The \textsc{VAM}_{AI} – an Abstract Machine for Incremental Global Dataflow Analysis of Prolog

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

Abstract

A commonly used technique for global flow analysis of Prolog programs is abstract interpretation. Until now nearly all abstract interpretation systems for Prolog are research prototypes and very slow. These systems are not suitable for the integration in Prolog compilers. So we developed the \textsc{VAM}_{AI}, an abstract machine for the abstract interpretation of Prolog. 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 \textsc{VAM}_{AI}, an abstract machine suited for abstract interpretation. The \textsc{VAM}_{AI} does the data flow analysis by a factor of two hundred faster than the previous used meta interpreters written in Prolog. Preliminary results of intermediate code size and analysis time are presented.

1 Introduction

Information about types, modes, trailing, reference chain length and aliasing of variables of a program can be inferred using abstract interpretation. Abstract interpretation was introduced by [6] 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] [3] [5] [8] [7] [11] [12] [14]. 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. Correctness is achieved by iterating the interpretation until the computed information reaches a fixpoint. Termination can be assured by limiting the size of the domain. Most of the previously cited systems are either meta-interpreters written in Prolog or general abstract interpretation systems and are very slow.

A practical implementation of data flow analysis has been done by Tan and Lin [13]. 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.

Section 2 presents the \textsc{VAM}_{AI} in detail with the complete instructions set, the memory model and the execution principles. Section 3 has results which show the efficiency of the \textsc{VAM}_{AI}.

2 The \textsc{VAM}_{AI}

We followed the way of Tan and Lin and designed an abstract machine for abstract interpretation, the \textsc{VAM}_{AI}. The design goal of this abstract machine was to develop a very fast global analysis system which collects the information necessary for optimizing the machine code generated by the \textsc{VAM}_{1P} [9] Prolog compiler. The current implementation of the abstract machine computes information about mode, type,
reference chain length and aliasing of variables, but it can be changed for other domains. The VAMAI is optimized for this particular domain. It is based on top down abstract interpretation with tabulation. An abstract machine breaks up the complex operation like abstract unification or updating the extension table into more atomic instructions. The VAMAI has an instruction for each argument of a goal. The instruction set is based on the VAM2P [10] and benefits from the fast decoding mechanism of this machine. But the run time data structures and the operation of the instructions are completely different. For support of incremental analysis the inferred data flow information is stored directly in the intermediate code of the VAMAI. So the intermediate code is used like the extension table of the abstract interpretation algorithm. We choose the VAM as the basis for an abstract machine for abstract interpretation because it is much better suited for our purpose than the WAM [15]: The parameter passing of the WAM via registers and storing registers in a choice 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 VAM2P-like intermediate code is much simpler and faster than WAM code generation. A VAM2P-like interpreter enabled us to model low level features of the VAM. Furthermore, the VAM2P intermediate code is needed for the generation and optimization of machine code.

2.1 An informal interpretation model

We use a top-down approach 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 head of the clauses are copied. The body of the clauses are shared. This duplication of the code gives a more precise analysis for the use in the VAM2P which generates specialized code for each call, and simplifies many parts of the VAMAI.

Abstract interpretation with the VAMAI is demonstrated with the following short example. Fig. 1 shows a simple Prolog program part, and a simplified view of its code duplication for the representation in the VAMAI intermediate code.

Prolog program:

\[ A_1 := B^1 \]
\[ B_1 := C^1 \]
\[ B_2 := B^2, C^2 \]
\[ C_1 := \text{true} \]

Code representation:

\[ A_1^1 := B^1 \]
\[ B_1^1 := C^1 \]
\[ B_2^1 := B^2, C^2 \]
\[ B_2^2 := C^1 \]
\[ B_2^3 := B^2, C^3 \]
\[ C_1^1 := \text{true} \]
\[ C_1^2 := \text{true} \]

Figure 1: Prolog program part and its representation in VAMAI

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 data flow 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 bodies 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 \( B_1^1 \) 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 $B^2_2$ 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 $B^2_2$ there exists a recursive call for $B$, named $B^2_2$ 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 data flow information for the recursion is assigned to the clauses $B^1_1$ and $B^2_2$. After all inferable information is 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 left to the current subgoal.

2.2 The abstract domain

The goal of the VAMA$1$ 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** 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 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.

2.3 The VAMA$1$ instruction set

Variables are classified into void, temporary and local variables. Void variables occur only once in a clause and no information has to be collected for them. Different to the WAM, temporary variables occur only in the head or in one subgoal, counting a group of built-in predicates as one 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 stored in a fixed sized data structure. All other variables are local and are allocated on the stack.

The representation for the arguments of a Prolog term is the same for VAMA$1$ (see fig. 2) 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. This information
fields replaces the extension table of conventional abstract interpretation algorithms. Local variables of the head have split information fields, because they store both the information at the entry of the clause and the information after a successful computation of this clause. Both 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 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.

- The instruction `const` has been split into `integer` and `atom`.

The translation of Prolog source code to VAMAI instructions is simple due to the direct mapping between source code and VAMAI code. The head arguments of a clause are translated to unification instructions. Each clause 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` instruction. The following example shows the VAMAI code for the `append` procedure.

```prolog
append([], nil, L, L), nogoal
```
The translation of terms to the intermediate code is done in two passes (see fig. 3). 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 VAMAI instructions. Between this two passes the variable classes and offsets are determined.

![Translator Structure Diagram](image)

**Figure 3: translator structure**

### 2.4 The VAMAI memory model

Another significant difference between the VAM2P and the VAMAI concerns the data areas, i.e. the stacks (see fig. 4). While the VAM2P needs three stacks, in VAMAI a modified environment stack and a trail are sufficient. Similar to CLP systems the trail is implemented as a value trail. It contains both the address of the variable and its content.

The VAMAI also needs less machine registers (see fig. 5). Machine registers are the goalptr and headptr (pointer to the code of the calling goal and the head of the called clause, respectively), the goalframeptr and the headframeptr (frame pointer of the clause containing the calling goal and the called clause, respectively) and the trailptr (top of trail). The headframeptr can be used as top of stack pointer. Because every stack frame is a choice point, no choice point register is needed.

Fig. 6 shows a stack frame for the stack of the VAMAI. Note that every stack frame is a choice point because all alternatives for a call are considered for the result of the computation. 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 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 the goalframeptr points to the stack frame of the calling procedure. goalframeptr is not strictly necessary because the stack frame size is contained in the VAMAI code.

Fig. 7 describes the stack entry for a local variable. The fields reference, domain, n-f-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 hold the union of all previously analysed alternatives.
The reference field connects the caller’s variables with the callee’s variables. Possibly 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 least upper bound of the informations of all alternatives analysed so far.

### 2.5 Handling of recursion

The information in the fields of local variables of a clause’s head is used for fixpoint computation. This fields hold both the information that is available at the entry of the clause and the informations, that are available for this local variables after a successful computation of the clause, i.e. the success pattern. When the interpreter reaches the last instruction of a clause (\textit{nogoal}), the success pattern has to be updated. The clause’s head variable success pattern fields are replaced with the least upper bound of its actual entries (the old success pattern) and the new variable domains. The new domains (for the success
Figure 7: a local variable on the stack

During abstract unification of goal and head arguments the entry pattern for head local variables is stored in the intermediate code of the head, if this call is computed the first time. If the intermediate code information contains already entry pattern information, then the old information is replaced by the least upper bound of the new and the old information. If the information in the head’s intermediate code fields does not change, i.e. the new entry pattern contains more special or equal information than formerly applied patterns, 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 calling clauses of this clause. This flag is used to mark these clauses for recomputation. Interpretation is iterated, until the success patterns do not change any more.

2.6 Instruction execution

Like the VAM sp the VAMAI fetches a head instruction, fetches a goal instruction and executes the combined instruction (e.g. the unification of two variables). Only if a structure is combined with a variable, the instructions for the arguments of the structure are executed in single fetch mode. A typical combined unification or structure unification instruction executes the following subtasks:

- If a goal variable is involved in the instruction, store the current value of this variable in the intermediate code.
- Execute abstract unification, eventually fail.
- If a head variable is involved in the instruction and the new value of this variable is more general than the value stored in the intermediate code, then store this value in the intermediate code and set the changed entry pattern flag.

If the combined call-goal instruction is executed with an unchanged entry pattern, the success pattern is merged into the union field and the other alternatives for this call are tried (artificial fail). Otherwise a new stack frame is allocated and this subgoal is called. If a call is combined with a nogoal instruction, the union is constructed and the alternatives are tried. If the last alternative has been executed, a single goal instruction is fetched. If this instruction is a goal instruction, the current value of the union field is copied into the variable, the union fields are initialized and the goal is called. If this instruction is a nogoal instruction, the success pattern is stored, the stack frame is popped and the next calling single goal instruction is fetched and executed.

2.7 An example

Consider the following program part:
top :-
append( [1,2], [3,4], L),
    write( L).

append( [], L, L).
append( [X|L1], L2, [X|L3]) :-
append( L1, L2, L3).

This program is compiled to the following \texttt{VAM\textsubscript{A1}} instructions:

append1:
    head: nil, fsttmp(0), tmp(0), nogoal
    head: list, fsttmp(0), fstvar(2), fstvar(1), list, tmp(0), fstvar(0),
        goal(append2), var(2), var(1), var(0), call, nogoal.

append2:
    head: nil, fsttmp(0), tmp(0), nogoal
    head: list, fsttmp(0), fstvar(2), fstvar(1), list, tmp(0), fstvar(0),
        goal(append2), var(2), var(1), var(0), call, nogoal.

top:
    head:
        goal(append1), list, int(1), list, int(2), nil,
        list, int(3), list, int(4), nil, fstvar(0), call,
        builtin(write), var(0), nogoal.

Below follows a simplified trace of the analyser. Usually the interpreter executes in the combined mode (e.g. \texttt{call-goal}, or unification of two variables). The first clause of \texttt{append1} is tried, it fails and the second clause is tried. Some instructions are executed in combined mode and the elements of the list are executed in list unification mode. Then the first clause of \texttt{append2} is tried, it fails and the second clause of \texttt{append2} is tried. It calls recursively the first clause of \texttt{append2}, succeeds and executes the second clause which calls recursively \texttt{append2}. But this time the entry pattern is equal to the last one and the success pattern is used. The end of the second clause of \texttt{append2} is reached and the success pattern is stored. Now the end of the second clause of \texttt{append1} is reached and the success pattern is stored. The success patterns have changed and a second iteration is started. In this small example the second iteration is identical to the first one since both clauses have a changed success pattern. But this time no pattern changes and the analysis stops.

*************** first iteration ***************
call  goal(append1)
fail  goal(append1) true fail (list/nil)
list  nil
    list unify
        int(1) fsttmp(0) unify
        list h_fstvar(2) unify
            int(2) unify list
            nil unify list
        list h_fstvar(1) unify
            int(3) unify list
            list unify list
            int(4) unify list
            nil unify list
g_fstvar(0) list unify
nxttmp(0) unify list
h_fstvar(0) unify list
call  goal/(append2/) \\
g_nxtvar(2) nil  \hspace{1em} unify \\
fail  goal/(append2/) \hspace{1em} true fail (list/nil) \\
g_nxtvar(2) list  \hspace{1em} unify \\
fsttmp(0) \hspace{1em} unify list \\
h_fstvar(2) \hspace{1em} unify list \\
g_nxtvar(1) h_fstvar(1) \hspace{1em} unify \\
g_nxtvar(0) list \hspace{1em} unify \\
nxttmp(0) \hspace{1em} unify list \\
h_fstvar(0) \hspace{1em} unify list \\
call  goal/(append2/) \\
g_nxtvar(2) nil \hspace{1em} unify \\
g_nxtvar(1) h_fstvar(1) \hspace{1em} unify \\
g_nxtvar(0) list \hspace{1em} unify \\
nxttmp(0) \hspace{1em} unify list \\
h_fstvar(0) \hspace{1em} unify list \\
call  nogoal \\
fail  goal/(append2/) \hspace{1em} try all alternatives \\
g_nxtvar(2) list \hspace{1em} unify \\
fsttmp(0) \hspace{1em} unify list \\
h_fstvar(2) \hspace{1em} unify list \\
g_nxtvar(1) h_fstvar(1) \hspace{1em} unify \\
g_nxtvar(0) list \hspace{1em} unify \\
nxttmp(0) \hspace{1em} unify list \\
h_fstvar(0) \hspace{1em} unify list \\
call  goal/(append2/) \hspace{1em} use success pattern \\
next  nogoal \\
fail  goal/(append2/) \hspace{1em} last alternative tried \\
next  nogoal \\
fail  goal/(append1/) \hspace{1em} last alternative tried \\
next  builtin/(write/) \\
next  nogoal \\

*************** second iteration ***************

equivalent to the first iteration

2.8  Incremental abstract interpretation

The VAM$_{AI}$ is also well suited for incremental abstract interpretation. Incremental abstract interpretation is similar to the 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, for each procedure a pointer to the caller of this procedure is stored in the VAM$_{AI}$ code. This pointer is used to find the top goal of the whole program. This pointer chain can be used to reconstruct 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 can lead to the interpretation of the whole program.

2.9  Design alternatives

The current implementation uses one value trail. For each trailed variable the complete value of the variable is stored together with the address of the variable. Since in most cases only a part of the information has changed a tagged trail can be used. The information in the variables is divided into groups which usually change at the same time. Each group get a special tag and only the changed group
together with the tag is stored on the trail. If the number of groups is very small, also more than one
trail can be used to avoid the tagging of values.

We investigated also an implementation without trailing. The variable gets an additional field (called
in-field) which holds the value of a variable prior to the call. At a call referenced variables have to copy
their contents to the callee's variables. At call completion the contents have to copied back.

It has to be evaluated if the duplication of clause bodies increases the accuracy of the analysis. The
sharing of the bodies eliminates the construction of the least upper bound of the bodies before code
generation. An interesting alternative would also be the elimination of the stack and storing the stack
frames directly in the intermediate code. Different calls would be handled by the same stack frames in
the code.

3 Results

Before developing the VAMAI we developed a prototype analyser in Prolog. Due to the single assignmen-
ture of Prolog the information about the program has to be stored in the database using assert. The
prototype analyser is implemented as a deterministic recursive procedure so that it was possible to store
intermediate representations in Prolog data structures. The problem is that this structures are stored
on the copy stack, destructive assignment must be replaced by copying part of the structures and that
our Prolog interpreter does not support garbage collection. So this interpreter was very slow and needed
a stack size greater than 32 MB when analysing a program which was bigger than 50 clauses. So it is
evident that an analyser based on the VAMAI is on average more than a factor of two hundred faster than
the prototype analyser (see table 1). For evaluation of the VAMAI we used the well known benchmarks
described in [2]. These benchmarks were executed on a DECStation 5000/200 (25 MHz R3000) with
40 MB Memory. A direct comparison with the GAIA system [5] is not possible, because only data for
one benchmark we use is available and the domains are not comparable. Compensating the speed of the
different benchmark machines, the VAMAI is about a factor of 20 faster.

<table>
<thead>
<tr>
<th>test</th>
<th>Prolog ms</th>
<th>Prolog scaled</th>
<th>VAMAI scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>det. append</td>
<td>648</td>
<td>1</td>
<td>1115</td>
</tr>
<tr>
<td>naive reverse</td>
<td>789</td>
<td>1</td>
<td>639</td>
</tr>
<tr>
<td>quicksort</td>
<td>977</td>
<td>1</td>
<td>264</td>
</tr>
<tr>
<td>8-queens</td>
<td>815</td>
<td>1</td>
<td>241</td>
</tr>
<tr>
<td>serialize</td>
<td>1630</td>
<td>1</td>
<td>206</td>
</tr>
<tr>
<td>differentiate</td>
<td>2122</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>query</td>
<td>781</td>
<td>1</td>
<td>194</td>
</tr>
<tr>
<td>bucket</td>
<td>923</td>
<td>1</td>
<td>230</td>
</tr>
<tr>
<td>permutation</td>
<td>732</td>
<td>1</td>
<td>503</td>
</tr>
</tbody>
</table>

Table 1: global analysis time, factor of improvement

The VAMAI was integrated in the VAMIP compiler. Table 2 shows the improvement of the generated
code due to global analysis. The high improvement for quicksort is only partly due to global analysis,
the main improvement comes from a better clause indexing (built-in predicate indexing). To show the
efficiency of the compiler the compile time were compared to that of the VAMIP and SICStus intermediate
code translators [4]. The VAMIP compiler is about ten times slower than the VAMIP translator but about
two times faster than the SICStus compiler (see table 3). The Aquarius compiler [12] is by a factor of
2000 slower than the VAMIP 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 VAMIP interpreter with the VAMAI shows that the size of the annotated
VAMAI intermediate code is about three times larger than the simple VAMIP intermediate code (see
table 4).
<table>
<thead>
<tr>
<th>test</th>
<th>VAM2P</th>
<th>VAM2P</th>
<th>VAM1P</th>
<th>VAM1popt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ms</td>
<td>scaled</td>
<td>scaled</td>
<td>scaled</td>
</tr>
<tr>
<td>det. append</td>
<td>0.25</td>
<td>1</td>
<td>26.1</td>
<td>26.1</td>
</tr>
<tr>
<td>naive reverse</td>
<td>4.17</td>
<td>1</td>
<td>19.3</td>
<td>20.0</td>
</tr>
<tr>
<td>quicksort</td>
<td>6.00</td>
<td>1</td>
<td>7.23</td>
<td>18.1</td>
</tr>
<tr>
<td>8-queens</td>
<td>65.4</td>
<td>1</td>
<td>12.4</td>
<td>13.5</td>
</tr>
<tr>
<td>serialize</td>
<td>3.90</td>
<td>1</td>
<td>5.76</td>
<td>6.84</td>
</tr>
<tr>
<td>differentiate</td>
<td>1.14</td>
<td>1</td>
<td>6.32</td>
<td>8.14</td>
</tr>
<tr>
<td>query</td>
<td>41.7</td>
<td>1</td>
<td>7.58</td>
<td>9.70</td>
</tr>
<tr>
<td>bucket</td>
<td>247</td>
<td>1</td>
<td>5.02</td>
<td>5.24</td>
</tr>
<tr>
<td>permutation</td>
<td>2660</td>
<td>1</td>
<td>5.08</td>
<td>6.48</td>
</tr>
</tbody>
</table>

Table 2: execution time, factor of improvement compared to the VAM2P

<table>
<thead>
<tr>
<th>test</th>
<th>VAM2P</th>
<th>VAM2P</th>
<th>VAM1P</th>
<th>SICStus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ms</td>
<td>scaled</td>
<td>scaled</td>
<td>scaled</td>
</tr>
<tr>
<td>det. append</td>
<td>5.78</td>
<td>1</td>
<td>11.43</td>
<td>21.5</td>
</tr>
<tr>
<td>naive reverse</td>
<td>7.31</td>
<td>1</td>
<td>10.5</td>
<td>19.3</td>
</tr>
<tr>
<td>quicksort</td>
<td>9.30</td>
<td>1</td>
<td>9.9</td>
<td>23.1</td>
</tr>
<tr>
<td>8-queens</td>
<td>9.18</td>
<td>1</td>
<td>11.6</td>
<td>19.7</td>
</tr>
<tr>
<td>serialize</td>
<td>11.36</td>
<td>1</td>
<td>11.22</td>
<td>19.2</td>
</tr>
<tr>
<td>differentiate</td>
<td>13.71</td>
<td>1</td>
<td>11.41</td>
<td>30.3</td>
</tr>
<tr>
<td>query</td>
<td>21.05</td>
<td>1</td>
<td>7.5</td>
<td>13.4</td>
</tr>
<tr>
<td>bucket</td>
<td>15.59</td>
<td>1</td>
<td>7.25</td>
<td>12.7</td>
</tr>
<tr>
<td>permutation</td>
<td>4.88</td>
<td>1</td>
<td>8.88</td>
<td>18.1</td>
</tr>
</tbody>
</table>

Table 3: compile time, compared to the VAM2P

4 Conclusion

We presented the VAMAI, an abstract machine for abstract interpretation of Prolog. 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 analysis of Prolog.

Acknowledgement

We express our thanks to Thomas Berger, Anton Ertl and Franz Puntigam for their comments on earlier drafts of this paper.

References

<table>
<thead>
<tr>
<th>Test</th>
<th>VAM$_{2P}$</th>
<th>VAM$_{3P}$</th>
<th>VAM$_{AM}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>det. append</td>
<td>288</td>
<td>380</td>
<td>1044</td>
</tr>
<tr>
<td>naive reverse</td>
<td>764</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>quicksort</td>
<td>536</td>
<td>1064</td>
<td>2084</td>
</tr>
<tr>
<td>serialize</td>
<td>1044</td>
<td>1</td>
<td>8.37</td>
</tr>
<tr>
<td>differentiate</td>
<td>1064</td>
<td>1</td>
<td>8.37</td>
</tr>
<tr>
<td>query</td>
<td>2084</td>
<td>1</td>
<td>0.89</td>
</tr>
<tr>
<td>bucket</td>
<td>996</td>
<td>1</td>
<td>1.96</td>
</tr>
<tr>
<td>permutation</td>
<td>296</td>
<td>1</td>
<td>2.77</td>
</tr>
</tbody>
</table>

Table 4: code size of intermediate representations


