An extension of the WAM for uninitialized variables

Dante Baldan* Nicola Civran† Gilberto Filé‡

Abstract

An extension of the WAM for handling uninitialized variables is presented. Our modification of the WAM is based on a static analysis designed within the *classical* Cousot's abstract interpretation framework. The information inferred by our analysis supports the optimization of the WAM code produced for the analyzed programs. The optimization consists in modifying some instructions taking into account the possibility that their inputs can be uninitialized variables.

For illustrating the usefulness of our method, we show that it allows to improve the WAM code of the *append* program. In general, analyses that infer very concrete properties of programs are important because such properties can support significant and easily implementable optimizations of the code generated by compilers. The main contributions of this work are the simplicity of the domain of the analysis, the WAM-like extension supporting the optimization and the fact that the analysis is provably correct according to the well-known proof-scheme of Cousot's framework.

Keywords: Static Analysis, Abstract Interpretation, Logic Programming, WAM, BAM, GAIA.

1. Introduction

We present a static analysis of logic programs that supports the production of optimized WAM code [12, 1] for these programs. This analysis is designed within the *classical* Cousot's Abstract Interpretation framework [6, 7]. The information that is inferred by this static analysis is that certain program variables are not initialized as self-referencing pointers.

The goal of the optimization consists in postponing the initialization of the variables as much as possible. The phenomenon that the analysis aims to avoid is the initialization of a variable as self-referencing memory cell followed by a binding of that variable with another value. This situation is very common as stated in [10]. We follow [10] considering the memory cell of an uninitialized variable as containing garbage. Therefore specialized WAM instructions are needed to cope with the fact that the *uninitialized* variables cannot

*E-mail: dante@math.unipd.it Phone:+ 39 49 827 5899

†E-mail: cvrncl26@leonardo.math.unipd.it

‡E-mail: gilberto@math.unipd.it Phone:+ 39 49 827 5989 Fax: + 39 49 827 8596

be dereferenced. Thus, the specified instructions are more efficient then the normal ones because dereferencing is expensive [10, 11, 3].

Our analysis allows to discover variables that can be considered as uninitialized. This information supports the optimization consisting in specialized/simplified versions of existing WAM instructions manipulating terms.

Van Roy [10] proposed an uninitialized variable analysis for the BAM implemented in the Aquarius Compiler. The analysis needs two abstract domains: one made of tuples of modes associated with the argument variables of the predicates and one made of 4-tuples of sets of program variables. The elements of the domain of modes are functions that assign a mode to each argument of each predicate. The elements in the domain of the sets of variables are 4-tuple (G, N, U, R) carrying information about the variables that are (bound to) ground terms, structured terms, uninitialized variables and recursively dereferenced terms. Those two domains cooperate in the analysis. The domain of modes is used to perform the least upper bound operation and the clause call-exit operations while the one made of 4-tuples of sets is used to compute the values inside the clause. A first contribution of our work is that we use only one domain to perform all the computation. Our domain is made of 3-tuples of sets of variables (G, N, U) that represent the variables that are (bound to) ground terms, structured terms and that are uninitialized, respectively. A second contribution of our work is that our analysis supports a modification of the WAM without requiring the design of an abstract machine (and then a new compiler). The third contribution is that our analysis is designed in Cousot's abstract interpretation framework that allows to prove the correctness of the analysis according to a well-known proof-scheme.

Our analysis is described as a non-standard semantics (of logic programs) that is defined by an abstract domain and by abstract operations that correspond to the concrete domain and operations of the concrete semantics.

A formal correctness proof of uninitialized variable analysis and of other similar low-level analyses is the subject of future work. Thus, the present work is a first step towards an important objective: the design of analyses that are both, formally correct and *practical* in the sense that they produce significant optimizations that are also easy to introduce in compilers.

In the present work we outline the analysis by means of the *append* program. The analysis for the introduction of uninitialized variables is performed using GAIA [5]. The system where we implemented the extension is the Emulator of SICStus 2.1 extended with an instruction set that handles the uninitialized variables. As a first step of the integration of the analysis within the Emulator we did not take care of the garbage collecting phase. This means that the modified version of the Emulator does not collect garbage as much as possible. We ran the system on a SparcStation Classic 40Mb RAM powered by a MicroSparc Processor equipped with SunOS 4.1.3.

The paper is organized as follows. Section 2 defines some basic notations. In Section 3 the well-known *append* program is used for presenting both the concrete and the abstract semantics for our analysis. In Section 4 the result of the fixpoint computation is presented and compared with the information computed by the Aquarius dataflow analyzer. The optimized version of *append* is outlined together with performance results.

A short conclusion closes the paper.

2. Preliminaries

In this section we introduce some terminology used in the paper. A complete introduction to lattice theory is in [4] and the theory of Abstract Interpretation can be found in [7].

2.1. Logic programming

We introduce some notation about *logic programming* [9].

It is well-known that compilers for logic programs are, in most cases, based on an abstract machine that has been introduced by D.Warren [12] and that is called the Warren Abstract Machine (WAM). For reading this paper a good knowledge of the WAM is required. A crystal-clear introduction to the WAM is presented by Hassan [1].

We will consider logic programs in normalized form as illustrated in the following example.

Example

Let P be the following logic program.

cl : `p(f(f(X)),Y,Y) :- q(X,g(g(g(Y))),Z), q(Z,Z,Z).`

The normalized form of P is as follows:

cl : `p/3 :-` A_0^0`=f(f(`X_0`))`, A_1^0`=`X_1`,` A_2^0`=`X_1`,`
A_0^1`=`X_0`,` A_1^1`=g(g(g(`X_1`)))`, A_2^1`=`Y_0`,` `q(`A_0^1`,`A_1^1`,`A_2^1`),`
A_0^2`=`Y_0`,` A_1^2`=`Y_0`,` A_2^2`=`Y_0`,` `q(`A_0^2`,`A_1^2`,`A_2^2`).`

This normalized form is convenient for the static analysis because in this way unifications are expressed by the equality built-ins inside the clauses that can be handled one at a time. Few remarks about such programs are useful.

- The body of a normalized clause is logically divided into blocks. In the first clause of the previous example there are two blocks. The first is:

  A_0^0`=f(f(`X_0`))`, A_1^0`=`X_1`,` A_2^0`=`X_1`,` A_0^1`=`X_0`,` A_1^1`=g(g(g(`X_1`)))`, A_2^1`=`Y_0`,` `q(`A_0^1`,`A_1^1`,`A_2^1`)`

  and the second is:

  A_0^2`=`Y_0`,` A_1^2`=`Y_0`,` A_2^2`=`Y_0`,` `q(`A_0^2`,`A_1^2`,`A_2^2`)`

  Each block is called *body goal*. Observe that the built-ins corresponding to the head unifications belong to the first body goal.

- The variables A_j^i are called *argument variables* because they correspond to the argument registers used in the WAM. Clearly, there is an important difference between these variables and the argument registers of the WAM: in the WAM the same registers are used for passing the arguments of all calls, whereas in a normalized clause

each atom has argument variables with distinct superscripts. Obviously, this difference is unavoidable: normalized programs are still programs and thus they do not contain registers. However, when later we will describe the concrete semantics of the normalized programs (that is very close to the WAM), it will be very simple to treat the argument variables as registers simply ignoring the superscripts.

- It is easy to see that a clause in normalized form contains one (non argument) variable for each variable of the original (non normalized) clause. As in the WAM we distinguish two types of these non-argument variables: the *temporary* and the *permanent* variables. The temporary variables are those that occur in at most one of the body goals of the clause, whereas the permanent ones are those that occur in at least two distinct body goals of the clause.

  Following the notation used in [1], the argument variables are always indicated with the letter `A`, the temporary ones with `X` and the permanent ones with `Y`, eventually with appropriate superscripts and subscripts.

- Given a normalized clause $cl : p/n : -\zeta_1, \ldots, \zeta_n$ where ζ_i can be a built-in or an atom, its *program points* are those indicated below:

$$cl : p/n : -(0)\zeta_1(1)\ldots(n-1)\zeta_n(n).$$

  To indicate program point i in clause cl we will write (cl, i).

- Let t be a term in the first order logic.

  - $vars(t)$ is the set of variables of the term t;
  - $var(t)$ is true if and only if t is a variable in the first order logic;
  - $dups(t)$ is the set of the variables occurring at least twice in t.

3. Uninitialized Variable Analysis

This section introduces the concrete and the abstract semantics for the uninitialized variable analysis. Both semantics are characterized by a domain and operations (respectively, concrete and abstract). The concrete semantics is an abstraction of the operational semantics of the WAM so as it is presented in [1], thus familiarity with this work is required for following our description. The concrete semantics is illustrated simply sketching the computation of the meaning of the *append* program according to this semantics.

In the WAM setting, a variable is *uninitialized* if either its representation is a self-referencing cell possibly pointed to by argument or temporary variables or its representation is a cell pointing to a self-referencing location possibly pointed to by registers or argument variables. In other words, we define *uninitialized* those variables that we can directly (i.e. within one step) access through an argument or a temporary variable.

cl_0 : `main:-append(X,Y,[a,b,c]).`
cl_1 : `append([],L,L).`
cl_2 : `append([X|L],L1,[X|L2]):- append(L,L1,L2).`

Figure 1: Non-normalized form of `append`

cl_0 : `main/0:-` (0) A_0^1=X_1 (1) A_1^1=X_2 (2) A_2^1=`[a,b,c]` (3) `append(`A_0^1`,`A_1^1`,`A_2^1`)` (4).
cl_1 : `append/3:-` (0) A_0^0=`[]` (1) A_1^0=A_2^0 (2).
cl_2 : `append/3:-` (0) A_0^0=`[`X_3`|`X_0`]` (1) A_1^0=X_1 (2) A_2^0=`[`X_3`|`X_2`]`
(3) A_0^1=X_0 (4) A_1^1=X_1 (5) A_2^1=X_2 (6) `append(`A_0^1`,`A_1^1`,`A_2^1`)` (7)

Figure 2: Normalized form of `append`

3.1. The Concrete Semantics

As uninitialized variable analysis is concerned only with the variables of the clauses of a program, the concrete semantics is obtained from the operational semantics of the WAM [1] focusing only on the heap, on the variables of the local stack, and on the argument and temporary variables and ignoring all the other data structures used in the WAM such as the trail, the choice points etc.. In the following we will follow [1] calling *stack* the memory part of the WAM made of the heap, the local stack and the argument and temporary variables.

The concrete semantics of a given logic program P, computes for each program point (cl, i) of P the set of all stacks that are computed in the executions that reach (cl, i) (starting from some given initial goal). In this semantics the operations are as follows:

- equality built-ins are interpreted as operations that transform these sets of stacks just as the corresponding WAM operations modify them.
- the calls appearing in clause bodies are interpreted as operations that push on the current stack a new environment for the called clause (if necessary).

3.1.1. Append

For illustrating the concrete semantics we will apply it to the *append* program that is shown in Fig. 1. Fig. 2 shows the normalized version of the program and Fig. 3 illustrates the corresponding WAM code produced by the SICStus [2, 8] Compiler 2.1.

3.1.2. The concrete semantics of Append

In what follows we will briefly describe the computation of the concrete values at some of the program points of *append*.

The computation starts from the goal `?-main`, following the style of the benchmarks in [10]. The changes from one point to another will be indicated with $\leftarrow$. Clause cl_0 is called and the situation at program point $(cl_0, 3)$ is depicted in Fig. 4(a).

```
A2versionB121C31902I-1J1J3L-1
clause(main/0/1, [ifshallow, neck(0), else, endif, put_list(2), unify_constant(a),
unify_list, unify_constant(b), unify_list, unify_constant(c), unify_nil,
put_x_void(0), put_x_void(1), execute(append/3)]).
clause(append/3/1, [get_nil_x0, get_x_value(1,2), neck(3), proceed]).
clause(append/3/2, [ifshallow, get_list_x0, unify_x_variable(4),
unify_x_variable(3), get_list(2), unify_x_value(4), neck(3), unify_x_variable(2),
put_x_value(3,0), execute(append/3), else, get_list_x0, unify_x_variable(3),
unify_x_variable(0), get_list(2), unify_x_value(3), unify_x_variable(2),
execute(append/3), endif]).
predicate(append/3, [switch_on_term([append/3/1,append/3/2],
[append/3/2],[[]-append/3/1],[])]).
predicate(main/0, [switch_on_term([main/0/1],[main/0/1],[],[main/0/1])]).
```

Figure 3: WAM code for `append`

The call to procedure `append` is executed. Let us first consider the execution of cl_1. The two built-ins of cl_1 produce the following effect:

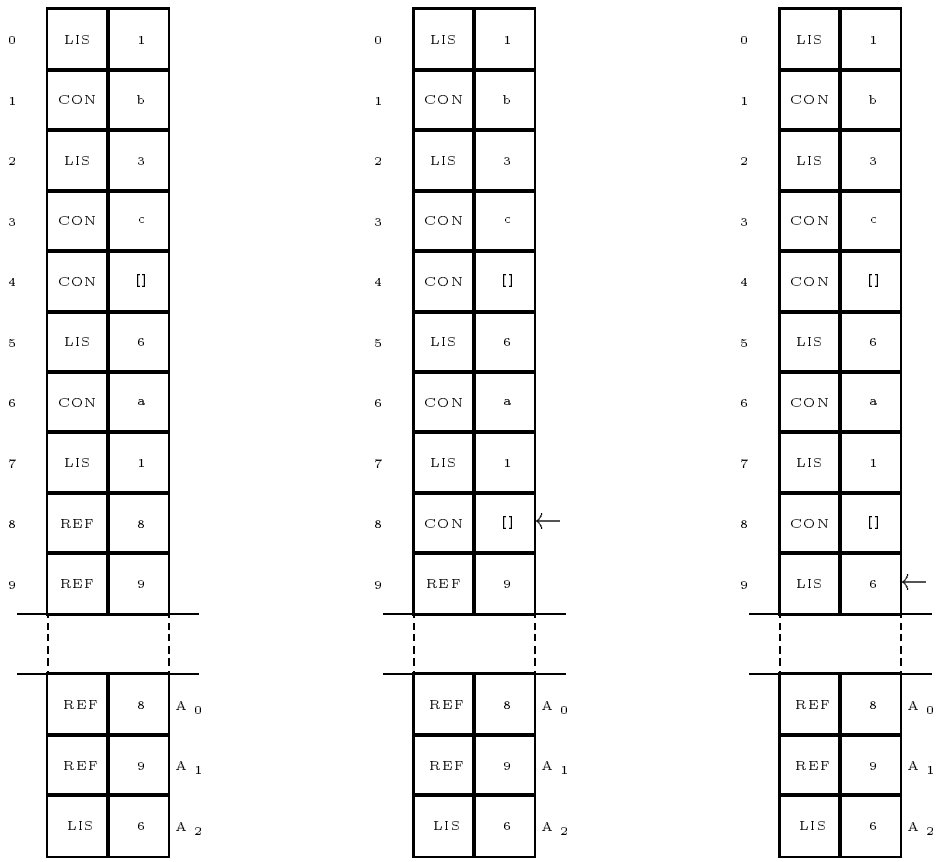
- A_0^0=[]: it corresponds to executing `get_nil_x0` of the WAM code in Fig. 3 which assigns the empty list to (the location pointed by) the first argument register; the stack obtained after this operation is shown in Fig. 4(b).
- A_1^0=A_2^0: it corresponds to executing `get_x_value (1,2)`, (see Fig. 3); Fig. 4(c) shows the new stack.

We can see that the location pointed to by A_0^1 is assigned a CON value and that the location pointed to by A_1^1 is assigned a LIS structure. The optimization will show how to exploit this observation.

Let us consider now the call of cl_2 (from $(cl_0, 3)$). The built-ins until program point $(cl_2, 7)$, are handled as follows:

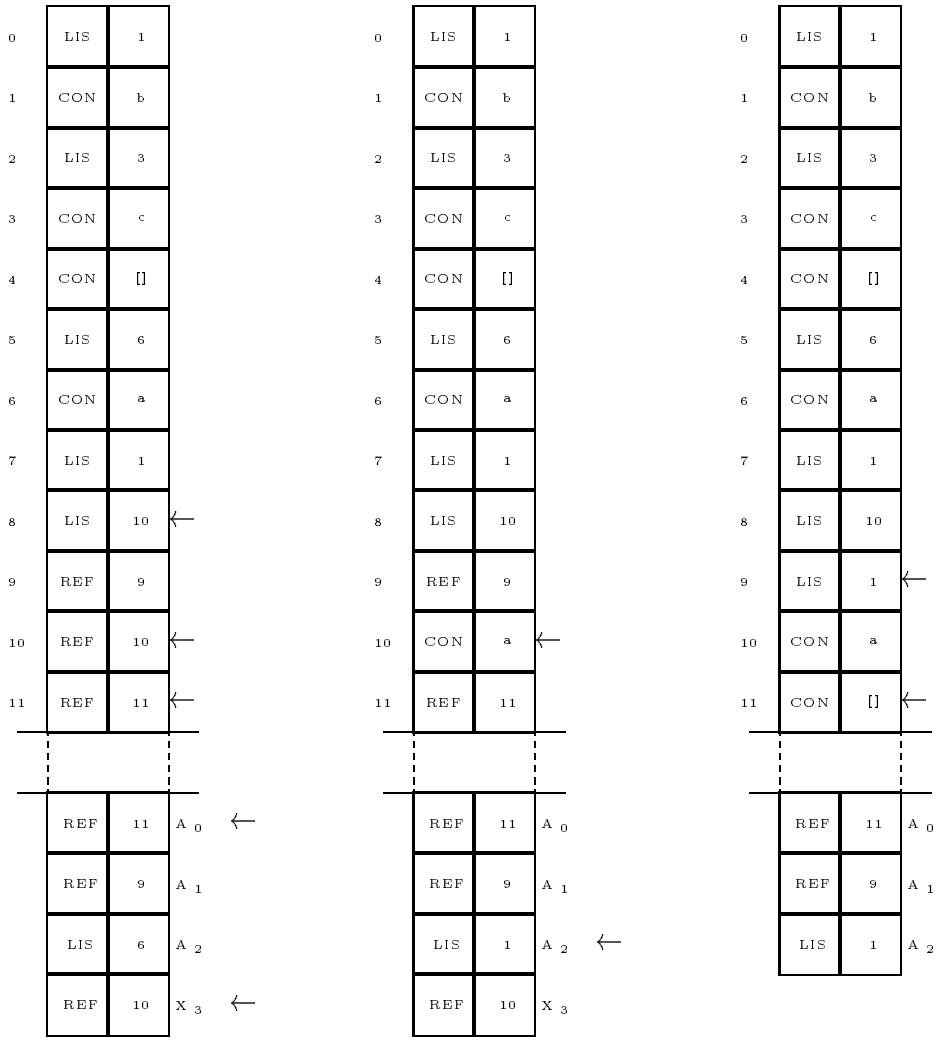
- A_0^0=[X_3|X_0]: this corresponds to executing the sequence constituted by the instructions `get_list_x0`, `unify_x_variable(3)` and `unify_x_variable(0)` (see Fig. 3). The stack it produces is shown in Fig. 5(a).
- the built-in A_1^0=X_1 has no correspondent WAM instruction since the second argument is simply read and then reloaded in A_1^1=X_1 with the same content on the same WAM register;
- the built-in A_2^0=[X_3|X_2] corresponds to executing `get_list(2)`, `unify_x_value(3)` and `unify_x_variable(2)` (see Fig. 3).

We describe the stack at $(cl_2, 7)$ in Fig. 5(b). The WAM semantics applies and clause cl_1 is called. The output is depicted in Fig. 5(c). In a similar way we can continue the computation according to the WAM semantics. We will not show the details of the computation since they are not relevant for the rest of the paper.



$(a) - (cl_0, 3)$ $(b) - (cl_1, 1)$ $(c) - (cl_1, 2)$

Figure 4: Stacks at program point $(cl_0, 3)$, $(cl_1, 1)$, $(cl_1, 2)$ for `append`



$(a) - (cl_2, 1)$ $(b) - (cl_2, 7)$ $(c) - (cl_1, 2)$

Figure 5: Stacks at program point $(cl_2, 1)$, $(cl_2, 7)$, $(cl_1, 2)$ for `append`

As you can see, there are many cases where a self-referencing cell is allocated and then assigned a value. What [10] proposes is the uninitialization of such memory cells since they will be eventually assigned a 'real' value.

3.2. The Abstract Domain for Uninitialized Variable Analysis

The analysis in [10] requires two abstract domains: one made of 4-tuples of sets of variables and one made of tuples of modes associated with the argument variables of the predicates. We adopted an abstract domain made of 3-tuples of sets of variables according to the Abstract Interpretation framework. Therefore our domain is simpler than the one in [10] and closer to the Cousot's framework.

3.3. The definition of the abstract domain

Let, in what follows, cl be any clause in normalized form. Recall that $vars(cl)$ is the set of the variables of cl. We will describe the values of the abstract domain for cl.

A (non bottom) value of the domain is a 3-tuple $\Gamma = (G, N, U)$ with the following meaning:

- G and N are subsets of $Vars(cl)$. G and N are sets of variables bound to *surely* ground and *surely* non variable terms, respectively.
- U is a subset of $Vars(cl)$. U is a set of *surely* uninitialized variables; recalling the definition of uninitialized variables, we can say that U is the set of variables over which the optimization applies.

The abstract domain $\mathcal{D}$ for cl consists in a bottom element $d_\perp$ and in all the 3-tuples $\Gamma = (G, N, U)$ satisfying the following (soundness) condition:

$$G \subseteq N \wedge U \cap N = \emptyset$$

$d_\perp$ is assumed to be the minimum element and the ordering between two 3-tuples is given by the componentwise superset (partial) relation. The top element is $(\emptyset, \emptyset, \emptyset)$.

The least upper bound (*lub*) operation is the componentwise intersection, while the greatest lower bound can be defined as the dual of the *lub* since $\mathcal{D}$ has the minimum element. It is easy to see that $\mathcal{D}$ forms a complete lattice.

Now we present briefly the definition of the abstract operations. To do that we need an auxiliary partition on the set $Vars(cl)$ expressing which variables are alias by means of the equations of the form $\mathtt{A}_j^i\mathtt{=V}_k$ of cl. Given a variable A and a partition C, $alias_C(A)$ is the element of C that contains A. Moreover, for each program point (cl_i, j) we will need the set of the variables appearing in the part of the body of cl coming before (cl_i, j). Let (G, N, U) and (G', N', U') [C and C'] be the abstract values [partitions] associated with the program points preceding and following $A_j^i = t$, respectively, where t is a first order logic term. The following rules are to be interpreted as follows: next line condition has to be evaluated if the current line condition does not hold. The new abstract value is the following:

- Groundness component: $G' \equiv \begin{cases} G \cup \bigcup_{Z \in vars(t)} alias_{C'}(Z) & \text{if } A_j^i \in G \\ G \cup alias_{C'}(A) & \text{if } vars(t) \subseteq G \\ G & \text{otherwise} \end{cases}$

- Nonvar component: $N' \equiv \begin{cases} N \cup G' \cup \{A\} & \text{if } \neg var(t) \\ N \cup G' \cup alias_{C'}(A) & \text{if } var(t) \wedge t \in N \\ N \cup G' \cup alias_{C'}(t) & \text{if } var(t) \wedge A_j^i \in N \\ N \cup G' & \text{otherwise} \end{cases}$

- Uninitialized variable component. Let t be a variable V. The equation is $A_j^i = V$ and let S be the set of variables appearing in the body part coming before $A_j^i = V$ in cl. The following rules apply:

  - $i > 0$ and V is temporary: $U' \equiv \begin{cases} U \setminus alias_{C'}(V) & \text{if } V \in S \text{ and } V \notin U \\ U & \text{otherwise} \end{cases}$

  - $i > 0$ and V is permanent: $U' \equiv \begin{cases} U \setminus alias_{C'}(V) & \text{if either the WAM executes a } \texttt{put_unsafe_value} \text{ or (the WAM executes a } \texttt{put_value} \text{ and } Y \notin U) \\ U & \text{otherwise} \end{cases}$

  - $i = 0$ and V is temporary: $U' \equiv \begin{cases} U \setminus alias_{C'}(V) & \text{if } V \in S \text{ or if } A_j^i \notin U \\ U & \text{otherwise} \end{cases}$

  - $i = 0$ and V is permanent: $U' \equiv U \setminus alias_{C'}(V)$

- Uninitialized variable component. Let $A = t \equiv A_j^i = f(t_1, \ldots, t_m)$. Let $\mathcal{Y}(cl)$ be the set of permanent variables in cl. The following rules applies:

  - $i > 0$: $U' \equiv U \setminus (\{A_j^i\} \cup \bigcup_{Z \in vars(t)} alias_{C'}(Z))$

  - $i = 0$: $U' \equiv \begin{cases} U \setminus (\{A_j^i\} \cup \bigcup_{Z \in dups(t) \cup (vars(t) \cap (S \cap \mathcal{Y}(cl)))} alias_{C'}(Z)) & \text{if } A_j^i \in U \\ U \setminus \bigcup_{Z \in vars(t)} alias_{C'}(Z)) & \text{otherwise} \end{cases}$

These are the abstract operations for the built-ins execution. Now we present the abstract operations for the clause entry and exit phases. Let cl_1 and cl_2 be the calling and the called clauses, respectively. Let (G, N, U) be the abstract value holding at the calling point in cl_1 and let $p(A_0^i, \ldots, A_m^i)$ be the calling atom. Then the abstract value associated with the program point 0 in cl_2 is:

- $G' \equiv \{A_j^0 \in vars(cl_2) : A_j^i \in vars(cl_1) \cap G\}$

- $N' \equiv \{A_j^0 \in vars(cl_2) : A_j^i \in vars(cl_1) \cap N\}$

- $U' \equiv vars(cl_2) \setminus \{A_j^0 \in vars(cl_2) : A_j^i \in vars(cl_1) \setminus U\}$

Finally, we present the clause exit operation. Let cl_1 and cl_2 be the calling clause and the clause just executed, respectively. Let (G, N, U) be the abstract value holding at the calling point in cl_1, let $p(A_0^i, \ldots, A_m^i)$ be the calling atom, and let (G', N', U') be the abstract value holding at the end of cl_2. Then the abstract value associated with the program point following $p(A_0^i, \ldots, A_m^i)$ in cl_1 is:

- $G'' \equiv G \cup \bigcup_{A_j^i \in vars(cl_1) \wedge A_j^0 \in vars(cl_2) \cap G'} alias(A_j^i)$

- $N'' \equiv N \cup \bigcup_{A_j^i \in vars(cl_1) \wedge A_j^0 \in vars(cl_2) \cap N'} alias(A_j^i)$

- $U'' \equiv U \setminus \bigcup_{A_j^i \in vars(cl_1) \wedge A_j^0 \in vars(cl_2) \setminus U'} alias(A_j^i)$

It is easily verified that the above described operations returns values in $\mathcal{D}$.

4. Uninitialized Variable Analysis applied to Append

In this section we apply the abstract semantics to the program of Fig. 2. We implemented our domain in GAIA obtaining a modified version of that software. There are two main differences with respect to GAIA: the normalization and the association of abstract values with the program points. Moreover, we adopted a single-input/single-output granularity for the predicates since our goal was not the one of performing multiple specialization of the programs. In the second part we present a comparison with the analysis and with the speed-up obtained by [10].

4.1. The Fixpoint

We present the fixpoint computed by our implementation. Each program point is associated with a 3-tuple (G, N, U).

Clause cl_0				
main:-	{} {} {X1,X2,A_0^1,A_1^1,A_2^1}	A_0^1=X1	{} {} {X1,X2,A_0^1,A_1^1,A_2^1}	A_1^1=X2
{} {} {X1,X2,A_0^1,A_1^1,A_2^1}	A_2^1=.(a,.(b,.(c,[]))))	{A_2^1} {A_2^1} {X1,X2,A_0^1,A_1^1}	append(A_0^1,A_1^1,A_2^1)	{X2,A_1^1,A_2^1} {X1,X2,A_0^1,A_1^1,A_2^1 {}

Clause cl_1					
append(A_0^0,A_1^0,A_2^0):-	{A_2^0} {A_2^0} {A_0^0,A_1^0}	A_0^0 = []	{A_0^0,A_2^0} {A_0^0,A_2^0} {A_2^0}	A_1^0 = A_2^0	{A_0^1,A_1^1,A_2^1} {A_0^1,A_1^1,A_2^1} {A_0^1,A_1^1,A_2^1}

Clause cl_2			
append(A_0^0,A_1^0,A_2^0):-	{A_2^0} {A_2^0} {X1,X2,X3,X4,A_0^0,A_1^0,A_0^1,A_1^1,A_2^1}	A_0^0=.(X1,X2)	{A_2^0} {A_0^0,A_2^0} {X1,X2,X3,X4,A_1^0,A_0^1,A_1^1,A_2^1}
A_2^0 = X3	{A_2^0} {A_0^0,A_2^0} {X1,X2,X3,X4,A_1^0,A_0^1,A_1^1,A_2^1}	A_2^0=.(X1,X4)	{X1,X4,A_2^0} {X1,X4,A_0^0,A_2^0} {X2,X3,A_1^0,A_0^1,A_1^1,A_2^1}
A_0^1=X2	{X1,X4,A_2^0} {X1,X4,A_0^0,A_2^0} {X2,X3,A_1^0,A_0^1,A_1^1,A_2^1}	A_2^1 = X3	{X1,X4,A_2^0} {X1,X4,A_0^0,A_2^0} {X2,X3,X4,A_1^0,A_0^1,A_1^1,A_2^1}
A_2^1=X4	{X1,X4,A_2^0,A_2^1} {X1,X4,A_0^0,A_2^0,A_2^1} {X2,X3,A_1^0,A_0^1,A_1^1}	append(A_0^1,A_1^1,A_2^1)	{X1,X3,X4,A_1^0,A_2^0,A_1^1,A_2^1} {X1,X2,X3,X4,A_0^0,A_1^0,A_2^0,A_0^1,A_1^1,A_2^1} {}

The input mode for append(A_0^0,A_1^0,A_2^0) is (G={A_2^0},N={A_2^0},U={A_0^0,A_1^0}), while the output is (G={A_1^0,A_2^0},N={A_0^0,A_1^0,A_2^0},U={}).

Each of the concrete values computed by the concrete semantics is approximated by the abstract value associated with the corresponding program point. The approximation is by the means of the given semantics of the abstract element.

In the following we describe our optimization technique supported by the abstract values computed in the fixpoint. As an example, consider A_0^1 = X1 in clause cl_0. The abstract value associated with the program point $(cl_0, 1)$ (i.e. the following one) carries information stating that X1 is uninitialized. Thus, we substitute the put_x_void(0) with put_uninit_x_void(0) whose semantics differs from the one of put_x_void(0) only in that it does not initializes the top of the heap (but it simply increments the top pointer by 1). A similar reasoning applies for put_uninit_x_void(1) and put_x_void(1).

Since the first and the second arguments of *append* are uninitialized, the get instructions have to cope with arguments that cannot be dereferenced. As a side effect, the switch_* operations had to be modified, too. The switching operations do not have to dereference the first argument if it is uninitialized.

At program point $(cl_1, 0)$ the fixpoint expresses the fact that the first and the second arguments are uninitialized. Thus, we exploit this information to design two new instructions. We introduced get_uninit_nil_x0 whose semantics differs from the one of get_nil_x0 in that it does not perform the dereferencing of the first argument but it simply assigns the empty list to the location pointed to by the first argument variable. Moreover we designed get_uninit_x_str_value(1,2) whose semantics is simpler than the one of get_x_value(1,2) in that it exploits the knowledge about the uninitialization of the second argument and the knowledge about the compoundness of the third one.

```
A2versionB121C31902I-1J1J3L-1
clause(main/0/1, [ifshallow, neck(0), else, endif, put_list(2), unify_constant(a),
unify_list, unify_constant(b), unify_list, unify_constant(c), unify_nil,
put_uninit_x_void(0), put_uninit_x_void(1), execute(append/3)]).
clause(append/3/1, [get_uninit_nil_x0, get_uninit_x_str_value(1,2), neck(3),
proceed]).
clause(append/3/2, [ifshallow, get_uninit_list_x0, unify_uninit_x_variable(4),
unify_uninit_x_variable(3), get_list(2), unify_uninit_x_value(4), neck(3),
unify_uninit_x_variable(2), put_uninit_x_value(3,0), execute(append/3), else,
get_uninit_list_x0, unify_uninit_x_variable(3), unify_uninit_x_variable(0),
get_list(2), unify_uninit_x_value(3), unify_uninit_x_variable(2), execute(append/3),
endif]).
predicate(append/3, [switch_on_term([append/3/1,append/3/2],
[append/3/2],[[]-append/3/1],[])]).
predicate(main/0, [switch_on_term([main/0/1],[main/0/1],[],[main/0/1])]).
```

Figure 6: Optimized WAM code for `append`

At program point $(cl_2, 0)$ the fixpoint expresses the same information about the append arguments as in $(cl_1, 0)$. Thus we introduced a new instruction `get_uninit_list_x0` whose semantics differs from the one of `get_list_x0` since we know that the first argument is uninitialized.

Finally `unify_uninit_x_variable(I)` and `unify_uninit_x_value(I)`, if executed in write mode, simply make the variable `I` pointing to an uninitialized location, else, if executed in read mode, they perform the binding of `I` with the input value `S` (in the notation of [1]).

The whole optimized code is shown in Fig. 6.

4.2. The BAM and the speed-up

Now we compare the information computed by Aquarius and by our analysis. First we consider the input values for the predicate `append`. The BAM analysis computes the same information as our analysis about the first argument: the first argument is uninitialized. The information returned by the BAM about the second argument is `uninit_reg` which is a special kind of uninitialized variable. Our analysis returns the information that the second argument is uninitialized. Finally the third argument is known to be a recursively dereferenced structure by Aquarius while it is known to be a structure by our analysis.

As you can see, Aquarius is more precise than our analysis. If you look closely you can see that the differences arise where the recursively dereferenced analysis comes in. Therefore our work is an analysis that (at least on append) is as precise as Aquarius uninitialized variable analysis and, in addition, our analysis is simpler and its use does not require the definition of a new abstract machine but only a minor extension of the WAM.

We ran the code and the optimized code of *append* on a SUN workstation. The timing results indicate a speed-up of about 7-8% which is comparable with the execution speed-ups

of the BAM enhanced with the global dataflow analysis described in [10] which contains the recursively dereferenced term analysis and a mode analysis.

We ran our analyzer also on an ad_hoc program and the optimization results are analogous as in `append`. The extension of the optimization to other benchmarks will be the next part of the work.

5. Conclusion

A static analysis of logic programs has been presented for the introduction of uninitialized variables in the WAM. The concrete domain we proposed is an abstraction of the WAM memory model, that focuses only on the stack and on the registers.

The abstract domain consists of 3-tuples that express information about the groundness of the variables. Moreover it expresses information about the variables bound to non variable term and about the variables that are uninitialized.

The present work is a step of an ambitious project whose aims are the following:

1. The design of a very low-level generic concrete semantics of logic programs that can be instantiated in different ways in order to express different properties of the execution of logic programs.

2. Show that the concrete semantics expressing a particular property can be very useful for defining an abstract semantics for automatically inferring approximations of that property.

3. Use the method described in the previous point for designing correct static analyses for inferring other low-level properties.

We would like to thank Dr. Mats Carlsson for his kindness, timeliness and patience in answering our beginner-like questions about SICStus and the anonymous referees for their precious and interesting advices.

References

[1] Hassan Ait-Kaci. *Warren's Abstract Machine*. The MIT Press, Cambridge, MA, 1991. Prepared with LaTeX.

[2] J. Almgren, S. Andersson, L. Flood, C. Frisk, H. Nilsson, and J. Sundberg. *SICStus Prolog Library Manual*. Swedish Institute of Computer Science, October 1991.

[3] Joachim Beer. *The Occur-Check Problem Revisited. Journal of Logic Programming*, 5(3):243–261, September 1988.

[4] Garrett Birkhoff. *Lattice Theory*, volume 25 of *American Mathematical Society Colloquium Publications*. American Mathematical Society, New York, N.Y., third edition edition, 1967.

[5] Baudouin Le Charlier and Pascal Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for PROLOG. *ACM Transactions on Programming Languages and Systems*, 16(1):35–201, January 1994.

[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 ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977. ACM Press, New York.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–279, July 1992.

[8] Ralph Clarke Haygood. Native code compilation in SICStus Prolog. In Pascal Van Hentenryck, editor, *Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming*, pages 190–204, Massachusetts Institute of Technology, 1994. The MIT Press.

[9] R. Kowalski. *Logic for Problem Solving.* North-Holland, New York, 4 edition, 1983.

[10] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming.* PhD thesis, Computer Science Division, University of California Berkeley, December 1990.

[11] A. Taylor. Removal of dereferencing and trailing in prolog compilation. In Levi, Giorgio; Martelli, Maurizio, editor, *Proceedings of the 6th International Conference on Logic Programming (ICLP '89)*, pages 48–62, Lisbon, Portugal, June 1989. The MIT Press.

[12] David H. D. Warren. An Abstract PROLOG Instruction Set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Menlo Park, CA, October 1983.