame

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

Figure 8: a local variable on the stack

goal the information contained in the intermediate code fields is more general or equal to the fields of the variables.

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). If the domain has not changed, new code is only generated for the changed part of the program. If the domain has been changed and the new domain is a subtype of the old domain, the previously generated code fits for the changed program part. The information derived for the program before the change is less exact than the possibly derivable information. Incremental interpretation can stop now. If the new domain is instead a supertype of the old one, the assertion of the new clause made the possible information less precise and the old code part at this program point wrong for the new program. Therefore, incremental abstract interpretation must continue to analyse the callers of the clause.

The retraction of a clause has a similar effect as the assertion of a new one. The only difference is that there might not be an information loss which makes the previously generated code of the callers of the calling clauses wrong. Therefore, it is sufficient to start the incremental analysis with the calling clauses of the changed procedure and interpret top-down until the derived information is equal to the previously inferred one.

3.2 Compilation to Machine Code

The abstract interpretation pass has annotated a part of the VAM_{AI} intermediate representation with type, mode and dereferencing information. This parts which have changed must be translated to machine code. The intermediate representation is traversed and translated to machine code instructions representing VAM_{1P} instructions without generating VAM_{1P} intermediate code. The first occurrences of temporary variables are replaced by their unification partner. If temporary variables occur in two or more structures which have to be unified with local variables, the arguments are reordered so that these temporary variables can be identified by an offset from the top of the copy stack.

The next step is instruction scheduling. Our scheduler is based on list scheduling [19]. This is a heuristic method which yields nearly optimal results. First, the basic block and the dependence graph for each basic block is determined. The alias information is used in recognizing the independence of load and store instructions. Then, the instructions are reordered to fill the latency of load instructions and the delay slot of a branch instruction starting with the instructions on the longest path. After scheduling, the branch offsets and address constants are updated. For this purpose the relocation information stored in the VAM_{AI} intermediate representation is used. So the address changes resulting from scheduling and from incremental compilation can be handled together.

4 Results

To evaluate the performance of our incremental compiler we executed the well known benchmarks described in [2]. We executed these benchmarks on a DECStation 5000/200 (25 MHz R3000) with 40 MB Memory. We compared our compiler with the Aquarius compiler of Peter Van Roy [14] and the Parma system of Andrew Taylor [17]. The Parma system was not available for us, so we used the benchmark data reported in [13]. As you can see in table 1 our compiler produces faster code than the Aquarius system.

We compared the compile time of the VAM_{1P} compiler to that of the VAM_{2P} and SICStus intermediate code translators [5]. It shows that it is about ten times slower than the VAM_{2P} translator but about two times faster than the SICStus compiler (see table 2). The Aquarius compiler is by a factor of 2000 slower than the VAM_{2P} translator. A direct comparison is not possible since it is a three pass compiler which communicates with the assembler and linker via files.

The comparison of the VAM_{2P} interpreter with the VAM_{AI} 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 3). The annotated VAM_{AI}

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

Table 1: execution time, factor of improvement compared to the VAM_{2P}

test	VAM_{2P} ms	VAM_{2P} scaled	VAM_{1P} scaled	SICStus 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 2: compile time, compared to the VAM_{2P}

intermediate code is about three times larger than the simple VAM_{2P} intermediate code.

5 Conclusion

We presented an incremental compiler based on the Vienna Abstract Machine. This compiler uses the VAM_{AI} as an abstract machine for abstract interpretation. This abstract machine has a very compact representation and short analysis times. The fast analysis and the storage of additional information enables the incremental global compilation of Prolog.

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 3: code size of intermediate representations

Acknowledgement

We express our thanks to 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] Joachim Beer. *Concepts, Design, and Performance Analysis of a Parallel Prolog Machine*. Springer, 1989.

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

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

[5] Mats Carlsson and J. Widen. SICStus Prolog user's manual. Research Report R88007C, SICS, 1990.

[6] 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.

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

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

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

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

[11] Micha Meier. Compilation of compound terms in Prolog. In *Eighth International Conference on Logic Programming*, 1991.

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

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

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

[15] 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.

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

[17] Andrew Taylor. LIPS on a MIPS. In *Seventh International Conference on Logic Programming*, Jerusalem, 1990.

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

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