Precise Garbage Collection in Prolog

Jan Wielemaker[1] and Ulrich Neumerkel[2]

[1] Universiteit van Amsterdam, The Netherlands
J.Wielemaker@uva.nl
[2] Technische Universität Wien, Austria
ulrich@complang.tuwien.ac.at

Abstract. In this paper we present a series of tiny programs that verify that a Prolog heap garbage collector can find specific forms of garbage. Only 2 out of our tested 7 Prolog systems pass all tests. Comparing memory usage on realistic programs dealing with finite datastructures using both poor and precise garbage collection shows only a small difference, providing a plausible explanation why many Prolog implementors did not pay much attention to this issue. Attributed variables allow for creating infinite lazy datastructures. We prove that such datastructures have great practical value and their introduction requires 'precise' garbage collection. The Prolog community knows about three techniques to reach at precise garbage collection. We summarise these techniques and provide more details on scanning virtual machine instructions to infer reachability in a case study.

1 Introduction

All modern Prolog systems come with a heap garbage collector, no longer limiting the programmer to revert to failure driven loops or **findall/3** to free unneeded memory through backtracking. For this article, we define a 'precise' garbage collector as a garbage collector that reclaims all data that can no longer be reached considering all possible execution paths from the current state without considering semantics. I.e. in `1==2, A=ok`, A is unreachable due to the semantics of ==/2, but we consider all parts of a conjunction reachable and therefore A is considered reachable. Our survey of 7 popular Prolog systems (Sect. 3) reveals that only two satisfy this definition. We compared the memory requirements between the poorest and best performance of GC on 5 very different real-world programs (Tab. 2). The comparison indicates that precise GC is unimportant for many programs, which provides a plausible explanation why precise GC is not widespread.

Precise GC becomes important for processing infinite datastructures, in this case distinguished from *cyclic* structures. A truly infinite structure clearly never fits into finite physical memory. We are concerned with datastructures that grow due to further instantiation while (older) parts of the datastructure become unreachable after processing and can be reclaimed by the garbage collector. A typical example is processing input using a list: the list is expanded as new

input becomes available, while the head of the list becomes unreachable after being processed deterministically. This approach is used in [9], where infinite lists are used for communication between concurrent processes. Similar consideration motivated improvements in functional languages [17].

This article is organised as follows. First, in Sect. 2 we make a case for the practical value of infinite lazy datastructures and the requirement of precise GC. In Sect. 3 we identify possible leaks and test 7 Prolog implementations for them, 5 of which exhibit two or more leaks. This is followed by a survey of known existing techniques to reach precise GC and the description and evaluation of a case study adding precise GC to SWI-Prolog.[3]

2 A case for infinite lazy datastructures: pure input

Prolog DCG and other parsing techniques are based on processing lists. Unfortunately, the data that needs to be parsed is often provided as a Prolog stream that accesses data from the outside world. This problem has been identified long ago and many implementations of DCG provide a hook 'C'/3 to read an input character. This hook is of little practical use, notably due to the poor combination of non-determinism and side-effects. The current proposal for an ISO standard on DCGs [8] no longer mentions 'C'/3. Fortunately, extended unification [13, 11, 10, 6] using attributed variables as found in many modern Prolog systems provides a straightforward mechanism to remedy this problem.

Figure 1 presents the simple algorithm to apply a grammar rule on input from a file as it appears in the SWI-Prolog library `pure_input.pl`. Besides standard ISO predicates, the implementation depends on **freeze**(*Var, Goal*), which delays *Goal* until *Var* becomes instantiated (coroutining); **call_cleanup**(*Goal, Cleanup*) which allows for closing the input handle when *Goal* becomes inaccessible due to deterministic termination, an exception or pruning of a choicepoint and finally **read_pending_input**(*Handle, Head, Tail*) which reads a block of buffered input into the difference-list `Head\Tail`. Freeze or a substitute is available in all systems with attributed variables. Call_cleanup is available in multiple Prolog implementations and has been discussed for inclusion in the upcoming revision of Part I of the ISO Prolog standard.[4] A block-read operation is not defined by the ISO standard but trivial to implement while it provides a very significant speedup (12× in SWI-Prolog 5.6.59) because it only needs to validate and lock the stream handle once.

The **phrase_from_file**(*:DCG, +File*) definition in Fig. 1 allows for applying an arbitrary non-deterministic DCG completely transparently on the content of a file while, given precise GC, the memory usage is independent from the size of this file. We compared the use of a DCG on a file with a carefully handcrafted program to count words in a text-file. We summarise the key results in the table below and conclude that the DCG version is much easier to read and very comparable in performance.

[3] http://www.swi-prolog.org

[4] Inclusion is stalled because the precise semantics prove hard to describe.

```
read_to_input_stream(Handle, Pos1, Stream0) :-
    set_stream_position(Handle, Pos1),
    (   at_end_of_stream(Handle)
    ->  Stream0 = []
    ;   read_pending_input(Handle, Stream0, Stream1),
        stream_property(Handle, position(Pos2)),
        freeze(Stream1, read_to_input_stream(Handle, Pos2, Stream1))
    ).

phrase_from_file(Phrase, File) :-
    open(File, read, Handle),
    stream_property(Handle, position(Pos)),
    freeze(Stream, read_to_input_stream(Handle, Pos, Stream)),
    call_cleanup(phrase(Phrase, Stream), close(Handle)).
```

Fig. 1. Implementation of input streams.

	traditional	**DCG on file**
Code size (lines)	31	22
Time (sec., 25MB file)	16.1	17.1
GC time (sec.)	0.9	1.4

From the above, we conclude that infinite (lazy) terms have great practical value and it is therefore desirable that garbage collection is capable of reclaiming the no-longer-accessible part of the term.

3 State of the art

Can pure input as described above be used in current Prolog systems with coroutining? We reviewed 7 Prolog implementations. The first obvious requirement is that there is no memory leak after a deterministic wakeup of a delayed goal (Sect. 3.1). The other requirements are about reclaiming unneeded parts of the input list within and-control and or-control. I.e. we must be able to create a list of arbitrary size if there are no references to the entire list. The simplest form is the test below. Predicate **f/1** builds a list, but as nobody uses it, GC reclaims it and **run/0** runs forever in constant space.

```
run :- f(_).
f([f|X]) :- f(X).
```

This is the simplest case, where the initial list is created through a singleton variable. In WAM-based systems with registers, the list resides in a register that is overwritten in each recursion. On virtual machines such as the ZIP [3, 14] and ATOAM [18] that pass arguments over the stack, last-call optimization overwrites the arguments, making the head inaccessible.

We will now go systematically through requirements to deal with infinite (lazy) datastructures. The first property validates that deterministic instantiation of an attributed variable does not leak. The remaining properties validate that various scenarios where the head of the list becomes inaccessible are detected by the garbage collector. Each test case considers a situation that requires special attention in one or more virtual machines, based on our understanding of, notably, the WAM and ZIP. As the number of possible virtual machines is unbounded, it is not possible to be sure that these cases cover all cases in all possible virtual machines. Each property is accompanied by a program that must run forever in constant space. A test is considered 'failed' if the system aborts or memory usage exceeds 1Gb. The given programs are very simple, using a fact **dummy/1** to pretend access to a variable. We assume that **dummy/1** cannot be optimized away by the compiler, otherwise a more complex replacement is needed.

3.1 Property 1: Permanent removal of attributes

Attributed variables that have been unified deterministically with a non-variable term must be reclaimed completely. This property can be tested using the program below. It creates delayed goals and executes them through deterministic binding. Note that for most constraint solvers, complete reclamation of attributed variables is not strictly necessary. Most CLP(FD) programs are concerned with finding solutions nondeterministically via a labeling procedure, thus most volatility stems from backtracking and not from forward recursion.

```
run :- run(_).
run(X) :- freeze(X, dummy(X)), X = 1, run(T).
dummy(_).
```

3.2 Property 2: And-control (head variables)

Variables appearing in the head of a rule and in the body must be discarded as soon as possible. We test this using the following which, like the previous test, must run forever in bounded memory. The call to **dummy/2** ensures $L0$ is not made inaccessible due to last call optimization.

```
run :- run(_,_).
run(L0, L) :- f(L0, L1), dummy(L1, L).
f([g|X], Y) :- f(X, Y).
dummy(Xs, Xs).
```

3.3 Property 3: And-control (existential variables)

Existential variables that occur in several goals, but not the last one. Ideally such variables should be covered by environment trimming [5] in the WAM. Careful environment trimming avoids more complex treatment.

```
run :- run(_,_).
run(L0, L) :- dummy(L0, L1), f(L1, L2), dummy(L2, L).
f([f|X], Y) :- f(X, Y).
dummy(Xs, Xs).
```

3.4 Property 4: Or-control

Or-control covers the case where a variable is only accessible from a choicepoint. A well behaved garbage collector will reset such variables and discard their current value (early-reset, [1]). This situation arises in disjunctions in grammars. E.g. `(...,"a"|...,"b")`, where .../0 is defined to match an unbounded string.

```
run :- run(_).
run(X) :- f(X).
run(X) :- X == [].
f([f|X]) :- f(X).
```

3.5 Property 5: Branching inside a clause

Branching (*A*;*B* and *If*`->`*Then*`;`*Else*) using different ordering of the variables in both branches cannot be handled optimally with the WAM environment trimming as the branches require different environment layout. This test is only of interest for systems that open code disjunctions, avoiding an auxiliary internal definition.

```
run(Z) :- p(_,_,Z).
p(X,Y,Z) :- (Z > 0 ->  f(X), g(Y), dummy ; g(Y), f(X), dummy).
f([f|X]) :- f(X).
g([g|X]) :- g(X).
dummy.
```

3.6 Conclusion from our survey

	1	2	3	4	5_0	5_1	VM
SICStus 3.12.5	ok	ok	ok	ok	ok	ok	WAM
Ciao 1.10p8	ok	ok	ok	ok	ok	ok	WAM
YAP 5.0.1	n	ok	ok	ok	ok	n	WAM
ECLiPSe 5.10	ok	ok	n	ok	n	n	WAM
SWI 5.6.54	n	n	n	n	n	n	ZIP
BProlog 7.1	n	ok	n	n	n	n	ATOAM
XSB 3.1	n	ok	n	n	n	n	WAM

Table 1. Evaluation of GC in some popular Prolog systems with coroutining. The numbers correspond to the properties. Property 5 is tested for both branches.

In the above sections we have provided tests for the main properties of Prolog coroutining and garbage collection needed to be able deal with infinite lazy datastructures. The results are shown in Tab. 1. As detailed descriptions of GC in these systems is either not in the literature or the description is likely to be outdated and we do not have access to the source code of all these systems we have not examined *why* tests succeed or fail. We merely conclude that precise GC has not been given much attention by the respective developers. Table 2 justifies this behaviour in the absence of infinite datastructures.

To the best of our knowledge, SICStus[5] and the derived Ciao system [9] reach a precise result using WAM registers, environment trimming and the implementation of in-clause alternative execution paths using anonymous predicates. The YAP VM uses virtual machine instructions for in-clause alternative execution paths, which cannot be handled perfectly with only environment trimming as explained in Sect. 4. It is hard to explain the behaviour of the other systems.

Our study started with providing a pure input library for SWI-Prolog. SWI-Prolog *design* was ok for property 1, but the implementation was proven flawed. As the SWI-Prolog virtual machine passes arguments over the stack and does not use environment trimming, it failed on all test.

4 Related work on data reachability in Prolog

Prolog systems discard data during backtracking. During forwards execution, discarding data is achieved by the heap garbage collector. The garbage collector preserves all data that is accessible through a set of *root pointers* [2]. The precise set of root pointers depends on the Virtual Machine (VM) architecture, where we distinguish between VMs that pass arguments in registers (WAM) and VMs that pass arguments using the stack (ZIP, ATOAM). The current stack frame and choice point are always root pointers. Registers and global variables are other examples. There are several mechanisms by which data becomes inaccessible from the set of root pointers that are part of the normal Prolog (forward) execution:

- Temporary variables allocated in registers become inaccessible when they are overwritten.
- Arguments (on machines passing arguments over the stack) and environment slots become inaccessible if the frame is discarded due to last-call optimization.
- Environment trimming (see below) shrinks the environment, discarding unneeded parts as the execution of the clause progresses.

Environment trimming [5] allocates variables in the environments ordered by the last subgoal that references the variable. Each call to a subgoal has an additional numeric argument that states that the first N variables of the

[5] www.sics.se/sicstus/ explained to one of the authors by Mats Carlsson.

environment are still valid. Together with registers for argument passing and last-call optimization, environment trimming reaches a precise result if there are *no alternative execution paths* in the VM instructions. This implies that disjunction $(A;B)$ and *If*`->`*Then*`;`*Else* must be translated into pure (anonymous) predicates with some additional machinery to deal with proper scoping of the cut. This technique is used by SICStus Prolog and Ciao (see Sect. 3.6.

Many virtual machines realise disjunction and if-then-else using branch instructions in the VM. As different subgoal ordering in the alternate execution paths may require different ordering of variables in the environment (Sect. 3.5), there is no longer a perfect order and garbage collection that scans the entire environment will mark data that is no longer reachable because there is no instruction that refers to some variable. Table 1 suggests this is the status in YAP 5.0.1.

Environment trimming cannot deal with arguments that are passed over the stack as their order is determined by the calling convention and, analogous to in-clause branching, different clauses of the predicate generally require a different ordering.

VMs that pass arguments over the stack as well as VMs that use branching instructions to code in-clause alternate execution paths need additional measures to regain precise GC. Two techniques to achieve this have been part of the Prolog folklore for some time.[6] One scans the VM instructions from the continuation points to find the accessible variables. It was used by old versions of BIM Prolog. With native code this became very hard to maintain. The other uses compiler generated bitmaps for each possible continuation point that represent all reachable variables. This is used by BIM Prolog and hProlog.

In systems based on 'Binary Prolog' [15], continuations take the place of environments. They are represented by ordinary Prolog terms and therefore profit from the same data representations [16]. Garbage collection in such systems [7] do not require any special treatment. On the other hand, Binary Prolog requires more space for representing variables within continuations than traditional implementations. Every occurrence of a variable is now represented separately, while traditional environments represent each variable only once.

5 Our case study: SWI-Prolog

SWI-Prolog is based on the ZIP VM which passes arguments over the stack and uses branching instructions inside a clause. Like most today's Prolog systems, the VM is emulated. We briefly examine these properties under the assumption that the optimal choice depends on the specific setting: desired performance, portability, transparency for debugging, simplicity and speed of the compiler.

- *Argument passing*
  The use of registers for argument passing as the WAM has some clear advantages. It keeps the environment small and simplifies last-call optimization.

[6] according to Bart Demoen

This comes at a price: the compiler is more complicated and it is harder to provide a (graphical) debugger that provides access to variables in the parent frames.

- *Branching instructions*
  Using branching instructions to code disjunction and if-then-else prohibits precise trimming of the environment as we have seen in Sect. 4. On the other hand, execution is generally faster as no environment needs to be created for the anonymous predicates that otherwise replace different in-clause execution paths. We have no information on the implementation effort associated with these approaches.
- *Emulated VM vs. native code*
  An emulated VM is clearly easier to implement and if the VM is written in a portable language, portability of the system comes for free. In addition, it allows for simple decompilation [4] and simplifies two tasks in GC: identify not-yet-initialized variables in the environment and identify variables that can still be accessed from a given program counter (PC) location. SICStus has dropped native code in release 4[7]

The above observations make it clear that scanning VM instructions to remedy the reachability problem is the most obvious approach for SWI-Prolog. Because most todays Prolog implementation use an emulated VM and Tab. 1 proves that several systems still need to realise precise GC we believe a description of our case study will help persuading other implementors to implement precise GC and will help them to take the correct decisions right away.

6 Implementation

The SWI-Prolog VM differs considerable from the much more widely adopted WAM. SWI-Prolog's garbage collector however closely follows the SICStus Prolog garbage collector, which is described excellently in [1]. The fact that our GC closely follows a GC for a WAM-based system gives some confidence that our findings are applicable to a wider range of Prolog implementations. This section only concentrates on the modifications to the algorithm described in [1] and cannot be understood without detailed understanding of this paper.

Our modifications only affect the *marking* phase of GC. The modified algorithm is provided in pseudo code in Fig. 2 and discussed below. Added lines and deleted lines are marked with `+/-` at the start of the line.

First, initialize_and_mark() marks all data that is accessible from the continuation PC and at the same time initialises variables for which it finds a 'first-access' instruction, finishing the initialization of the environment. All environments are marked as 'seen'. This is the same as in [1], except

[7] Mats Carlsson has confirmed that SICStus 4 uses VM code scanning to deal with uninitialized variables in the environment. See also http://www.sics.se/sicstus/docs/latest4/pdf/relnotes.pdf

```
procedure mark_environments(env, PC)
   while ( env )
       if ( not_seen(env) )
           set_seen(env)
-          initialize(env, PC)
+          initialize_and_mark(env, PC)
           PC = env->PC
           env = env->parent
       else
+          mark(env, PC)
           return

procedure mark_choices(ch)
    env = ch->environment
    early_reset_trail()
    while ( ch )
        if ( pc_choice(ch) )
            mark_environments(env, ch->PC)
        else if ( alt_clause(ch) )
+           unmarked = count_unmarked_arguments(env)
+           while ( unmarked > 0 && clause )
+               mark_arguments(env, clause->code)
+               clause = next_visible(clause)
            if ( not_seen(env) )
                set_seen(env)
                mark_environments(env->parent, env->PC),
+       else if ( foreign_choice(ch) )
+           mark_all_arguments(env);

procedure mark_stacks(env, ch, PC)
    mark_environments(env, PC)
    mark_choices(ch)
```

Fig. 2. Pseudo code for the marking algorithm

- Mark variables in the environment that are referred to by instructions reachable from the PC instead of all variables in the environment.
- If we find a reference pointer to a parent environment, we mark the pointer and continue marking the referenced destination. In the traditional algorithm the variable in the parent is marked if we mark the parent environment. Now we must cover the case where the corresponding variable is accessed in this frame, but not in the parent frame.
- When called from mark_choices(), that marking is normally aborted if the frame has already been seen. Now we must continue to mark the first seen environment as this continuation may have a different PC and thus access to different variables. There is no need to continue with the parent frame as that has already been marked using the same PC.

Marking choicepoints is also similar to [1]. It resets trail entries that point to garbage cells (early reset, dealing with property 4) and then marks the associated environment. As SWI-Prolog passes arguments over the stack, if an alternate clause is encountered we need to keep all arguments that are used by the remainder of the clause list (possibly reduced due to indexing). Simply scanning the code of each clause could scan a lot of code on, for example, predicates with many facts. We avoid this by computing the number of unmarked arguments and abort the scan if all arguments are marked. Note that a clause without singleton variables in the head accesses all arguments and thus stops the search. Ground facts are a common example. Finally, as we have no information on how a foreign predicate accesses its arguments we must mark all arguments as accessible.

Sweeping an environment has been changed slightly. In [1], all heap references in the environment are inserted into relocation chains. Now, we first check whether the heap reference is marked. If so, we put it into a relocation chain as before, otherwise we assign the atom `'<garbage_collected>'` to the variable. This ensures consistency of the environment variable after heap relocation and is needed by the debugger if execution switches from normal mode to debug mode after a user interrupt or explicit call to **trace/0** inside code running in no-debug mode. In such cases, the debugger may show arguments of parent goals that were executed in normal mode as `'<garbage_collected>'` and the graphical debugger may show variables from the environment this way.

Note that if the program was started in debug mode, all data remains accessible through extra 'debug' choicepoints that also facilitate 'retry' at goals that were started deterministically. Figure 6 illustrates the problem using an explicit call to **garbage_collect/0** and **trace/0**. Explicitly calling **trace/0** is common practice to start debugging in a very specific state. The explicit call to **garbage_collect/0** is there only to illustrate what happens if GC was invoked at that specific point, while the system still operates in no-debug mode.

```
test :-
        read_line_to_codes(user_input, List),
        all_spaces(List).

all_spaces([]).
all_spaces([0' |T]) :- !, all_spaces(T).
all_spaces(_) :- garbage_collect, trace, fail.

1 ?- test.
|:   xx.
   Call: (11) fail ? goals
     [11] fail
     [10] all_spaces('<garbage_collected>')
      [1] '$toplevel'
   Call: (11) fail ?
```

Fig. 3. The debugger showing a garbage collected argument

7 Evaluation

Our evaluation considers four aspects: time, space, implementation effort and maintenance. In the tradition of SWI-Prolog, we consider mainly real and large applications. We selected the following applications because of diversity, size and the amount of garbage collection involved: CHAT80 (Pereira & Warren, 1986) running its test-suite in a forward chaining loop to force GC, Back52 (Thomas Hoppe et all., 1993) running its test suite, CHR compiler (Tom Schrijvers) compiling itself, k123.pl (Peter Vanbroekhoven) and pgolf.pl (Mats Carlsson).

The results are shown in Tab. 2. The first set of columns describe the overall timing, the last set describes characteristics of the code scanning version only and is discussed in Sect. 7.3. All timings are executed on an AMD Athlon X2 5400+; 64-bit Linux 2.6 using the 64-bit development version of SWI-Prolog based on 5.6.55. Reported time is in seconds. Frequency stepping was disabled during the tests.

7.1 Time evaluation

Table 2 shows that the overall execution time is only slightly affected by our changes. Note that the logic to trigger GC depends on the amount of memory that is accessible after the previous GC and therefore different effectiveness of GC leads to unpredictable overall behaviour of the program in terms of time and number of garbage collections.

We obtained a detailed breakdown of the garbage collector using valgrind [12] with the *callgrind* tool and *kcachegrind* to explore the results. The overhead of analysing instructions is approximately 1% of the garbage collector marking

Test	Time	#GC	GCLeft	GCTime	AvgScan	AvgCls	AvgInstr
Without code scanning							
k123	8.88	164	1,594,534	1.35			
chat80	2.56	109	18,661	0.10			
back52	2.31	406	5,589	0.17			
pgolf	13.22	53	7,328,689	3.44			
chr	6.41	36	3,466,387	1.17			
With code scanning							
k123	8.71	209	1,111,646	1.20	1.51	0.09	12.10
chat80	2.42	111	12,301	0.08	1.68	0.60	14.52
back52	2.21	420	3,360	0.15	1.56	0.12	11.19
pgolf	11.06	53	7,151,304	3.19	1.42	0.01	12.37
chr	6.29	38	3,265,471	1.15	1.91	0.32	14.52

Table 2. Effects of code scanning. *Time* is the total execution time (including GC time); *#GC* the number of garbage collections; *GCLeft* the average amount of memory (heap+trail) immediately after GC and *GCTime* the time spent on GC. *AvgScan* is the average number of continuation points that must be explored for an environment; *AvgCls* the average number of additional clauses scanned; *AvgInstr* the average number of instructions scanned before reaching the end of the clause.

time. These timing are slightly distorted because gcc's inline function optimization needs to be disabled to analyse the breakdown of execution time over the various functions.

7.2 Space evaluation

Our approach based on marking accessible data by scanning the VM instructions obviously reaches the 'precise' result as defined in the introduction for the heap and trail stack. It does not provide the optimal result for the environment stack. Only the approach as taken by SICStus is optimal here in the sense that the stack contains no variables that are not accessible, while using our marking approach the variables remain in the environment, bound to `'<garbage_collected>'`. Environment stack usage is in practice rarely a bottleneck and our deficiency is a constant amount rather than the difference between finite and infinite stack usage.

Table 2 also explains why precise GC is not widespread. Except for memory usage of the k123.pl test, we find no noticeable differences in the memory usage after GC. The k123 program is a small program (75 lines after cleanup of unreachable code). The central predicate **mmul/3** in Fig. 4 is deterministic. Lacking temporary registers and environment trimming, the old SWI-Prolog, could not dispose the intermediate matrices.

Implementation and maintenance Only the code for marking environments and clearing uninitialised variables was extended from originally 150 lines (C), to 557 including comment and debugging statements. Total implementation effort was

```
mmul(M, M6) :-
        mmul(M, M, M1),   mmul(M1, M1, M2), mmul(M2, M2, M3),
        mmul(M3, M3, M4), mmul(M4, M4, M5), mmul(M5, M5, M6).
```

Fig. 4. Main routine of k123.pl

4 days. One of the problems associated with VM instruction interpretation is maintenance that results from changing the instruction set. SWI-Prolog maintains information of the instruction format for each instruction. This is used to list VM instructions, deal with saving and loading and simplifies VM instruction scanning as it allows enumerating the instructions using a generic loop. Four instructions have variable length data associated with them (packed string and unbounded integer) and need (uniform) special attention.

In addition to the generic code walking, 36 out of 89 instructions require special attention as described in table Tab. 3. The table states the number of instructions the marking algorithm needs to understand, the number of groups of instructions that require different treatment (especially the variable accessing functions are often handled using the same code) and the number of lines of C-code involved.

Description	instructions	groups	lines
Identify flow control	6	5	44
Realise initialization of uninitialised variables	3	1	10
Identify variable access for marking (body)	14	6	30
Identify variable access for marking (head)	13	6	27

Table 3. Statistics on interpreting VM instructions

7.3 Discussion

Before we arrived at the current implementation we had two worries: prohibitive costs of multiple scans of the same code from different continuations and prohibitive scans of code from multiple clauses to identify the still-reachable arguments. Column *AvgCls* of Tab. 2 (page **??**) indicates that scanning alternative clauses is cheap, while the value of equal GC behaviour between in-clauses disjunctions and alternative clauses is obvious.

Our first prototype avoided multiple scans of the same code from different continuations. Not correctly dealing with early-reset, this code was flawed and abandoned. Nevertheless, it executed the above programs correctly and we obtained statistics on its effectiveness. On the above test cases, multiple scans increase the number of scanned instructions by 0, 58%, 7%, 7% and 3% (same order as Tab. 2). As the scanning itself is responsible for less than 1% of the

time of the mark phase, it is considered neglectable. This conclusion can also be drawn from column *AvgScan* and *AvgInstr* together with the 1% time spent on code scanning.

8 Conclusions

We have defined a set of five properties, each of which accompanied with a very simple test case, that must be satisfied to deal with infinite (lazy) datastructures in Prolog. We have proven that such datastructures are of significant practical value as they can be used to realise processing a repositionable input stream using the full power of non-deterministic grammar rules (DCGs). The majority of Prolog implementations that provide the required attributed variables to realise a lazy datastructure does not provide the required precise garbage collector. Precise GC can be realised using a VM that uses registers to pass arguments, implements environment trimming and codes in-clauses disjunction using anonymous predicates. Our case study indicates that other virtual machines can be remedied by scanning virtual machine instructions to identify reachable variables in the environment. This technique is viable for any Prolog system based on emulating virtual machine instructions. Next to supporting infinite datastructure, the approximately 1% extra cost in the marking phase is more than compensated for in the compacting phase of the garbage collector.

The current version of SWI-Prolog is shipped with the described enhancements to the garbage collector and a library to use DCGs on repositionable input streams.

Acknowledgements

We would like to thank Mats Carlsson for explaining to one of the authors how the reachability problem is solved in SICStus Prolog, Bart Demoen for explaining some folklore and the bitmap technique and Paulo Moura for investigating the state of the art in some popular Prolog systems as shown in Tab. 1. Vitor Santos Costa has confirmed property 1 for YAP, which is planned to be fixed soon.

References

1. Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, 1988.
2. Yves Bekkers, Olivier Ridoux, and Lucien Ungaro. Dynamic memory management for sequential logic programming languages. In *Workshop on Memory Management*, 1992. LNCS 627.
3. D. L. Bowen, L. M. Byrd, and WF. Clocksin. A portable Prolog compiler. In L. M. Pereira, editor, *Proceedings of the Logic Programming Workshop 1983*, Lisabon, Portugal, 1983. Universidade nova de Lisboa.
4. Kevin A. Buettner. Fast decompilation of compiled prolog clauses. In Ehud Y. Shapiro, editor, *ICLP*, volume 225 of *Lecture Notes in Computer Science*, pages 663–670. Springer, 1986.

5. Luís Fernando Castro and Vítor Santos Costa. Understanding memory management in Prolog systems. In Philippe Codognet, editor, *ICLP*, volume 2237 of *Lecture Notes in Computer Science*, pages 11–26. Springer, 2001.
6. Bart Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium, oct 2002. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.
7. Bart Demoen, Paul Tarau, and Geert Engels. Segment order preserving copying garbage collection for wam based prolog. In *Symposion on Applied Computing (SAC)*, pages 380–386. ACM, 1996.
8. Paulo Moura et. al. Prolog, 2006. ISO/IEC DTR 132113:2006.
9. Manuel V. Hermenegildo, Daniel Cabeza Gras, and Manuel Carro. Using attributed variables in the implementation of concurrent and parallel logic programming systems. In *ICLP*, pages 631–645, 1995.
10. Christian Holzbaur. Metastructures versus attributed variables in the context of extensible unification. In *PLILP*, volume 631, pages 260–268. Springer-Verlag, 1992. LNCS 631.
11. Serge Le Huitouze. A new data structure for implementing extensions to prolog. In *PLILP*, volume 456, pages 136–150. Springer-Verlag, 1990. LNCS 456.
12. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 89–100. ACM, 2007.
13. Ulrich Neumerkel. Extensible unification by metastructures. In Maurice Bruynooghe, editor, *Proceedings of META90, Workshop on Meta-Programming in Logic*, Leuven, Belgium, April 1990.
14. Ulrich Neumerkel. The binary WAM, a simplified Prolog engine. Technical report, Technische Universität Wien, 1993. http://www.complang.tuwien.ac.at/ulrich/papers/PDF/binwam-nov93.pdf.
15. Paul Tarau and Michel Boyer. Elementary logic programs. In *PLILP*, pages 365–381. Springer-Verlag, 1990. LNCS 456.
16. Paul Tarau and Ulrich Neumerkel. A novel term compression scheme and data representation in the binwam. In *PLILP*, pages 73–87. Springer-Verlag, 1994. LNCS 844.
17. Philip L. Wadler. Fixing some space leaks with a garbage collector. *Software Practice and Experience*, 17(9):595–609, 1987.
18. Neng-Fa Zhou. Garbage collection in B-Prolog. In *Proc. of the First Workshop on Memory Management in Logic Programming Implementations*, 2000.