Proceedings of the Workshop on

Principles of Abstract Machines

19. September, Pisa, Italy

Editors: Stephan Diehl, Peter Sestoft

Technischer Bericht A 05/98

FB 14 - Informatik
Universität des Saarlandes, Postfach 15 11 50
66041 Saarbrücken , GERMANY
Phone: ++49-681-3023915
diehl@cs.uni-sb.de

Workshop Proceedings

Workshop on Principles of Abstract Machines

in conjunction with the
Joint International Symposia SAS'98 and PLILP/ALP'98 (14-18.9)

19. September 1998, Pisa, Italy

Recently the topic of abstract machines has got a new boost by the success of the Java Virtual Machine. For many years abstract machines have been designed for different sorts of languages including imperative, object-oriented, eager functional, lazy functional, constraint and logic languages, as well as hybrid languages. The goal of this workshop is to bring together researchers and developers working on different language paradigms.

Organizers

Stephan Diehl (Saarbrücken, Germany) diehl@cs.uni-sb.de
Peter Sestoft (Copenhagen, Denmark) sestoft@dina.kvl.dk

Program Committee

- Michael Franz (Irvine, California)
- Michael Hanus (Aachen, Germany)
- Pieter Hartel (Southampton, UK)
- Peter van Roy (Louvain-la-Neuve, Belgium)

Table of Contents

A better CAT made-in-Belgium: CHAT* (or KAT**)

Bart Demoen and Konstantinos Sagonas

Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium
e-mail: {bmd,kostis}@cs.kuleuven.ac.be

Abstract. The Copying Approach to Tabling, abbrv. CAT, is an alternative to SLG-WAM and based on total copying of the areas that the SLG-WAM freezes to preserve execution states of suspended computations. The disadvantage of CAT as pointed out in a previous paper is that in the worst case, CAT must copy so much that it becomes arbitrarily worse than the SLG-WAM. Remedies to this problem have been studied, but a completely satisfactory solution has not emerged. Here, a hybrid approach is presented: CHAT. Its design was guided by the requirement that for non-tabled (i.e. Prolog) execution no changes to the underlying WAM engine need to be made. CHAT combines certain features of the SLG-WAM with features of CAT, but also introduces a technique for freezing WAM stacks without the use of the SLG-WAM's freeze registers that is of independent interest. Empirical results indicate that CHAT is a better choice for implementing the control of tabling than SLG-WAM or CAT. However, programs with arbitrarily worse behaviour exist.

1 Introduction

In [2], we developed a new approach to the implementation of the suspend/resume mechanism that tabling needs: CAT. The essential characteristic of the approach is that freezing of the stacks (as in SLG-WAM [4]) was replaced by copying the state of suspended computations. One advantage is that this approach to implementing tabling does not introduce new registers, complicated trail or other inefficiencies in an existing WAM: CAT does not interfere at all with Prolog execution. Another advantage is that CAT can perform completion and space reclamation in a non-stack based manner without need for memory compaction. Finally, experimentation with new strategies seems more easy within CAT. On the whole, CAT is also easier to understand than SLG-WAM. The main drawback of CAT, as pointed out in [2], is that its worst case performance renders it arbitrarily worse than SLG-WAM: CAT might need to copy arbitrary large parts of the stacks; the SLG-WAM's way of freezing in contrast is an operation with

* the Copy-Hybrid Approach to Tabling (to be pronounced in French).
** the K.u.leuven Approach to Tabling (to be pronounced in Flemish).

constant cost. Although this bad behaviour of CAT has not shown up as a real problem in our uses of tabling (see [2] and the performance section of this paper), in [3] we have described a partial remedy for this situation. Restricted to the heap, it consists of performing a minor garbage collection while copying; that is, preserve only the useful state of the computation by copying just the data that are used in the continuation of the consumer. The same idea can be applied to the environment stack as well. [3] contains some experimental data which show that this technique is quite effective at reducing the amount of copying in CAT. This is especially important in applications which consist of a lot of Prolog computation and few consumers. However, even this memory-optimised version of CAT suffers from the same worst case behaviour compared to SLG-WAM. Nevertheless, for most applications CAT is still a viable alternative to SLG-WAM.

We therefore felt the need to reconsider the underlying ideas of CAT and SLG-WAM once more. In doing so, it became quite clear that all sorts of hybrid methods are also possible, e.g. one could copy the environment stack while freezing the heap, trail and choice point stack, etc. However, we are convinced that the guiding principle behind any successful design of a (WAM-based) tabling implementation must be that the necessary extensions to support tabling should not impair the efficiency of the underlying abstract machine for (strictly) non-tabled execution, and support for tabled evaluation should be possible without requiring difficult changes: CAT was inspired by this principle and provides such a design. CHAT, the hybrid CAT we present here enjoys the same property.

If the introduction of tabling must allow the underlying abstract machine to execute Prolog code at its usual speed, we have to preserve and reconstruct execution environments of suspended computations without using SLG-WAM's machinery; in other words we have to get rid of the freeze registers and the forward trail (with back pointers as in the SLG-WAM). The SLG-WAM has freeze registers for heap, trail, environment stack (also named local stack) and choice point stack. These are also the four areas which CAT selectively copies. What CHAT does with each of these four areas is described in Section 3 which is the main section of this paper. Section 4 shows best and worst cases for CHAT compared to SLG-WAM. Section 5 discusses the combinations possible between CHAT, CAT and SLG-WAM. Section 7 shows the results of some empirical tests with CHAT and Section 8 concludes.

2 Notation and Terminology

Due to space limitations we assume familiarity with the WAM (see e.g. [1, 5]), SLG-WAM [4] and CAT [2]. We also assume a four stack WAM, i.e. an implementation with separate stacks for the choice points and the environments as in SICStus Prolog or in XSB. This is by no means essential to the paper and whenever appropriate we mention the necessary modifications of CHAT for the original WAM design. We will also assume stacks to grow downwards; i.e. higher in the stack means older, lower in the stack (or more recent) means younger.

We will use the following notation: **H** for top of heap pointer; **TR** for top of trail pointer; **E** for current environment pointer; **EB** for top of local stack pointer; B for most recent choice point; the (relevant for this paper) fields of a choice point are H and EB, the top of the heap and local stack respectively at the moment of the creation of the choice point; for a choice point of type T pointed by **B**, these fields are denoted as $\mathbf{B}_T$[H] and $\mathbf{B}_T$[EB] — T is either Generator, Consumer or Prolog choice point. The SLG-WAM uses four more registers for freezing the WAM stacks; however only two of them are relevant for this paper. We denote them by **HF** for freezing the heap, and **EF** for freezing the environment stack.

In a tabling implementation, some predicates are designated as *tabled* by means of a declaration; all other predicates are *non-tabled* and are evaluated as in Prolog. The first occurrence of a tabled subgoal is termed a *generator* and uses resolution against the program clauses to derive answers for the subgoal. These answers are recorded in the table (for this subgoal). All other occurrences of identical (e.g. up to variance) subgoals are called *consumers* as they do not use the program clauses for deriving answers but they consume answers from this table. Implementation of tabling is complicated by the fact that execution environments of consumers need to be retained until they have consumed all answers that the table associated with the generator will ever contain.

To partly simplify and optimize tabled execution, implementations of tabling try to determine *completion* of (generator) subgoals: i.e. when the evaluation has produced all their answers. Doing so, involves examining dependencies between subgoals and usually interacts with consumption of answers by consumers. The SLG-WAM has a particular stack-based way of determining completion which is based on maintaining *scheduling components*; that is, sets of subgoals which are possibly inter-dependent. A scheduling component is uniquely determined by its *leader*: a (generator) subgoal G_L with the property that subgoals younger than G_L may depend on G_L, but G_L depends on no subgoal older than itself. Obviously, leaders are not known beforehand and they might change in the course of a tabled evaluation. How leaders are maintained is an orthogonal issue beyond the scope of this paper; see [4] for more details. However, we note that besides determining completion, leaders of a scheduling component are usually responsible for scheduling consumers of all subgoals that they lead to consume their answers.

3 The Anatomy of CHAT

We describe the actions of CHAT by means of an example. Consider the following state of a WAM-based abstract machine for tabled evaluation. A generator G has already been encountered and a *generator choice point* has been created for it immediately below a (Prolog) choice point P_0; then execution continued with some other non-tabled code (P and all choice points shown by dots in Figure 1). Eventually a consumer C was encountered and let us, without loss of generality, assume that G is its generator and G is not completed.[1] Thus, a *consumer choice point* is created for C; see Figure 1. The heap and the trail are shown segmented according to the values saved in the corresponding fields of choice points. The same segmentation is not shown for the environment stack as it is a spaghetti stack; however the EB values of choice points are also shown by pointers.

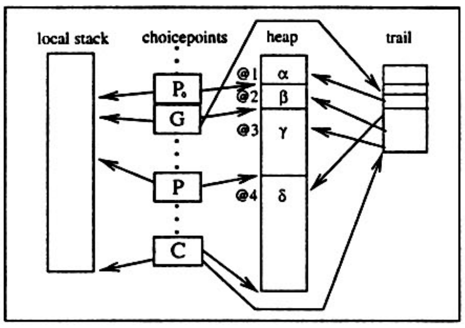


Fig. 1. CHAT stacks immediately upon laying down a consumer choice point.

Without loss of generality, let us assume that C is the only consumer. The whole issue is how to preserve the execution environment of C. CAT does this very simply through (selectively and incrementally) copying all necessary information in a separately allocated memory area — see [2]. The SLG-WAM employs *freeze registers* and freezes the stacks at their current top; allocation of new information occurs below these freeze points — see [4]. We next describe what CHAT does.

3.1 Freezing the heap without a heap freeze register

As mentioned, we want to prevent that on backtracking to a choice point P that lies between the consumer C and the nearest generator G (included), $\mathbf{H}$ is reset to the $\mathbf{B}_P[\mathrm{H}]$ as it was on creating P. However, the WAM sets:

$\mathbf{H} := \mathbf{B}_P[\mathrm{H}]$

upon backtracking to a choice point pointed to by $\mathbf{B}_P$. We can achieve that no heap lower than $\mathbf{B}_C[\mathrm{H}]$ is reclaimed on backtracking to P, by manipulating its $\mathbf{B}_P[\mathrm{H}]$ field, i.e. by setting:

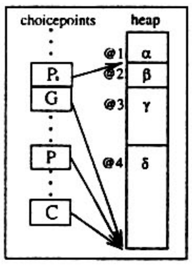
$\mathbf{B}_P[\mathrm{H}] := \mathbf{B}_C[\mathrm{H}]$

at the moment of backtracking out of the consumer. Note that rather than waiting for execution to backtrack out of the consumer choice point, this can happen immediately upon encountering the consumer (see also [4] on why this is correct).

[1] Otherwise, if G is completed, the whole issue is trivial as a *completed table optimization* can be performed and execution proceeds as in Prolog; see [4].

More precisely, upon creating a consumer point for a consumer C the action of CHAT is:

```
for all choice points $P$ between $C$ and its generator (included)
    $\mathbf{B}_P[\mathrm{H}] := \mathbf{B}_C[\mathrm{H}]$
```



The picture on the right shows which H fields of choice points are adapted by CHAT in our running example.
To see why this action of CHAT is correct, compare it with how the SLG-WAM freezes the heap using the freeze register **HF**:

```
when a consumer is encountered, the SLG-WAM sets $\mathbf{HF} := \mathrm{B}_C[\mathrm{H}]$
on backtracking to a choice point $P$, the SLG-WAM resets $\mathbf{H}$ as follows:
    if older($\mathbf{B}_P$[H],$\mathbf{HF}$) then $\mathbf{H} :=$ $\mathbf{HF}$ else $\mathbf{H} := \mathbf{B}_P$[H]
```

In this way, CHAT neither needs the freeze register **HF** of the SLG-WAM, nor uses copying for part of the heap as CAT.

The cost of setting the **B**[H] fields by CHAT is linear in the number of choice points between the consumer and the generator up to which it is performed. In principle this is unbounded, so the act of freezing in CHAT can be arbitrarily more costly than in SLG-WAM. However, our experience with CHAT is that this is not a problem in practice; see the experimental results of Section 7.

3.2 Freezing the local stack without EF

The above mechanism can also be used for the top of the local stack. Similar to what happens for the H fields, CHAT sets the EB fields in affected choice points to $\mathbf{B}_C$[EB]. In other words, the action of CHAT is:

```
for all choice points $P$ between the consumer $C$ and its generator (included)
    $\mathbf{B}_P[\mathrm{EB}] := \mathbf{B}_C[\mathrm{EB}]$
```

The top of the local stack can now be computed at any moment as in the WAM:

```
if older(B[EB],E) then E+length(environment) else B[EB]
```

and no change to the underlying WAM is needed.

Again, we look at how the SLG-WAM employs a freeze register **EF** to achieve freezing of the local stack: **EF** is set to **EB** on freezing a consumer. Whenever the first free entry on the local stack is needed, e.g. on backtracking to a choice point **B**, this entry is determined as follows:

```
if older(B[EB],EF) then EF else B[EB]
```

The code for the allocate instruction is slightly more complicated as a three-way comparison between **B**[EB], **EF** and **E** is needed.

It is worth noting at this point that this schema requires a small change to the retry instruction in the original three stack WAM, i.e. when choice points and environments are allocated on the same stack. The usual code (on backtracking to a choice point **B**) can set **EB** := **B** while in CHAT this must become **EB** := B[EB].

As far as the complexity of this scheme of preserving environments is concerned, the same argument as in Section 3.1 for the heap applies. In the sequel we will refer to CHAT's technique of freezing a WAM stack without the use of freeze registers as *CHAT freeze*.

3.3 The choice point stack and the trail

CHAT borrows the mechanisms for dealing with the choice point stack and the trail from CAT: from the choice point stack, CAT copies only the consumer choice point. The reason is that at the moment that the consumer C is scheduled to consume its answers, all the Prolog choice points (as well as possibly some generator choice points) will have exhausted their alternatives, and will have become redundant. This means that when a consumer choice point is reinstalled, this can happen immediately below a scheduling generator which is usually the leader of a scheduling component (see [2] for a more detailed justification why this is so). CHAT does exactly the same thing: it copies in what we call a *CHAT area* the consumer choice point. This copy is reinstalled whenever the consumer needs to consume more answers.

Also for the trail, CHAT is similar to CAT: the part of the trail between the consumer and the generator is copied, together with the values the trail entries point to. However, as also the heap and local stack are copied by CAT, CAT can make a selective copy of the trail, while CHAT must copy all of the trail between the consumer and the generator. This amounts to reconstructing the forward trail of the SLG-WAM (without back-pointers) for part of the computation.

For a single consumer, the cost of reconstructing the forward trail (only partly) is not greater (in complexity) than what the SLG-WAM has incurred while maintaining the forward trail. Figure 2 shows the state of CHAT immediately after creating the consumer and doing all the actions described above; the shaded parts of the stacks show exactly the information that is copied by CHAT.

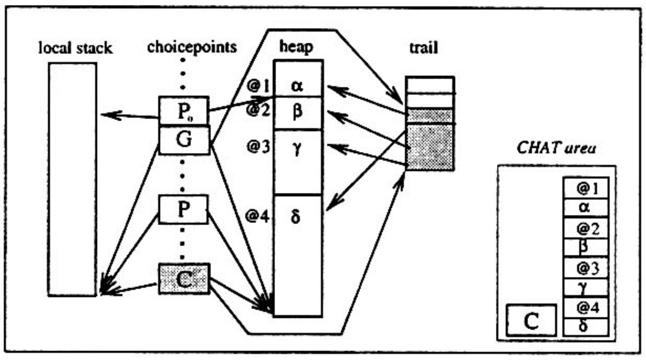


Fig. 2. Stacks and CHAT area after making the CHAT copy and adapting the choice points.

3.4 More consumers and change of leader: a more incremental CHAT

The situation with more consumers, as far as freezing the heap and local stack goes, is no different from that described above. Any time a new consumer is encountered, the **B**[EB] and **B**[H] fields of choice points **B** between the new consumer and its generator are adapted. Note that the same choice point can be adapted several times, and that the adapted fields can only point lower in the corresponding stacks. From now on, we will drop the assumption that there is only one consumer.

It is also worth considering explicitly a coup: a change of leaders. Note that as far as the heap and local stack is concerned, nothing special needs to be done if each consumer performs CHAT freeze till its current leader at the time of its creation. For the trail, a similar mechanism as for CAT applies: an incremental part of the trail between the former and the new leader needs to be copied. In [2] it is shown that this need not be done immediately at the moment of the coup, but can be postponed until backtracking happens over a former leader so that the incremental copy can be easily shared between many consumers. It also leads directly to the same incremental copying principle as in CAT: each consumer needs only to copy trail up to the nearest generator and update this copy when backtracking over a non-leader generator occurs.

The incrementality of copying parts of the trail, also applies to the change of the EB and H fields in choice points: instead of adapting choice points up to the leader, one can do it up to the nearest generator. In this scheme, if backtracking happens over a non-leader generator, then its EB and H fields have to be propagated to all the choice points up to the next generator. Our current implementation employs incremental copying of the trail and non-incremental adaptation of the choice points.

3.5 Reinstalling a consumer

As in CAT, CHAT can reinstall a consumer C by copying the saved consumer choice point just below the choice point of a scheduling generator G. Let this copy happen at a point identified as $\mathbf{B}_C$ in the choice point stack. The CHAT trail is reinstalled also exactly as in CAT by copying it from the CHAT area to the trail stack. There remains the installation of the correct top of heap and local stack registers: since the moment C was first copied, it is possible that more consumers were frozen, and that these consumers are still suspended (i.e. their generators are not complete) when C is reinstalled. It means that C must protect also the heap of the other consumers. This is achieved by installing in $\mathbf{B}_C$ the EB and H fields of G at the moment of reinstallation. This will lead to correctly protecting the heap, as G cannot be older than the leader of the still suspended consumers and G was in the active computation when the other consumers were frozen. Figure 3 gives a rough idea of a consumer's reinstallation; shaded parts of the stacks show the copied information.

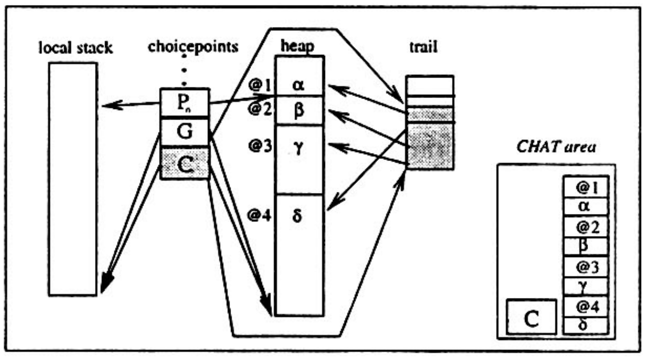


Fig. 3. Memory areas upon reinstalling the CHAT area for a consumer C.

3.6 Releasing frozen space and the CHAT areas upon completion

The generator choice point of a leader is popped only at completion of its component. At that moment, the CHAT areas of the consumers that were led by this leader can be freed: this mechanism is again exactly the same as in CAT. Also, there are no more program clauses to execute for the completed leader and backtracking occurs to the previous choice point, say P_0.[2] P_0 contains the correct top of local stack and heap in its EB and H fields: these fields could have been updated in the past by CHAT or not. In either case they indicate the correct top of heap and local stack.

The SLG-WAM achieves this space reclamation at completion of a leader by resetting the freeze registers from the values saved in the leader choice point. Indeed, the SLG-WAM saves **HF**, **EF**, etc. in all generator choice points; see [4].

4 Best and Worst Cases

As noted in [2], a worst case for CAT can be constructed by making CAT copy and reinstall arbitrary often, arbitrary large amounts of heap to (and from) the CAT area. Since CHAT does not copy the heap, this same worst case does not apply. Still, CHAT can be made to behave arbitrarily worse than SLG-WAM. We also show an example in which the SLG-WAM uses arbitrary more space than CHAT.

4.1 The worst case for CHAT

There are two ways in which CHAT can be worse than SLG-WAM:

1. every time a consumer is saved, the choice point stack between the consumer and the leader is traversed; such an action is clearly not present in SLG-WAM neither CAT

[2] If there is no previous choice point, the computation is finished.

2. trail chunks are copied by CHAT for each save of a consumer; the inefficiency lies in the fact that consumers in the SLG-WAM can share a part of the trail even strictly between the consumer and the nearest generator; this is a direct consequence of the forward trail with back pointers; both space and time complexity are affected. Note that the same source of inefficiency is present in CAT.

The following example program shows both effects. The subscripts g and c denote the occurrence of a subgoal that is a generator or consumer for p(_).

```
query(Choices,Consumers) :-
    p_g(_),
    make_choices(Choices,_),
    make_consumers(Consumers,[]).
make_choices(N,trail) :-
    N > 0, M is N - 1, make_choices(M,_).
make_choices(0,_).
make_consumers(N,Acc) :-
    N > 0, M is N - 1,
    p_c(_), make_consumers(M,[a|Acc]).
:- table p/1.
p(1).
```

Predicate make_choices/2 is supposed to create choice points; if the compiler is sophisticated enough to recognize that it is indeed deterministic, a more complicated predicate with the same functionality can be used. The reason for giving the extra argument to make_consumers is to make sure that on every creation of a consumer, H has a different value and an update of the H field of choice points between the new consumer and the generator is needed — otherwise, an obvious optimization of CHAT would be applicable. The query is e.g. ?- query(100,200). CHAT uses $(Choices * Consumers)$ times more space and time than SLG-WAM for this program. If the binding with the atom trail were not present in the above program, CHAT would also use $(Choices * Consumers)$ times more space and time than CAT.

At first sight, this seems to contradict the statement that CHAT is a better CAT. However, since for CHAT the added complexity is only related to the trail and choice points, the chances for running into this in reality are lower than for CAT.

4.2 A best case for CHAT

The best case space-wise for CHAT compared to SLG-WAM happens when lots of non tabled choice points get trapped under a consumer: in CHAT, they can be reclaimed, while in SLG-WAM they are frozen and retained till completion. The following program shows this:

```
query(Choices,Consumers) :-
    p_g(_), create(Choices,Consumers), fail.
create(Choices,Consumers) :- Consumers > 0,
    ( make_choicepoints(Choices), p_c(Y), Y = 2
    ; C is Consumers - 1, create(Choices,C) ).
make_choicepoints(C) :-
    C > 0, C1 is C - 1, make_choicepoints(C1).
make_choicepoints(0).
:- table p/1.
p(1).
```

When called with e.g. ?- **query(25,77)**. the maximal choice point usage of SLG-WAM contains at least $25 * 77$ Prolog choice points plus 77 consumer choice points; while CHAT's maximal choice point usage is 25 Prolog choice points (and 77 consumer choice points reside in the CHAT areas). Time-wise, the complexity of this program is the same for CHAT and SLG-WAM.

One should not exaggerate the impact of the best and worst cases of CHAT: in practice, such contrived programs rarely occur and probably can be rewritten so that the bad behaviour is avoided.

5 A Plethora of Implementations

After SLG-WAM and CAT, CHAT offers a third alternative for implementing the suspend/resume mechanism that tabled execution needs. It shares with CAT the characteristic that Prolog execution is not affected and with SLG-WAM the high sharing of execution environments of suspended computations. On the other hand, CHAT is not really a mixture of CAT and SLG-WAM: CHAT copies the trail in a different way from CAT and CHAT freezes the stacks differently from SLG-WAM, namely with the CHAT freeze technique. CHAT freeze can be achieved for the heap and local stack only. Getting rid of the freeze registers for the trail and choice point stacks can only be achieved by means of copying; the next section elaborates on this.

Thus, it seems there are three alternatives for the heap (SLG-WAM freeze, CHAT freeze and CAT copy) and likewise for the local stack, while there are two alternatives for both choice point and trail stack (SLG-WAM freeze and CAT copy). The decisions on which mechanism to use for each of the four WAM stacks are independent. It means there are at least 36 possible implementations of the suspend/resume mechanism which is required for tabling !

It also means that one can achieve a CHAT implementation starting from the SLG-WAM as implemented in XSB, get rid of the freeze registers for the heap and the local stack, and then introduce copying of the consumer choice point and the trail. This was our first attempt: the crucial issue was that before making a complete implementation of CHAT, we wanted to have some empirical evidence that CHAT freeze for heap and local stack was correct. As soon as we were convinced of that, we implemented CHAT by partly recycling the CAT implementation of [2] which is also based on XSB as follows:

- replacing the selective trail copy of CAT with a full trail copy of the part between consumer and the closest generator
- not copying the heap and local stack to the CAT area while introducing the CHAT freeze for these stacks; this required a small piece of code that changes the H and EB entries in the affected choice points at CHAT area creation time and consumer reinstallation

It might have been nice to explore all 36 possibilities, with two or more scheduling strategies and different sets of benchmarks but unlike cats, we do not have nine lives.

6 More Insight

One can wonder why CHAT can achieve easily (i.e. without changing the WAM) the freezing of the heap and the environment stack (just by changing two fields in some choice points) but the trail has to be copied and reconstructed. There are several ways to see why this is so. In WAM, the environments are already linked by back-pointers, while trail entries (or better trail entry chunks) are not. Note that SLG-WAM does link its trail entries by back-pointers; see [4]. Another aspect of this issue is also typical to an implementation which uses untrailing (instead of copying) for backtracking (or more precisely for restoring the state of the abstract machine): it is essential that trail entry chunks are delimited by choice points; this is not at all necessary for heap segments. Finally, one can also say that CHAT avoids the freeze registers by installing their value in the affected choice points: The WAM will continue to work correctly, if the H fields in some choice points are made to point lower in the heap. The effect is just less reclamation of heap on backtracking. Similarly for the local stack. On the other hand, the TR fields in choice points cannot be changed without corrupting backtracking.

7 Tests

All measurements were conducted on an Ultra Sparc 2 (168 MHz) under Solaris 2.5.1. Times are reported in seconds, space in KBytes.[3] Space numbers measure the maximum use of the stacks (for SLG-WAM) and the total of max. stack + max. C(H)AT area (for C(H)AT). The benchmark set is exactly the same as in [2] where more information about the characteristics of the benchmarks and the impact of the scheduling can be found.

[3] While writing this paper, we are finding on an almost daily basis new opportunities for better memory reclamation in XSB's implementation of the SLG-WAM; this affects also CAT's and to a lesser extent CHAT's implementation; therefore, the space figures are bound to improve.

7.1 A benchmark set dominated by tabled execution

Tables 1 and 2 show the time and space performance of SLG-WAM, CHAT and CAT for the batched (indicated by B in the tables) and local scheduling strategy (indicated by L). The benchmark set is dominated by tabled execution, i.e. minimal Prolog execution is going on.

	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o
SLG-WAM(B)	0.23	0.45	0.13	0.17	0.15	0.44	0.12	0.58
CHAT(B)	0.21	0.42	0.13	0.15	0.15	0.46	0.14	0.73
CAT(B)	0.22	0.41	0.13	0.15	0.14	0.50	0.15	0.92
SLG-WAM(L)	0.23	0.43	0.13	0.16	0.16	0.42	0.12	0.61
CHAT(L)	0.22	0.42	0.12	0.15	0.14	0.40	0.11	0.53
CAT(L)	0.22	0.42	0.12	0.15	0.14	0.40	0.11	0.55

Table 1. Time performance of SLG-WAM, CAT & CHAT under batched & local scheduling.

For the local scheduling strategy, CAT and CHAT are the same time-wise and systematically better than SLG-WAM. Under the batched scheduling strategy, the situation is less clear, but CHAT is never worse than the other two. Taking into account the uncertainty of the timings, it is fair to say that except for read_o all three implementation schemes perform the same time-wise in this benchmark set.

	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o
SLG-WAM(B)	9.7	11.4	8.8	20.6	40	317	119	512
CHAT(B)	9.6	11.6	8.4	24.7	35.1	770	276	1080
CAT(B)	13.6	19.4	11.7	45.3	84	3836	1531	5225
SLG-WAM(L)	6.7	7.6	5.8	17.2	13.3	19	15.8	93
CHAT(L)	5.8	7.2	5.6	19	8.2	16	13.2	101
CAT(L)	7.9	10.7	7.1	29.5	12.5	17	23.5	246

Table 2. Space performance of SLG-WAM, CAT & CHAT under batched & local scheduling.

Space-wise, CHAT wins always from CAT and 6 out of 8 times from SLG-WAM (using local scheduling). However, as noted before, the space figures should be taken *cum grano salis.*

7.2 A more realistic mix of tabled and Prolog execution

The next set of programs is more balanced, i.e. 75–80% of the execution concerns Prolog code. We consider this mix a more "typical" use of tabling. We note at this point that CHAT (and CAT) have faster Prolog execution than SLG-WAM by around 10% according to the measurements of [4] — this is the overhead that the SLG-WAM incurs on the WAM. In the following tables all figures are for the local scheduling strategy; batched scheduling does not make sense for this set of benchmarks — see [2] on why this is so.

	akl	color	bid	deriv	read	browse	serial	rdtok	boyer	plan	peep
SLG-WAM	1.48	0.67	1.11	2.56	9.64	32.6	1.17	3.07	10.02	7.61	9.01
CHAT	1.25	0.62	1.03	2.54	9.73	32	0.84	2.76	10.17	6.14	8.65
CAT	1.24	0.62	0.97	2.50	9.56	32.2	0.83	2.75	9.96	6.38	8.54

Table 3. Time Performance of SLG-WAM, CHAT & CAT.

Table 3 shows that CAT wins on average over the other two. CHAT comes second.

	akl	color	bid	deriv	read	browse	serial	rdtok	boyer	plan	peep
SLG-WAM	998	516	530	472	5186	9517	279	1131	2050	1456	1784
CHAT	433	204	198	311	4119	7806	213	746	819	963	1187
CAT	552	223	206	486	8302	7847	227	821	1409	1168	1373

Table 4. Space Performance (in KBytes) of SLG-WAM, CHAT & CAT.

Space-wise, CHAT wins from both SLG-WAM and CAT in all benchmarks. It has lower trail and choice point stack consumption than SLG-WAM and saves considerably less information than CAT in its copy area.

8 Conclusion

CHAT offers one more alternative to the implementation of the suspend/resume mechanism that tabling requires. Its main advantage over SLG-WAM's approach is that no freeze registers are needed and in fact no complicated changes to the WAM. As with CAT, the adoption of CHAT as a way to introduce tabling to an existing logic programming system does not affect the underlying abstract machine and the programmer can still rely on the full speed of the system for non-tabled parts of the computation. Its main advantage over CAT is that CHAT's memory consumption is lower and much more controlled. The empirical results show that CHAT behaves quite well and CHAT is a better candidate for replacing SLG-WAM (as far as the control goes) than CAT. CHAT also offers the same advantages as CAT as far as flexible scheduling strategies goes.

Acknowledgements

The research of the second author is supported by a junior scientist fellowship from the K.U. Leuven Research Council and CCR-9702681.

References

1. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, Massachusetts, 1991. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.
2. B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In H. Glaser, K. Meinke, and C. Palamidessi, editors, *Proceedings of the Joint Conference on PLILP/ALP'98*, number 1490 in LNCS, pages 21–35, Pisa, Italy, Sept. 1998. Springer-Verlag.
3. B. Demoen and K. Sagonas. Memory Management for Prolog with Tabling. In *Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management*, Vancouver, Canada, Oct. 1998. ACM Press.
4. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20, 1998. To appear.
5. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.

An Abstract Machine for Fast Parsing of Typed Feature Structure Grammars

John C. Brown and Suresh Manandhar

Department of Computer Science, University of York, York, YO1 5DD, UK,
{johnb,suresh}@cs.york.ac.uk,
Tel.(Fax.) +44 1904 432757 (432767)

Abstract. Constraint-based grammars based on Head Driven Phrase Structure Grammar (HPSG) employ typed feature structures as the representation language. Although the unification operation over typed feature structures can be implemented in Prolog, this is significantly inefficient. A recent abstract machine [18] for typed feature structures translates grammatical descriptions into abstract code sequences, reducing parse times by a factor of 11 to around 1 second for 12 words as compared to an earlier Prolog parser.
We present a new method for compiling typed feature structure grammars that can bring significant improvement in parsing times. A Prolog prototype of our abstract machine described in this paper already parses faster than the existing abstract machine. Our approach is based on creating composite types which are a combination of bitmaps and pointers. Composite types are created for lexical and phrasal categories. Precompiled tables reduce speculative unification, and built-in predicates achieve further speed-ups by more efficient array representations of tables and phrases. Applicability to constraint programming in general is discussed.
Keywords: Constraints, types, feature structures, parsing, HPSG, ALE, abstract machine.

1 Introduction

Modern grammars for natural languages based on the *Head-driven Phrase Structure Grammar* (HPSG) [1] [2] employ *typed-feature structures* [3] as their description language. *Feature structures* such as the one in Figure 1 are employed for describing linguistic objects. Feature structures consist of a conjunction of *feature-value* pairs where *features* are atomic symbols and *values* can be either atomic or a feature structure. Feature structures can be thought of as the analog of Prolog terms used in linguistic descriptions, and were popularised by unification grammar implementations such as PATR [4][5]. Features can be thought of as indexes to argument positions of a complex term.

Typed feature structures (TFS) are complex terms with functors organised into a type hierarchy: this makes unification more complex than for Prolog terms. In this paper, by a TFS we mean a well-formed formula in a typed-feature logic. Typed feature logic can be viewed as a constraint language that extends feature logic [6] by providing types (as in [7]) and appropriateness constraints [3].

1.1 TFS Grammars

A typed feature structure grammar is best understood as an extension of the familiar context free grammar (CFG) where terminal and non-terminal symbols are replaced by TFSs. We assume that a TFS is given by the following definition:

$$\begin{array}{lll} \langle TFS \rangle & \rightarrow & \langle FS \rangle \mid \langle type \rangle \& \langle FS \rangle \mid \langle type \rangle \\ \langle FS \rangle & \rightarrow & \langle path \rangle : \langle value \rangle \mid \langle FS \rangle \& \langle FS \rangle \end{array}$$

where

$$\begin{array}{lll} \langle value \rangle & \rightarrow & \langle TFS \rangle \mid \langle constant \rangle \mid \langle variable \rangle \\ \langle path \rangle & \rightarrow & \langle feature \rangle \mid \langle feature \rangle : \langle path \rangle \end{array}$$

Figure 1 shows the TFS for the category of a verb in an HPSG grammar for English [8] [9], in attribute value matrix (AVM) notation. Types and feature names are in lower and upper-case respectively. In the example shown in figure 1 the values of SUBJ and COMPS are each a list of structures of type *synsem*. Appropriateness [3] is employed to restrict the types that a feature can take, and is defined by a function *Approp:* $F \times T \rightarrow T$ where F denotes the set of feature symbols and T denotes the set of types T.

A feature appropriate to any type is also appropriate to all its sub-types. For example, in the type hierarchy at the bottom left of Figure 4, tb is the appropriate type of f1 which is appropriate to t2, t5 and t10. ALE uses the restriction that each feature should uniquely determine the most general type for which it is appropriate. Where all features appropriate to a type are supplied, the TFS is said to be ***totally well-typed***, as in Figure 3 but not 6, where comparison with Figure 5 shows additional features for the value of index.

A typed feature formalism, such as ALE, has a type inference system, a TFS definite clause theorem prover, and a bottom-up chart parser.

1.2 Bottom-up chart parsing

The bottom left of Figure 7 shows a phrase structure analysis of the sentence "Peter likes Paul" using the grammar rules and lexical entries as given below:

$$\begin{array}{lll} vp \rightarrow v\ np & s \rightarrow np\ vp & \\ np \rightarrow \text{“Peter”} & np \rightarrow \text{“Paul”} & v \rightarrow \text{“likes”} \end{array}$$

In this example np and v are ***lexical categories*** or ***terminal symbols*** and vp and s are ***phrasal categories*** or ***non-terminal symbols.***

The nodes in the phrase structure analysis are known as *constituents* and the immediate children of a given constituent will be referred to as its *sub-constituents.* The sub-string dominated by a constituent is called a *phrase*, and the structure of that phrase is given in the tree-structure rooted in the constituent. A *parser* constructs the phrase structure analysis of a sentence. This paper will refer to ***bottom-up parsing*** where an analysis is constructed, a phrase at a time, upwards from the words. In *chart parsing,* illustrated in Figure 2, completed constituents are entered into a table or chart, which indexes them by the start and end positions of their associated strings, so in this case the chart will contain:

0 np 1   1 v 2   2 np 3   1 vp 3   0 s 3

Chart parsers are treated in Prolog in [10][11] and in pseudocode in [12].

$$\begin{bmatrix}
\text{SYNSEM synsem} & \begin{bmatrix}
\text{LOCAL local} & \begin{bmatrix}
\text{CAT cat} & \begin{bmatrix}
\text{HEAD verb} & \begin{bmatrix} \text{PRD boolean} \\ \text{MOD none} \\ \text{VFORM bse} \\ \text{AUX minus} \\ \text{INV minus} \end{bmatrix} \\
\text{SPR ne\_list\_synsem} & \begin{bmatrix} \text{LOC loc} & \begin{bmatrix} \text{CAT cat} & \begin{bmatrix} \text{MARKING comp} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\
\text{SUBJ ne\_list\_synsem} & \begin{bmatrix} \text{LOC loc} & \begin{bmatrix} \text{CAT cat} & \begin{bmatrix} \text{HEAD noun} \begin{bmatrix} \text{CASE nom} \end{bmatrix} \\ \text{SPR e\_list} \\ \text{SUBJ e\_list} \\ \text{COMPS e\_list} \\ \text{MARKING unmarked} \end{bmatrix} \\ \text{CONTENT nom\_obj} & \begin{bmatrix} \text{INDEX } \textit{Ind1} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\
\text{COMPS ne\_list\_synsem} & \begin{bmatrix} \text{LOC loc} & \begin{bmatrix} \text{CAT cat} & \begin{bmatrix} \text{HEAD noun} \begin{bmatrix} \text{CASE acc} \end{bmatrix} \\ \text{SPR e\_list} \\ \text{SUBJ e\_list} \\ \text{COMPS e\_list} \\ \text{MARKING unmarked} \end{bmatrix} \\ \text{CONTENT nom\_obj} & \begin{bmatrix} \text{INDEX } \textit{Ind2} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\
\text{MARKING unmarked} &
\end{bmatrix} \\
\text{CONTENT psoa} & \begin{bmatrix} \text{NUCLEUS like} & \begin{bmatrix} \text{LIKER } \textit{Ind1} \\ \text{LIKEE } \textit{Ind2} \end{bmatrix} \\ \text{QUANTS e\_list} & \end{bmatrix}
\end{bmatrix} \\
\text{NONLOCAL nonlocal} & \begin{bmatrix} \text{INHERITED nonloc1} \begin{bmatrix} \text{SLASH e\_set} \end{bmatrix} \\ \text{TO\_BIND nonloc1} \begin{bmatrix} \text{SLASH e\_set} \end{bmatrix} \end{bmatrix}
\end{bmatrix} \\
\text{word} &
\end{bmatrix}$$

Fig. 1. The verb "like" in HPSG, shown as an attribute value matrix

17

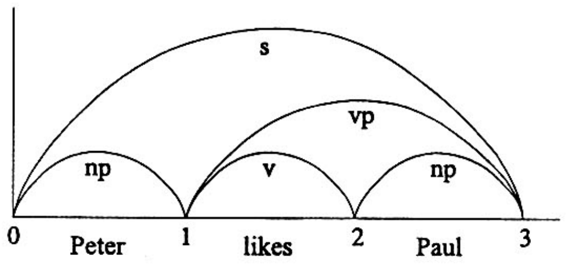


Fig. 2. The conventional view of a chart in the parse of a simple sentence

2 Existing Abstract Machine Techniques

Use of the chart or "well-formed sub-string table" avoids multiple computation of the same constituent arising from ambiguities in natural language. A sub-constituent in the chart can have multiple parent constituents in a forest of phrase structure trees, reflecting its combination with alternative phrases making differing feature bindings. An efficient solution, by J.A.Carroll, [12] that minimises copying for unification grammars and appears extensible to TFS grammars, is to keep these bindings in the parent, and copy them temporarily into the sub-constituent during retrieval.

The ALE environment [16] makes a complete copy of the TFS of a new constituent including its components in the sub-constituents: the computational cost is considerable. For the constituents in the chart to be globally addressable which a bottom-up algorithm requires, in Prolog they must be represented by dynamically asserted clauses. Many unifications fail, so it is desirable to delay copying until a rule has successfully unified with a sub-constituent. This means making bindings in the sub-constituent and then undoing them on failure or after copying. However, the WAM code [14] [15] or interpreted Prolog representing an asserted clause cannot be modified. One CFG parser in Prolog [17] passed the chart as an argument in a recursive algorithm: extended to TFS unification this would allow bindings in a sub-constituent without copying, but removing these on backtracking would also remove any new edge copied to the chart. A clause cannot be partially invoked either, so when a rule unifies with an asserted clause, a complete copy of its TFS is made on the heap, even if unification fails early in the structure.

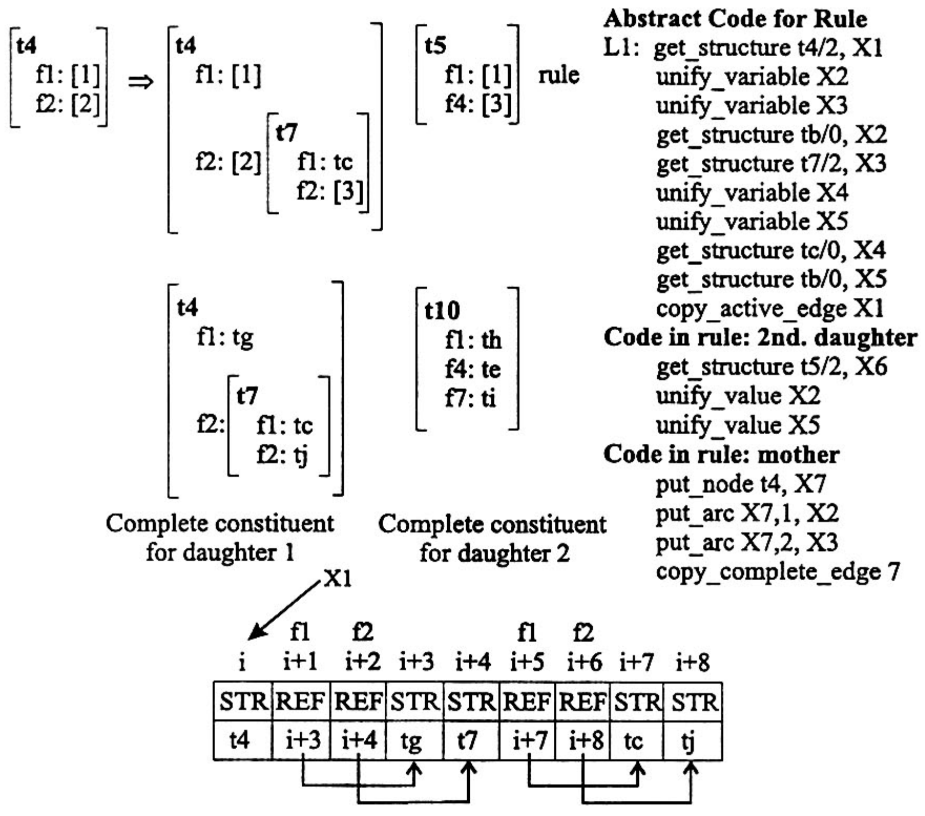
These problems can be easily overcome in an abstract machine, such as Amalia [18], illustrated in Figures 3 and 4. This parses a subset of ALE 10 times faster, although a faster Prolog version, ALE 3.0[1] is in beta release.

2.1 The Amalia Abstract Machine for ALE

Figure 3 shows the abstract code generated by Amalia for a single TFS grammar rule (also shown in 3). Before any abstract code is generated, consistency of every

[1] Details of ALE 3.0 on **http://www.sfs.nphil.uni-tuebingen.de/~gpenn/ale.html**

TFS in a rule is checked against inheritance and appropriateness declarations. This process completes partial structures by adding omitted types and features. Unification is more complex than in Prolog. Two TFSs also unify if they both have a common join (as with *tg* and *th*, *t5* and *t10* in Figure 4).


t4
f1: [1]
f2: [2]
⇒
t4
f1: [1]
f2: [2]
t7
f1: tc
f2: [3]
t5
f1: [1]
f4: [3]
rule
t4
f1: tg
f2:
t7
f1: tc
f2: tj
t10
f1: th
f4: te
f7: ti
Complete constituent for daughter 1
Complete constituent for daughter 2
X1
f1 f2 f1 f2
i i+1 i+2 i+3 i+4 i+5 i+6 i+7 i+8
STR REF REF STR STR REF REF STR STR
t4 i+3 i+4 tg t7 i+7 i+8 tc tj

Abstract Code for Rule

```
L1: get_structure t4/2, X1
    unify_variable X2
    unify_variable X3
    get_structure tb/0, X2
    get_structure t7/2, X3
    unify_variable X4
    unify_variable X5
    get_structure tc/0, X4
    get_structure tb/0, X5
    copy_active_edge X1
```

Code in rule: 2nd. daughter

```
    get_structure t5/2, X6
    unify_value X2
    unify_value X5
```

Code in rule: mother

```
    put_node t4, X7
    put_arc X7,1, X2
    put_arc X7,2, X3
    copy_complete_edge 7
```

Fig. 3. A rule with constituents, and their underlying abstract code and heap representations in the Amalia abstract machine for ALE

For simplicity in coding, Amalia stores all arguments (*i.e.* the value of appropriate features) of a TFS structure separately on the heap, referenced from argument positions inside the structure (for example, STR node i+3 in Figure 3), in contrast to WAM that does this only for non-atomic arguments. Registers are used to address these arguments in both cases. On the first occurrence of a co-indexed variable in a rule, *unify_variable* sets a register referencing the appropriate REF node. On subsequent occurrences *unify_value* initiates full recursive unification of two dereferenced structures. Trailing of bindings in Amalia involves the original value so this can be restored. This contrasts with the WAM where variables are simply unbound.

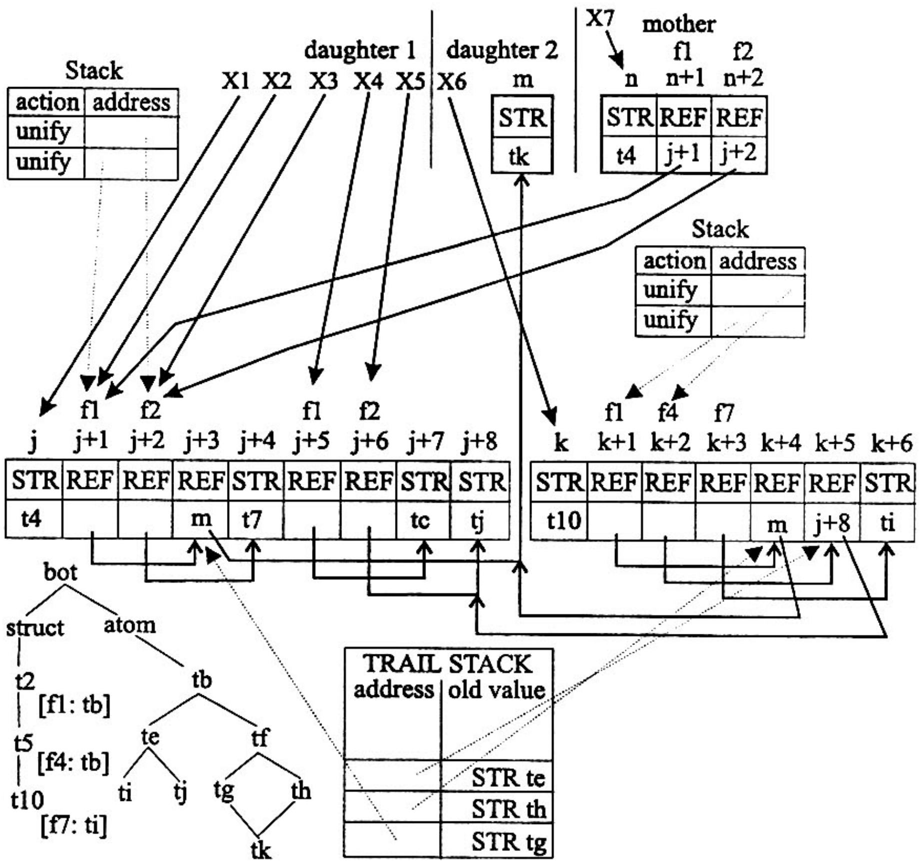


Fig. 4. The heap after the application of the rule of Figure 2, showing the stacks during its application

The *get_structure* instruction differs from that in the WAM in that it must cope with possibly differing numbers of features in the rule and sub-constituent. It addresses a table of procedures using two types: one from the instruction, arising from the rule, and one found in the sub-constituent by dereferencing the register appearing in the instruction. Abstract code instructions in the procedure push references onto a stack, as shown in 4, and each one is popped off the stack by one of the following *unify_** instructions. Where the type in the sub-constituent is equal to (e.g. t4), or is subsumed (e.g. t10) by the type in the rule, the references are to all, or to some of the features respectively in the sub-constituent. By this means, the *unify_** instructions in the rule always reference the correct features, whatever the type in the sub-constituent.

Abstract instructions corresponding to the mother (the LHS of a rule) build a structure on the heap, with features referencing nodes in the sub-constituents, this co-indexing being achieved using registers. The final operation is to copy the structure that is rooted in the mother, after which the bindings in the sub-constituents are undone using the trail stack.

Significant overheads in Amalia include building (invoked by *get_structure*) and reducing the stacks, as well as fetching and decoding a considerable number of abstract instructions, including those within the procedure addressed by *get_structure*. The final copying operation (*e.g copy_complete_edge 7*) is considerable, since for example the lexical entry of Figure 1 contains 53 nodes (STR cells) and 54 incoming arcs (REF cells), occupying 428 bytes, where structures labelled with Ind1 or Ind have features number, gender and person. In 6 out of 7 of the HPSG rules in [9], a *synsem* structure in the first or second daughter, as in *subj* and *comps* in Figure 1, ensures agreement with another sub-constituent. Co-indexing demands a path as deep as this structure, involving many abstract instructions to traverse it, and more in unifying the structure. Following success, all of the TFS less this structure is expensively copied into the mother: for a verb this occurs twice in forming the sentence.

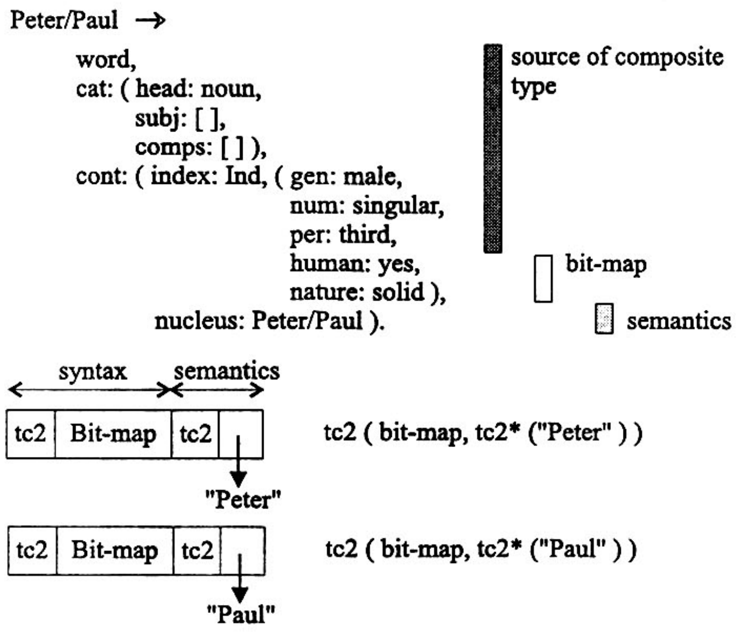
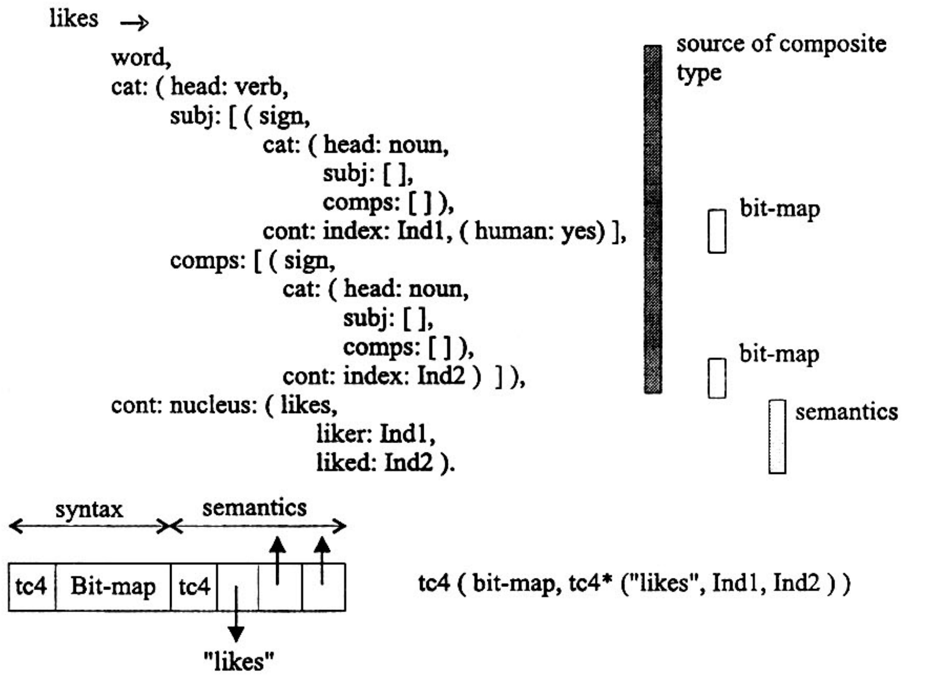


Fig. 5. Two noun lexical entries in simplified HPSG, shown in constraint and in compressed forms

3 Compressed Categories

Much of the structure in a category is invariant during parsing, a crucial observation exploited in our approach. This implies that most of these structures can be stored in a suitable data structure retrieved after parsing, avoiding multiple copying. HPSG partitions the category into syntactic structure reachable by the *synsem: local: category* path, and semantic structure reachable by *synsem: local: content*. Structure is usually discarded once unification has been checked between sub-constituents, so there is often less and never more structure in a phrase than in either of its subconstituents. Therefore unique phrasal syntactic structures number less than unique lexical structures, ignoring the orthographic string identifying the word, which does not affect combination with other categories.

Current grammars do not generally examine semantic structure to decide if a phrase is well-formed, so this does not affect the applicability of a rule to sub-constituents. An exception is the *ind* structure of a nominal. Semantic structure grows as words are added to the analysis, by combining semantic graphs originating from the lexical categories, so lexical categories must contain some skeleton representation of these graphs so the semantics of the sentence can later be retrieved. This skeleton involves a composite type to enable full semantic structure to be re-built from the *retrieval graph*, discussed in section 4, and a pointer for each node in the structure that could be co-indexed with semantic structure in another constituent. There is also a one-byte reference to the string, indirectly through an array with one entry per word in the sentence.



```
likes →
      word,
      cat: ( head: verb,
             subj: [ ( sign,
                       cat: ( head: noun,
                              subj: [ ],
                              comps: [ ] ),
                       cont: index: Ind1, ( human: yes) ],
             comps: [ ( sign,
                        cat: ( head: noun,
                               subj: [ ],
                               comps: [ ] ),
                        cont: index: Ind2 ) ] ),
      cont: nucleus: ( likes,
                       liker: Ind1,
                       liked: Ind2 ).
```

tc4 (bit-map, tc4* ("likes", Ind1, Ind2))

Fig. 6. A verb lexical entry in simplified HPSG, in constraint and compressed forms

Figures 5 and 6 show examples of compressed categories for lexical entries with their constraint expressions in a simplified HPSG. The compressed syntactic component comprises a composite type, from which the invariant syntactic structure can be retrieved, and an optional bit-map for a subset of paths not included in the type. The semantic skeleton has just been described. An equivalent Prolog representation, used in the working prototype, is shown alongside.

Current grammars allow encoding of complete syntactic structure in an acceptable number of composite types, without a bit-map. To reduce the number of types, some features would be encoded in a compressed bit-map in the category as in [19]: features usually have only 2 or 3 possible values.

In Figure 5, both noun categories have identical composite types since only the orthographic string varies, and the semantic components contain only the reference to this string. Figure 6 illustrates the verb "likes" again. The *subj* and *comps* features each specify a noun, involving a bit-map of features. The semantics has a pointer for the subject category and one for the complement category, irrespective of the complexity of the semantic structure.

4 Parsing Using Compressed Categories

Figure 7 shows a phrase structure analysis by the proposed abstract machine, with a CFG summary at bottom left. A verb phrase derives "likes Paul", and combines with the preceding noun to produce a sentence phrase deriving "Peter likes Paul". Each compressed phrase has a composite type reflecting its syntactic structure. Its semantic structure is formed by copying the semantic structure from the semantic head, here the verb, followed by a grammar rule identifier, here s3, and the semantic structure from the other sub-constituent: the appropriate pointer is set to point to this structure. Concise categories allow small self-relative pointers, and block copying rather than expensive graph traversal. Modern CPU's support fast block copying for high-speed graphics.

The figure shows a complete semantic structure being copied into the new phrase, as in Amalia. The Prolog prototype uses an alternative approach, where only the structure from the semantic head is copied, and this references semantic structure in the sub-constituents via pointers bound to edge numbers. This is possible without excessive copying since bindings are made mostly in the root node, or in semantic nodes close to the root. This second case will involve edge copying to allow for shared subtrees, which has low overheads for compact edges.

After parsing, each complete semantic structure can be re-created by access to a precompiled ***retrieval graph***, following arcs chosen by the composite type. The current prototype implements this with dynamically asserted Prolog clauses:

```
  arc(Composite_type, Arc_no, Type, Type_node_no)
  type_node(Type_node_no, Type, Arc_no_list)
```

The semantic structure in a composite type starts at a known node in the *retrieval graph*, and contains nodes that a rule will bind to nodes in structures of other types. The prototype contains a table recording these associations and during retrieval this is referenced to link together lexical semantic structures to build that of the sentence.

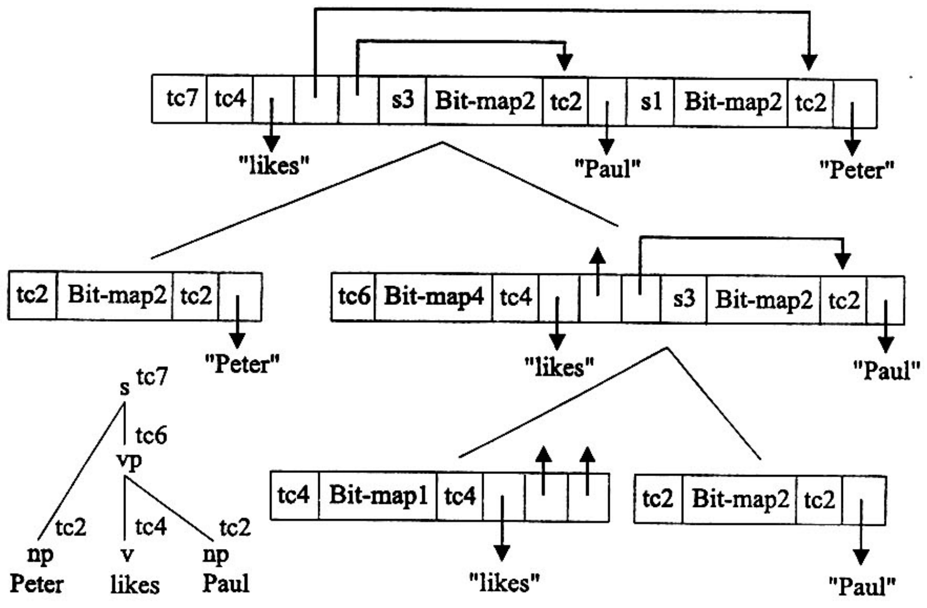


Fig. 7. Building a complete phrase structure analysis in the proposed abstract machine

4.1 Avoiding Unification During Parsing

The prototype parser developed in SWI Prolog is similar to the chart parser in [16] or [10], except that precompiled tables indicate which sequences of consecutive constituents in the chart can be combined. The following code is invoked after selecting an edge of composite type `Type1` from the chart, spanning words between `Start1` and `Right1`, where `Sem1` is its semantic structure: the edge is identified by the unique integer `Edge_no1`. An edge to combine with this is being sought by the code.

```
apply_rule(Edge_no1,Start1,Right1,Type1,Sem1,N):-
  compact_edge(Edge_no2,Right1,Right2,Type2,Sem2,_,_),
  double_tuple(Type1, Type2, Rule, Mother_type),
  update_compact_edge_no(J),
  Edges = [Edge_no1,Edge_no2],
  build_semantics([Sem1,Sem2],Rule,Edges,Sem),
  assert(compact_edge(J,Start1,Right2,Mother_type,Sem,Edges,Rule)),
```

– `compact_edge/7` references another edge in the chart, starting at `Right1` where the first edge ends: arguments resemble those in the clause head.

- `double_tuple/4` invokes clauses implementing precompiled tables, indicating if edges of `Type1` and `Type2` combine under `Rule` to make a phrase of `Mother_type`.
- `update_compact_edge_no/1` increments the edge-count in a dynamic clause.
- the next clause builds a list of edge numbers once, to be referenced later.
- `build_semantics/4` builds the semantic structure of a new edge from `Sem1` and `Sem2`.
- `assert(compact_edge/7)` stores a new edge numbered J, made by combining `Edge_no1` and `Edge_no2` under `Rule`.

Indexing on the first two arguments of `double_tuple/4` speeds access, but hash tables are completely re-addressed with both types when only the second edge is changed on backtracking. All arguments of `compact_edge/7` are instantiated although `Right1` has been checked by indexing, and only `Type2` is immediately needed. A wide-coverage grammar and an ambiguous sentence can generate hundreds of edges with the correct `Right1` that will not combine, making this a bottleneck. The less frequent assertion of edges and the edge-count are also computationally expensive.

A solution is to write new built-in predicates in imperative code, supporting parse tables structured into a tree of arrays, and the more compact edge representations of Figure 7. Overall table size for a full-scale grammar is estimated at < 1Mbyte, based on the number of combinations of linguistic feature values under each rule. Tables are small mainly because complements agree only with the verb and not each other, and phrase structures derive from a single head daughter, so tables need not treat a full cross-product of 3 categories. The hash tables generated in Prolog indexing could not exploit these restrictions.

5 Precompilation

The precompilation method should be:
- insensitive to the type signature/feature geometry employed.
- portable across different typed feature formalisms (apart from ALE).

Composite type mapping should not be onto CFG categories, or rely on specific paths governing agreement. For example, a verb phrase typically has:

synsem: local: category: (head: verb, subj: not(e_list), comps: e_list)

but such structure geometry can vary between grammars. Composite types can be integers arbitrarily assigned to distinct structures, ignoring orthographic strings and semantic structure except for *ind* in nominals, and in categories combining with nominals to make nominals. In time $O(n.m)$ a discrimination network can treat n categories of m nodes, allocating a new type to any category that differs in any respect from those already treated: a full-scale lexicon would take a few minutes with an efficient imperative implementation. In the Prolog prototype, network arcs are dynamically asserted clauses of the form:

```
arc_disc(Node_no_source, Type_destination, Node_no_destination)
```

whilst nodes appear only in the source or destination fields of the arcs. Since structures are totally well-typed, feature names can be derived and only types are stored. An arc and node are added when no destination node reachable from the current source node matches the type of the next node in the feature structure. Terminals are labelled with the composite type using dynamic clauses:

```
type_node_disc(Leaf_node_no, Composite_type)
```

Then compilation involves recursively applying each rule to every combination of categories, generating tuples like `double_tuple` of the last section. `Mother_type` is assigned using the discrimination network when a new phrasal category is generated. Compilation is a bounded task, since the discharge of *subj* and *comps* lists results in there being fewer phrasal than lexical categories.

The second objective is important so general linguistic principles of HPSG like the head-feature principle can be implemented in new ways by the linguist, rather than by the explicit invocation of definite clauses used in current ALE grammar rules. Principles could be attached to the type hierarchy and invoked when a rule treats a particular type, or be globally declared and automatically implemented in every rule [20]. Whatever the method the heap shown in Figure 4 will reflect the principle following the successful application of a grammar rule to a category sequence. Currently ALE compiles grammar rules into Prolog clauses, and further compilation in our system adds a list of arguments accessing the rule daughters, so the feature structure of each daughter on the heap can be traversed and analysed after rule application.

Each co-indexed node is tagged with a unique integer reflecting the first daughter in which it is referenced and the tag number in that daughter, for example 203 for tag 3 in daughter 2. Tag detection during TFS traversal in a later daughter or the mother determines which daughter provides the semantic structure in the mother, or the paths in other daughters that unify with pointers in that structure, like Ind1 in Figure 6. Such inter-daughter unification is clearly seen in Figure 4. Computational cost is small compared to unsuccessful rule application, and the technique is implementable in Prolog without recourse to the heap data structure at WAM level, although this would speed-up compilation.

6 Experiments and Results

Initial tests have used an abbreviated version of a small HPSG grammar [9], with six schemas, and 83 types. The type signature is realistic for a wide-coverage grammar as described in [2]. Tests were made with varying length unambiguous sentences of the form *"kim believes kim believes ... kim likes sandy"*. Parses contain all generated edges in a vine structure which is an extension of Figure 7, with 2 arcs per node. By repeatedly parsing 100 and 1000 times respectively for sentences of 31 and 5 words, on a Pentium II 266MHz machine, Table 1 was obtained. Both parsers destroyed edges using `retractall`: the shortest time was rechecked with 10,000 repetitions.

Neither parser adds null edges, the ALE code being modified to ensure this, and the lexical rules that generate slashes and the head-filler schema that resolve them were removed from the grammar. Inequality constraint checking was left in

Table 1. Relative experimental parse times for a Prolog prototype high-speed parser

	Parse	Parse + Retrieve	Parse + Retrieve + Co-index
		5-words(ms)/31-words(ms)	
ALE parser			58/434
High-speed parser	1.1/7.5	5.1/22	9.2/52
Speed-up	53/58	11/20	6.3/8.3

ALE, since it is inherent in the compiled grammar rules, but since the grammar does not use this feature the execution overhead of checking for empty lists is small. A call was added to the ALE parser to traverse the feature structure of the sentence to number co-indexed nodes, since further processing would demand this: without this ALE times would be 26%/16% shorter for the two sentences. Co-indexing itself is automatic in the graph-based approach of ALE, but must be generated in the high-speed parser by detecting revisited nodes in the retrieval graph. Currently, recursive serial search is used, as with numbering in ALE, leading to the higher parse times shown. Implementation in an imperative code built-in predicate will eliminate this cost, marking visited nodes in a sparse array.

Although retrieval costs appear high, programming effort in Prolog could not reduce them, and they are realistic since the ALE parser copies the growing semantic structure at each phrase, while the high-speed parser traverses the largest structure once after parsing, leading to calculated ratios of 3 and 15 for the copying overheads for the short and long sentences respectively. Most sentences treated by wide-coverage grammars are ambiguous [12], and so generate both unincorporated and shared sub-trees, increasing the potential speed-up of the high-speed parser which delays traversal and does not repeat it.

Imperative built-ins will further increase speed-up as discussed in section 4.1, and block-copying of an array representation of an edge will be faster than compiling a Prolog clause to WAM: in space alone, a binary edge can be coded in 16 bytes rather than the 52 bytes of Prolog arguments. Adding semantic structure and asserting edges each form 25% of parse times, leaving 50% for the recursive algorithm with argument passing and clause reference. An imperative solution allows edge details to be randomly extracted given the edge number, eliminating most of the algorithm costs.

6.1 Estimated Size of Retrieval Graph for a Wide-Coverage Grammar

Because a new composite type is allocated to each distinct structure, an upper bound on the number of composite types can be calculated independently of the lexicon size. It is equal to the number of different combinations of linguistic feature values in each category, including those controlling agreement. Assuming that a verb agrees with its subject in number and person but not with its complements, and that a specifier and adjunct agree with the noun on number, gender, and case which is assumed to have four values, the number of composite types can be shown to be <45,000. Further reasonable assumptions are a limited range of HPSG structures, a branching factor of 3 in the type hierarchy, and 53 arcs in a large structure like Figure 1.

Encoding transitive verbs then requires 500Kbytes, involving mainly 53 sparse arrays, in an imperative implementation: this assumes each arc in the *retrieval graph* has an OR node with a sparse array of arc numbers, packed 4 to a byte, addressed with the composite type. Ditransitive verbs are assumed to be 20% larger, repeating the *synsem* structure. For a noun, arcs leading to *e_list* for *subj* and *comps* are added, with some minor *case* data reachable by *head*, and the small semantic structure illustrated in Figure 5. Assuming a generous 20% extension for each of 20 CFG categories, a total of 2.5Mbytes would be required, and a lot less if binary search replaced direct access.

7 Conclusions and Wider Application of the Techniques

The high-speed parser in its Prolog prototype is about 50 times faster than the ALE parser in Prolog, ignoring retrieval costs: implementing key sections in imperative code will improve this and also the retrieval costs which should be relatively smaller with commonly occurring ambiguous grammars and sentences. The time-cost of implementing HPSG principles is already included. Adding these to Amalia is likely to increase its parse times. The speed-up of the high-speed parser relative to ALE therefore compares favourably with the 11 times speed-up of Amalia.

7.1 Application of the Techniques to Constraint Programming

The pre-compilation of TFSs into compact representations with composite types, and of constraints into tables, can be applied to constraint logic programming over typed feature structures, CLP(TFS), provided that:

- There is some strategy governing the combination of constraints, and the number of intermediate steps over all possible programs is closely bounded.
- Recursion is not precluded, but decisions about which constraints to combine next do not increase in complexity with the number of recursions.

This would allow the precompilation of a fixpoint of TFSs over the clauses starting with the ground ones, as in [21] for Prolog, and composite types to be passed as terms at run-time. However, this is an area for further study.

Acknowledgments This research has been funded by a legacy from Miss Nora Brown. The authors are grateful to Shuly Wintner and Gerald Penn for technical conversations on the material covered.

References

1. Pollard, C., Sag, I.A.: Information-Based Syntax and Semantics, Volume 1: Fundamentals. CSLI Lecture Notes, No. 10, Leland Stanford Junior University: Centre for the Study of Language and Information (distributed by the University of Chicago press) (1987)
2. Pollard, C., Sag, I.A.: Head-Driven Phrase Structure Grammar. University of Chicago Press, Chicago (1994)
3. Carpenter, B.: The Logic of Typed Feature Structures. Cambridge University Press, Cambridge, UK (1992)
4. Shieber, S.M.: An Introduction to Unification-Based Approaches to Grammar. CLSI Lecture Notes No. 4, Leland Stanford Junior University (1986)
5. Shieber, S.M.: Constraint Based Grammar Formalisms. MIT Press, Cambridge, USA (1992)
6. Smolka, G.: A Feature Logic with Subsorts. Lilog-Report 33, IBM-Deutschland GmbH, Stuttgart (1988)
7. Aït-Kaci, H.: A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures. PhD Thesis, University of Pennsylvania (1984)
8. Manandhar, S., Grover, C.: A Course on HPSG Grammars and Typed Feature Formalisms. Human Communication Research Centre, University of Edinburgh (1996)
9. Matheson, C.: Developing HPSG Grammars in ALE. Course Notes, Human Communication Research Centre, University of Edinburgh (1996)
10. Gazdar, G., Mellish, C.: Natural Language Processing in Prolog. Addison-Wesley, Wokingham, England (1989)
11. Covington, A.: Natural Language Processing For Prolog Programmers. Prentice-Hall, New Jersey, USA (1994)
12. Carroll, J.A.: Practical Unification-Based Parsing of Natural Language. University of Cambridge Computer Laboratory, Technical Report No. 314 (1993)
13. Carpenter, B., Penn, G.: ALE: The Attribute Logic Engine User's Guide: Version 2.0. Technical Report, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA (1994)
14. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA (1993)
15. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (1991)
16. Carpenter, B., Penn, G.: Compiling Typed Attribute-Value Logic Grammars. In: Bunt, H, Tomita, M., eds.: Current Issues in Parsing Technologies, Vol. 2, Kluwer (1995)
17. Kay, M.: Head-Driven Parsing. In International Workshop on Parsing Technologies, Carnegie Mellon (1990) 52–62
18. Wintner, S., Francez, N.: Efficient Compilation of Unification-Based Grammars. BCN SummerSchool 97 on Topics in Constraint-Based Natural Language Processing, Groningen University (1997) 90–126.
19. Brown, J.C.: High Speed Feature Unification and Parsing. Natural Language Engineering Vol. 1 (4) (1995) 309–338
20. Götz, T., Meurers, D.: Interleaving Universal Principles and Relational Constraints over Typed Feature Logic. Proc. of the 35th. Meeting of the ACL and 8th. Conference of the EACL, Madrid, Spain (1997)
21. Van Roy, P.L.: Can Logic Programming Execute as Fast as Imperative Programming? Ph.D. thesis, University of California, Berkeley, TR CSD-90-600 (1990)

Abstract Machine Construction through Operational Semantics Refinements

Frédéric Cabestre, Christian Percebois and Jean-Paul Bodeveix

IRIT, Université Paul Sabatier
118, route de Narbonne
31062 Toulouse Cedex 4 France
e-mail: {cabestre,perceboi,bodeveix}@irit.fr

Abstract. This article describes the derivation of an abstract machine from an interpreter describing the operational semantics of a source language. This derivation process relies on the application of a set of gradual transformations to the interpreter written in a functional language. Through pass separation, the derivation process leads to the extraction of a compiler and an abstract machine from the transformed interpreter.

1 Introduction

The increasing interest in abstract machines consecutive to their portability, advocated by the popularity of the Java language [AG96], raises the problem of their design. The design of an abstract machine intended to the compilation and the execution of a high-level language is an arduous task. Generally, it needs an empiric survey of the semantics of the high-level language to study. In other words, one analyses the working of an interpreter embodying the operational semantics of the language to find the data structures and an emulator for an intermediate language constituting the abstract machine.

Some propositions have been made aiming at organizing this conceptual study. Among the methods suggested, we put forward those proposing an interpreter or an operational semantics as starting point and a set of gradual transformations as principle.

The rest of the paper is organized as follows: first we discuss about existing design methods. Section 3 presents the concepts supporting the derivation process. In section 4, we illustrate the method through a concrete example. Section 5 deals with abstract machine improvements.

2 Design methods of abstract machines

An abstract machine is composed of an interpreter for the intermediate language that it defines and of a **run-time environment** [ASU87] on which relies the interpreter. In what follows, the instructions of the language interpreted by the abstract machine will be called **abstract instructions** and their interpreter will be called **emulator**. To design an abstract machine, we must conceive an abstract data type **environment**, a suited instruction set and an emulator for this instruction set using the environment.

In this section, we present studies connected to the design of an abstract machine and present the guiding lines of the derivation process.

2.1 Principles of existing methods

In the framework of Prolog, we can note the work of Kursawe [Kur87] and Nilsson [Nil93]. Kursawe designs an abstract machine dedicated to the compilation of unification. However, abstract instructions are discovered through the partial evaluation of the interpreter on a given user program: some instructions may not be discovered. Nilsson performs pass separation [JS86] on an interpreter of the control of Prolog and obtains an exhaustive set of instructions but his methodology is still dedicated to Prolog.

Concerning functional languages, Hannan and Miller [HM90] propose an ad hoc progressive transformation of an operational semantics described by inference rules over lambda terms. The obtained result is an abstract machine defined by rewriting rules over the source language. Hannan [Han91] pursues this work and proposes a systematic way to perform pass separation over rewriting rules: the result is an intermediate target language and its emulator.

Sestoft [Ses97] transforms, in only one stage, the natural operational semantics of a lazy functional language in an abstract machine. Then, he introduces within the machine optimizations derived from the observation of its working. The final result is a machine similar to the TIM machine [FW87], but the derivation steps are still ad hoc.

Diehl [Die96] introduces a specification formalism, the two level big-step semantics (`2BIG`), over first order judgments. He defines and validates a set of progressive and automatic transformations leading to a compiler and an emulator. His pass separation step relies on Hannan's proposition.

All these approaches use natural semantics as specification language and rewrite rules as target language. Wand [Wan82] uses a continuation-based denotational semantics applied on the derivation of an abstract machine for a procedural language. He introduces combinators to eliminate free variables from the semantic equations. Then, each combinator is replaced by a first order term, and so the interpreter becomes a compiler with these terms as target language. An emulator associates to each term its semantics defined by its corresponding combinator. However, all these steps are highly empiric.

In conclusion, the starting point of these studies is a simple expression of the semantics of a language. It is simple because some mechanisms of the language to be implemented are implicitly taken into account by the power of the implementation language. Then, gradually, by finer steps, the implementation of these mechanisms is clarified while relying less and less on the power of the implementation language. Consequently, data structures constituting the abstract machine appear. Finally pass separation extracts a compiler and an emulator.

2.2 Overview of the derivation process

The derivation followed along this paper shares the same fundamental principles as most of the works cited above i.e. writing a first simple interpreter, giving a gradual clarification of its mechanisms and extracting an emulator and a compiler through pass separation. These works can be classified according to the specification formalism used to express the semantics of the source language [Die96]: natural semantics, translational semantics, denotational semantics, action semantics, operational semantics, etc. Here, we consider an executable formalism for operational semantics description which is a functional language. Thus, we join Wand's approach [Wan82] concerning the choice of the description formalism, altogether stating more precisely each transformation step.

Thus, an initial interpreter is written in a high-level language allowing to get a simple and quickly comprehensible expression of it. In a functional framework, the first writing makes heavy use of lexical bindings, recursion and higher order constructs. Then, we try not to rely on the high-level features of the implementation language in order to write new versions of the same interpreter. For that, step by step, we provide an implementation of the previous capabilities using low level constructs. The use of continuation passing style allows a uniform expression of the control flow which is rather machine like. However, the power of lexical binding still allows a high level description of the data flow, similar to recursion for the control flow. This mechanism must be eliminated in order to obtain low level machine code, thus enlightening internal data structures of the abstract machine. This step is completed by introducing a run-time environment encapsulating dynamic data. Therefore, abstract instructions and their emulator can be extracted through pass separation.

To sum up, the derivation of an abstract machine will proceed according to the following steps:

- Writing of a first interpreter.
- Binding time analysis which annotates dynamic variables.
- Elimination of lexical bindings of dynamic variables.
- Normalization of the interpreter which introduces the `environment` abstract data type.
- Elimination of the environment through η-reduction.
- Pass separation.

3 Concepts supporting the derivation process

The derivation process uses techniques issued from compilation, partial evaluation (binding time analysis) and program transformation (pass separation). In this section, we present the most specific concepts: pass separation used in the derivation of a compiler and an emulator, and the notion of abstract machine and its representation in a functional language.

3.1 Pass separation

Pass separation [JS86] consists in dividing a program into two distinct programs achieving two different activities of the initial program. Let us consider a program **P** and its data composed of **s** and **d**. Applying pass separation to **P** consists in building $\mathbf{P}_1$ and $\mathbf{P}_2$ such as eval **P s d** = eval $\mathbf{P}_2$ (eval $\mathbf{P}_1$ **s**) **d**. There are several solutions for $\mathbf{P}_1$ and $\mathbf{P}_2$, but our purpose is that $\mathbf{P}_1$ does as much work as possible. For example, if **s** is the static data of **P**, $\mathbf{P}_2$ will be limited to computations from the dynamic data **d**.

If **P** denotes an interpreter for a given language, $\mathbf{P}_1$ is then the compiler applied to the static data (the program) and $\mathbf{P}_2$ the compiled code emulator.

3.2 The implementation language

The implementation language must be sufficiently powerful to get a clear and concise writing of the first interpreter of the language to implement. Functional languages have some features allowing this first realization. They are suitable to a natural implementation of a transition system that describes our abstract machine. We kept for our survey Caml, a dialect of ML, whose features are [Ler96]:

- strong typing and possibility to define abstract data types,
- pattern matching,
- higher order functions,
- lexical binding of identifiers.

Among the transformations applied to the first interpreter, some of them aim at abandoning the use of the high-level features of the implementation language. Hence, this leads to explicit, in a lower level fragment of the language, mechanisms of interpretation of the language to implement. Consequently, the complexity of the interpreter structure will be gradually increased, hence, leading to the emergence of the abstract machine structure.

3.3 Caml expression of an abstract machine

An abstract machine is defined by two kinds of transition rules:

$$\begin{array}{ll} \text{(I)} & \langle i; C, e\rangle \Rightarrow \langle i_0; \ldots; i_n; C, e'\rangle \text{ with } e' = f_i e \\ \text{(II)} & \langle j; C, e\rangle \Rightarrow \langle C, e'\rangle \text{ with } e' = f_j e \end{array}$$

where i_k are terms of T_Σ with Σ a first order signature, and e and e' run-time environments. The first type of rule (I) adds new instructions to the sequence of code to be interpreted while modifying the run-time environment, whereas the second (II) only modifies the environment. In a general way, such a system can result in an continuation passing style interpreter of the form:

```
let rec eval = fun
  (* type I rules *)
  i -> fun cont env -> eval i0 (eval i1 (...(eval in cont)...) (fi env)
  (* type II rules *)
| j -> fun cont env -> cont (fj env);;
```

4 Design of abstract machines

We now present in detail the derivation steps leading to the definition of an abstract machine. As a demonstration example, we consider all along this section an interpreter of arithmetic expressions.

4.1 The example of a calculator

We illustrate the different transformations through a simple example dealing with an interpreter for arithmetic expressions. This interpreter includes a construction `try ... with ...` allowing to catch exceptions raised during the computation. In this example, the only exception that can be raised is the attempt of division by zero. The expression `try expr1 with expr2` means: "return the result of the evaluation of `expr1`, or the one of `expr2` if the evaluation of `expr1` raises a division by zero exception".

The abstract syntax of the language interpreted by this calculator is expressed by the following Caml abstract data type:

```
type expression =
    ID of string                        (* Identifier of a variable *)
  | INT of int                          (* Integer value *)
  | TRY of expression * expression      (* Construction try ... with ... *)
  | ADD of expression * expression      (* Operator + *)
  | SUB of expression * expression      (* Operator - *)
  | MUL of expression * expression      (* Operator * *)
  | DIV of expression * expression;;    (* Operator / *)
```

4.2 Writing a first interpreter

The first stage consists in writing an interpreter for the language to implement. This interpreter is either written in continuation passing style if special control flow must be specified, or results of the transformation of a direct style interpreter into continuation passing style [Plo75].

In our example, the first interpreter of the calculator described by figure 1 defines the function `eval1` having the following arguments:

- a term of the abstract syntax of type `expression` described at section 4.1, which is the expression to be evaluated.
- a forward continuatiuon `cont` associated to the normal progress of the computation. It returns an integer and has three arguments: an escape continuation, a value corresponding to the result of the previous evaluation of a sub-expression of the current expression and the list of variable bindings. However, the use of η-reduction may hide the third argument of some continuations.
- an escape continuation `exs`, which takes as argument the list of variable bindings and returns an integer. It pursues the computation from the inclosing `try ... with ...` if an exception is raised.
- a list `bnds` of couples `(name, value)` where `name` is the name of a variable identifier and `value` is the value associated to this variable i.e. an integer.

Note that the variable bindings remaining unchanged during the computation are transmitted from continuations to continuations, avoiding useless duplications. The same holds for escape continuations. A more natural writing of this first interpreter, corresponding to the translation of a recursive writing using the Caml `try-with` construct, would be the following:

```
let rec eval1 = fun
    INT(n) -> fun cont exs bnds -> cont n
  | ADD(e1,e2) ->
      fun cont exs bnds ->
        eval1 e1
          (fun v1 -> eval1 e2 (fun v2 -> cont (v1 + v2)) exs bnds) exs bnds
  | ...
  | TRY(e1,e2) ->
      fun cont exs bnds -> eval1 e1 cont (fun () -> eval1 e2 cont exs bnds) bnds
```

```
let rec eval1 = fun
    INT(n) -> fun cont exs -> cont exs n
  | ID(i) -> fun cont exs bnds -> cont exs (List.assoc i bnds) bnds
  | ADD(e1,e2) ->
      fun cont ->
        eval1 e1
          (fun exs1 v1 -> eval1 e2 (fun exs2 v2 -> cont exs2 (v1 + v2)) exs1)
  | DIV(e1,e2) ->
      fun cont ->
        eval1 e2
          (fun exs2 v2 ->
            if v2 = 0 then exs2
            else eval1 e1 (fun exs1 v1 -> cont exs1 (v1 / v2)) exs2)
  | TRY(e1,e2) ->
      fun cont exs -> eval1 e1 (fun _ -> cont exs) (eval1 e2 cont exs)
```

Fig. 1. The first interpreter

However, **exs** and **bnds** are lexically transmitted to nested abstractions entailing duplications. We also note the use of the unit parameter `()` to delay the evaluation of the escape continuation, as specified by the operational semantics of the `try-with` construct. Thus, the use of the continuation passing style allows a precise management of data and control flows.

The `eval` function of figure 2, which uses the continuation passing style interpreter, transmits to `eval1` two functions: a forward continuation (line 1) which takes three arguments, among which the result of the evaluation, and an escape continuation (line 2) which displays an error message.

```
 let eval term bnds =
   eval1 term
1    (fun _ v _ -> v)
2    (fun _ -> print_string "Uncaught exception" ; 0)
     bnds
```

Fig. 2. Top level evaluation function

4.3 Binding-time analysis

This stage performs binding-time analysis in order to identify variables whose values are known at compile-time and at run-time. For this purpose, we introduce two annotations related to the binding-time classification of [JGS93] introduced for the study of a partial evaluator for Scheme:

- v^d designates a dynamic variable only known at run-time,
- $v^{\rightarrow}$ designates a partially static closure whose code part is statically known and whose binding environment is known at run-time.

The values of unmarked variables are thus supposed to be known at compile-time. Furthermore, annotations are positioned on the declaration point of lambda variables.

Supposing that `eval1` is called with compile-time known values `term`, `cont` and `exs`, a handmade marking of our interpreter is illustrated by figure 3.

4.4 Lexical bindings elimination

This stage eliminates the use of lexical bindings to access annotated variables. These variables depend on the run-time environment of the interpreted program. If a continuation has access to such a data through

```
let rec eval1 = fun
    S_INT n -> fun cont→ exs→ -> cont exs n
  | S_ID id -> fun cont→ exs→ bnds^d -> cont exs (List.assoc id bnds) bnds
  | S_ADD(e1,e2) -> fun cont→ ->
      eval1 e1
        (fun exs1→ v1^d -> eval1 e2 (fun exs2→ v2^d -> cont exs2 (v1 + v2)) exs1)
      ...
  | S_DIV(e1,e2) -> fun cont→ ->
      eval1 e2
        (fun exs2→ v2^d ->
            if v2 = 0
              then exs2
              else eval1 e1
                 (fun exs1→ v1^d -> cont exs1 (v1 / v2)) exs2)
  | S_TRY(e1,e2) -> fun cont→ exs→ ->
      eval1 e1 (fun _ -> cont exs) (eval1 e2 cont exs)
```

Fig. 3. Binding-time annotations

lexical binding, it implicitly makes an access to a previous computation state and not to the current one. Eliminating lexical bindings allows to put forward new components of the run-time environment.

In order to eliminate lexical bindings of annotated data in a typed context, we use a variant of Reynolds's proposal [Rey72] where only the environment part of the closure is transmitted to nested abstractions. Thus, the code part, as well as static data, remain lexically accessed. We proceed as follows:

- We introduce the recursive sum type `stack` with a variant for each abstraction having free annotated variables. The variant associated to an abstraction contains the type of statically accessed annotated data where closures are represented by the type `stack`. For simplification purpose, a variant with one entry of type `stack` is not considered. Furthermore, variants of equal types are not duplicated.
- Each annotated functional variable becomes a pair build from the original variable and a `stack` variable. Then, for each application, the code part of the pair must be applied to its stack part in order to transmit lexical information.
- Each nested function is associated to its `stack` environement encapsulating lexical data. Consequently, a new formal parameter of type `stack` is added to these functions and is bound at call time to their lexical environment.

In our example, we introduce the type `stack` of figure 4. It defines three variants: `Empty` for abstractions without lexical data, `PushInt` for `(fun exs2`$^{\rightarrow}$ `v2`^{d} `...)` that lexically access to the two annotated variables `v1` and `cont`, and `PushStack` for `(fun _ -> cont exs)` which lexically access to `cont` and `exs`. Other nested functions use one of these variants.

```
type stack =
    Empty
  | PushInt of int * stack
  | PushStack of stack * stack
;;
```

Fig. 4. The type `stack`

The code of the `eval1` function is then transformed as shown by figure 5. Note that the functional variable `cont` becomes the pair `(cont,cs)` when applied (line 1). Similarly, `exs` becomes `(exs,xs)` (line 4). Lexical data are transmitted by building closures (line 3). An abstraction extracts its lexical data using pattern-matching on its new formal parameter `PushInt(v1,cs)` (line 2).

```
 let rec eval1 = function
1     INT(n) -> fun (cont,cs) exs -> cont cs exs n
    | ID(i) -> fun (cont,cs) exs bnds -> cont cs exs (List.assoc i bnds) bnds
    | ADD(e1,e2) ->
        fun (cont,cs) ->
          eval1 e1
            ((fun cs exs1 v1 ->
              eval1 e2
2               ((fun (PushInt (v1,cs)) exs2 v2 -> cont cs exs2 (v1 + v2)),
3                (PushInt (v1,cs)))
                exs1),
             cs)
    | DIV(e1,e2) ->
        fun (cont,cs) ->
          eval1 e2
4           ((fun cs (exs2,xs2) v2 -> if v2 = 0 then exs2 xs2
              else eval1 e1
                  ((fun (PushInt (v2,cs)) exs1 v1 -> cont cs exs1 (v1 / v2)),
                   (PushInt (v2,cs)))
                  (exs2,xs2)), cs)
    | TRY(e1,e2) ->
        fun (cont, cs) (exs, xs) ->
          eval1 e1
            ((fun (PushStack (xs,cs)) _ -> cont cs (exs,xs)), (PushStack (xs,cs)))
            ((fun (PushStack (xs,cs)) -> eval1 e2 (cont,cs) (exs,xs)), (PushStack (xs,cs)))
 ;;

 let eval e bnds =
   eval1 e ((fun _ _ v _ -> v), Empty)
           ((fun _ _ -> print_string "Uncaught exception"; 0), Empty) bnds
 ;;
```

Fig. 5. Lexical bindings elimination

4.5 Normalization of the interpreter

The normalization step aims at gathering all dynamic data inside a unique data structure, the run-time environment. For this purpose, a sum type is introduced where the type of each variant is the cartesian product of the dynamic parameter types of the functions used by the interpreter. Then, accessing to individual data is now performed through the environment. Thus, the interpreter will have the type

static data `-> continuation -> environment -> answer`,

where

`continuation =` *static data* `-> environment -> answer`.

This corresponds to the type of an interpreter equivalent to a transition system.

In our example, for the sake of simplicity, we only consider one variant. The environment defined by figure 6 is introduced as a record type including dynamic data detected by the binding time analysis: the environment part of closures (`cs` and `xs`), the variable bindings (`bnds`) and the integer accumulator `v` which stores the result of previous computations (`v1` and `v2`).

The primitives of the abstract data type `environment`, having the current environment as parameter, are introduced as follows:

- For each call, a primitive computes the environment representing its dynamic arguments.
- For each call where the function is dynamic, a primitive returns the function to be called.
- For each `if` statement, a primitive returns a boolean value corresponding to the condition of the test.

In our example, the functions of figure 7 are introduced.

```
type environment = {
  cs: stack;                   (* execution stack *)
  xs: stack;                   (* backup of the execution stack *)
  bnds: (string * int) list;   (* variable bindings *)
  v: int;                      (* accumulator *)
};;
```

Fig. 6. The type environment

```
let load n env = {cs=env.cs; xs=env.xs; bnds=env.bnds; v=n};;
let loadv v env = {cs=env.cs; xs=env.xs; bnds=env.bnds; v=List.assoc v bnds};;
let push env = {cs=PushInt(env.v,env.cs); xs=env.xs; bnds=env.bnds; v=env.v};;
let add {cs=PushInt(v1,cs); xs=xs; bnds=bnds; v=v} = {cs=cs; xs=xs; bnds=bnds; v=v+v1};;
let div {cs=PushInt(v1,cs); xs=xs;bnds=bnds;v=v} = {cs=cs; xs=xs; bnds=bnds; v=v/v1};;
let restore env = {cs=env.xs; xs=env.xs; bnds=env.bnds; v=env.v};;
let pushex env = let s = PushStack(env.xs,env.cs) in {cs=s; xs=s; bnds=env.bnds; v=env.v};;
let popex {cs=PushStack(xs,cs); xs=_; bnds=bnds; v=v} = {cs=cs; xs=xs; bnds=bnds; v=v};;
let zerop {cs=cs; xs=xs; bnds=bnds; v=v} = (v=0);;
```

Fig. 7. Environment access functions

The normalized interpreter including the so-defined environment access functions is given by the figure 8. All dynamic data is now encapsulated inside a unique variable env of type environment which is locally accessed.

```
let rec eval1 = function
    INT(n) -> fun cont exs env -> cont exs (load n env)
  | ID(i) -> fun cont exs env -> cont exs (loadv i env)
  | ADD(e1,e2) ->
      fun cont ->
        eval1 e1 (fun exs1 env -> eval1 e2 (fun exs2 env -> cont exs2 (add env))
              exs1 (push env))
  | DIV(e1,e2) ->
      fun cont ->
        eval1 e2
          (fun exs2 env ->
            if (zerop env) then exs2 (restore env)
            else eval1 e1 (fun exs1 env -> cont exs1 (div env))
                exs2 (push env))
  | TRY(e1,e2) ->
      fun cont exs env ->
        eval1 e1
          (fun _ env -> cont exs (popex env)) (fun env -> eval1 e2 cont exs (popex env))
          (pushex env)
```

Fig. 8. The normalized interpreter

Introducing this run-time environment implies to modify the eval function as defined by figure 9. It transmits to eval1 a forward continuation cont0 which gets a result from the accumulator, an escape continuation (ex0) and an initial run-time environment initialized using the bnds parameter.

```
let cont0 env = env.v;;
let ex0 env = (print_string "Uncaught exception"; 0);;

let eval term bnds = eval1 term (fun _ -> cont0) ex0 {cs=Empty; xs=Empty; bnds=bnds; v=0};;
```

Fig. 9. Top level for normalized evaluation function

4.6 η-reduction of the interpreter

This step eliminates from the interpreter all explicit references to the environment: all variables of type `environment` are discarded through η-reduction. For this purpose, we must consider sequence, selection, and indirect jumps control structures:

- For the sequence, we introduce a composition combinator, noted ++ and defined by:

  ```
  let (++) f g env = g (f env);;
  ```

- For each test function of a selection, a combinator is defined as follows:

  ```
  let if_test true_cnt false_cnt env =
    if (test env) then true_cnt env else false_cnt env;;
  ```

- For each indirect jump corresponding to the application of a function computed from the current environment, we introduce a combinator having as many parameters as the function. The combinator performs the jump and all the actions previously done on the environment.

After the introduction of these combinators, η-reduction eliminates all occurrences of parameters of type `environment`. In our example, the obtained code is described by figure 10.

```
let rec eval1 = function
    INT(n) -> fun cont exs -> (load n) ++ (cont exs)
  | ID(i) -> fun cont exs -> (loadv i) ++ (cont exs)
  | ADD(e1,e2) ->
      fun cont ->
        eval1 e1 (fun exs1 -> push ++ (eval1 e2 (fun exs2 -> add ++ (cont exs2)) exs1))
  | DIV(e1,e2) ->
      fun cont ->
        eval1 e2
          (fun exs2 ->
            if_zerop
              (restore ++ exs2)
              (push ++ (eval1 e1 (fun exs1 -> div ++ (cont exs1)) exs2)))
  | TRY(e1,e2) ->
      fun cont exs ->
        pushex ++ (eval1 e1 (fun _ -> popex ++ (cont exs)) (popex ++ (eval1 e2 cont exs)))
```

Fig. 10. The η-reduced interpreter

4.7 Pass separation

This stage splits the interpreter into a compiler and an emulator. Until now, the interpreter is a function which, given a program and a continuation, returns a function taking an environment as parameter. It must now return a sequence of abstract instructions which is the compiled code for the program. It can be transmitted to an emulator to get the final result of the computation.

In other words, the initial type of the interpreter was:

static data -> continuation -> environment -> answer

We split the interpreter into the two functions **comp** and **emul** with respective types:

```
comp : static data -> continuation -> abstract code
   emul : abstract code -> environment -> answer
```

The function comp compiles a program into abstract code, and the function emul interprets this abstract code and updates the environment to compute the result. We have performed a pass separation and extracted a compiler and an emulator from the initial interpreter.

The method used here for pass separation consists in generalizing the interpreter with respect to all the introduced combinators: we define a new *generic* interpreter having all the combinators as formal parameters. In such a way, the type environment no more occurs within the type of the generic function. Furthermore, the type environment -> environment is abstracted into a polymorphic Caml type 'a. In the same way, environment -> answer is abstracted into 'b.

In our example, we get the interpreter of figure 11, whose type is described by figure 12.

```
let geval e load loadv (++) push pushex popex add div if_zerop cont0 ex0 =
  let rec eval1 = function
    INT(n) -> fun cont exs -> (load n) ++ (cont exs)
  | ID(i) -> fun cont exs -> (loadv i) ++ (cont exs)
  | ADD(e1,e2) ->
      fun cont ->
        eval1 e1
          (fun exs1 ->
            push ++ (eval1 e2 (fun exs2 -> add ++ (cont exs2)) exs1))
  | DIV(e1,e2) ->
      fun cont ->
        eval1 e2
          (fun exs2 ->
            if_zerop (push ++ (eval1 e1 (fun exs1 -> div ++ (cont exs1)) exs2))
              exs2)
  | TRY(e1,e2) ->
      fun cont exs ->
        pushex ++ (eval1 e1 (fun _ -> popex ++ (cont exs)) (popex ++ (eval1 e2 cont exs)))
  in eval1 e (fun _ -> cont0) ex0
;;
```

Fig. 11. The generic interpreter

It remains to notice that the type of the generic interpreter defines the signature of an abstract data type. Each type parameter defines a sort, and the type of each argument defines the signature of an operator. Names must be chosen for each sort and each operator. Hence, the abstract instruction set is defined.

In our example, we introduce two types: instruction and control and their associated constructors, as defined by the figure 13.

We are now able to perform pass separation. The compiler of figure 14 is obtained by instanciating the generic interpreter with the operators of the introduced abstract data type. The emulator of figure 15 is an interpretation of the abstract data type. It associates to each operator its meaning given by the corresponding combinator. The abstract data type defining two sorts, the emulator is composed of two functions emul and ins_emul. Finally, we get the caracteristic equation of pass separation:

```
let eval e bnds = emul (comp e) {cs=Empty; xs=Empty; bnds=bnds; v=0};;
```

We now illustrate the result of the transformation process through the compilation of an arithmetic expression:

```
geval;;
- :   exp ->
  (int -> 'a) ->              (* load     *)
  (string -> 'a) ->           (* loadv    *)
  ('a -> 'b -> 'b) ->         (* ++       *)
  'a ->                       (* push     *)
  'a ->                       (* pushex   *)
  'a ->                       (* popex    *)
  'a ->                       (* add      *)
  'a ->                       (* div      *)
  ('b -> 'b -> 'b) ->         (* if_zerop *)
  'b ->                       (* cont0    *)
  'b ->                       (* ex0      *)
  'b = <fun>
```

Fig. 12. Synthetized type of the generic interpreter

```
type instruction =                          type control =
    I_LOAD of int                               I_SEQ of instruction * control
  | I_LOADV of string                         | I_IFZEROP of control * control
  | I_ADD                                     | I_CONT0
  | I_DIV                                     | I_EX0
  | I_PUSH                                  ;;
  | I_PUSHEX
  | I_POPEX
;;
```

Fig. 13. The abstract instruction set

```
let comp e = geval e
    (fun n -> I_LOAD n)
    (fun s -> I_LOADV s)
    (fun i c -> I_SEQ (i,c)) I_PUSH I_PUSHEX I_POPEX I_ADD I_DIV
    (fun c1 c2 -> I_IFZEROP (c1,c2)) I_CONT0 I_EX0;;
```

Fig. 14. The compiler

```
let ins_emul = function
    I_LOAD n -> load n
  | I_LOADV s -> loadv s
  | I_ADD -> add
  | I_DIV -> div
  | I_PUSH -> push
  | I_PUSHEX -> pushex
  | I_POPEX -> popex
;;

let rec emul = function
    I_SEQ(i,c) -> (ins_emul i) ++ (emul c)
  | I_IFZEROP(c1,c2) -> if_zerop (emul c1) (emul c2)
  | I_CONT0 -> cont0
  | I_EX0 -> ex0
;;
```

Fig. 15. The emulator

```
# comp (ADD(INT 1,TRY(ADD(INT 2,DIV(INT 3,INT 0)),INT 4)));;
- : control =
I_SEQ (I_LOAD 1, I_SEQ (I_PUSH, I_SEQ (I_PUSHEX, I_SEQ (I_LOAD 2,
   I_SEQ (I_PUSH, I_SEQ (I_LOAD 0, I_IFZEROP (
      I_SEQ (I_PUSH, I_SEQ (I_LOAD 3, I_SEQ (I_DIV, I_SEQ (I_ADD,
             I_SEQ (I_POPEX, I_SEQ (I_ADD, I_CONTO)))))),
      I_SEQ (I_POPEX, I_SEQ (I_LOAD 4, I_SEQ (I_ADD, I_CONTO)))))))))
```

5 Towards an efficient abstract machine

By a set of gradual transformations, we attempt to extract what exists potentially in the initial interpreter. It makes it possible to separate from the interpreter the static part, the compiler, of the dynamic part, the emulator. The degree of realism of the compiler and of the obtained machine is related to the writing of the initial interpreter.

Let us consider a problem raised because the initial interpreter is not optimized by taking into account all the information contained in the abstract syntax. The evaluation of an addition, whose code is:

```
| ADD(e1,e2) -> fun cont excp ->
    eval e1 (fun exs1 v1 -> eval e2 (fun exs2 v2 -> cont exs2 (v1 + v2)) exs1) excp
```

performs two recursive calls to `eval` which in certain cases could be completely avoided. Indeed it is enough to operate an unfolding of these calls in the particular cases where either `e1` or `e2` are integers. Thus we could have written the four following clauses:

```
| ADD(INT n1, INT n2) -> fun cont excp -> cont excp (n1 + n2)
| ADD(INT n1, e2) -> fun cont -> eval e2 (fun exs2 v2 -> cont exs2 (n1 + v2))
| ADD(e1, INT n2) -> fun cont -> eval e1 (fun exs1 v1 -> cont exs1 (v1 + n2))
| ADD(e1,e2) -> fun cont ->
    eval e1 (fun exs1 v1 -> eval e2 (fun exs2 v2 -> cont exs2 (v1 + v2)) exs1)
```

Consequently, applying the same method would have led to an abstract machine with one more instruction, `ADD_INT(n)`, which adds a constant to the accumulator.

From an operational point of view, the semantics defined by this optimized interpreter is not the same as that defined by the basic interpreter. Indeed, if these two semantics calculate the same result, they proceed in a different way. Thus, the operational specification of the language to be implemented heavily governs the obtained result.

6 Conclusion

We have presented a process for deriving an abstract machine from a source language whose operational semantics is defined by an interpreter written in a functional language. It is based on a progressive transformation of this interpreter into a compiler and an emulator of abstract code.

The initial interpreter is written by means of a high level functional language. Then it is transformed gradually so that it does not rely on the features of the implementation language. So, we must explicit mechanisms of interpretation that were taken implicitly into account. These transformations lead to the emergence of data structures which will be the heart of the abstract machine.

Pass separation is then applied to the interpreter, aiming at splitting it into two complementary activities: on the one hand the compilation of the source language that produces the intermediate code, and on the other hand the interpretation of this intermediate code. Thus, we get a compiler and an emulator of abstract code.

The formalism used along this paper is functional and higher order continuation-based, as Wand's. The guideline of the transformations is similar but the details differ in our framework as Wand's work is untyped. Furthermore, Wand's transformations are only studied with respect to his examples.

We now plan to study the automatization of the presented study and its correctness. Mainly, a specific binding time analyser must be implemented. The second step will be the design of a validated abstract machine development tool on top of a proof assistant [HKPM97]. Then more realistic languages could be considered, and more precisely a subset of Java and λProlog [NM88].

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison Wesley, 1996.

[ASU87] A. Aho, R. Sethi, and J. Ullman. *Compilers : principles, techniques and tools.* Addison Wesley, 1987.

[Die96] S. Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines.* PhD thesis, University Saarbrücken, Germany, 1996.

[FW87] J. Fairbain and S. C. Wray. TIM : a simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Functional programming languages and computer architecture*, volume 274. Springer Verlag, 1987.

[Han91] J. Hannan. Staging Transformations for Abstract Machines. *Partial Evaluation and Semantic Based Program Manipulation*, 26(9):130–141, september 1991.

[HKPM97] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.1. rapport technique 204, INRIA, Août 1997. Version révisée distribuée avec Coq.

[HM90] J. Hannan and D. Miller. From Operational Semantics to Abstract Machines : Preliminary Results. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 323–332. ACM Press, 1990.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall Int., 1993.

[JS86] U. Joerring and W.L. Scherlis. Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages*, pages 86–96, St Petersburg, Florida, 1986.

[Kur87] P. Kursawe. How to Invent a Prolog Machine. *New Generation Computing*, 5:97–114, 1987.

[Ler96] X. Leroy. *The Objective Caml System Release 1.03.* INRIA, october 1996.

[Nil93] U. Nilsson. Towards a Methodology for the Design of Abstract Machines for Logic Programming Languages. *Journal of Logic Programming*, 16:163–189, 1993.

[NM88] G. Nadathur and D. Miller. An Overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming.* MIT Press, 1988.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theorical Computer Science*, 1:125–159, 1975.

[Rey72] J.-C. Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740, 1972.

[Ses97] P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3), 1997.

[Wan82] M. Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

Brunette: Brute Force Rewriting Engine

Makoto Ishisone, Ataru T. Nakagawa

SRA Software Engineering Laboratory

Abstract. This paper presents an order-sorted conditional term rewriting engine which supports equational theories such as associativity and commutativity. It is designed to be simple and fast. Rewriting is performed by executing bytecode instructions specialised for pattern matching, term construction and sort (re)computation. To be simple enough, a brute force method is used for matching modulo equational theories. This may lead to inefficiency, but with some technique and in most cases the rewriting speed is pretty good.

1 Introduction

Term rewriting[1] is a simple but powerful mechanism to compute functions and to prove equality. At the simplest level, given a rewrite rule set R and a term t, term rewriting proceeds as

1. find in R a rule whose left-hand side matches a subterm (redex) of t,
2. if such a rule was found, substitute the corresponding instance of the right-hand side into t, and go back to 1. Otherwise stop.

In terms of implementation, therefore, all you need are

- A manager of a rewrite rule set,
- A unification procedure, and
- A substitution procedure.

In case of ground term rewriting, the mechanism is simpler, requiring only a pattern matching procedure, instead of a general unification algorithm.

In useful applications, however, there tend to arise a couple of complications, depending on the definition of terms and the admissible form of rules. Several term rewriting engines that have been developed (OBJ3[3], Maude[2], ELAN[4], CafeOBJ[5]) attempt to accommodate those complications without compromising efficiency. For example, in the case of CafeOBJ,

- Each term is sorted; moreover, each term may have more than one sort; 1 is a natural number, an integer, and a rational, at the same time.
- Several distinct terms may be identified, due to associativity, commutativity, identity, and/or idempotency of operators; $1 + (2 + 3)$ is identical to $(3 + 1) + 2$.

- An arbitrary evaluation strategy may be associated to an operator; in evaluating a conditional operator if-then-else, the then part and the else part need not be evaluated before reducing the entire term. Thus you may have lazy operators, eager operators, and partially lazy operators.
- A rewrite rule may be conditional, and such a rule is applicable only when the condition evaluates to true.

Our goal has been to develop a term rewriting engine that covers as many of these features as possible, yet is fast enough and scalable. For that purpose, we defined an abstract machine architecture, where to evaluate a term is to execute bytecode instructions corresponding to the basic procedures of term rewriting. The idea of using an abstract machine was originated from TRAM[7], which, however, uses a quite different architecture, and could not handle associative/commutative matching.

In the rest of the paper, we first give an overview of our approach (Section 2), followed by an explanation of the machine architecture (Section 3). Then we give an account of implementation (Sections 4 and 5). Section 6 shows the results of preliminary evaluation and future works.

2 Overview

Our term rewriting engine, called Brunette, works via standard i/o interface. A rewrite rule set is written in the form of Lisp's S-expressions, and is preceded by a signature (sort and operator declarations). For example, a definition of addition over natural numbers may be supplied as

```
(sort Zero PosNat Nat)
(sort-order (Zero Nat) (PosNat Nat))
(op 0 () Zero (0))
(op s (Nat) PosNat (1 0))
(op + (Nat Nat) Nat (1 2 0) (:assoc :comm)))
(rule ((M Nat)) (+ (0) M) M)
(rule ((M Nat) (N Nat)) (+ (s M) N) (s (+ M N)))
```

where

- `sort` introduces a set of sorts.
- `sort-order` imposes an order over the introduced sorts. In the above, `Zero` is included in `Nat`; so is `PosNat`.
- `op` declares an operator with an arity, a coarity, an evaluation strategy, and an optional list of equational attributes. In the above, `+` has the arity `Nat Nat`, the coarity `Nat`, the strategy `1 2 0` — evaluate the first argument first, the second second, and then the whole term (`0`) —, and is associative and commutative.
- `rule` introduces a rewrite rule, which is a list of a variable declaration, a left-hand side, a right-hand side, and an optional condition term. For example, the last rule consists of variables `M` and `N` of sort `Nat` and two terms `s(M)+N` and `s(M+N)` (if we use the usual notation). This rule is unconditional.

To summarise, Brunette admits (1) ordered sorts, (2) equational attributes such as associativity, (3) evaluation strategies, and (4) conditional rules. The following restrictions are imposed on rules.

- The right-hand side or condition must not contain variables that do not appear in the left-hand side. Brunette is strictly for ground term rewriting.
- The left-hand side and right-hand side may be of different sorts, but these sorts must belong to the same connected component with respect to sort orders.

Given a sequence of S-expressions as above, the Brunette compiler translates the left-hand side of each rule into a bytecode sequence, as explained in Section 4. Given a term to evaluate, then, the Brunette rewrite engine runs the sequence to see whether the term matches the left-hand side of a rule. The right-hand side of a rule is compiled into another bytecode sequence (for a substitution procedure). Thus rewriting is essentially an execution of bytecode sequences.

To provide a more pleasant interface, the Brunette compiler and its engine were incorporated in CafeOBJ language processor, so that you may act entirely within CafeOBJ codes (Figure 1).

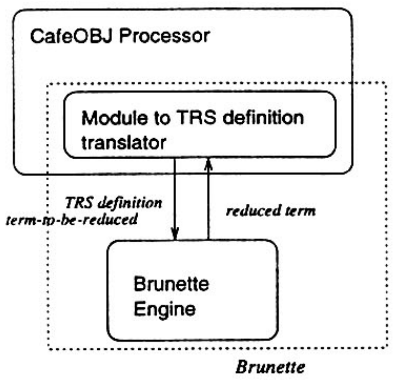


Fig. 1. Brunette as an CafeOBJ engine

In using Brunette in this setting, you may dispose of lots of parentheses, like a code equivalent to the above addition example shows:

```
[ Zero PosNat < Nat ]
op 0 : -> Zero
op s : Nat -> PosNat { strat: (1 0) }
op + : Nat Nat -> Nat { strat: (1 2 0) assoc comm }
eq +(0, M:Nat) = M .
eq +(s(M:Nat), N:Nat) = s(+(M, N)) .
```

For legibility, in the sequel the examples are written in a CafeOBJ-like syntax.

3 Architecture

Figure 2 shows the architecture of Brunette. It consists of two major components, the bytecode compiler and the rewrite engine.

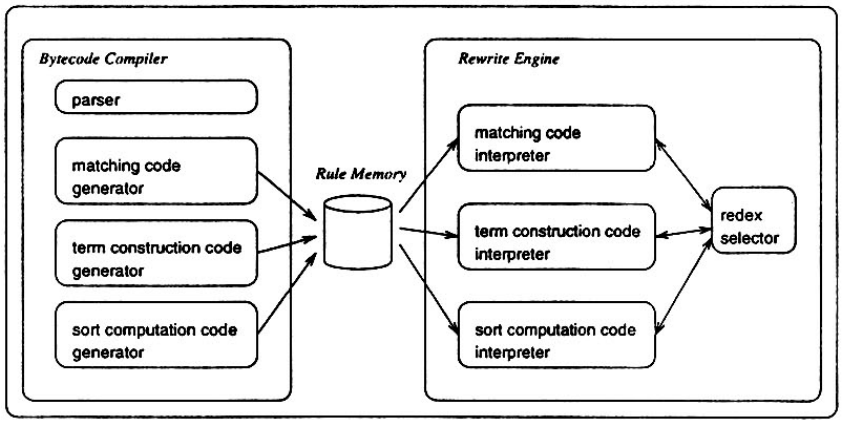

Bytecode Compiler
parser
matching code generator
term construction code generator
sort computation code generator
Rule Memory
Rewrite Engine
matching code interpreter
term construction code interpreter
sort computation code interpreter
redex selector

Fig. 2. Brunette architecture

3.1 Bytecode Compiler

The bytecode compiler parses the given rewrite rules and produces bytecode[1] sequences to be interpreted by the rewrite engine. There are 3 types of bytecodes: *m-code* for pattern matching, *r-code* for term construction, and *s-code* for sort computation. They will be explained in detail in Section 4.

3.2 Rewrite Engine

The rewrite engine contains 4 major components, (1) a redex selector, (2) a pattern matcher, (3) a term constructor, and (4) a sort calculator, and the engine pursues rewriting as follows. Given a term to evaluate,

1. Let the redex selector see the evaluation strategy of the dominant operator. An evaluation strategy is a list of non-negative integers, such as 1 0 2. $n > 0$ means the n-th argument, and 0 means the whole term.

[1] Each code is a 32-bit word, so strictly, what is produced is "wordcode".

2. If the list is empty, stop. Otherwise, the selector takes the first number. Call it n.
   a. If n is 0, invoke the pattern matcher to find a rewrite rule applicable to the term.
      i If there is no rule to apply, go to 3.
      ii Otherwise, call the term constructor to perform the substitution. Additionally, if the sort of the newly created term is not determined, call the sort calculator. Then go back to 1.
   b. Otherwise, the selector chooses the n-th immediate subterm, and recursively call this procedure with this subterm.
3. Go back to 2. with the rest of the list.

In case of a conditional rule, after the match succeeds, the condition itself is evaluated with this procedure.

4 Implementation in Detail

4.1 Term Representation

A term is internally represented as a directed acyclic graph, where each node contains (1) an operator symbol, (2) a sort identifier, (3) a set of flags, (4) the number of subnodes (arity), and (5) pointers to subnodes (Figure 3).

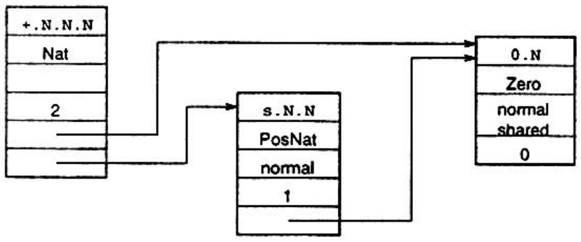


Fig. 3. an internal representation of "0 + s(0)"

Each operator symbol is qualified with a rank — a list of sort families[2] — that distinguishes different operators with the same name. For example, if the sorts `Nat` and `Int` are in the same sort family while `String` is not, the first two operators are recognised as the same, which is distinct from the last one:

```
op + : Nat Nat -> Nat
op + : Int Int -> Int
op + : String String -> String
```

[2] A sort family is a connected component of sorts ordered by inclusion.

The sort identifier represents the sort of the term. Since an operator symbol is always qualified by sort families, the sort *family* of the term is self-evident. The rôle of the sort identifier is to indicate which exact sort in the family the term belongs to. Continuing the example above, given a term dominated by +, the identifier indicates whether the term belongs to `Nat` or `Int`[3].

The flags indicate the state of the term:

- whether the term is already in a normal form,
- whether its sort is already computed, and
- whether it is shared, i.e., is pointed to by more than one nodes.

4.2 Pattern Matcher

As previously stated, the Brunette rewrite engine contains a pattern matcher, a term constructor, and a sort calculator. These parts are realised as interpreters of bytecode instructions.

In the following three sections, these 3 parts and their instruction sets are described briefly, with some examples. We focus on their basic features first, and the issue of associative/commutative matching is to be treated in a separate section (Section 5).

The pattern matcher is an interpreter of *m-code*. It has a register called **NODE** and a binding table called **BIND**. **NODE** points to the current node which many *m-code* instructions operate on, and is initialised to the node passed from the redex selector. The table **BIND** is an array of registers, used to store variable bindings as well as to store temporary values as a scratch memory.

Table 1 lists the basic *m-code* instructions.

Instruction	Description
`match_sym` *SYM*	If symbol of NODE is not *SYM*, matching will fail.
`match_sort` *SORTS*	If sort of NODE is not included in *SORTS*, matching will fail.
`match_var` *INDEX*	If NODE is not equivalent to BIND's *INDEX*th element, matching will fail.
`child` *N*	Go to *N*th child – i.e. set NODE to *N*th argument of NODE.
`bind` *INDEX*	Save NODE in BIND's *INDEX*th element.
`ref` *INDEX*	Set NODE to BIND's *INDEX*th element.
`reset`	Reset NODE to its initial value.
`ret` *N*	Return from the interpreter with success. *N* is the number of valid elements in BIND.

Table 1. basic *m-code* instructions

[3] In all the actual examples we have examined, a sort family consists of rather small number of sorts. Hence Brunette uses a small integer for a sort identifier.

For example, suppose we have the following rewrite rule:

```
+(X:Nat, 0) -> X
```

The left-hand side of this rule will be translated into the following *m-code.*

```
match_sym +          ; Top-most operator must be '+'.
child 0              ; Go to the first subnode.
match_sort Nat       ; This node must be of sort Nat.
bind 0               ; Bind this node to variable X.
reset                ; Go to the top node.
child 1              ; Go to the second subnode this time.
match_sym 0          ; The operator of this node must be '0'.
ret 1                ; Matching succeeds. BIND contains 1
                     ; entry (X).
```

For a more complicated example, the *m-code* corresponding to

```
foo(X:S, bar(X:S, Y:S))
```

is

```
match_sym foo        ; Is the top-most operator 'foo'?
child 0              ; Go to the first subnode.
match_sort S         ; Is the sort S?
bind 0               ; Bind this node to X.
reset                ; Go back to the top node.
child 1              ; Got to the second subnode.
match_sym bar        ; Is the operator 'bar'?
bind 2               ; Save NODE to get back here later.
child 0              ; Go to the first subnode.
match_var 0          ; Is this node equivalent to X?
ref 2                ; Go back to the parent node.
child 1              ; Then go to the 2nd subnode.
match_sort S         ; Is the sort S?
bind 1               ; Bind this node to Y.
ret 2                ; Success with 2 valid entries (X and Y)
```

Optimising The bytecode sequence of the last example is rather long. Since the length of the sequence is a major factor that affects the interpreter performance, it is important to reduce the number of instructions whenever possible. Brunette has many macro instructions which perform sequences of basic instructions in one go. The last example is actually compiled into this *m-code*:

```
match_sym foo
match_and_bind0 S 0  ; Is the sort of first subnode S?
```

```
                        ; If so, bind it to X. Otherwise, fail.
child1                  ; Go to the 2nd subnode.
match_sym bar           ; Is the operator 'bar'?
match_var0 0            ; Is 1st subnode equivalent to X?
match_and_bind1 S 1     ; Is the sort of 2nd subnode S?
                        ; If so, bind it to Y.
ret 2                   ; Matching is succeeded.
```

4.3 Term Constructor

The term constructor is an interpreter of *r-code*. It is used for creating a new term according to the right-hand side of a matched rule. It is also used to construct terms that appear in the condition part of a conditional rule.

The constructor has registers called **NODE** and **TOP**, and a binding table **BIND**. **NODE** is a working register and points to the current node. **TOP** points to the topmost node. After execution, the node pointed by **TOP** is returned as the result.

The table **BIND** is an array of registers, and is shared between the pattern matcher and the term constructor. Some of its entries have been initialised by the matcher, which has the main responsibility of determining variable bindings.

Table 2 lists the basic *r-codes*.

Instruction	Description
`cr_top` *SYM*	Create a node with symbol *SYM*. Set NODE and TOP to the node.
`cr_child` *N SYM*	Create a node with symbol *SYM*. Set it as *N*th argument of the node pointed by NODE, then set NODE to the node.
`put_top` *INDEX*	Set NODE and TOP to *INDEX*th element of BIND.
`put_child` *N INDEX*	Set NODE's *N*th argument to point *INDEX*th element of BIND, then set NODE to the node.
`bind` *INDEX*	Save NODE in BIND's *INDEX*th element.
`ref` *INDEX*	Set NODE to BIND's *INDEX*th element.
`reset`	Set NODE to TOP.

Table 2. basic *r-code* instructions

As an example, suppose we have a rewrite rule

```
+(X:Nat, s(Y:Nat)) -> s(+(X, Y))
```

and assume the pattern matcher stores the values of X, Y in the first 2 elements (index 0 and 1) of **BIND**. Then the right-hand side of this rule will be translated into the following *r-code*.

Instruction	Description
`set` *SORT*	Set the node's sort to *SORT* and return.
`jump_unless_match` *N SORTS OFFSET*	If the sort of the node's *N*th argument doesn't match *SORTS*, skip *OFFSET* words.

Table 3. basic *s-code* instructions

```
cr_top s              ; Create a node with operator 's'
                      ; and make it top.
cr_child 0 +          ; Create a node with + as its 1st subnode.
put_child 0 0         ; Set contents of X as the 1st subnode.
put_child 1 1         ; Set contents of Y as the 2nd subnode.
```

4.4 Sort Calculator

The sort calculator is an interpreter of *s-code*. It is used for computing the sort of a specific node. Such computation is necessary since, for example, if `Nat` is a subsort of `Int`, a term originally of sort `Int` may become a term of sort `Nat` (as well as of `Int`) during computation. Such a change affects applicability of rewrite rules.

Table 3 lists the basic *s-code* instructions.

Before executing a given *r-code*, the calculator checks if the sort of each subnode has already been computed. If it hasn't, the calculator calls itself recursively to compute it.

Unlike *m-code* and *r-code*, which are generated for each rewrite rule, *s-code* is generated for each operator. For example, suppose we have an operator `+` declared as

```
+ : Nat Nat -> Nat
+ : Int Int -> Int
```

where, as usual, `Nat` is a subsort of `Int`. Then *s-code* for `+` is

```
   jump_unless_match 0 Nat L1    ; Jump to L1 if arg0 is not Nat.
   jump_unless_match 1 Nat L1    ; Jump to L1 if arg1 is not Nat.
   set Nat                       ; Set Nat and return.
L1:jump_unless_match 0 Nat,Int   ; Jump to L2 if arg0 is
                     L2          ; neither Nat nor Int.
   jump_unless_match 1 Nat,Int   ; Jump to L2 if arg1 is
                     L2          ; neither Nat nor Int.
   set Int                       ; Set Int and return.
L2:set ErrorSort                 ; Invalid. Mark it as ErrorSort.
```

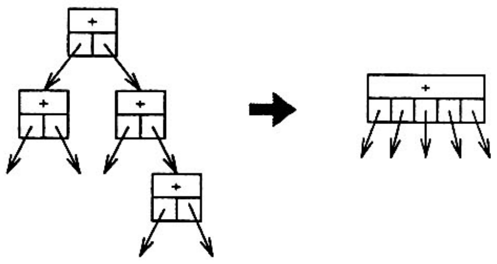


Fig. 4. Flattened representation

5 Associative/Commutative Matching

In this section we describe a matching procedure modulo equational theories. As the name implies, Brunette resorts to brute force, except for a couple of heuristic tricks. To simplify the explanation, we restrict ourselves to associativity and the combination of associativity and commutativity, although Brunette also deals with identities and other combinations. And to save space, we say just AC instead of associative and commutative.

5.1 Flat Term Representation

To make associative or AC matching faster, every node with an associative or AC operator is flattened, as if the operator accepts an arbitrary number of arguments (Figure 4).

5.2 Matching Instructions

In Brunette, AC matching is done on trial-and-error basis with backtracking. To support this procedure, the semantics of some *m-code* instructions described in Section 4 are slightly changed. For example, the instruction match_sym does not make the match fail, but causes backtracking.

In addition, several new instructions are added, some of which are listed in Table 4.

For example, if + is AC, *m-code* for the pattern "a + X + Y" is

```
match_sym +           ; top-most operator must be +.
try_ac 0              ; Prepare AC match.
ac_find_sym 0 a       ; Find a subnode with operator a.
ac_bind_nodes 0 0 1   ; Bind a combination of subnodes
                      ; to variable X, leaving at least one
                      ; subnode unmatched for Y.
ac_bind_nodes 0 1 0   ; Bind a combination of subnodes to
```

Instruction	Description
`try_ac` *ID*	Prepare AC matching on NODE.
`ac_find_sym` *ID SYM*	Find an unmatched subnode of AC node identified by *ID* having symbol *SYM*, mark it as matched and set NODE to it. Do this to all the unmatched nodes via backtracking.
`ac_bind_node` *ID INDEX*	Mark one of the unmatched subnodes of *ID* as matched and save it in BIND's *INDEX*th element. Process all of them repeatedly via backtracking.
`ac_bind_nodes` *ID INDEX MIN*	Get a combination of unmatched subnodes of *ID* and mark them matched (but leaving at least *MIN* subnodes unmatched), and save the combined node to BIND[*INDEX*]. Process every possible combination through backtracking.

Table 4. Instructions for AC matching

```
                          ; variable Y.
    ret 2                 ; Matching succeeds. BIND contains 2
                          ; entries (X, Y).
```

Note that matching with the instruction **ac_bind_nodes** can become extremely inefficient as the number of subnodes increases.

5.3 Avoiding Excessive Search

Brute-force AC matching can result in an excessive search. Which, moreover, is often futile. It is most important to avoid such an excess as much as possible.

An excessive search occurs when an argument to an AC operator is a plain variable. For example, if + is AC, suppose the pattern

```
foo(X) + Y
```

is being matched against a term $t_1 + \ldots + t_n$ (n arguments when flattened). The number of possible matches is at most n for `foo(X)`, which is tolerable even by trial-and-error basis; but it is $2^n - 2$ for Y[4]. Thus we should prepare a disciplinary action for such an unwieldy variable.

Brunette uses the following techniques to avoid an excessive search with such a variable:

- Optimising the matching strategy,
- Use of sort information,

[4] You may choose any number, up to $n - 1$, of t_i's, so the number of possibilities is $m(n,1) + m(n,2) + \ldots + m(n, n-1)$ where $m(n,k)$'s are binomial coefficients.

– Use of shared information, and
– Cutting useless candidates.

We explain below the first two techniques, which are useful in many cases.

Optimising the Matching Strategy Even if a variable is placed directly under an AC operator, an excessive search is unnecessary (1) if it is already bound, or (2) when a variable must match to the rest of the term,

For example, assuming + is AC, consider the following rewrite rule.

```
foo(X + Y, bar(X)) -> bar(X + Y)
```

If the matching of the second argument `bar(X)` of `foo` is tried first (instead of `X + Y`), the variable `X` is already bound when `X + Y` is to be matched; thus an excessive search with `X` is avoided. Moreover, since `X` is already bound, binding for `Y` is also determined uniquely — to match `X + Y` against $t_1 + \ldots + t_n$, if `X` is bound to, say, t_1, `Y` have to match the "rest" of the term, i.e. $t_2 + \ldots + t_n$. So in this case, no excessive search is needed.

Brunette generates the matching code for each subpattern in the following order[5]:

1. subpatterns that have no variables and no AC operators.
2. subpatterns that have variables but no AC operators.
3. subpatterns that have AC operators but no variables.
4. subpatterns that have both variables and AC operators.

Using Sort Information Consider the following rule for sorting natural numbers:

```
[ Nat < NatList ]
op . : NatList NatList -> NatList { assoc }
vars N N' : Nat
N . N' -> N' . N if N' < N
```

The operator "`.`" requires associative matching. If the sort information were not used and if — when flattened — a long term were given, a large number of trial-and-errors would occur. For example, the term "`0.1.2.3`" has 10 possible matches for the variables `N` and `N'`, if their sorts were not taken into account.

[5] In fact, it is a little more complicated.

N	N'
`0`	`1`
`0`	`1.2`
`0`	`1.2.3`
`0.1`	`2`
`0.1`	`2.3`
`0.1.2`	`3`
`1`	`2`
`1`	`2.3`
`1.2`	`3`
`2`	`3`

However, Brunette uses the sort information and deduces that any term whose outer-most operator is "`.`" can match neither **`N`** nor **`N'`** — since `N` is of sort `Nat`, which is not the coarity of "`.`". Hence Brunette generates a code sequence that try only 3 matches (with "`0.1`", "`1.2`", and "`2.3`").

6 Performance and Future Works

Brunette is written in C, and the machine instructions are compiled into structures in C. All the declarations and evaluations are processed via a simple command interpreter. Thus Brunette is fairly portable.

We made a preliminary assessment of Brunette using Ackermann's function and factorial. The measurements below were obtained on a PC with Pentium-Pro 200MHz running FreeBSD-2.2.2.

Firstly, Ackermann's function as shown below gives a measurement of Brunette's simple rewriting (i.e. rewriting in which no equational theories are involved) performance.

```
ack(0, M) -> s(M)
ack(s(M), 0) -> ack(M, s(0))
ack(s(M), s(N)) -> ack(M, ack(s(M), N))
```

Table 5 shows CPU time for computation. It also has some measurements of the original CafeOBJ engine[5], and of some other popular interpreters (Scheme VM48 and Perl 5.00401). Brunette achieves 450-460k rewrite/sec.

	Brunette (# of rewrites)	CafeOBJ	Scheme	Perl
ack(3, 4)	24ms (10307)	640ms	47ms	164ms
ack(3, 5)	99ms (42438)	2.39s	171ms	664ms
ack(3. 6)	374ms (172233)	10.2s	687ms	2.95s
ack(3, 7)	1520ms (693964)	42.0s	2900ms	11.0s

Table 5. ackermann's function

Secondly, the factorial function is used to see the performance of the AC matching. The function is defined with addition and multiplication declared or not declared AC. Column "non-AC" and "AC" in Table 6 shows the result. You may notice the big difference in the number of rewrites between non-AC version and AC version. Where does it come from? Consider these rules:

```
a) X * 0 -> 0
b) X * s(Y) -> (X * Y) + X
```

In the non-AC version, the term "`s(0) * 0`" only matches the rule a), so that it is always rewritten to "`0`". But in the AC version, this term can also match the rule b), rewritten to "`(0 * 0) + 0`", which would require two more rewrites to become "`0`". Rules for `+` also has a similar problem.

You can remedy this situation by modifying the rules as:

```
a) X * 0 -> 0
b') s(X) * s(Y) -> (s(X) * Y) + s(X)
```

Now terms of the form "`s(..) * 0`" would not match with the rule b'), so unnecessary rewriting explained above won't happen. Note that these rules no longer work without AC attributes (commutativity, to be more precise). The "opt-AC" column of Table 6 shows the result using this optimised program.

	non-AC	#rewrites	AC	#rewrites	opt-AC	#rewrites
fact(7)	17ms	7676	248ms	41087	57ms	7648
fact(8)	147ms	58078	2484ms	368572	469ms	58042

Table 6. factorial number

The last example is a sorting algorithm, where an associative operator is involved. It has a conditional rule

```
N . N' -> N' . N if N' < N
```

and "`.`" is associative. For sorting 20 numbers in reverse order, it took 40ms with 13680 rewrites, which is about 340k rewrite/sec.

Brunette is a result of an attempt to make a small, simple, portable and fast term rewriting engine. Compared with the original CafeOBJ engine, Brunette runs 10 to 50 times as faster; it also compares well with other similar engines. Even with associative/commutative matching, where Brunette resorts to a brute force method, its performance seems not too bad.

We began to use Brunette on larger and larger rule sets and terms, but so far found no problem. At the same time, we are improving Brunette's performance based on those experiments. Among other things, we are investigating how a memoisation mechanism, where the results of previous evaluation of subterms are cached, affects its performance. It is almost obvious that, with this mechanism, some recursively defined functions, such as factorial, are computed much faster. What we want to know is how much advantage workaday examples will gain.

The one major issue we have not confronted is modularisation, where a rewrite rule set is compiled incrementally, or separately compiled rule sets are merged into a whole. For a large rule set, this issue is as important as the speed of rewriting, if not more so. The current design of Brunette is difficult to modify to allow modular compilation, and a thorough revision is necessary.

References

1. Dershowitz, N. and Jouannaud, J.-P., "Rewrite Systems", *Handbook of Theoretical Computer Science, Vol.B: Formal Models and Semantics*, The MIT Press/Elsevier Science Publishers, 1990, pp.245–320
2. Meseguer, J., "A Logical Theory of Concurrent Objects and its Realization in the Maude Language", *Research Directions in Object-Based Concurrency*, The MIT Press, 1993
3. Goguen, J., Kirchner, C., Kirchner, H., Mégrelis, A., Meseguer, J., and Winkler, T., "An Introduction to OBJ-3", *Proc. of the 1st International Workshop on Conditional Term Rewriting Systems*, Lecture Notes in Computer Science 308, Springer-Verlag, 1987, pp.258–263
4. Kirchner, C., Kirchner, H., and Vittek, M., "Designing CLP Using Computational Systems", *Principles and Practice of Constraint Programming*, The MIT Press, 1995
5. Futatsugi, K. and Nakagawa, A.T., "An Overview of CAFE Specification Environment", *Proc. of the 1st International Conference on Formal Engineering Methods*, IEEE, 1997
6. Nakagawa, A.T., Sawada, T., and Futatsugi, K. *CafeOBJ Manual*, JAIST/SRA, 1997; available at
`ftp://www.sra.co.jp/pub/lang/CafeOBJ/Manual/manual.ps`
7. Ogata, K., Ohhara, K. and Futatsugi, K., "TRAM: An Abstract Machine for Order-Sorted Conditional Term Rewriting Systems", *Proc. of the 8th Int. Conf. on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 1232, Springer-Verlag, 1997, pp.335–338

Super-Closures

Frédéric Lang, Zino Benaissa and Pierre Lescanne

Laboratoire de l'Informatique du Parallélisme
École Normale Supérieure de Lyon
46, Allée d'Italie, F-69364 Lyon Cedex 07, FRANCE
E-mail: {Frederic.Lang, Pierre.Lescanne}@ens-lyon.fr
Phone: +33 (0)4 72 72 86 43 fax: +33 (0)4 72 72 80 80

Pacific Software Research Center
Oregon Graduate Institute of Science & technology
P.O. Box 91000 Portland, Oregon 97291-1000 USA
E-mail: benaissa@cse.ogi.edu
Phone: +1 503 690 1361 fax: +1 503 690 1548

Abstract. We propose abstract machines for lazy functional programming languages, based on the Krivine machine, and initially designed for eliding the creation of useless closures. We introduce the notion of *super-closure*, a data structure representing a *list of closures* in a compact way. We give four machines: the first three of them implement call by name. They illustrate a basic issue not directly related to sharing, namely *splitting*, and two possible solutions to this problem. Then, we give the proofs of correctness of these machines, using a weak λ-calculus with explicit substitution. Finally, we propose a machine implementing call by need and claim that it is correct w.r.t. the call-by-need strategy of environment machines. However, we also show that against our naive expections, our system has many weaknesses with respect to space consumption as well as execution speed.

Keywords. Abstract machines, functional programming, optimization, shared environments, explicit substitution.

Table of Contents

1 Introduction

Functional abstract machines that perform weak normal order evaluation – *i.e.* call by name (Plotkin 1975) or call by need (Launchbury 1993) – use closures to store unevaluated arguments (Fairbairn and Wray 1987; Crégut 1991; Curien 1991; Peyton Jones 1992). A closure is made of two parts, namely a pointer to a piece of code, and an environment which is a snapshot (possibly trimmed) of the configuration of the machine at the time the closure was built. Therefore, there are as many closure creations in a particular environment as unevaluated arguments encountered in this particular environment. We think that this number may be big in some applications, for instance automatically generated programs, and that therefore, such a strategy may be very expensive. We propose here to exploit this observation by investigating a new direction.

To present our ideas, we use a well known machine performing call by name and designed for pedagogy, namely the Krivine machine (Krivine 1985). This is a very straightforward machine not intended to design efficient and realistic implementations since *e.g.*, sharing, data-structures, recursion, *etc.* are missing. However its simplicity just makes it easier to point out the essence of our work. Then, we add sharing and discuss the related problems.

After some preliminaries in Section 2, we present the Krivine machine and develop the problematics in Section 3. Section 4 shows what exactly super-closures are and how the Krivine machine can be modified to handle them. We will see that this new machine raises a difficulty, namely *splitting*. In Section 5, we propose two solutions to address it: The first one is based on a program transformation close to what is known in the λ-calculus as expansion to η-long normal form. Unfortunately, this transformation is only applicable to simply typed terms. This is too restrictive in programming languages where expressivity really needs type polymorphism. The second solution is much more satisfactory, and uses what we call *access windows*. We then show in Section 6 that all our machines are sound *w.r.t.* the λ-calculus, and compute terms until weak head normal form. We then propose in Section 7 a machine performing call by need, *i.e.* in which evaluation is shared.

2 Preliminaries

In this section we set our notations. Naturals are denoted by n, m, p, q. If not stated otherwise, they denote any natural greater or equal to 0. In general, $n + 1$ denotes any natural greater or equal to 1 if n is not bound to any value, and $n + q$ denotes any natural greater or equal to n and/or q if at least one of them is not bound to a value.

In the following sections we use vectors. A vector of elements of type A denoted by $a, b, c, \ldots, a_0, \ldots, a_n$ may be denoted by $\vec{a}$ (any vector), or $\{a_n; a_{n+1} \ldots; a_m\}$ (any vector of size $m - n + 1$, empty if $m < n$), also denoted by $\vec{a}_n^m$. The empty vector is denoted by ϵ. Vectors may be appended such that $\vec{a}_0^i + \vec{a}_{i+1}^n = \vec{a}_0^n$ or single elements may be added to vectors such that $a_0 \cdot \vec{a}_1^n = \vec{a}_0^n$. The access to the n^{th} element of a vector is denoted by $\vec{a}(n)$ and defined such that $\vec{a}_i^{i+n+m}(n) = a_{i+n}$. The modification of the n^{th} element of a vector is denoted by $\vec{a}(n \leftarrow b)$ and defined such that $\vec{a}_i^{i+n+m}(n \leftarrow b) = \{a_i; \ldots; a_{i+n-1}; b; a_{i+n+1}; \ldots; a_{i+n+m}\}$.

We talk about λ-terms in (classical) de Bruijn notation (de Bruijn 1972). A λ-term in de Bruijn notation is either a *variable* or *index* $\underline{n}$ with n a natural, an *abstraction* λM with M a term, or an *application* $M_1 M_2$ with M_1 and M_2 two terms. The index $\underline{n}$ represents the variable bound by the $(n+1)^{th}$ λ encountered when looking backwards in the current context.

λ-terms may also be denoted in *vector notation*. The λ-term $M N_0 \ldots N_n$ where M is not an application is denoted by $M\vec{N}_0^n$. Abstractions are grouped as well, in which case the term $\lambda \ldots \lambda M$ (n nested abstractions) where M is not an abstraction is denoted by $\lambda^n M$. $\lambda^0 M$ denotes a term M that is not an abstraction. For parenthesis, we use the same convention as Barendregt (1984), *i.e.* λ has the lowest precedence and the application is left associative. Thus a β-redex has the form $(\lambda^{n+1} M)\vec{N}$, and $\lambda^{n+1} M\vec{N}$ must be read $\lambda^{n+1}(M\vec{N})$.

$$(\lambda M[e]\ ;\ N[e']\cdot s) \to (M[N[e']\cdot e]\ ;\ s) \qquad \textsf{(Lam)}$$
$$(MN[e]\ ;\ s) \to (M[e]\ ;\ N[e]\cdot s) \qquad \textsf{(App)}$$
$$(\underline{n}[M_0[e_0]\cdot \ldots \cdot M_n[e_n]\cdot e']\ ;\ s) \to (M_n[e_n]\ ;\ s) \qquad \textsf{(Access)}$$

Fig. 1. The Krivine machine.

τ_i^j is the renaming operator of de Bruijn λ-terms, defined as follows on terms in vector notation:

$$\tau_i^j(M\ \vec{N_0^n}) = \tau_i^j(M)\,\{\tau_i^j(N_0);\ldots;\tau_i^j(N_n)\}$$
$$\tau_i^j(\lambda^n M) = \lambda^n \tau_{i+n}^j(M)$$
$$\tau_i^j(\underline{n}) = \begin{cases}\underline{n+j} & \text{if } n \geq i\\ \underline{n} & \text{otherwise}\end{cases}$$

We define *mappings* as applications from elements $a_1,\ldots,a_n$ of a set A to elements $b_1,\ldots,b_n$ of a set B, and denote it by $[a_1 \mapsto b_1;\ldots;a_n \mapsto b_n]$. We call the set of the a_i the *domain* of the mapping. A mapping with empty domain is denoted by $[\ ]$. Mappings may be applied to elements of A, such that $[a_1 \mapsto b_1;\ldots;a_n \mapsto b_n]\,a_i = b_i$. A mapping ρ of type $A \to B$ may also be extended or updated with a new map $\rho' = [a_0 \mapsto b_0;\ldots;a_n \mapsto b_n]$ where $\forall 0 \leq i \leq n : a_i \in A$ and $b_i \in B$. This is denoted by $\rho[a_0 \mapsto b_0;\ldots;a_n \mapsto b_n]$ or $\rho\rho'$ where $\rho\rho'\ a = b_i$ if $a = a_i, 0 \leq i \leq n$ and $\rho\rho'\ a = \rho\, a$ otherwise. We may write $\rho[\vec{a_0^n} \mapsto \vec{b_0^n}]$ instead of $\rho[a_0 \mapsto b_0;\ldots;a_n \mapsto b_n]$, and $\rho\ \vec{c_0^n} = \{\rho\ c_0;\ldots;\rho\ c_n\}$.

3 The Krivine machine

The Krivine machine is shown on Figure 1. It works on *states* $(M[e]\ ;\ s)$ made of:

- The *code* M, a λ-term in the classical de Bruijn notation.
- The *environment* e, a vector of closures.
- The *stack* s, with the same structure as environments.

A *closure* is a pair of a code M and environment e written $M[e]$. When evaluating an application (App), a closure is built with the code of the argument and the current environment, and pushed on the stack. An abstraction λM represents a function that takes at least one argument. When evaluating (Lam), the closure on top of the stack is popped and bound to the variable $\underline{0}$. Any value which was at index $\underline{n}$ in the previous environment is now at index $\underline{n+1}$.

As one can see, if an application has n arguments, then n closures are built, which is illustrated by the following derivation:

$$(MN_1\ldots N_n[e]\ ;\ s) \to \ldots \to (M[e]\ ;\ N_1[e]\cdot \ldots \cdot N_n[e]\cdot s)$$

These closures all have the same structure, but their code. That feature happens in most existing machines: many similar data structures are built. We believe this work could be saved. For instance in STG, a let binding of n expressions involves the creation of n closures. We thus aim to study the behaviour of a machine in which those closures are grouped in a single structure, which we call *super-closure*. Therefore, our main aim is not to share e (even if it is actually partially shared) but to make the creation of n closures a unique task which can be done in a time which does not depend on n.

$$\begin{aligned}
(M\vec{N}[e]\ ;\ s) &\to (M[e]\ ;\ \langle \vec{N}, e\rangle \cdot s) && \text{(App)}\\
(\lambda^{n+1} M[e]\ ;\ \langle \vec{N}_0^{n+m+1}, f\rangle \cdot s) &\to (M[\langle \vec{N}_0^{n}, f\rangle \cdot e]\ ;\ \langle \vec{N}_{n+1}^{n+m+1}, f\rangle \cdot s) && \text{(Split)}\\
(\lambda^{n+m+1} M[e]\ ;\ \langle \vec{N}_0^{m}, f\rangle \cdot s) &\to (\lambda^{n} M[\langle \vec{N}_0^{m}, f\rangle \cdot e]\ ;\ s) && \text{(Lam)}\\
(\underline{n}[\langle \vec{N}_0^{n+m}, f\rangle \cdot e]\ ;\ s) &\to (N_m[f]\ ;\ s) && \text{(Access)}\\
(\underline{n+m+1}[\langle \vec{N}_0^{m}, f\rangle \cdot e]\ ;\ s) &\to (\underline{n}[e]\ ;\ s) && \text{(Skip)}
\end{aligned}$$

Fig. 2. The SC-machine.

4 A first machine using super-closures

First of all, we define *super-closures*. A super-closure is a structure composed with a code vector $\vec{N}$, and an environment e, denoted by $\langle \vec{N}, e\rangle$. In general we will not deal with super-closures where $\vec{N}$ is empty. Super-closures may appear in stacks or in environments. In a stack, $\langle \vec{N}_0^{n}, e\rangle$ can be interpreted as $N_0[e]\cdot\ldots\cdot N_n[e]$ in the Krivine machine, and in an environment as the reverse $N_n[e]\cdot\ldots\cdot N_0[e]$. The SC-machine is defined on Figure 2. If one compares it to the Krivine machine, one sees that only one super-closure is built instead of n closures. This means that only one structure is built, thus saving n units of time. Therefore, the environment e is instanciated only once for the $n+1$ arguments.

The SC-machine is thus a machine in which environements are (at least partially) shared. In that, it has to be compared with the TIM (Fairbairn and Wray 1987) which also shares environments. However, there are some differences between the two approaches: first of all, our machine inherits from the Krivine machine that it can reduce any term whereas the TIM is designed to only reduce supercombinators. However, we can imagine extensions of the TIM which reduce any term, using *e.g.*, linked environments: both machines would still be quite different. In our SC-machine, environments are shared by the mean of super-closures, whereas in the TIM they are shared through a heap of frames. This heap of frame is the structure that allows sharing of computations, whereas we still need to introduce a heap of (super-)closures to enable sharing. In fact our environments are shared with some limitation (see Section 7).

Super-closures are an obvious optimisation of the Krivine machine for call by name w.r.t. the number of steps of reduction performed to reduce a term. However, some steps of reduction are now more expensive, since for instance rules (Lam) and (Split) imply an n-ary abstraction check on whether the super-closure on top of the stack is smaller in size than n. For the same reason, access to variables is more expensive, since it may not be performed in a single step (rules (Access, Skip)). At last, our super-closures do not allow a classical optimisation that avoids the creation of closures of the kind $\underline{n}[\ldots\cdot M_n[e_n]\cdot\ldots]$, which are replaced by the already allocated closure $M_n[e_n]$. This loss may be a flaw since this optimisation would avoid many useless updates.

Moreover, we are aware that our vision is yet too simplist for a good analysis on call by need. Indeed, one trouble with sharing is in the (Split) rule. Suppose an abstraction needs less arguments than those available in the super-closure on top of the stack. Therefore, the super-closure has to be split to return only the required number of arguments to the environment. The risk is that super-closures are eventually split into super-closures of unitary code vectors, namely closures! This is not what we expect, since we have in mind to create closures only when necessary, *i.e.*, only when they are accessed (See Section 7 for more details on sharing). Moreover, because of (Split), a super-closure may not be shared since it may change after its creation, into a particular environment. We present in the next section two solutions to avoid the inconveniences of such a rule.

5 Avoiding split: two solutions

In this section, we propose two solutions to the split problem: The first one is based on a static transformation inspired from expansion to η-long normal form. However, this solution is not satisfactory in the case of a polymorphic typed language, and we give another solution that we call *access windows*.

5.1 First attempt: A static program transformation

A straightforward solution to the split problem is to consider only programs that *fit* super-closures. By fit, we mean that we impose abstractions to take as many or more arguments than provided in the super-closure on top of the stack. In other words, there is no more acceptable term of the form $(\lambda^n M)\vec{N}^{n+m+1}$ where $n > 0$. The question is whether such a program transformation exists.

In the simply typed λ-calculus, there exists a transformation known as expansion to η-long normal form (Snyder and Gallier 1989), which is a type directed transformation. It is defined on β-normal forms, as a way to unify them modulo η-conversion. To describe it, we first need to define what simple types are: a simple type (or, in what follows, a type) is either a constant type γ, or a functional type $\alpha_1 \to \alpha_2$ where α_1 and α_2 are simple types. Every expression e of the simply typed λ-calculus has a type, which can be denoted by $\alpha_1 \to \ldots \to \alpha_n \to \gamma$. We then say that e has *arity* n, that we denote by $\mathsf{ar}(e) = n$. This means that e can not be applied to more than n arguments. The expansion to η-long normal form is defined as follows: Let $\lambda^n \underline{m}\,\vec{N}_1^p$ be a simply typed term of arity $n+q$ in β-normal form, then its η-long normal form is the term

$$\lambda^{n+q}\underline{m+q}\,\{\tau_0^q(N_1);\ldots;\tau_0^q(N_p);\underline{q-1};\ldots;\underline{0}\}$$

This transformation inspires us to define an extended transformation, generalized on all simply typed terms, not only on β-normal forms. We call it the generalized expansion to η-long normal form, that for brevity we denote by gel. Let M be a simply typed term, we define $\mathsf{gel}(M)$ as follows:

- If M is $\lambda^n N\,\vec{N}_0^p$ with $\mathsf{ar}(M) = n+q$, then
$$\mathsf{gel}(M) = \lambda^{n+q}\mathsf{gel}(\tau_0^q(N))\,\{\mathsf{gel}(\tau_0^q(N_0));\ldots;\mathsf{gel}(\tau_0^q(N_p));\underline{q-1};\ldots;\underline{0}\}$$
- If M is $\lambda^{n+1}\underline{m}$ with $\mathsf{ar}(M) = n+q+1$, then
$$\mathsf{gel}(M) = \lambda^{n+q+1}\underline{m+q}\,\{\underline{q-1};\ldots;\underline{0}\}$$
- If M is $\underline{n}$, then $\mathsf{gel}(M) = M$

We make here two remarks: First of all, the reader should easily convince oneself that any term in vector notation can be transformed by this transformation, *i.e.* it matches one of these three cases. Second, it is worth to notice that variables are not expanded when they are not the body of an abstraction. This is because expansion of variables is useless for our purpose: a variable is only a code to access a closure, and has nothing to do with operations on the stack. Therefore, gel is not a strict generalization of expansion to η-long normal form.

Example 1. Consider the term $(\lambda\underline{0})\{\lambda\underline{0};\lambda\underline{0}\}$ of type $\iota \to \iota$ where ι is a basic type.

$$\begin{aligned}\mathsf{gel}((\lambda\underline{0})\{\lambda\underline{0};\lambda\underline{0}\}) &= \mathsf{gel}(\lambda\underline{0})\,\{\mathsf{gel}(\lambda\underline{0});\mathsf{gel}(\lambda\underline{0})\}\\ &= (\lambda^3\mathsf{gel}(\underline{2})\{\underline{1};\underline{0}\})\{\lambda^2\mathsf{gel}(\underline{1})\{\underline{0}\};\lambda\underline{0}\}\\ &= (\lambda^3\underline{2}\{\underline{1};\underline{0}\})\{\lambda^2\underline{1}\{\underline{0}\};\lambda\underline{0}\}\end{aligned}$$

Now dealing with η-expanded terms, we can define our new machine, called SCη-machine (Figure 3). In fact, this is no more than the SC-machine, without (Split).

However, this machine is not fully satisfactory. Indeed, as mentioned earlier, it is designed for simply typed terms, in particular terms without type variables. This means that our machine can not be used as basis for the implementation of a type polymorphic language. It is well known that typed functional programming without type polymorphism is not expressive enough to be of interest. See (Leroy 1992) for a

$$
\begin{aligned}
&(M\vec{N}[e]\ ;\ s) \to (M[e]\ ;\ \langle \vec{N}, e\rangle \cdot s) && \textsf{(App)}\\
&(\lambda^{n+m+1} M[e]\ ;\ \langle \vec{N}^{n+1}, f\rangle \cdot s) \to (\lambda^{m} M[\langle \vec{N}^{n+1}, f\rangle \cdot e]\ ;\ s) && \textsf{(Lam)}\\
&(\underline{n}[\langle \vec{N}_0^{n+m}, e\rangle \cdot f]\ ;\ s) \to (N_m[e]\ ;\ s) && \textsf{(Access)}\\
&(\underline{n+m+1}[\langle \vec{N}_0^{m}, e\rangle \cdot f]\ ;\ s) \to (\underline{n}[f]\ ;\ s) && \textsf{(Skip)}
\end{aligned}
$$

Fig. 3. The SCη-machine.

good introduction to type polymorphism in functional languages. Now the question is whether there exists a similar transformation to a more powerful type system, as a polymorphic one? The answer is no, since one does not know the actual arity of a type expression $\alpha_0 \to \ldots \to \alpha_n$ where α_n is a type variable.

At last, notice that any expression of functional type –thus of non null arity– is transformed to an abstraction, *i.e.* a value. Thus, its evaluation through the machine is trivial! This may have a bad effect on sharing, since a term M of arity $n+1$ whose evaluation might be shared can be transformed to an abstraction $\lambda^{n+1} M\{\underline{n}; \ldots ; \underline{0}\}$ in weak head normal form. Thus, the value of M can not be shared anymore.

5.2 Second attempt: Access windows

We propose here an alternative, called *access windows*. The main idea is that in (Split), there is no need to actually split the super-closure. Only the access into the super-closure must be restricted. Thus a super-closure may be shared, and the bounds between which access is permitted in the code vector of the super-closure is the specificity of a super-closure. We call these bounds an *access window*. In what follows, we are not going to express sharing, and we define a machine in which super-closures are copied. But we keep in mind that the ultimate goal is an actual implementation. We invite the reader to see super-closures as pointers to super-closures and to accept waiting til Section 7.

Let $\langle \vec{N}_0^n, e\rangle$ be a super-closure on which we want to restrict access to the part $\langle \vec{N}_i^{i+j}, e\rangle$. We denote it by $\langle \vec{N}_0^n, e\rangle_i^{i+j}$, where ${}_i^{i+j}$ is the access window of the super-closures. Figure 4 gives the dynamics of our machine with access windows.

$$
\begin{aligned}
&(M\vec{N}_0^n[e]\ ;\ s) \to (M[e]\ ;\ \langle \vec{N}_0^n, e\rangle_0^n \cdot s) && \textsf{(App)}\\
&(\lambda^{n+1} M[e]\ ;\ \langle \vec{N}, f\rangle_i^{i+n+m+1} \cdot s) \to (M[\langle \vec{N}, f\rangle_i^{i+n} \cdot e]\ ;\ \langle \vec{N}, f\rangle_{i+n+1}^{i+n+m+1} \cdot s) && \textsf{(Split)}\\
&(\lambda^{n+m+1} M[e]\ ;\ \langle \vec{N}, f\rangle_i^{i+n} \cdot s) \to (\lambda^{m} M[\langle \vec{N}, f\rangle_i^{i+n} \cdot e]\ ;\ s) && \textsf{(Lam)}\\
&(\underline{n}[\langle \vec{N}, e\rangle_i^{i+n+m} \cdot f]\ ;\ s) \to (N_{i+m}[e]\ ;\ s) && \textsf{(Access)}\\
&(\underline{n+m+1}[\langle \vec{N}, f\rangle_i^{i+m} \cdot e]\ ;\ s) \to (\underline{n}[e]\ ;\ s) && \textsf{(Skip)}
\end{aligned}
$$

Fig. 4. The SC-machine with access windows.

Syntax.

$$\begin{aligned} M, N, P &::= \lambda M \mid MN \mid \underline{n} \mid M[s] && \text{(Code)} \\ s, t &::= \mathsf{id} \mid M \cdot s && \text{(Substitution)} \end{aligned}$$

Rules.

$$\begin{aligned} (\lambda M)[s]\, N &\xrightarrow[\lambda\sigma_w]{} M[N \cdot s] && \text{(Bw)} \\ (MN)[s] &\xrightarrow[\lambda\sigma_w]{} M[s]N[s] && \text{(App)} \\ \underline{0}[M \cdot s] &\xrightarrow[\lambda\sigma_w]{} M && \text{(FVar)} \\ \underline{n+1}[M \cdot s] &\xrightarrow[\lambda\sigma_w]{} \underline{n}[s] && \text{(RVar)} \end{aligned}$$

Fig. 5. $\lambda\sigma_w$-calculus

Example 2. Here is the example of the reduction of the term $(\lambda\underline{0})\{\lambda\underline{0}; \lambda\underline{0}\}$:

$$\begin{aligned} ((\lambda\underline{0})\{\lambda\underline{0}; \lambda\underline{0}\}[\epsilon]\ ;\ \epsilon) &\rightarrow (\lambda\underline{0}[\epsilon]\ ;\ \langle\{\lambda\underline{0}; \lambda\underline{0}\}, \epsilon\rangle_0^1 \cdot \epsilon) && \text{(App)} \\ &\rightarrow (\underline{0}[\langle\{\lambda\underline{0}; \lambda\underline{0}\}, \epsilon\rangle_0^0 \cdot \epsilon]\ ;\ \langle\{\lambda\underline{0}; \lambda\underline{0}\}, \epsilon\rangle_1^1 \cdot \epsilon) && \text{(Split)} \\ &\rightarrow (\lambda\underline{0}[\epsilon]\ ;\ \langle\{\lambda\underline{0}; \lambda\underline{0}\}, \epsilon\rangle_1^1 \cdot \epsilon) && \text{(Access)} \\ &\rightarrow (\underline{0}[\langle\{\lambda\underline{0}; \lambda\underline{0}\}, \epsilon\rangle_1^1 \cdot \epsilon]\ ;\ \epsilon) && \text{(Lam)} \\ &\rightarrow (\lambda\underline{0}[\epsilon]\ ;\ \epsilon) && \text{(Access)} \end{aligned}$$

Access windows are undoubtly a better solution than transformation to generalized η long normal form, since it does not result in a loss in the amount of sharing that can be performed, and it is not tied to a particular type system. However, access windows are not for free, since they will increase the size of environments, each reference to an argument being made of two integers (an access window) in addition to a reference to a super-closure.

6 Correctness

We give in this section proofs of the soundness of our machines. Moreover, we prove that they reduce terms to their weak head normal form. To this aim, we need to translate states to terms (readback), and show that these properties are satisfied. Working with the λ-calculus would be quite laborious, since the substitution process is a meta-operation, thus making the translation non trivial. Instead, we use an explicit substitution calculus especially designed for weak reduction, namely $\lambda\sigma_w$ (Curien 1991; Hardin, Maranget and Pagano 1996). Since this calculus is sound *w.r.t.* the λ-calculus, soundness *w.r.t.* $\lambda\sigma_w$ implies soundness *w.r.t.* the λ-calculus. $\lambda\sigma_w$ is presented on figure 5. Notice that it is a weak calculus, *i.e.* not performing β-reduction under abstractions: Indeed, consider a pure λ-term M (here pure means without substitution) and give it a substitution $[\mathsf{id}]$. Then $M[\mathsf{id}]$ may not be reduced under abstractions. Moreover, unlike the weak λ-calculus, $\lambda\sigma_w$ is confluent (there is no critical pairs) and the $\lambda\sigma_w$-normal form of a pure term M, if it exists, is called the *weak normal form* of M. A *head normal form* is a $\lambda\sigma_w$ term of the form $\underline{n}[\mathsf{id}]M_1 \ldots M_n$ or $(\lambda^{n+1}M)[s]$.

6.1 Correctness of the sc-machine

Definition 1. We define the function μ translating states of the SC-machine to terms of $\lambda\sigma_w$ as follows:

$$\mu(M[e]\ ;\ s) = \kappa((M)[\rho(e)], s)$$
$$\text{where } \kappa(M, \epsilon) = M$$
$$\kappa(M, \langle \vec{N}_0^n, f\rangle \cdot s) = \kappa(M\, N_0[\rho(f)] \ldots N_n[\rho(f)], s)$$

and ρ is a function from environments of the SC-machine to substitutions of $\lambda\sigma_w$:

$$\rho(\epsilon) = \mathsf{id}$$
$$\rho(\langle \vec{N}_0^n, e\rangle \cdot f) = N_n[\rho(e)] \cdot \ldots \cdot N_0[\rho(e)] \cdot \rho(f)$$

We need the following lemma to establish the main property:

Lemma 2. *If $M \xrightarrow[\lambda\sigma_w]{} M'$ then $\kappa(M, s) \xrightarrow[\lambda\sigma_w]{} \kappa(M', s)$.*

Proof. Easy induction on s, to show that M is a subterm of $\kappa(M, s)$.

Theorem 3 (soundness). *If $(M[e]\ ;\ s) \to (M'[e']\ ;\ s')$ then $\mu(M[e]\ ;\ s) \xrightarrow[\lambda\sigma_w]{}\!\!\!\!\rightarrow \mu(M'[e']\ ;\ s')$.*

Proof. Systematic verification of each rule of the SC-machine. We give the proof for (Split) and leave the easy verification of the other rules to the reader.

$$\begin{aligned}
\mu(\lambda^{n+1}M[e]\ ;\ \langle \vec{N}_0^{n+m+1}, f\rangle \cdot s) &= \kappa((\lambda^{n+1}M)[\rho(e)], \langle \vec{N}_0^{n+m+1}, f\rangle \cdot s) \\
&= \kappa((\lambda^{n+1}M)[\rho(e)]\, N_0[\rho(f)] \ldots N_{n+m+1}[\rho(f)], s) \\
&\xrightarrow[\lambda\sigma_w]{}\!\!\!\!\rightarrow \kappa(M\big[N_n[\rho(f)] \cdot \ldots \cdot N_0[\rho(f)] \cdot \rho(e)\big]\, N_{n+1}[\rho(f)] \ldots N_{n+m+1}[\rho(f)], s) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{(by Lemma 2)} \\
&= \kappa(M[\rho(\langle \vec{N}_0^n, f\rangle \cdot e)], \langle \vec{N}_{n+1}^{n+m+1}, f\rangle \cdot s) \\
&= \mu(M[\langle \vec{N}_0^n, f\rangle \cdot e]\ ;\ \langle \vec{N}_{n+1}^{n+m+1}, f\rangle \cdot s)
\end{aligned}$$

To show that the SC-machine computes terms to their weak head normal form, we first establish the following lemma:

Lemma 4. *The stopping states of the SC-machine are of the form $(\lambda^m M[e]\ ;\ \epsilon)$ or $(\underline{n}[\epsilon]\ ;\ s)$.*

Proof. Trivial since there exists a rule for every other state.

Thus we can state the following:

Corollary 5. *The SC-machine reduces terms until reaching a state representing a $\lambda\sigma_w$ head normal form.*

Proof. Direct from Theorem 3, Lemma 4 and the translation function μ.

6.2 Correctness of the scη-machine

Soundness of the machine and soundness of **gel** are straightforward:

Theorem 6. *The SCη-machine is sound.*

Proof. Trivial, its rules are rules of the SC-machine.

Theorem 7. *Let M be a simply typed term. M and $\mathsf{gel}(M)$ are η-equivalent.*

Proof. This is trivial since gel only performs η-expansions.

Therefore, the only thing to show is that the machine computes terms to their $\lambda\sigma_w$ head normal form. The main lemma is the following:

Lemma 8. *The machine may not stop in a state of the form* $(\lambda^{n+1}M[e]\ ;\ \langle \vec{N}_0^{n+m+1}, f\rangle \cdot s)$.

Proof. Suppose there exists such a state. It is easy to show that $\lambda^{n+1}M$ has arity $n+1$. Thus, since the machine is sound *w.r.t.* β-reduction in the λ-calculus, and since β-reduction preserves simple types, there may not be more than $n+1$ arguments on the stack.

Corollary 9. *The* scη*-machine reduces terms to their* $\lambda\sigma_w$ *head normal form.*

6.3 Correctness of the sc-machine with access windows

The correctness for the sc-machine with access windows is very easy to prove with the following function:

Definition 10. ζ is the translation function from states of the sc-machine with access windows to states of the sc-machine defined as follows:

$$\zeta(M[e]\ ;\ s) = (M[\xi(e)]\ ;\ \xi(s))$$

where ξ is the translation function from environments or stacks of one machine to the other.

$$\xi(\epsilon) = \epsilon$$
$$\xi(\langle \vec{N}_0^{i+j+k}, e\rangle_i^{i+j} \cdot s) = \langle \vec{N}_i^{i+j}, e\rangle \cdot \xi(s)$$

Theorem 11. *If* $(M[e]\ ;\ s) \to (M'[e']\ ;\ s')$ *then* $\zeta(M[e]\ ;\ s) \to \zeta(M'[e']\ ;\ s')$.

Proof. Trivial verification on each rule of the sc-machine with access windows.

Lemma 12. *The stopping states of the* sc*-machine with access windows are of the form* $(\lambda^m M[e]\ ;\ \epsilon)$ *or* $(\underline{n}[\epsilon]\ ;\ s)$.

Proof. Same argument as for the sc-machine.

Corollary 13. *The* sc*-machine with access windows reduces terms to their* $\lambda\sigma_w$ *head normal form.*

7 Sharing

In this section, we add sharing to the sc-machine with access windows. We use a common technique, namely a *heap*, and *addresses* that allows sharing data structures representing arguments, with a notion of *update*. This is the same technique as used by *e.g.* Crégut (1991) or Peyton Jones (1992)[1], adapted to the use of super-closures.

States have now five components:

- The *code* M, still a λ-term in de Bruijn vector notation.
- The *environment* e, a vector of *addresses with access windows.*
- The *argument stack* s, like an environment, but a stack ...
- The *update stack* u, a stack of *update frames.*
- The *heap* h, a mapping from *addresses* to *heap objects.*

[1] Crégut uses actually marks on the heap, like in the TIM, instead of an update stack. But the difference is small.

Access windows are now associated to *addresses* of super-closures. a_i^j denotes the super-closure at address a, restricted to the access window $_i^j$. For sharing, we need to manipulate closures *and* super-closures. We call them *heap objects*. Indeed, a super-closure is a data structure that represents several closures sharing the same environement. When a member of a super-closure is evaluated, it has to be updated in the heap. Since its environment has changed, it cannot anymore be shared with the other members of the super-closure. Thus, one has to create a closure, and an indirection from the super-closure to the closure to enable sharing. Therefore, super-closures are a bit different as before: They are now composed of a vector of items that are either code or addresses of closures. Elements of this union type are denoted by $l, l_1, \ldots, l_n, \ldots$.

We need an operation on the heap which provides *fresh* addresses, where *fresh* intuitively means *unused*. In a state $(M[e] \; ; \; s \; ; \; u \; ; \; h)$, a fresh could be taken such that a does not belong to the domain of h. However, this is a bit too restrictive since in a real implementation, one also would like to reuse addresses pointing to useless heap objects.

We first give the *easy* rules, namely the rules that are immediate translations from rules of the machine without sharing:

$$(M\vec{N}_0^n[e] \; ; \; s \; ; \; u \; ; \; h) \rightarrow (M[e] \; ; \; a_0^n \cdot s \; ; \; u \; ; \; h[a \mapsto \langle \vec{N}_0^n, e\rangle]) \qquad a \text{ fresh} \qquad \text{(App)}$$

$$(\lambda^{n+1}M[e] \; ; \; a_i^{i+n+m+1} \cdot s \; ; \; u \; ; \; h) \rightarrow (M[a_i^{i+n} \cdot e] \; ; \; a_{i+n+1}^{i+n+m+1} \cdot s \; ; \; u \; ; \; h) \qquad \text{(Split)}$$

$$(\lambda^{n+m+1}M[e] \; ; \; a_i^{i+n} \cdot s \; ; \; u \; ; \; h) \rightarrow (\lambda^m M[a_i^{i+n} \cdot e] \; ; \; s \; ; \; u \; ; \; h) \qquad \text{(Lam)}$$

$$(\underline{n+m+1}[a_i^{i+m} \cdot e] \; ; \; s \; ; \; u \; ; \; h) \rightarrow (\underline{n}[e] \; ; \; s \; ; \; u \; ; \; h) \qquad \text{(Skip)}$$

The less trivial rules are the rules to update and access to data. The first rule of access is quite classical, as follows:

$$(\underline{n}[a_i^{i+n+m} \cdot f] \; ; \; s \; ; \; u \; ; \; h) \rightarrow (N[e] \; ; \; \epsilon \; ; \; (s, a, i+m) \cdot u \; ; \; h) \qquad \text{(Access1)}$$
$$\text{where } h \; a = \langle \vec{l}, e\rangle \text{ and } \vec{l}(i+m) = N$$

Since the argument may be updated, we push an update frame on the update stack, memorizing the current argument stack and the reference of the argument in the super-closure : the address of the super-closure and an index in this super-closure. However, one can imagine that the argument is *not updatable*, since it is already a value, or it is proven used at most once (Launchbury, Gill, Hughes, Marlow, Peyton Jones and Wadler 1993). In this case one can imagine an optimisation as follows:

$$(\underline{n}[a_i^{i+n+m} \cdot f] \; ; \; s \; ; \; u \; ; \; h) \rightarrow (N[e] \; ; \; s \; ; \; u \; ; \; h) \qquad \text{(Access1')}$$
$$\text{where } h \; a = \langle \vec{l}, e\rangle \text{ and } \vec{l}(i+m) = N \text{ not updateable, } \textit{e.g.}\text{, a value}$$

Anyway, in (Access1) or in (Access1'), the environment e needs to be copied, since it has to be modified with respect to the evaluation of this particular argument N. This could be yet another flaw of our machine, since until now we have tried to avoid copying environments. However, since copies of environments are performed *by need*, we may hope that that will not happen too often.

Update frames have to be popped from the update stack when the evaluation has reached a value, thus leading to the (Update) rule. Let $\lambda^{n+1}M$ be the value of the closure. There, we have to create a closure, since the environment changed with respect to the shared environment.

$$(\lambda^{n+1}M[e] \; ; \; \epsilon \; ; \; (s, a, i) \cdot u \; ; \; h) \rightarrow (\lambda^{n+1}M[e] \; ; \; s \; ; \; u \; ; \; h[a \mapsto \langle \vec{l'}, f\rangle; a' \mapsto \lambda^{n+1}M[e]]) \quad a' \text{ fresh} \quad \text{(Update)}$$
$$\text{where } h \; a = \langle \vec{l}, f\rangle \text{ and } \vec{l'} = \vec{l}(i \leftarrow a')$$

In this last rule, the environment e is once again copied since it is going to be modified in the particular context of evaluation of the function $\lambda^{n+1}M$. This is one of the main differences with the TIM, in which environments are always kept shared.

As suggested by rule (Update) there is now a second case to consider when accessing the argument bound to a variable: the case of an indirection to a closure. If a closure is already built for the argument to evaluate, then one can consider that it is already updated. Thus, the following rule:

$$(\underline{n}[a_i^{i+n+m} \cdot f] \; ; \; s \; ; \; u \; ; \; h) \to (N[e] \; ; \; s \; ; \; u \; ; \; h) \qquad \text{(Access2)}$$
$$\text{where } h \; a = \langle \vec{l}, e' \rangle$$
$$\vec{l}(i+m) = a' \text{ and } h \; a' = N[e]$$

That last rule thus suggests that to perform an access, the machine has to check whether there is an indirection or not. That unfortunately may lead to a severe loss in the performance of the machine. Moreover, the indirection itself is expensive since it necessitates two accesses in the heap instead of only one.

We do not give the correctness proof of the SC-machine with sharing, since this would uselessly extend this paper. However, the reader should convince oneself that one can easily define a translation function from the SC-machine with sharing to KP and show that one simulates the other. There already exist proofs of the correctness of KP, like in (Crégut 1991) or in (Sestoft 1997). Moreover, we are currently working on a new elegant proof of KP using the calculus $\lambda\sigma_w^a$ (Benaissa, Rose and Lescanne 1996; Benaissa 1997), which is $\lambda\sigma_w$ with global addresses. $\lambda\sigma_w^a$ is a good tool to reason about implementations, since it makes a clear distinction between the description of the calculus, and the description of the actual strategy. In this sense it is generic. The proof of Sestoft uses the natural semantics of Launchbury (Launchbury 1993) which is tied to a particular class of strategies (namely call-by-need strategies) and is not intended to modelize some important aspects of implementations like global addresses in particular. Therefore, one of the particularities of our proof is that not only we prove the correctness of our machine w.r.t. (lazy) λ-calculus, but also w.r.t. a particular call-by-need strategy, namely the call-by-need of environment machines. The whole proofs will be part of the first author's Phd, soon to be published.

8 Conclusion

We have proposed a new concept for implementing lazy functional languages, namely *super-closures*, and we have shown in a simple framework that it is correct.

However, we have left open issues of efficiency. We have shown that our machine may have severe flaws, but we have not compared the gains to the losses in practice. Are there cases in which super-closures are really an optimization? We are not yet able to answer this question as it strongly depends on the structure of programs and only experiments would prove its efficiency. It seems an evidence that super-closures may only be efficient for programs defining functions with a large number of arguments. This makes us think that use of closures and super-closures should be mixed in the same implementations, *i.e.* some applications should be allowed to create closures (small number of arguments and little environment) whereas other applications should create super-closures as a consequence of a static analysis. We are aware that there already exists static program transformations that try to avoid the creation of closures, such as *strictness analysis*, or *deadcode analysis*. Our future work must take them into consideration.

Moreover, we did not talk about memory allocation, namely trimming and garbage collection, which are big issues in real implementations. In particular we think that trimming is made difficult by the use of super-closures. It has been shown by Benaissa et al. (1996) that the *Spineless Tagless G-machine* of Peyton Jones (1992) does not leak space. Would it be the same with super-closures, and if not, what would be the loss on memory allocation? Indeed, super-closures may be very harmful for space consumption, since environments are bound to vectors of code. Therefore, it may be very difficult, if possible, to determine whether an address is garbage or not, or whether even a part of a super-closure is garbage or not. Consider the following example (due to an anonymous referee), namely the term (in the λ-calculus with names for readability) $(\lambda f.\lambda g.f)M[x]\,N[y]$ in an environment $\rho = [x \mapsto C_1; y \mapsto C_2]$ where $M[x]$ is a term that has only x as free variable and $N[y]$ a term that has only y as free variable. A super closure $\langle \{M[x]; N[y]\}, \rho \rangle$ is built but only the first component $M[x]$ is used. Moreover, ρ still contains x and y whereas the mapping to the value of y is useless and will remain. Now if y is bound to a large data-structure, this could dramatically increase the cost in space by introducing space leaking.

At last, we have not addressed recursion and data structures. We think that this problem is quite orthogonal to the creation of closures. Indeed, recursion can be realized with a μ operator that does not involve closure creation. The instruction (letrec $x = e_1$ in e_2) compiles to something like $((\lambda M_2)\,\mu M_1)$ where M_1

and M_2 are the compiled versions of e_1 and e_2, in which $\underline{0}$ is the representation of x. Then, the rule for μ binding is

$$(\mu M[e] \; ; \; \epsilon \; ; \; (s,a) \cdot u \; ; \; h) \rightarrow (M[a \cdot e] \; ; \; \epsilon \; ; \; (s,a) \cdot u \; ; \; h)$$

which does not involve closure creation. Moreover, super-closures can be very easily generalized to data structures, since they can be handled exactly like other arguments. Thus we have chosen not to address them in this paper.

To conclude, we may say that super-closures are probably not a good many purpose optimization for abstract machines since they raise many problems with costly solutions. However, we believe that there may exist specific cases in which super-closures are an optimization, and they are worth to be studied further. Anyway, we have tried to exploit this simple idea that closures could be grouped, and have shown the problems it raises. On another hand, we have shown after Hardin et al. (1996) that an explicit substitution calculus like $\lambda\sigma_w$ is a very useful tool for reasoning with implementations and proving their correctness. Its limitation is on its unability to express sharing, recursion and data structures. We are currently working on machine proofs that use the calculus $\lambda\sigma_w^a$ (Benaissa et al. 1996; Benaissa 1997), and that allow to prove correctness of implementations with respect to particular and precise strategies with sharing.

Acknowledgements

The authors wish to thank Kristoffer Rose and Judicaël Courant for useful discussions, and the anonymous referees for very relevant comments and suggestions.

References

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics.* Revised edn. North-Holland.

Benaissa, Z. (1997). *Les calculs de substitutions explicites comme fondement des implantations des langages fonctionnels.* PhD thesis. Université Henri Poincaré, Nancy 1.

Benaissa, Z., Rose, K. H. and Lescanne, P. (1996). Modeling sharing and recursion for weak reduction strategies using explicit substitution. *In* Kuchen, H. and Swierstra, D., eds., *8th PLILP—Symposium on Programming Language Implementation and Logic Programming.* Aachen, Germany. pp. 393–407.

de Bruijn, N. G. (1972). Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Koninkl. Nederl. Akademie van Wetenschappen* 75(5): 381–392.

Crégut, P. (1991). *Machines à environnement pour la réduction symbolique et l'évaluation partielle.* PhD thesis. Université de Paris 7.

Curien, P.-L. (1991). An abstract framework for environment machines. *Theoretical Computer Science* 82: 389–402.

Fairbairn, J. and Wray, S. C. (1987). TIM: A simple, lazy abstract machine to execute supercombinators. *In* Kahn, G., ed., *Functional Programming and Computer Architecture.* Portland, Oregon. Number 274 in *Lecture Notes in Computer Science.* pp. 34–45.

Hardin, T., Maranget, L. and Pagano, B. (1996). Functional back-ends within the lambda-sigma calculus. *Proceedings of International Conference on Functional Programming.* pp. 25–33.

Krivine, J.-L. (1985). Un interpréteur pour le λ-calcul. Notes de cours de DEA (Master lecture notes). Université de Paris 7.

Launchbury, J. (1993). A natural semantics for lazy evaluation. *20th POPL.* pp. 144–154.

Launchbury, J., Gill, A., Hughes, j., Marlow, S., Peyton Jones, S. and Wadler, P. (1993). Avoiding unnecessary updates. *Proceedings of the Glasgow Workshop on Functional Programming.* Springer-Verlag Workshops in Computing.

Leroy, X. (1992). *Typage polymorphe d'un langage algorithmique.* PhD thesis. Université de Paris 7.

Peyton Jones, S. L. (1992). Implementing lazy functional programming languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming* 2(2): 127–202.

Plotkin, G. D. (1975). Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science* 1: 125–159.

Sestoft, P. (1997). Deriving a lazy abstract machine. *Journal of Functional Programming* 7(3): 231–264.

Snyder, W. and Gallier, J. (1989). Higher order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation* 8(1 & 2): 101–140.

This article was typeset using the LaTeX macro package with the LLNCS2E class.

An Abstract Machine for Module Replacement

Chris Walton, Dilsun Kırlı, and Stephen Gilmore

Laboratory for Foundations of Computer Science, The University of Edinburgh, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK.

Abstract. In this paper we define an abstract machine model for the $m\lambda$ typed intermediate language. This abstract machine is used to give a formal description of the operation of run-time module replacement from the programming language Dynamic ML. The essential technical device which we employ for module replacement is a modification of two-space copying garbage collection.

1 Introduction

We have previously presented the high-level design of Dynamic ML, a variant of the Standard ML programming language which incorporates a facility for the replacement of modular components during program execution [1]. This useful facility builds upon existing compiler technology which permits the separate compilation of modular units of a Standard ML program. A suitable application problem for Dynamic ML would be the implementation of a distributed system where it is necessary to correct errors, improve run-time performance or reduce memory use, without interrupting the execution of the system.

Standard ML has a formal definition [2]. The Definition of Standard ML acts as a solid scientific platform where experiments in programming language design may be conducted. Any alteration to the Standard ML language such as ours should be investigated in the terms of the Definition. However, as readers of the Definition will know, it is silent on the topic of memory management except to say that "there are no (semantic) rules concerning disposal of inaccessible addresses" [2, page 42]. The Definition also separates the static and the dynamic semantics in such a way that the typing information inferred at compile-time is discarded before run-time. However, Dynamic ML needs some type information at run-time. These differences from Standard ML have motivated our work on a novel semantic model that would form a suitable setting for the formal definition of Dynamic ML. That model is presented in this paper.

Other authors have argued for the usefulness of a semantic model of memory management in making precise implementation notions such as memory leaks and tail recursion optimisation, developing suitable abstract machine models of memory management for this purpose [3]. Our abstract machine model for Dynamic ML serves a different purpose and this has led to the creation of a significantly different abstract machine than those used by previous authors. An essential feature of our machine is the modelling of user program exceptions, which other authors do not include.

2 A Model for Module Replacement

We introduce our first-order module-level replacement by an example to give the reader an informal understanding of its use in practice. Standard ML has interfaces called *signatures* and modules called *structures*. In our replacement model we allow the replacement of signatures by other signatures and structures by other structures, under reasonably generous conditions [1]. As our running example we consider the replacement of one implementation of a name table with another which is functionally equivalent but offers improved performance. Both implementations match the TABLE signature shown below.

```
signature TABLE = sig
    type table
    type name = string
    val empty: table
    val insert: name × table → table
    val member: name × table → bool
end;
```

We provide a facility for expressing such a replacement which ensures that the data values already present in memory cannot be used in ways which are not allowed by their type. The replacement operation is expressed by allowing the user to abstract over a Tbl structure which is specialised to implement a name table as a list of character strings. The Standard ML terminology for a structure abstraction is a *functor*. The functor body describes a structure which implements name tables as binary search trees and in addition contains functions to convert from the types of the given structure to the types of the new. We place the conversion functions inside an Install structure and follow a convention of mapping values from their old representation to their new one using functions which have the same identifier as the type which they update. This method of structure replacement is encoded as a Dynamic ML functor below.

```
functor InstallTable (Tbl: TABLE where type table = string list) :> TABLE =
struct
    type name = string
    datatype table = empty | node of table × name × table

    fun insert (s, empty) = node (empty, s, empty)
      | insert (s, node (l, v, r)) =
            if s < v then node (insert (s, l), v, r)
            else if s > v then node (l, v, insert (s, r)) else node (l, v, r)

    fun member (s, empty) = false
      | member (s, node (l, v, r)) =
            if s < v then member (s, l)
            else if s > v then member (s, r) else true

    structure Install = struct
        val name: Tbl.name → name = fn x ⇒ x
        val table: Tbl.table → table = List.foldr insert empty
    end
end;
```

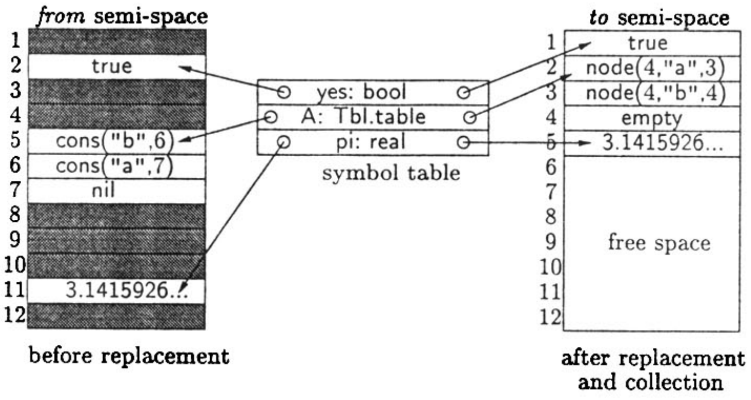


Fig. 1. Code replacement with type update

Through the use of the InstallTable functor, a Dynamic ML programmer could replace a structure which implemented tables as (either sorted or unsorted) lists with one which implemented them as binary search trees. This is an example of a very simple modification which would improve the performance of the insert and member operations. However, more sophisticated improvements would be made by the same method: defining a functor which maps the old implementation to the new one and provides functions to convert from the old types to the new. In both cases, it is critical that the types under replacement are abstract ones (with only the type identifier given in the signature) in order that functions outside the structure were not able to depend on a particular choice of concrete representation for a type, thereby preventing its replacement later.

We propose to perform the code replacement operation during garbage collection. A functor, such as the one shown above, is compiled separately. We then invoke the garbage collection operation extended with the application of the replacement functions from the Install structure to any values of the type under replacement. After completion of the copying with replacement, it is possible to dispose of the outdated version of the structure under modification (in the *from* semi-space), and switch to use the new version (in the *to* semi-space) which now contains the data values of the newly introduced replacement types. This is illustrated in Fig. 1 where a list representation of a name table containing the names *b* and *a* is replaced by the corresponding tree representation. Values of types not under replacement are unaltered: this includes values of built-in types such as booleans and real numbers.

The functions which are executed during code replacement are unrestricted Standard ML functions which may diverge upon application or raise an exception to signal an inability to continue processing. Our method of recovery is to rollback the garbage collection operation when any exception is raised. We revert to using the *from* semi-space of data values and the old types and we continue with the execution of the old program code.

Types	τ	::=	$tn \mid tn(\tau) \mid \{\overline{\tau}^k\} \mid \tau_1 \to \tau_2$
Program	P	::=	$(\overline{D}^k,\ X)$
Datatype	D	::=	**datatype** tn **of** $\overline{(con,\ \tau)}^k$
Expression	X	::=	**scon** *scon*
		$\mid$	**con** *con* $\mid$ **con** $(con,\ X)$
		$\mid$	**decon** $(con,\ X)$
		$\mid$	**excon** *excon* $\mid$ **excon** $(excon,\ X)$
		$\mid$	**dexcon** $(excon,\ X)$
		$\mid$	**record** $\overline{X}^k$
		$\mid$	**select** $(i,\ X)$
		$\mid$	**var** v
		$\mid$	**let** $v = X_1$ **in** X_2
		$\mid$	**fix** $\overline{(v,\ \tau) = X_1}^k$ **in** X_2
		$\mid$	**fn** $(v,\ \tau_1 \to \tau_2) = X$
		$\mid$	**app** $(X_1,\ X_2)$
		$\mid$	**switch** X_1 **case** $(c \overset{map}{\longmapsto} X_2,\ X_3)$
		$\mid$	**exception** $(excon,\ \tau)$ **in** X
		$\mid$	**raise** X
		$\mid$	**handle** X_1 **with** X_2

Fig. 2. Syntax of $m\lambda$ language

3 The $m\lambda$ Language

In order to formalise the replacement operation described in the previous section we first define a call-by-value lambda language $m\lambda$. This language is representative of a typical typed intermediate language used in the current state-of-the-art Standard ML compilers [4–7]. By basing replacement on such an intermediate language, we obtain an operation that is applicable to the whole of Standard ML, yet avoid a great deal of the complexity. For example, pattern matching is converted into switch statements by the higher-level match compiler. Furthermore, we can assume that the $m\lambda$ program is well-typed. For brevity, we have restricted our attention here to a purely-functional monomorphic lambda language. However, we note that including polymorphism and side-effects does not change the resulting replacement operation.

The syntax of the $m\lambda$ language is given in Fig. 2. The syntactic categories of the language include special constants of types unit, integer, real, and string; value constructors such as c_true; exception constructors such as e_match; and type names such as t_bool. Variables are bound uniquely to values generated by the evaluation of expressions. The types are constructor types (which may be either nullary or unary), record types, and function types. Constructor types include the basic types, as required by the special constants; value constructor types; and exception constructor types.

A program consists of a sequence of datatype declarations followed by a single expression. A datatype declaration consists of a unique type name and a sequence of typed constructors. The expressions divide into those for constructing and de-constructing values, defining and manipulating variables, and controlling the order of evaluation.

Notation: A set is defined by enumerating its members in braces, for example, $\overline{\overline{x}} = \{a,\ b,\ c,\ d\}$ with $\emptyset$ for the empty set. A sequence is an ordered list of members of a set, e.g. $\overline{x}^k = (a,\ b,\ c,\ a)$. The *i*th element of a non-empty sequence is written x^i, where $0 < i \leq k$. A finite-map from $\overline{x}^k$ to $\overline{y}^k$ is defined: $x \overset{map}{\longmapsto} y = \{x^1 \mapsto y^1,\ \ldots,\ x^k \mapsto y^k\}$ (the elements of $\overline{x}^k$ must be unique). The domain (*Dom*) and range (*Rng*) are the sets of elements of $\overline{x}^k$ and $\overline{y}^k$ respectively. A stack is written as a dotted sequence, e.g. $S = (a{\cdot}b{\cdot}c)$. The left-most element of the sequence is the top of the stack, and a pair of adjacent brackets () is used to represent the empty stack.

We use the meta-variables *scon* for special constants, *con* and *excon* for value and exception constructors with c ranging over all three of these and i over special constants of integer type. We use *tn* for type names and v for variables. We use p for type heap pointers and l for value heap locations.

4 The $m\lambda$ Abstract Machine

The dynamic semantics of $m\lambda$ is formalised in this section by a transition relation between states of an abstract machine. The organisation of our abstract machine has some features in common with the $\lambda_{\text{gc}}^{\rightarrow\forall}$ abstract machine [3] which is used in the formal description of the behaviour of the TIL/ML compiler. However, the resulting transitions differ considerably as $m\lambda$ is significantly different from $\lambda_{\text{gc}}^{\rightarrow\forall}$. One important way in which it differs is that $m\lambda$ does not adopt the named-form representation of expressions and types.

The syntax of the abstract machine is given in Fig. 3. The state of the machine is defined by a 4-tuple $(H,\ E,\ ES,\ RS)$ of a heap, an environment, an exception stack, and a result stack. The heap is used to store all the run-time objects of the program, while the environment provides a view of the heap relevant to the fragment of the program being evaluated (for example, a mapping between the bound variables currently in scope, and their values on the heap). The exception stack stores pointers to exception handling functions (closures). The result stack holds pointers to temporary results.

The heap consists of a type-heap mapping pointers to allocated types, and a value-heap mapping locations to allocated values. The heap types correspond directly to types in the $m\lambda$ language, and the heap values correspond to the heap types. Nullary constructors *scon*, *con*, and *excon* all have type *tn*. Unary constructors $con(l)$ and $excon(l)$ have type $tn(p)$. Records $\{\overline{l}^k\}$ have type $\{\overline{p}^k\}$, and closures $\langle\langle E,\ v,\ X\rangle\rangle$ have type $p_1 \rightarrow p_2$. The type heap and value heap are

Machine State	M	$::=$	$(H,\ E,\ ES,\ RS)$
Heap	H	$::=$	$(TH,\ VH)$
Type Heap	TH	$::=$	$p \overset{map}{\longmapsto} ty$
Heap Types	ty	$::=$	$tn \mid tn(p) \mid \{\overline{p}^k\} \mid p_1 \rightarrow p_2$
Value Heap	VH	$::=$	$l \overset{map}{\longmapsto} val$
Heap Values	val	$::=$	$scon$
		$\mid$	$con \mid con(l)$
		$\mid$	$excon \mid excon(l)$
		$\mid$	$\{\overline{l}^k\}$
		$\mid$	$\langle\!\langle E,\ v,\ X \rangle\!\rangle \mid \Omega$
Environment	E	$::=$	$(TE,\ CE,\ EE,\ VE)$
Type Env.	TE	$::=$	$\overline{\overline{tn}}$
Constructor Env.	CE	$::=$	$con \overset{map}{\longmapsto} p$
Exception Env.	EE	$::=$	$excon \overset{map}{\longmapsto} p$
Variable Env.	VE	$::=$	$v \overset{map}{\longmapsto} (l,\ p)$
Exception Stack	ES	$::=$	$() \mid (l,\ p)\cdot ES$
Result Stack	RS	$::=$	$() \mid (l,\ p)\cdot RS$

Fig. 3. Syntax of $m\lambda$ abstract machine

represented by finite-maps, as locations and pointers may be bound only once. It is important to note that we can only determine the shape of the data at a particular location by examining its corresponding type. Thus, each heap location will be paired with a heap pointer: $(l,\ p)$. This is essential for implementing tag-free garbage collection in the following section.

The following syntactic conventions are used for allocating heap objects: $H[l_1 \mapsto val_1,\ \ldots,\ l_k \mapsto val_k]$ allocates values $val_1, \ldots, val_k$ on the value heap, binding them to fresh locations $l_1, \ldots, l_k$, and $H[p_1 \mapsto \tau_1,\ \ldots,\ p_k \mapsto \tau_k]$ allocates types $\tau_1, \ldots, \tau_k$ on the type heap, binding them to fresh pointers $p_1, \ldots, p_k$. There are no corresponding operations for removing objects from the heap as this is achieved through garbage collection. However, the implementation of the fixed-point expression which is used to implement recursive functions requires a heap-update operation. As a special case, $H[l \mapsto \Omega]$ allocates a dummy closure on the value heap bound to a fresh location l. This location can subsequently be updated with a mapping to a new closure.

The environment records the allocation of $m\lambda$ values, mapping them to heap locations/pointers. As identifiers and variables are unique, their corresponding

$$
\begin{array}{lcl}
M & = & (H,\ E,\ ES,\ RS) \\
\\
H & = & (TH,\ VH) \\
TH & = & \{p_1 \mapsto \mathrm{t\_unit} \rightarrow \mathrm{t\_bool},\ p_2 \mapsto \mathrm{t\_unit} \rightarrow \mathrm{t\_exn}\} \\
VH & = & \emptyset \\
\\
E & = & (TE,\ CE,\ EE,\ VE) \\
TE & = & \{\mathrm{t\_unit},\ \mathrm{t\_int},\ \mathrm{t\_real},\ \mathrm{t\_string},\ \mathrm{t\_bool},\ \mathrm{t\_exn}\} \\
CE & = & \{\mathrm{c\_true} \mapsto p_1,\ \mathrm{c\_false} \mapsto p_1\} \\
EE & = & \{\mathrm{e\_match} \mapsto p_2,\ \mathrm{e\_bind} \mapsto p_2\} \\
VE & = & \emptyset \\
\\
ES, RS & = & (),\ ()
\end{array}
$$

Fig. 4. Initial machine state

environments are represented by finite-maps with the exception of the type environment where it is sufficient just to use a set for type names. The following notational conventions are used for extending the environment: $E[tn]$ adds tn to the type environment, $E[con \mapsto p]$ binds the constructor con to the heap pointer p in the constructor environment. Similarly, $E[excon \mapsto p]$, and $E[v \mapsto (l,\ p)]$ denote the binding of exception constructors and lambda variables respectively to heap pointers/locations in the environment. There are no operations for removing objects from the environment. However, unlike the heap, a copy of the current environment may be made at any time, for example by creating a closure. Thus, objects can effectively be removed from the environment by reverting to an old copy of the environment.

Execution of the abstract machine is defined by a transition system between machine states. The individual transitions are listed in Appendix A. The top-level transition has the form $(H,\ E,\ ES,\ RS,\ P) \Rightarrow (H',\ E',\ ES',\ RS')$, where P is an $m\lambda$ program, $(H,\ E,\ ES,\ RS)$ is the initial machine state (as illustrated in Fig. 4), and $(H',\ E',\ ES',\ RS')$ is the final machine state. This top-level transition decomposes into a sequence of transitions of the form $(H,\ E,\ ES,\ RS,\ D) \Rightarrow (H',\ E',\ ES',\ RS')$ for processing the datatypes D, followed by a sequence $(H,\ E,\ ES,\ RS,\ X) \Rightarrow (H',\ E',\ ES',\ RS')$ for evaluating the expression X.

There are three possible outcomes which can result from evaluating this expression. Firstly, the sequence may terminate normally yielding a single pair $(l,\ p)$ in the result stack which references the result. Secondly, the sequence may terminate prematurely, through an uncaught exception, yielding a pair $(l,\ p)$ at the top of the result stack which references the exception. Thirdly, the machine may encounter an infinite sequence of transitions and fail to terminate.

5 Garbage Collection with Replacement

In Section 2 we have explained how we extend the traditional two-space copying garbage collection to implement our replacement operation. In this section, we give the formal definition of this extended garbage collection used in the abstract machine defined in Section 4. The replacement operation has been presented in terms of the use of the modular constructs of Standard ML. However, for brevity we restrict our discussion here to the simpler non-modular language presented in Section 3.

We will consider the case where we are equipped with the information represented by a semantic object defined as follows:

$$RM ::= p_{\mathsf{old}} \overset{map}{\longmapsto} (l_{\mathsf{rep}},\ p_{\mathsf{rep}})$$

The domain of the replacement map $Dom(RM)$ is the set of the pointers to the types that are to be dynamically replaced. Each element p_{old} of the domain is mapped to a location/type-pointer pair $(l_{\mathsf{rep}},\ p_{\mathsf{rep}})$. The location contains the closure of the function which is to execute the replacement operation and the type-pointer points to the type which is to replace the old type.

In Dynamic ML this information is extracted from the result of the evaluation of the sub-structure Install which contains the user defined functions dedicated to the replacement operation. The replacement map obtained from the Install structure of our example would be as follows:

$$\{p_{Tbl.name} \mapsto (l_{name}, p_{name}),\quad p_{Tbl.table} \mapsto (l_{table}, p_{table})\}$$

We define garbage as the objects that are not reachable either directly or indirectly from the environment, exception stack, or result stack. Garbage collection may take place before or after any transition of the $m\lambda$ abstract machine dropping the bindings of the unreachable objects provided that this does not change the observable behaviour of the program.

Garbage collection is defined as a rewriting system between the configurations of our abstract machine $(S,\ RM,\ H_f,\ H_t)$. The replacement map denoted by RM is the auxiliary data structure which provides the information necessary for the replacement operation. The traditional two-space copying garbage collection corresponds to the case where RM is empty.

Initially, the scan stack S contains all of the pointers p and $(l,\ p)$ pairs in E, ES, and RS. Heap objects are copied from the semi-space H_f to the semi-space H_t until the scan stack is empty according to the rules listed in Appendix B.

We can incorporate the garbage collection operation in the dynamic semantics of our language explicitly by means of the following evaluation rule:

$$\frac{\begin{array}{c}(ES{\cdot}RS{\cdot}FE(E),\ RM,\ H_f,\ \emptyset) \Rightarrow^{*}_{\mathsf{gc}} ((),\ \emptyset,\ H_f,\ H_t)\\ (H_t,\ E,\ ES,\ RS,\ X)\ \Rightarrow\ (H_1,\ E_1,\ ES_1,\ RS_1)\end{array}}{(H_f,\ E,\ ES,\ RS,\ X)\ \Rightarrow\ (H_1,\ E_1,\ ES_1,\ RS_1)}$$

where $\Rightarrow_{\text{gc}}^{*}$ stands for the repeated application of the $\Rightarrow_{\text{gc}}$ rules. The informal understanding of the $\Rightarrow_{\text{gc}}$ rules is as follows:

Rule R0 is applied when the scan stack is empty. This signals the end of the garbage collection operation. The replacement map is discarded in order for subsequent garbage collections to operate correctly.

Rules $\text{R1}, \text{R1}^{\dagger}$ and $\text{R1}^{\ddagger}$ are applied when the top of the scan stack is a location/type-pointer pair $(l,\ p)$ and the value in the location has not yet been copied to the H_t semi-space, i.e. $l \notin Dom(H_t)$.

In R1 the type of the value reveals that it need not be replaced. As a result, the value in the H_f semi-space is copied to the H_t semi-space. The free locations and the type pointers of the allocated value are added to the scan stack.

$\text{R1}^{\dagger}$ and $\text{R1}^{\ddagger}$ are variants of R1 where the type of the value indicates that the value is to be replaced i.e. $p \in Dom(RM)$. Consecutive lookups in the replacement map and the heap yield the closure of the replacement function that is to be applied to the value currently being scanned. The code of the closure is evaluated in the environment extended by the binding of the scanned value. Note also that the disjoint union of the two semi-spaces is assumed as the heap because the code may be referring to some location or type-pointer that has already been copied.

There are two possible outcomes for the garbage collection operation. Either evaluation ends successfully or an exception is raised by one of the functions which is updating the values from the old type to the new one. These two cases are distinguished by inspecting the type of the most recent result which is at the top of the result stack. The first case is captured by $\text{R1}^{\dagger}$. The new value is copied to the H_t semi-space and the scan stack is arranged as in R1. The second case is captured by $\text{R1}^{\ddagger}$ where the top of the stack indicates that a top level exception has been raised. According to our implementation model we rollback the garbage collection operation and revert to using the H_f semi-space values. This is indicated by setting the scan stack to empty and identifying H_t with H_f. The replacement map is discarded as in R0.

R2 is applied when the top of the scan stack is a location/type-pointer pair and the value in the location has already been copied to the H_t semi-space. It simply skips this location and continues with the rest of the scan stack. R4 is exactly like R2 but skips over a type pointer instead of a location.

R3 and $\text{R3}^{\dagger}$ are applied when the top of the scan stack is a type pointer and the type pointer has not yet been copied to the H_t semi-space. R3 deals with the case where the type need not be replaced. The free pointers of the allocated type are added to the scan stack and the old representation of the type is copied to the H_t semi-space. $\text{R3}^{\dagger}$ deals with the case where the old representation of the type is to be replaced by the new representation.

The functions FE, FP and FL employed in the rewriting rules compute the free location/type-pointer pairs $(l,\ p)$ and type-pointers p. They are given in Fig. 5.

$$FE(E) = Rng(CE) \cdot Rng(EE) \cdot Rng(VE)$$

$$\begin{aligned} FP(tn) &= () \\ FP(tn(p)) &= (p) \\ FP(\{\overline{p}^k\}) &= (p^1 \cdots p^k) \\ FP(p_1 \rightarrow p_2) &= FP(p_1) \cdot FP(p_2) \end{aligned}$$

$$\begin{array}{ll} FL(H,\ l,\ tn) = () & \\ FL(H,\ l_1,\ tn(p)) = (l_2,\ p) & \text{where } tn = \text{t\_exn and } H(l_1) = excon(l_2) \\ FL(H,\ l_1,\ tn(p)) = (l_2,\ p) & \text{where } tn \neq \text{t\_exn and } H(l_1) = con(l_2) \\ FL(H,\ l,\ \{\overline{p}^k\}) = (l^1,\ p^1) \cdots (l^k,\ p^k) & \text{where } H(l) = \{\overline{l}^k\} \\ FL(H,\ l,\ p_1 \rightarrow p_2) = FE(E) & \text{where } H(l) = \langle\!\langle E,\ v,\ X \rangle\!\rangle \end{array}$$

Fig. 5. Auxiliary functions for garbage collection

6 Practicality

Users of state-of-the-art compilers for modern programming languages have become accustomed to complex program analyses which safely deliver impressive performance benefits in terms of run-time and memory usage while simultaneously offering greater access to a more sophisticated model of computation which incorporates advanced features such as remote evaluation or code mobility. In this setting it is all too easy to invent a new paradigm for program execution and to claim that it can be implemented efficiently because modern compilers and run-time systems offer so much functionality and convenience. In this section we would like to provide a more concrete explanation of the key implementation technology which could be used to provide an efficient implementation of the code replacement operation which we have described.

Languages in the Standard ML family are strongly typed. In order to enforce the application of the type-checking stage these language make a strict distinction between *elaboration* and *evaluation*, insisting that programs which have not successfully elaborated cannot be evaluated at all. The rigid ordering of these two stages prohibits the execution of any programs which attempt to use data values in ways which are not allowed by their type and thus eliminates a large number of software errors which would manifest themselves at run-time if working in an untyped programming language. However, several authors have observed that two stages are not enough for complex applications such as program generators. This has led to approaches such as the *multi-stage* programming paradigm for MetaML [8], *staged type inference* [9] and the *staged compilation* paradigm for the language Modal ML [10]. The last of these is the most closely related to our own approach because it has demonstrated the effectiveness of the use of run-time code generation by Lee and colleagues in the development of the Fabius compiler for ML [11]. Using this technology it is possible for us to eliminate the run-time penalties incurred by the use of abstract types in module specifications by exploiting the underlying representation of an abstract type and re-compiling at run-time when the replacement module is available. Further, many other benefits come from the use of run-time code generation including those associated with *partial evaluation* [12] since it is possible to take advantage of values which are not known until run-time. Other standard compiler optimisations such as elimination of array-bounds checking and loop unrolling also become more profitable in this setting.

Our discussion of module replacement has been exclusively framed in the context of Standard ML but the same idea has recently been investigated by other authors working with other languages. Andersson and colleagues [13] have considered the dynamic replacement of loaded classes in the Java run-time system. Their approach to implementation differs from ours in that they perform replacement of objects of the outdated class as they are accessed, meaning that both versions of the class are active at the same time. Replaced objects are garbage-collected as the computation proceeds and whenever all of the objects of the old version of the class have been replaced the class object will have no more references and it can then be garbage-collected also.

It might seem that the idea of dynamic replacement is better suited to an embedded systems language for a system with a high availability requirement, making Java the better choice for investigating dynamic code replacement because that its intended application domain. Although we admire Java as a useful, soundly-engineered product the absence of a well-understood theory for the language makes it less well-suited for this issue. Other researchers are also considering the use of Standard ML in areas such as these [14].

7 Conclusions

Modern compilers for higher-order typed programming languages use typed intermediate languages to structure the compilation process. We have provided an abstract machine definition of a small functional language which is representative of these. This has allowed us to define precisely the operation of dynamic module replacement which is used in Dynamic ML.

In composing the Definition of Standard ML, the authors chose not to give an account of the operation of garbage collection, which most compilers for that language provide. This was the right decision when focusing upon the abstract description of a sophisticated high-level language such as Standard ML. Our concern was to describe part of the operation of an executing computation, with access to values described by concrete manifestations.

The use of an abstract machine notation has allowed us to isolate the novel feature of interest from our language. We have presented its definition separately from other aspects such as syntax and type-correctness. For our purposes, the use of an abstract machine has established the right level of detail. In addition, it provides an implementor with an unambiguous and precise description of the operation of module-level code replacement.

Acknowledgements

Dilsun Kırlı is supported by a University of Edinburgh scholarship from the Department of Computer Science. Chris Walton is supported by an EPSRC postgraduate studentship.

References

1. S. Gilmore, D. Kırlı, and C. Walton. Dynamic ML without Dynamic Types. Technical Report ECS-LFCS-97-378, Laboratory for Foundations of Computer Science, The University of Edinburgh, 1997.
2. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised 1997.* The MIT Press, 1997.
3. G. Morrisett and R. Harper. Semantics of Memory Management for Polymorphic Languages. Technical report, School of Computer Science, Carnegie Mellon University, 1996. Also published as Fox Memorandum CMU-CS-FOX-96-04.
4. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimising compiler for ML. In *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, 1996.
5. Z. Shao. An Overview of the FLINT/ML Compiler. Technical report, Department of Computer Science, Yale University, 1997.
6. M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. Olesen, P. Sestoft, and P. Bertelsen. Programming with Regions in the ML-Kit. Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen, 1997.
7. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Third ACM SIGPLAN International Conference on Functional Programming*, Baltimore, 1998.
8. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, June 1997.
9. M. Shields, T. Sheard, and S. Peyton Jones. Dynamic typing as staged type inference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, January 1998.
10. P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and Modal-ML. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 224–235, Montreal, Canada, June 1998.
11. P. Lee and M. Leone. Optimising ML with run-time code generation. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 137–148, Philadelphia, Pennsylvania, May 1996.
12. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
13. J. Andersson, M. Comstedt, and T. Ritzau. Run-time support for dynamic Java architectures. In *ECOOP'98 Workshop on Object-Oriented Software Architectures*, Brussels, July 1998.
14. R. Pucella. Reactive programming in Standard ML. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 48–57, Chicago, USA, May 1998. IEEE Computer Society Press.

A Abstract Machine Definition

A.1 Programs

$$\frac{\begin{array}{l} P = (\overline{D}^{k}, X) \\ (H, E, ES, RS, D^1) \Rightarrow (H_1, E_1, ES, RS) \ldots \\ \ldots (H_{k-1}, E_{k-1}, ES, RS, D^k) \Rightarrow (H_k, E_k, ES, RS) \\ (H_k, E_k, ES, RS, X) \Rightarrow (H_{k+1}, E_{k+1}, ES_1, RS_1) \end{array}}{(H, E, ES, RS, P) \Rightarrow (H_{k+1}, E_{k+1}, ES_1, RS_1)}$$

A.2 Datatypes

$$\frac{}{\begin{array}{l} (H, E, ES, RS, \textbf{datatype}\ tn\ \textbf{of}\ \overline{(con, \tau)}^{k}) \Rightarrow \\ \quad (H[p_1 \mapsto \tau^1 \to tn, \ldots, p_k \mapsto \tau^k \to tn], \\ \quad E[tn][con^1 \mapsto p_1, \ldots, con^k \mapsto p_k], ES, RS) \end{array}}$$

A.3 Expressions

$$\frac{}{(H, E, ES, RS, \textbf{scon}\ scon) \Rightarrow (H[l \mapsto scon][p \mapsto \tau_{scon}], E, ES, (l, p) \cdot RS)}$$

$$\frac{E(con) = p_1 \qquad H(p_1) = p_2 \to p_3}{(H, E, ES, RS, \textbf{con}\ con) \Rightarrow (H[l_1 \mapsto con], E, ES, (l_1, p_3) \cdot RS)}$$

$$\frac{\begin{array}{l} (H, E, ES, RS, X) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS) \\ E(con) = p_2 \qquad H_1(p_2) = p_3 \to p_4 \end{array}}{(H, E, ES, RS, \textbf{con}\ (con, X)) \Rightarrow (H_1[l_2 \mapsto con(l_1)], E, ES, (l_2, p_4) \cdot RS)}$$

$$\frac{\begin{array}{l} (H, E, ES, RS, X) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS) \\ E(con) = p_2 \qquad H_1(p_2) = p_3 \to p_4 \qquad H_1(l_1) = con(l_2) \end{array}}{(H, E, ES, RS, \textbf{decon}\ (con, X)) \Rightarrow (H_1, E, ES, (l_2, p_3) \cdot RS)}$$

$$\frac{E(excon) = p_1 \qquad H(p_1) = p_2 \to p_3}{(H, E, ES, RS, \textbf{excon}\ excon) \Rightarrow (H[l_1 \mapsto excon], E, ES, (l_1, p_3) \cdot RS)}$$

$$\frac{\begin{array}{l} (H, E, ES, RS, X) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS) \\ E(excon) = p_2 \qquad H_1(p_2) = p_3 \to p_4 \end{array}}{\begin{array}{l} (H, E, ES, RS, \textbf{excon}\ (excon, X)) \Rightarrow \\ \quad (H_1[l_2 \mapsto excon(l_1)], E, ES, (l_2, p_4) \cdot RS) \end{array}}$$

$$\frac{\begin{array}{l} (H, E, ES, RS, X) \Rightarrow (H_1, E, ES, (l_1, p_1) \cdot RS) \\ E(excon) = p_2 \qquad H_1(p_2) = p_3 \to p_4 \qquad H_1(l_1) = excon(l_2) \end{array}}{(H, E, ES, RS, \textbf{dexcon}\ (excon, X)) \Rightarrow (H_1, E, ES, (l_2, p_3) \cdot RS)}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X^1) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS)\ \ldots\\ \ldots\ (H_{k-1},\ E,\ ES,\ (l_{k-1},\ p_{k-1})\cdots(l_1,\ p_1)\cdot RS,\ X^k) \Rightarrow\\ \quad (H_k,\ E,\ ES,\ (l_k,\ p_k)\cdots(l_1,\ p_1)\cdot RS)\end{array}}{\begin{array}{l}(H,\ E,\ ES,\ RS,\ \textbf{record}\ \overline{X}^k) \Rightarrow\\ \quad (H[l \mapsto \{l_1,\ \ldots,\ l_k\}][p \mapsto \{p_1,\ \ldots,\ p_k\}],\ E,\ ES,\ (l,\ p)\cdot RS)\end{array}}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS)\\ H_1(l_1) = \{\bar{l}^k\} \qquad H_1(p_1) = \{\bar{p}^k\}\end{array}}{(H,\ E,\ ES,\ RS,\ \textbf{select}\ (i,\ X)) \Rightarrow (H_1,\ E,\ ES,\ (l',\ p')\cdot RS)}$$

$$\frac{E(v) = (l,\ p)}{(H,\ E,\ ES,\ RS,\ \textbf{var}\ v) \Rightarrow (H,\ E,\ ES,\ (l,\ p)\cdot RS)}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X_1) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS)\\ (H_1,\ E[v \mapsto (l_1,\ p_1)],\ ES,\ RS,\ X_2) \Rightarrow (H_2,\ E_2,\ ES,\ RS_2)\end{array}}{(H,\ E,\ ES,\ RS,\ \textbf{let}\ v = X_1\ \textbf{in}\ X_2) \Rightarrow (H_2,\ E,\ ES,\ RS_2)}$$

$$\frac{\begin{array}{l}(H[l_f^1 \mapsto \Omega,\ \ldots,\ l_f^k \mapsto \Omega][p_f^1 \mapsto \tau^1,\ \ldots,\ p_f^k \mapsto \tau^k],\ ES,\ RS,\ X_1^1) \Rightarrow\\ \quad (H_1,\ E_1,\ ES,\ (l_1,\ p_1)\cdot RS)\\ (H_1[l_f^1 \overset{upd}{\longmapsto} l_1],\ E_1,\ ES,\ RS,\ X_1^2) \Rightarrow (H_2,\ E_1,\ ES,\ (l_2,\ p_2)\cdot RS)\ \ldots\\ \ldots\ (H_{k-1}[l_f^{k-1} \overset{upd}{\longmapsto} l_{k-1}],\ E_1,\ ES,\ RS,\ X_1^k) \Rightarrow (H_k,\ E_1,\ ES,\ (l_k,\ p_k)\cdot RS)\\ (H_k[l_f^k \overset{upd}{\longmapsto} l_k],\ E_1,\ ES,\ RS,\ X_2) \Rightarrow (H_{k+1},\ E_1,\ ES,\ RS_{k+1})\end{array}}{(H,\ E,\ ES,\ RS,\ \textbf{fix}\ (v,\ \tau) = \overline{X_1}^k\ \textbf{in}\ X_2) \Rightarrow (H_{k+1},\ E,\ ES,\ RS_{k+1})}$$

$$\frac{}{\begin{array}{l}(H,\ E,\ ES,\ RS,\ \textbf{fn}\ (v,\ \tau_1 \to \tau_2) = X) \Rightarrow\\ \quad (H[l \mapsto \langle\!\langle E,\ v,\ X\rangle\!\rangle][p \mapsto \tau_1 \to \tau_2],\ E,\ ES,\ (l,\ p)\cdot RS)\end{array}}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X_1) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS)\\ (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS,\ X_2) \Rightarrow (H_2,\ E,\ ES,\ (l_2,\ p_2)\cdot(l_1,\ p_1)\cdot RS)\\ H_2(l_1) = \langle\!\langle E_1,\ v,\ X_3\rangle\!\rangle\\ (H_2,\ E_1[v \mapsto (l_2,\ p_2)],\ ES,\ RS,\ X_3) \Rightarrow (H_3,\ E_2,\ ES,\ RS_3)\end{array}}{(H,\ E,\ ES,\ RS,\ \textbf{app}\ (X_1,\ X_2)) \Rightarrow (H_3,\ E,\ ES,\ RS_3)}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X_1) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS)\\ cmap = c \overset{map}{\longmapsto} X_2 \qquad H(l_1) = val\\ X_4 = \text{if } val \in Dom(cmap) \text{ then } cmap(val) \text{ else } X_3\\ (H_1,\ E,\ ES,\ RS,\ X_4) \Rightarrow (H_2,\ E,\ ES,\ RS_2)\end{array}}{(H,\ E,\ ES,\ RS,\ \textbf{switch}\ X_1\ \textbf{case}\ (cmap,\ X_3)) \Rightarrow (H_2,\ E,\ ES,\ RS_2)}$$

$$\frac{(H[p \mapsto \tau],\ E[excon \mapsto p],\ ES,\ RS,\ X) \Rightarrow (H_1,\ E_1,\ ES,\ RS_1)}{(H,\ E,\ ES,\ RS,\ \textbf{exception}\ (excon,\ \tau)\ \textbf{in}\ X) \Rightarrow (H_1,\ E,\ ES,\ RS_1)}$$

$$\frac{(H,\ E,\ (),\ RS,\ X) \Rightarrow (H_1,\ E,\ (),\ RS_1)}{(H,\ E,\ (),\ RS,\ \textbf{raise}\ X) \Rightarrow (H_1,\ E,\ (),\ RS_1)}$$

$$\frac{\begin{array}{l}(H,\ E,\ (l_1,\ p_1)\cdot ES,\ RS,\ X) \Rightarrow (H_1,\ E,\ (l_1,\ p_1)\cdot ES,\ RS_1)\\ H_1(l_1) = \langle\!\langle E_1,\ v,\ X_1\rangle\!\rangle\\ (H_1,\ E_1[v \mapsto (l_1,\ p_1)],\ ES,\ RS,\ X_1) \Rightarrow (H_2,\ E_2,\ ES,\ RS_2)\end{array}}{(H,\ E,\ (l_1,\ p_1)\cdot ES,\ RS,\ \mathbf{raise}\ X) \Rightarrow (H_2,\ E,\ ES,\ RS_2)}$$

$$\frac{\begin{array}{l}(H,\ E,\ ES,\ RS,\ X_1) \Rightarrow (H_1,\ E,\ ES,\ (l_1,\ p_1)\cdot RS)\\ (H_1,\ E,\ (l_1,\ p_1)\cdot ES,\ RS,\ X_2) \Rightarrow (H_2,\ E,\ ES_2,\ RS_2)\end{array}}{(H,\ E,\ ES,\ RS,\ \mathbf{handle}\ X_1\ \mathbf{with}\ X_2) \Rightarrow (H_2,\ E,\ ES,\ RS_2)}$$

B Garbage Collection with Replacement

$$\frac{}{((),\ RM,\ H_f,\ H_t) \Rightarrow_{\mathbf{gc}} ((),\ \emptyset,\ H_f,\ H_t)} \quad \text{(R0)}$$

$$\frac{l \notin Dom(H_t) \qquad p \notin Dom(RM) \qquad H_f(l) = val}{((l,\ p)\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathbf{gc}} (p\cdot FL(H_f,\ l,\ p)\cdot S,\ RM,\ H_f,\ H_t[l \mapsto val])} \quad \text{(R1)}$$

$$\frac{\begin{array}{l}l \notin Dom(H_t) \qquad p \in Dom(RM)\\ RM(p) = (l_{\mathrm{rep}},\ p_{\mathrm{rep}}) \qquad H(l_{\mathrm{rep}}) = \langle\!\langle E_1,\ v,\ X\rangle\!\rangle\\ (H_f \uplus H_t,\ E_1[v \mapsto (l,\ p)],\ ES,\ RS,\ X) \Rightarrow (H_2,\ E_2,\ ES,\ (l_{\mathrm{new}},\ p_{\mathrm{new}})\cdot RS)\\ H_2(l_{\mathrm{new}}) = val \qquad H_2(p_{\mathrm{new}}) \neq \mathsf{t\_exn}\end{array}}{((l,\ p)\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathbf{gc}} (p\cdot FL(H_f,\ l,\ p)\cdot S,\ RM,\ H_f,\ H_t[l \mapsto val])} \quad \text{(R1}^{\dagger}\text{)}$$

$$\frac{\begin{array}{l}l \notin Dom(H_t) \qquad p \in Dom(RM)\\ RM(p) = (l_{\mathrm{rep}},\ p_{\mathrm{new}}) \qquad H(l_{\mathrm{rep}}) = \langle\!\langle E_1,\ v,\ X\rangle\!\rangle\\ (H_f \uplus H_t,\ E_1[v \mapsto (l,\ p)],\ ES,\ RS,\ X) \Rightarrow (H_2,\ E_2,\ ES,\ (l_{\mathrm{new}},\ p_{\mathrm{new}})\cdot RS)\\ H_2(p_{\mathrm{new}}) = \mathsf{t\_exn}\end{array}}{((l,\ p)\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathbf{gc}} ((),\ \emptyset,\ H_f,\ H_f)} \quad \text{(R1}^{\ddagger}\text{)}$$

$$\frac{l \in Dom(H_t)}{((l,\ p)\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathbf{gc}} (S,\ RM,\ H_f,\ H_t)} \quad \text{(R2)}$$

$$\frac{p \notin Dom(H_t) \qquad p \notin Dom(RM) \qquad H_f(p) = ty}{(p\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathbf{gc}} (FP(ty)\cdot S,\ RM,\ H_f,\ H_t[p \mapsto ty])} \quad \text{(R3)}$$

$$\frac{\begin{array}{l}p \notin Dom(H_t) \qquad p \in Dom(RM)\\ RM(p) = (l_{\mathrm{rep}},\ p_{\mathrm{rep}}) \qquad H_f(p_{\mathrm{rep}}) = ty\end{array}}{(p\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathbf{gc}} (FP(ty)\cdot S,\ RM,\ H_f,\ H_t[p \mapsto ty])} \quad \text{(R3}^{\dagger}\text{)}$$

$$\frac{p \in Dom(H_t)}{(p\cdot S,\ RM,\ H_f,\ H_t) \Rightarrow_{\mathbf{gc}} (S,\ RM,\ H_f,\ H_t)} \quad \text{(R4)}$$

Visualizing Principles of Abstract Machines by Generating Interactive Animations

Stephan Diehl, Thomas Kunze [*]

Abstract

In this paper we describe the design rationale of GANIMAM, a web-based system which generates interactive animations of abstract machines from specifications. Common principles of abstract machines come into play at three levels: the design of the specification language, the choice of graphical annotations to visualize higher-level abstractions and the use of the system to explore and better understand known and detect new principles.

Introduction

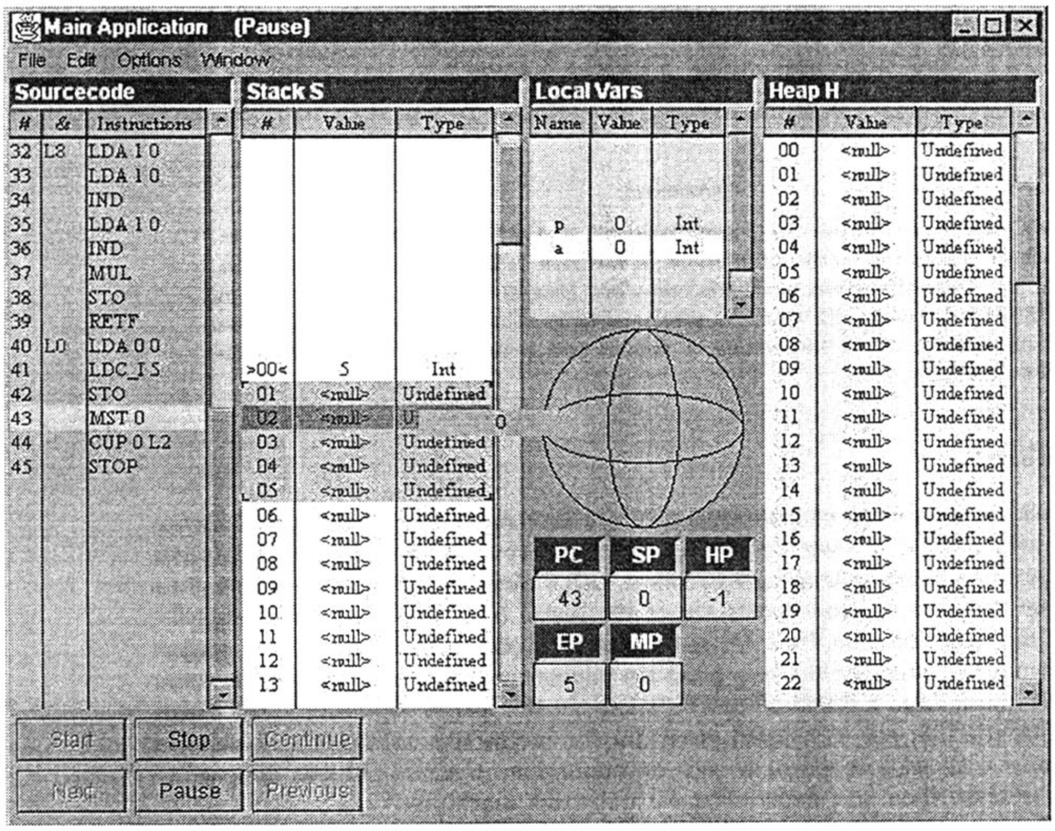
In the GANIMAL project we develop learning software for compiler design. Conceptually the computations performed by a compiler can be divided into several phases. For most of these phases there exist specification languages to define such a phase and generators which given the specification generate an implementation of the phase (e.g. LEX for lexical analysis, BISON for syntax analysis and PAG for semantical analysis). As a part of our project we develop generators, which do not only generate implementations, but also visualizations of the compiler phase from a standard specification. In this paper we describe the design rationale of GANIMAM, our web-based generator for interactive animations of abstract machines. Figure 1 shows a snapshot of such an animation. GANIMAM was designed to help students to learn about and experiment with abstract machines.
Abstract machines provide intermediate target languages for compilation. First the compiler generates code for the abstract machine, then this code can be interpreted or further compiled into real machine code. By dividing compilation into two stages, abstract machines increase portability and maintainability of compilers. The instructions of an abstract machine are tailored to specific operations required to implement operations of a source language or even better for languages of the same language paradigm.
In the following sections we describe how to use GANIMAM and what is generated by the system. Then we discuss the design of the specification language. Next we explain how we enhance animations by introducing annotations. Finally we discuss the benefits of using GANIMAM and its generated interactive animations both as a development tool and as a part of a learning software.

GANIMAM

GANIMAM can be accessed on a web page (http://www.cs.uni-sb.de/~diehl/GANI/). The user can enter a specification of an abstract machine, which is then send to the server. A

[*]FB 14 - Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, GERMANY, Email: diehl@cs.uni-sb.de, WWW: http://www.cs.uni-sb.de/~diehl

CGI script on the server generates Java code and using a Java compiler it translates this code into class files. In combination with the GANIMAM base package classes these class files form an interactive Java applet. This applet can be loaded over the internet and the user can enter machine programs, modify the layout of the different parts of the visualized abstract machines and control the animation of the execution of his abstract machine programs.



Sourcecode

#	&	Instructions
32	L3	LDA 1 0
33		LDA 1 0
34		IND
35		LDA 1 0
36		IND
37		MUL
38		STO
39		RETF
40	L0	LDA 0 0
41		LDC_I 5
42		STO
43		MST 0
44		CUP 0 L2
45		STOP

Stack S

#	Value	Type
>00<	5	Int
01	<null>	Undefined
02	<null>	U
03	<null>	Undefined
04	<null>	Undefined
05	<null>	Undefined
06	<null>	Undefined
07	<null>	Undefined
08	<null>	Undefined
09	<null>	Undefined
10	<null>	Undefined
11	<null>	Undefined
12	<null>	Undefined
13	<null>	Undefined

0

Local Vars

Name	Value	Type
p	0	Int
a	0	Int

PC	SP	HP
43	0	-1

EP	MP
5	0

Heap H

#	Value	Type
00	<null>	Undefined
01	<null>	Undefined
02	<null>	Undefined
03	<null>	Undefined
04	<null>	Undefined
05	<null>	Undefined
06	<null>	Undefined
07	<null>	Undefined
08	<null>	Undefined
09	<null>	Undefined
10	<null>	Undefined
11	<null>	Undefined
12	<null>	Undefined
13	<null>	Undefined
14	<null>	Undefined
15	<null>	Undefined
16	<null>	Undefined
17	<null>	Undefined
18	<null>	Undefined
19	<null>	Undefined
20	<null>	Undefined
21	<null>	Undefined
22	<null>	Undefined

Figure 1: Screenshot of an animated abstract machine

The automatic layout groups the different memories around the accumulator (those ellipses in the middle). Source code and stacks are placed to the left, stacks to the right, local variables above and registers below the accumulator. Associated with the accumulator is an accumulator window (see Figure 6), which shows the expressions which are currently evaluated and the definitions of the instructions or functions which are currently executed. Double clicking with the right mouse button at an instruction in the source code window, loads its definition into the accumulator window. Double clicking with the left mouse button at an instruction sets the value of the program counter to the address of that instruction, i.e. the execution of the abstract machine program continues at that address. Clicking at a cell of a stack, heap or register opens a window. In this window the user can change the value and type of that cell. For registers only the value can be changed.

```
program_unit      ::=  declarations definitions
declarations      ::=  [declaration {; declaration}]
declaration       ::=  REGISTER decl_REG | HEAP decl_HEAP | STACK decl_STACK
decl_REG          ::=  IDENTIFIER [, decl_REG]
decl_HEAP         ::=  array with IDENTIFIER {, array with IDENTIFIER }
decl_STACK        ::=  array with IDENTIFIER
array             ::=  IDENTIFIER [ CONSTANT ]
definitions       ::=  {definition}
definition        ::=  def command ≡ instructions fed
                       | fun funcommand ≡ instructions nuf
command           ::=  IDENTIFIER arguments
funcommand        ::=  type IDENTIFIER arguments
arguments         ::=  [( [identifiers] )]
identifiers       ::=  type IDENTIFIER {, type IDENTIFIER}
type              ::=  int | boolean | address | pointer | real | ...
instructions      ::=  [instruction {; instruction}]
instruction|      ::=  assignment | condition | for | case | call
                       | return expression | ie ( predefinedEvents )
predefinedEvents  ::=  markProcedureStackFrame ( register , CONSTANT )
                       | comment ( commenttext ) | ...
assignment        ::=  lval := rval
lval              ::=  register | memory
rval              ::=  expression
condition         ::=  if condit then instructions [else instructions] fi
for               ::=  for forinit to expression do instructions od
                       | for forinit downto expression do instructions od
forinit           ::=  IDENTIFIER := expression
case              ::=  case expression of expressionlists +esac
expressionlists   ::=  expressionlist expressionlists | defaultlist
expressionlist    ::=  CONSTANT : begin instructions end
defaultlist       ::=  [otherwise : instructions]
call              ::=  IDENTIFIER ( f_arguments )
f_arguments       ::=  [expression {, expression}]
```

Figure 2: Syntax of abstract machine specification language

Specification Language

Finding low-level principles and casting them into language constructs is the first step towards a specification language. A well-designed specification language enables us to generate implemenations and visualizations. A crucial point of our specification language is that it applies to abstract machines for programming languages of different paradigms. Our specification language is based on the notation used in the compiler design text book by Wilhelm and Maurer [9] to define abstract machines for imperative, logical and functional programming languages. Recently the notation was also used to describe the Java Virtual Machine [4]. The core of our specification language is an imperative language with assignments, expressions, conditionals and loops. Control flow languages are a standard specification method for imperative, functional and logical programming languages, e.g. [9, 1, 7]. For the specification of abstract machines for functional languages sometimes rewriting rules have been used, e.g. for the CAM [6], but usually they can be easily reformulated in a control flow language[5].
At the heart of our specification language is a general machine model. An abstract machine consists of a set of instructions, a program store, heaps, stacks[1] and registers. The machine runs in a loop executing the instruction currently pointed at by a special register, the program counter (PC).

```
while(true) {
 PC := PC + 1;
 execute instruction at CODE[PC-1]
}
```

In this model an abstract machine can be specified by declaring its heaps, stacks and registers and defining its instructions.
In Figure 2 the syntax of our specification language is given. A specification starts with declarations of stacks, heaps and registers. Then auxilliary functions (with `fun`) and machine instructions (with `def`) are defined. Functions must be defined before they are used. The predefined datatypes currently include integers, booleans, reals, addresses and pointers. Addresses refer to positions in the program code, whereas pointers point to cells in the stacks or heaps. One could imagine to have pointers to registers, but we haven't found an abstract machine which needs this. There is also a construct to declare structured data types:

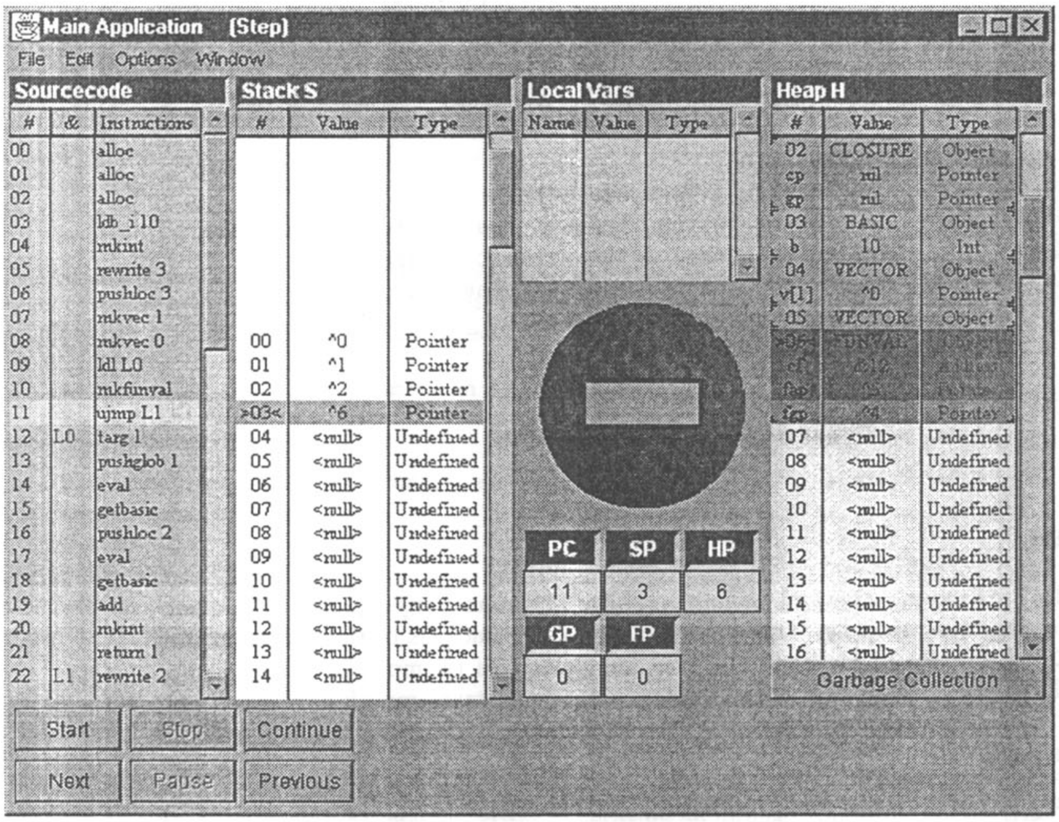
```
OBJECT  FUNVAL    (cf,fap,fgp),
        CLOSURE    (cp,gp),
        VECTOR []   (v);
```

It was heavily used in the specification of the MAMA, a variant of the G-machine, which is used as a target architecture for functional programming languages. In Figure 3 instances of these structured data types are visualized in the heap window.
The careful reader will notice, that labels are not part of the specification language, but that they occur in abstract machine programs (see source code window in Figure 1). Labels are used instead of concrete addresses and a preprocessor contained in the runtime system of GANIMAM maps these labels onto concrete addresses.
To initialize the state of an abstract machine, we allow initialization sequences in the abstract machine code:

[1]In some abstract machines there are several stacks, e.g., in the WAM we have the environment stack,the trail and the PDL for recursive implementations of unification.

```
.init_prog
#
source: S
offsetregister: MP
0:  3.14 Real
1: -3    Int
2: true  Boolean
#
```

In the above example the Stack S is initialized as follows: S[MP+0]=3.14 and its tag is set to Real, S[MP+1]=-3 and its tag is set to Int and S[MP+2]=true and its tag is set to Boolean. This means, that an abstract machine starts with all registers set to the default values given in the abstract machine specification and stacks and heaps are empty. But in addition each abstract machine program can have its own initialization sequence.



Sourcecode

#	&	Instructions
00		alloc
01		alloc
02		alloc
03		ldb_i 10
04		mkint
05		rewrite 3
06		pushloc 3
07		mkvec 1
08		mkvec 0
09		ldl L0
10		mkfunval
11		ujmp L1
12	L0	targ 1
13		pushglob 1
14		eval
15		getbasic
16		pushloc 2
17		eval
18		getbasic
19		add
20		mkint
21		return 1
22	L1	rewrite 2

Stack S

#	Value	Type
00	^0	Pointer
01	^1	Pointer
02	^2	Pointer
>03<	^6	Pointer
04	<null>	Undefined
05	<null>	Undefined
06	<null>	Undefined
07	<null>	Undefined
08	<null>	Undefined
09	<null>	Undefined
10	<null>	Undefined
11	<null>	Undefined
12	<null>	Undefined
13	<null>	Undefined
14	<null>	Undefined

Local Vars

Name	Value	Type

PC	SP	HP
11	3	6

GP	FP
0	0

Heap H

#	Value	Type
02	CLOSURE	Object
cp	nil	Pointer
gp	nil	Pointer
03	BASIC	Object
b	10	Int
04	VECTOR	Object
v[1]	^0	Pointer
05	VECTOR	Object
>06<	FUNVAL	Object
cf	@12	[illegible]
fap	^5	[illegible]
fgp	^4	Pointer
07	<null>	Undefined
08	<null>	Undefined
09	<null>	Undefined
10	<null>	Undefined
11	<null>	Undefined
12	<null>	Undefined
13	<null>	Undefined
14	<null>	Undefined
15	<null>	Undefined
16	<null>	Undefined

Figure 3: Screenshot of the animated MAMA

An Example Specification

In Figure 4 we show an excerpt of the specification of an abstract machine for imperative languages [9], a variant of the P-Machine. In this example the instruction mst (mark stack) is defined which pushes a frame for a procedure on top of the stack. In the specification a stack S, a heap H and the register PC, MP and EP are declared. The stack and heap declarations contain declarations of the special purpose registers SP and HP, which point to the top of the currently used memory area. The special purpose register PC is automatically defined by GANIMAM and its declaration is optional. Next auxilliary functions are defined. Here it is a function, which computes the static predecessor or the current procedure, in the WAM for example such functions include `unify()` or `deref()`.

```
// declarations
STACK    S[100] with SP=-1;
HEAP    H[100] with HP=-1;
REGISTER  PC,MP,EP;

// specification of auxilliary functions
fun int base (int p, int dv) =  // computes the static predecessor
 if (p=0) then                  // of the current procedure
  return dv;
 else
  return base(p-1,S[dv+1]);
 fi;
nuf

// specification of an instruction
def mst (int p) =        // create a procedure frame
 S[SP+2]:=base(p,MP);  // pointer to the frame of the static predecessor
 S[SP+3]:=MP;  // pointer to the frame of the dynamic predecessor
 S[SP+4]:=EP;  // max. depth for evaluation of expressions
 SP:=SP+5;  // procedure frame has at least 5 cells
fed
```

Figure 4: Example specification

Principles

Many principles are hard to define and explain verbally. There are many aspects (what, how, why) which belong to a principle. One has to distinguish principles of the programming language, e.g., inheritance of methods in Java, and principles to implement these in an abstract machine, e.g., chains of method tables. In the abstract machine we can only visualize the implementation principles. When visualizing a principle, we can visualize its different aspects:

- Show the information used and produced by the principle.
- Animate operations performed by the principle, e.g. dereferencing of variables in the WAM.
- Visualize properties and invariants enforced by the principle, e.g. in the WAM variables of higher addresses always reference that of lower addresses.

Visualizing Higher-Level Abstractions

Some of the principles of an abstract machine are not explicit at the abstraction level of our specification language. For example stack frames are common to all abstract machines we

considered. Stack frames are a means to implement recursion. Usually the stack cells of a stack frame are allocated by a sequence of one or more instructions, which push values on top of the stack.
Other instructions access information relative to the beginning of the stack frame or release the stack frame as a whole [2]. There is no single construct in our specification language to allocate a stack frame. When visualizing an abstract machine it is important that we do not only draw low-level abstractions captured by our specification language constructs, but also higher-level abstractions. In order to do this we added visualization annotations to our specification language. These annotations can be compared to interesting events in some algorithm animation systems[2].

```
// instruction including animation annotation

def mst (int p) =
 ie(markProcedureStackFrame(SP,5));
 // Starting a cell S[SP], the following 5 cells are
 // graphically marked as a procedure frame
 S[SP+2]:=base(p,MP);
 S[SP+3]:=MP;
 S[SP+4]:=EP;
 SP:=SP+5;
fed
```

A very general and useful annotation is a runtime comment. It produces a textual output which is shown in a console window. Using runtime comments the output in the console window can be used as a trace of the execution of the abstract machine, see Figure 5.

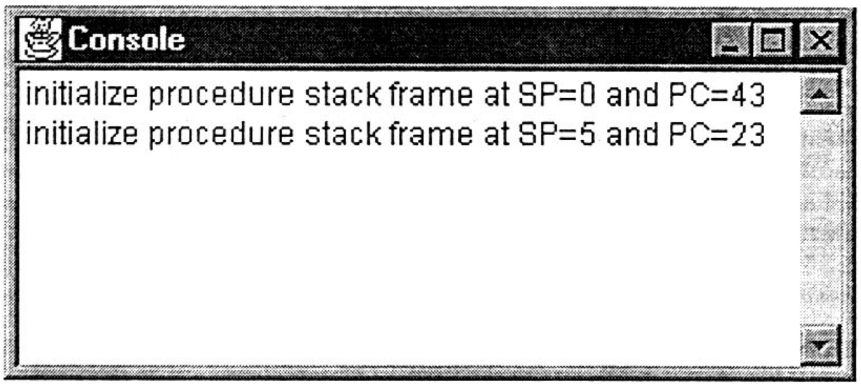
```
// instruction including animation annotation

def mst (int p) =
 ie(comment("Initialize procedure stack frame at SP="+SP+" and PC="+PC));
 S[SP+2]:=base(p,MP);
 S[SP+3]:=MP;
 S[SP+4]:=EP;
 SP:=SP+5;
fed
```

The Benefits of Interactive Animations

GANIMAM provides several ways of user interaction. First the user can enter or modify the specification of an abstract machine. After generating an implemenation of the abstract machine, the user can input an abstract machine program, execute it step by step and inspect the contents of each register or memory cell. When executing an instruction animations show the flow of information from registers or memory cells to a conceptual operation unit, called accumulator, and from the accumulator back to registers or memory cells. The evaluation done in the accumulator is shown in a special window, see Figure 6.
Annotations only help to visualize principles which we know upfront. GANIMAM can also be used to detect new principles by experimenting with specifications and abstract machine programs. Such an experimental approach can be used for two purposes:

[2]In the WAM stack frames are called environments and a special optimization called environment trimming decreases the number of stack cells of an environment during its live span.



```
initialize procedure stack frame at SP=0 and PC=43
initialize procedure stack frame at SP=5 and PC=23
```

Figure 5: Screenshot of console window

- As part of an explorative learning software it enables students to formulate hypotheses and validate or invalidate them by changing specifications or abstract machine programs. Additional text guides the learner, to make sure he doesn't miss the important issues. Such issues could be caller-save-registers vs. callee-save-registers, finding the frame of the static predecessor or lazy vs. eager evaluation.
- As a development tool it can help to detect errors and optimizations. As an example of such an optimization consider tail recursion optimization. By tracing the execution of example programs it might become apparent that the information stored in a frame is not needed after certain recursive calls.

GANIMAM is not meant to replace classical teaching or development approaches, but to supplement and enhance these. GANIMAM can also be used by researchers to present their new implementation techniques or for rapid prototyping.

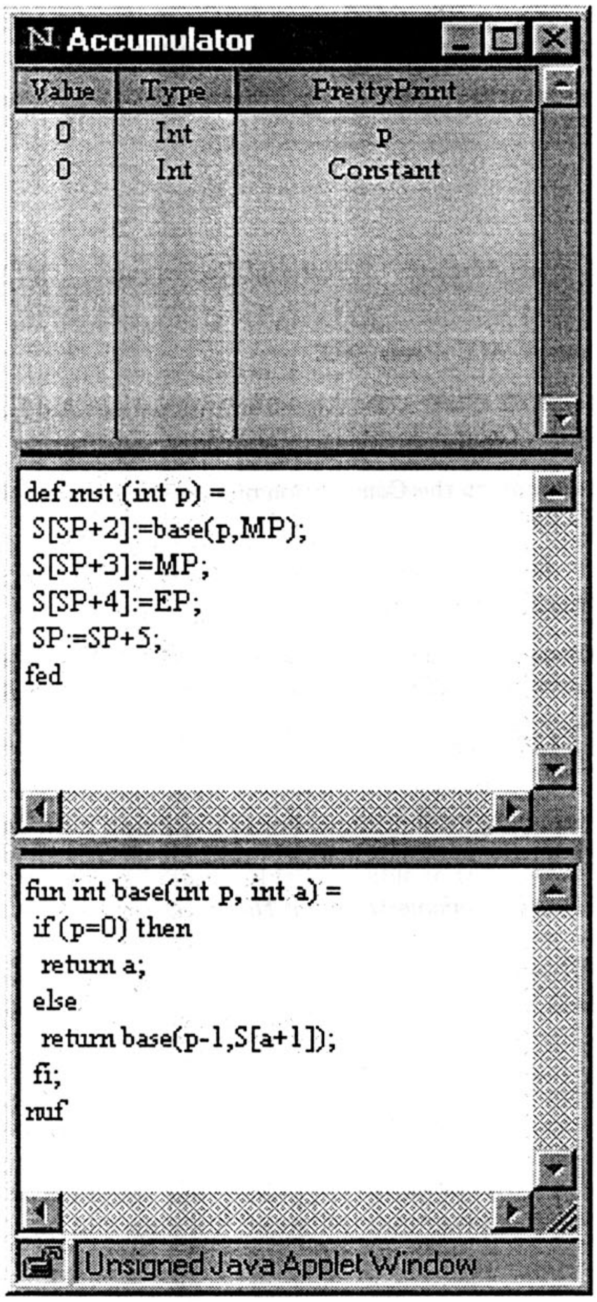
Current and Future Work

In the GANIMAL project[3] we will also develop generators for interactive animations of other compiler phases. We are currently looking into how to evaluate the software produced in the GANIMAL project. Those evaluations of algorithm animations we are aware of [8, 3] lack a serious approach both for collecting and evaluating the data. To avoid these problems we plan to cooperate with cognitive psychologists.
In the current version of GANIMAM structured datatypes like records, objects or ML datatypes can be built with the help of pointers but a coherent visualization does not yet exist. In a later version we will use the graph layouter which is currently under development in the GANIMAL project to visualize structured data types.

Conclusion

We introduced GANIMAM, a web-based system to generate interactive animations of abstract machines from specifications. During the development of GANIMAM common principles of abstract machines have been considered at three levels: the design of the specification language, the choice of graphical annotations to visualize higher-level abstractions and the

[3]This project is funded by the Deutsche Forschungsgemeinschaft and started in summer 1998.



Value	Type	PrettyPrint
0	Int	p
0	Int	Constant

```
def mst (int p) =
 S[SP+2]:=base(p,MP);
 S[SP+3]:=MP;
 S[SP+4]:=EP;
 SP:=SP+5;
fed
```

```
fun int base(int p, int a) =
 if (p=0) then
  return a;
 else
  return base(p-1,S[a+1]);
 fi;
nuf
```

Figure 6: Screenshot of the accumulator window

use of the system to explore and better understand known and detect new principles. Our final goal is to integrate GANIMAM into a learning software for compiler design and thus enabling students to solve exercises related to abstract machines by an experimental and explorative approach.

Acknowledgements The authors want to thank Andreas Placzek, who helped to implement a first prototype of GANIMAM.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine - A Tutorial Reconstruction.* MIT Press, 1991.

[2] M.H. Brown. *Algorithm Animation.* MIT Press, 1987.

[3] M.D. Byrne, R. Catrambone, and J.T. Stasko. Do Algorithm Animations Aid Learning. Technical Report GIT-GVU-96-18, Georgia Institute of Technology, 1996.

[4] Stephan Diehl. A Formal Introduction to the Compilation of Java. *Software – Practice and Experience*, 28(3):297–327, 1998.

[5] John Hannan. Making Abstract Machines Less Abstract. In *Proc. of FPCA'91*, volume LNCS 523, pages 618–635. Springer Verlag, 1991.

[6] M. Mauny and A. Suarez. Implementing functional languages in the categorial abstract machine. In *International Conference on LISP and Functional Programming*, 1986.

[7] St. Pemberton and M. Daniels. *Pascal Implementation, The P4 Compiler.* Ellis Horwood, 1982.

[8] John T. Stasko. Using Student-Built Algorithm Animations as Learning Aids. Technical Report GIT-GVU-96-19, Georgia Institute of Technology, 1996.

[9] Reinhard Wilhelm and Dieter Maurer. *Compiler Design: Theory, Construction, Generation.* Addison-Wesley, 1995.

Dynamic semantics of Java byte-code

Peter Bertelsen

Royal Veterinary and Agricultural University, Copenhagen, Denmark
Room T629, Thorvaldsensvej 40, DK-1871 Frederiksberg C
Tel. +45 3528 2693, Fax +45 3528 2350
E-mail: pmb@dina.kvl.dk

August 14, 1998

Abstract. We give a formal specification of the dynamic semantics of Java byte-code, in the form of an operational semantics for the Java Virtual Machine (JVM). For each JVM instruction we give a rule describing the instruction's effect on the machine state, and the conditions under which the instruction will execute without error.
This paper outlines the formalization of the JVM machine state, and illustrates our specification approach with a few select JVM instructions. Our full specification, covering the entire JVM instruction instruction set except for synchronization instructions, is available in [2].

Keywords: Java, JVM, formal specification, semantics.

1 Introduction

The Java Virtual Machine (JVM) is a virtual machine for safely implementing object oriented languages. It is rather complicated because each instruction must check a number of conditions. For instance, the `getstatic` instruction, which accesses a static field of a class, must check that the class is accessible to the current method, that the class has been loaded, that the class declares the requested field, that the field is accessible to the current method, etc.

The 'official' definition of the JVM is given in the book by Lindholm and Yellin [9]. To describe the pre-conditions and the effect of JVM instructions, the book uses natural language, pseudo-C constructs and runtime stack pictures. Consequently, Lindholm and Yellin's book is rather long (475 pages) and it is hard to fully understand the semantics of the JVM, e.g. the precise conditions for executing an instruction.

The objective of the present work was to gain a thorough understanding of the JVM as described by Lindholm and Yellin, and to express that understanding clearly and compactly. Another goal was that of uncovering possible omissions and ambiguities in the JVM specification. Hence, we describe the JVM at a higher level of abstraction, using ordinary mathematical concepts, and we use a semi-formal notation rather than natural language. Our full specification [2] is less than 80 pages long.

1.1 What is covered by our specification

We formalize the JVM machine state, and for every JVM instruction, we describe the conditions for its successful execution and its precise effect on the machine state. We do *not* describe the following aspects of the JVM:

- class file verification: what checks can or must be done at class loading time;
- multiple threads, the `monitorenter` and `monitorexit` instructions;
- what exception to throw when a pre-condition of an instruction fails;
- the Java Class Library, native methods, and garbage collection.

The JVM instruction set does not include any instructions for starting, suspending, or stopping a thread; these mechanisms are supported only via methods in the Java Class Library. Hence, a formalization of the JVM semantics with respect to multi-threading would have to cover the semantics of (parts of) the Java Class Library as well. In our specification, we focus solely on the JVM and consider only one single thread of execution.

1.2 Format of the specification

Our formalization uses ordinary mathematical concepts: partial functions[1], sets, sequences, disjoint sums, etc.

Alternatively, one could use a particular specification language, such as VDM [8], or develop a theory within a theorem prover, such as HOL [7], Isabelle [11] or PVS [10]. This would make the specification even more precise but probably less accessible to the general reader.

2 Modelling the JVM machine state

In this section we outline a formal specification of select JVM instructions. The details of our notation and the precise definition of auxilliary functions used in the following sections are presented in the full report [2].

Our specification is more abstract than that described in the 'official' JVM specification [9], yet describes the JVM semantics at a level very close to the actual instruction set, including many details of the byte-code instructions.

The run-time state of the JVM has two parts: the global environment, which remains fixed, and the thread state, which changes during execution.

Note that we model only a single thread of execution.

2.1 The global environment

The *global environment* maps class names to class files. Concretely, the global environment represents the file system and the network, from which class files may be loaded. A class file has eight components in our modelling:

$$C \;=\; \mathcal{P}(Acc_c) \times [Id_c] \times \mathcal{P}(Id_c) \times FD \times CV \times MD \times MI \times CP$$

[1] We use the notation $A \rightharpoonup B$ to designate a partial function from A to B.

The components of a class or interface C are its access modifiers (e.g. public), the name of its direct superclass (optional), the names of its direct superinterfaces, its field declarations, its constant values, its method declarations, its method implementations, and its constant pool. The constant pool holds string literals, other constants and symbolic references to classes and class members.

The precise definitions of the sets $\mathcal{P}(Acc_c)$, $[Id_c]$, etc. are given in the full report [2] in the same style.

2.2 The thread state

The *thread state* is the state of an executing thread. We model the thread states TS as follows:

$$TS = Frame^* \times Heap \times Env$$

The thread state consists of a frame stack ($Frame^*$), a heap ($Heap$), and an environment (Env). Each frame in the frame stack corresponds to a method invocation. The topmost frame is the frame of the method currently executing.

A frame $f \in Frame$ contains a program counter $pc \in PC$, an operand stack $s \in Oper^*$, a local variable table $l \in Locals$, and the current method's class name $id_c \in Id_c$ as well as its signature (method name and argument types) $sig \in Sig$:

$$Frame = PC \times Oper^* \times Locals \times (Id_c \times Sig)$$

An operand ($Oper$) is either a word (W) or a double-word (DW). A word is either a proper word (W_v) or a program counter value (PC). A proper word is either an integer[2] (Int), a float ($Float$) or a reference (Ref_0) which is possibly null. A double-word is either a long integer ($Long$) or a double ($Double$).

A local variable table ($Locals$) maps a non-negative integer index to the (one-word or two-word) local variable value ($Oper$) at that index. A two-word value occupies two entries in the table.

A heap $h \in Heap$ maps a non-null reference (Ref) to an object:

$$\begin{aligned}
Heap &= Ref \rightharpoonup Obj \\
Obj &= Obj_u \cup Obj_c \cup Obj_a \\
Obj_u &= Id_c \times IV \\
Obj_c &= Id_c \times IV \\
IV &= (Id_c \times Id_f) \rightharpoonup V \\
V &= W_v \cup DW
\end{aligned}$$

An object (Obj) is either an uninitialized (Obj_u) or initialized (Obj_c) instance of a class type, or an instance (Obj_a) of an array type.

An object (Obj_u or Obj_c) of class type consists of the name (Id_c) of the class, and its instance field values (IV). The latter maps a pair of a class name (Id_c) and a field name (Id_f) to the value (V) of that instance field. That value must be

[2] The JVM uses integers (Int) to represent Java's boolean, byte, char, short, and int types.

a proper value (V): either a proper word (W_v) or a double-word (DW). Hence, program counter values (PC) cannot be stored in an instance field.

Array objects are described similarly, as maps from non-negative indices to proper values (V). Multi-dimensional arrays are arrays of arrays: each element is itself an array object.

The environment $e \in Env$ in a thread state holds the classes that have been loaded by the JVM from the global environment (e.g. from the file system). It maps a class or interface name to its declaration (C) and static field values table (SV):

$$\begin{aligned} Env &= Id_c \rightarrow (C \times SV) \\ SV &= Id_f \rightarrow V \end{aligned}$$

A static field values table (SV) maps a field name (Id_f) to its value.

3 Formalizing the effect of JVM instructions

The effect of JVM instruction execution on the thread state is described in the style of small-step operational semantics [13]. Each JVM instruction is defined by an inference rule, as illustrated by rule (1) below. The premises above the line describe the conditions that must hold for the instruction to execute successfully, that is, without throwing an exception. The conclusion $ts \Rightarrow ts'$ below the line says that execution of the instruction will change the thread state from ts to ts'.

In each of the rules, the symbols s, l, m, fr, h, and e refer to the components of the thread state $ts = ((pc, s, l, m) :: fr, h, e)$.

By convention, the premises are read from the top down and from left to right. Although immaterial from a logical point of view, this supports a more operational interpretation of the rules.

3.1 Example 1: the dup instruction

The dup instruction duplicates the operand stack's topmost value:

$$\frac{\begin{array}{l} instr(ts) = \mathtt{dup} \\ s = (v : W) :: sr \\ size(s) + size(v) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', v :: v :: sr, l, m) :: fr, h, e)} \quad (1)$$

If all of the premises above the line hold, then the current thread state ts will change into the state specified after the $\Rightarrow$ symbol.

The premise $instr(ts) = \mathtt{dup}$ says that this rule applies to the dup instruction.

The premise $s = (v : W) :: sr$ asserts that the operand stack s in ts has top-most element v and remainder sr, and that v is a word (W), not a double-word[3].

[3] The dup instruction cannot be used for duplicating a double-word stack operand. Instead, the JVM instruction dup2 must be used.

The premise $size(s) + size(v) \leq max_s(ts)$ asserts that the stack will not overflow: the combined sizes of the old stack s and the duplicated value v does not exceed the maximal size $max_s(ts)$ of the stack.

The premise $succ(ts) = pc'$ asserts that there is a successor instruction and that its address is pc'.

For brevity, the rules use a number of semantic utility functions. For instance, *instr* finds the current instruction in the current thread state *ts*, *size* computes the size (in words) of a semantic object, max_s finds the declared maximal stack size for the method currently executing, *succ* finds the address of the next byte-code instruction to execute, etc. Formal definitions of these functions (and those used below) are given in the full report [2].

The notation $v : W$ in the second premise means that the value v has type W. This notation is used in two ways: (1) to assert a condition, and (2) to tag a value with a given type. For instance, 117 : *Double* is the number 117 of type *Double* (as opposed to e.g. *Int* or *Long*).

3.2 Example 2: the `istore` instruction

The `istore` instruction removes the stack's top-most (integer) value and stores it in a local variable:

$$\frac{\begin{array}{l} instr(ts) = \texttt{istore}\ j \\ s = (k : Int) :: sr \\ j < max_l(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', sr, rmDW(l, j) + \{j \mapsto k\}, m) :: fr, h, e)} \quad (2)$$

The premises state that this rule concerns the `istore` instruction, that the top-most value on the stack s must exist and be an integer k, that j must be within the declared range of local variable indexes, and that the current instruction must have a successor at pc'. If so, the thread state changes to $((pc', sr, rmDW(l, j) + \{j \mapsto k\}, m) :: fr, h, e)$ in which k has been popped off the stack, and local variable j has been changed to k.

If writing into local variable j happens to destroy the second half of a double-word value, then the first half of that double-word must be removed from l. This is handled by the semantic function $rmDW(l, j)$.

3.3 Example 3: the `baload` instruction

The `baload` instruction loads the value from an array of element type `byte` or `boolean`:

$$\frac{\begin{array}{l} instr(ts) = \texttt{baload} \\ s = (k : Int) :: (r : Ref) :: sr \\ h(r) = ((1, \texttt{byte}), k', av) : Obj_a \\ av(k) = k'' : Int \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k'' :: sr, l, m) :: fr, h, e)} \quad (3)$$

If the top-most stack operand is an integer k, if the second top-most stack operand is a reference r to a one-dimensional array object of element type `byte` or `boolean` and if k is a valid index into the array, then the value k and the array reference r are popped off the stack, the array component k'' at index k is pushed onto the stack and execution continues with the next instruction.

The component k' of the array object at position r in the heap h is the length of the array object. It is immaterial in the rule for `baload` since the premise $av(k) = \ldots$ ensures that k is in the domain of av (which maps an array index to the corresponding array element).

Note that an integer value (*Int*) is loaded from the array. It is assumed that a component of a `byte` or `boolean` array has already been truncated to a `byte` or `boolean` value, respectively, and then expanded back into an integer value[4].

3.4 Example 4: the `new` instruction

The `new` instruction instantiates the class specified by constant pool index i:

$$\frac{\begin{array}{l} instr(ts) = \texttt{new}\ i \\ pool(ts)(i) = id_c' : Id_c \\ e(id_c') = ((acc_c, id_c'', is, fd, cv, md, mi, cp), sv) \\ acc_c \cap \{\texttt{interface}, \texttt{abstract}\} = \emptyset \\ access(id_c', acc_c, id_c) \\ r \in Ref \backslash dom(h) \\ fields(id_c', e) = iv \\ (id_c', iv) = o : Obj_u \\ size(s) + size(r) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', r :: s, l, m) :: fr, h + \{r \mapsto o\}, e)} \quad (4)$$

If it holds that

- i is a valid index into the constant pool of the current class,
- the constant pool entry at index i is a class (or interface) reference id_c',
- the declaration of id_c' is in the domain of the environment e (i.e., has been loaded),
- id_c' is an instantiable class (not `abstract` or `interface`),
- the current class id_c can access class id_c', and
- there is an unused location r in the heap,

then the execution of the `new` instruction proceeds:

- the instance fields of class id_c' and all its superclasses are prepared using the auxiliary function *fields*;
- an instance o of class id_c' with instance field values iv is created;

[4] In our modelling, the truncation and expansion to/from `byte` and `boolean` values is specified in connection with the `bastore` instruction. See the full report [2] for details.

- the reference r is pushed onto the operand stack;
- o is bound at location r in the heap; and
- execution continues with the instruction at pc'.

The new object o is tagged with type Obj_u, which means that it is uninitialized. Members of the object cannot be accessed until it has been initialized by invocation of a constructor.

Note that the above rule does not specify any details with respect to memory allocation or garbage collection[5]. Instead, we assume an infinite heap in which a fresh heap location r is always available.

3.5 Example 5: the `getstatic` instruction

The `getstatic` instruction pushes the value of a class or interface field onto the stack:

$$\frac{\begin{array}{l} instr(ts) = \texttt{getstatic}\ i \\ pool(ts)(i) = (id_c', id_f, d) : Const_f \\ e(id_c') = ((acc_c, id_c'', is, fd, cv, md, mi, cp), sv) \\ access(id_c', acc_c, id_c) \\ fd(id_f) = (acc_f, d') \\ d = d' \\ \texttt{static} \in acc_f \\ \texttt{private} \in acc_f \Rightarrow id_c = id_c' \\ \texttt{protected} \in acc_f \Rightarrow id_c' \in supers(id_c, e) \\ sv(id_f) = v : V \\ size(s) + size(v) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', v :: s, l, m) :: fr, h, e)} \quad (5)$$

If it holds that

- i is a valid index into the constant pool of the current class,
- the constant pool entry at index i is a symbolic field reference, referring to a field id_f with type descriptor d in class or interface id_c',
- the name id_c' is in the domain of the environment e (i.e., the declaration of a class or interface of that name has been loaded),
- the current class id_c can access class/interface id_c',
- class/interface id_c' declares a field id_f of the same type as that specified by the descriptor d,
- the field id_f is `static`,
- the field id_f is not `private` unless the current class is the same as that declaring the field,

[5] The JVM specification [9] does not prescribe any particular memory management technique, but assumes some sort of automatic memory reclamation, e.g. garbage collection.

- the field id_f is not `protected` unless the current class is the same as that declaring the field or a subclass thereof, and
- the value of field id_f in the static values sv is v,

then the value v is pushed onto the operand stack, and execution proceeds with the next instruction.

It is possible to get a value from a field before putting anything into it; the value loaded from the field will then be the initial (default) value for the field's type.

3.6 Example 6: the `getfield` instruction

This example demonstrates one of the object-oriented features of the JVM. The `getfield` instruction pushes the value of an instance field onto the stack:

$$\frac{\begin{array}{l} instr(ts) = \texttt{getfield}\ i \\ pool(ts)(i) = (id_c{}', id_f, d) : Const_f \\ e(id_c{}') = ((acc_c, id_c{}'', is, fd, cv, md, mi, cp), sv) \\ \texttt{interface} \notin acc_c \\ access(id_c{}', acc_c, id_c) \\ fd(id_f) = (acc_f, d') \\ \texttt{static} \notin acc_f \\ \texttt{private} \in acc_f \Rightarrow id_c = id_c{}' \\ d = d' \\ s = (r : Ref) :: sr \\ h(r) = (id_c{}''', iv) : Obj_c \\ id_c{}' \in supers(id_c{}''', e) \\ \texttt{protected} \in acc_f \Rightarrow (id_c{}' \in supers(id_c, e)\ \wedge\ id_c \in supers(id_c{}''', e)) \\ iv(id_c{}', id_f) = v : V \\ size(sr) + size(v) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', v :: sr, l, m) :: fr, h, e)} \qquad (6)$$

The `getfield` instruction differs from the `getstatic` instruction in that the referenced field cannot be an interface or class field and that the value of the field is retrieved from a specific class instance rather than via the environment e of the thread state.

If it furthermore holds that

- the top-most stack operand is a non-`null` reference r to an initialized object,
- the object is an instance of the class $id_c{}'$ that declares the referenced field id_f [6],
- the field is not `protected` unless the current class is the same as $id_c{}'$ or a subclass thereof and the object is an instance of the current class, and
- the value of the field id_f in the instance values iv of the object is v,

[6] The field id_f must be declared in class $id_c{}'$; it is not sufficient for class $id_c{}'$ to inherit the field from a superclass.

then the object reference r is popped off the stack, the value v is pushed onto the stack and execution continues with the next instruction.

3.7 Example 7: the `invokespecial` instruction

This example shows one of the four JVM instructions for method invocation. The `invokespecial` instruction may be used for invoking a constructor, a `private` instance method, or a method of the current class[7]:

$$\frac{\begin{array}{l}
instr(ts) = \texttt{invokespecial}\ i \\
pool(ts)(i) = (id_c', sig', d) : Const_m \\
e(id_c') = ((acc_c, id_c'', is, fd, cv, md, mi, cp), sv) \\
\texttt{interface} \notin acc_c \\
access(id_c', acc_c, id_c) \\
sig' = (id_m, \texttt{<}d_1, d_2, d_3, \ldots, d_k\texttt{>}) \\
id_m \neq \texttt{<clinit>} \\
md(sig') = (acc_m, d', excs) \\
(id_m = \texttt{<init>} \ \vee\ \texttt{private} \in acc_m \ \vee\ id_c = id_c' \\
\qquad \vee\ id_c' \notin supers(id_c, e) \ \vee\ \texttt{super} \notin acc_c) \\
acc_m \cap \{\texttt{static}, \texttt{abstract}, \texttt{native}\} = \emptyset \\
\texttt{private} \in acc_m \Rightarrow id_c = id_c' \\
mi(sig') = (n_s, n_l, code', hdls') \\
s = as@sr \\
as = \texttt{<}a_k, a_{k-1}, a_{k-2}, \ldots, a_2, a_1\texttt{>}@\texttt{<}r : Ref\texttt{>} \\
size(as) \leq n_l \\
\forall 1 \leq j \leq k.(initialized(a_j, h) \wedge compatVal(d_j, a_j, h, e)) \\
h(r) = o : (Obj_u \cup Obj_c) = (id_c''', iv) \\
(id_m = \texttt{<init>} \wedge o \in Obj_u) \ \vee\ (id_m \neq \texttt{<init>} \wedge o \in Obj_c) \\
id_c' \in supers(id_c''', e) \\
\texttt{protected} \in acc_m \Rightarrow (id_c' \in supers(id_c, e) \ \wedge\ id_c \in supers(id_c''', e)) \\
(min_{pc}(code'), \texttt{<>}, args(as), (id_c', sig')) = f' : Frame \\
initObj(id_c', sig', r, h) = h'
\end{array}}{ts \Rightarrow (f' :: (pc, sr, l, m) :: fr, h', e)} \tag{7}$$

If it holds that

- i is a valid index into the constant pool of the current class
- the constant pool entry at index i is a symbolic method reference, referring to a method in class id_c' with signature sig' and return type descriptor d
- the declaration of class id_c' is in the environment e
- the current class has permission to access class id_c'
- the referenced method is not a static initializer (with the special method name `<clinit>`)
- class id_c' declares and implements a method with signature sig' and the same return type descriptor as that specified by the symbolic method reference

[7] The `invokespecial` instruction may also be used for invoking other 'special' instance methods; this is described in a separate rule in our full report [2].

- the method is not declared to be either `static`, `abstract` or `native`
- the method is not declared to be `private`, unless the current class is the same as that declaring the method
- the first stack operand is a reference r to a class instance o in the heap h
- the next k stack operands are the method arguments $a_k, \ldots, a_1$ which are assignment compatible with the corresponding k parameter type descriptors of the method signature sig'
- the total size of the method arguments as is less than or equal to the local variable limit n_l of the method
- the referenced object o is either uninitialized, in which case the method to be invoked must be a constructor (with the special method name `<init>`); or it has been initialized, in which case the invoked method must not be a constructor
- the class id_c''' of the object o must be the same as id_c', or a subclass thereof
- if the method is declared to be `protected`, then the current class must be the same as class id_c', or a subclass thereof, and the class id_c'''' implementing the method must be the same as the current class, or a subclass thereof.

then the method arguments as are popped from the operand stack of the current frame, a new frame f' representing the invoked method is pushed onto the frame stack, and execution continues with the first instruction in the invoked method.

The new frame f' initially contains an empty operand stack and the values of the method parameters in the first local variables (initialized by means of the utility function *args*).

The utility function *initObj* is used for tagging the object referenced by the stack operand r as initialized (that is, as having type Obj_c) in case the method being invoked is the constructor of class `java.lang.Object` (the superclass of all other classes).

3.8 Other instructions

Above we have given examples of instructions operating on the stack, the local variables, arrays, the constant pool, the static fields of a class and the instance fields of an object. Our full specification comprises 62 rules of which the most complicated one is the method invocation rule for `invokespecial` shown above.

We do not define separate rules for each of the 201 JVM instructions since many of these are *very* similar to other instructions. Instead, we define rules for instructions representing the different 'families' of instructions and describe (in less formal terms) how the remaining instructions in the JVM instruction set differ from those.

4 Future work

Topics for further work towards a complete JVM specification include:

- Specify which exception is thrown when a given premise of a rule fails to hold. For each instruction, and for each premise that can fail, one may introduce an additional rule which lists those premises that do hold, a single premise that fails and the exception that must be thrown in that case.
  This will considerably increase the number of rules, but should not affect the overall structure of our specification.
- Specify the semantics of parallelism (threads). This would require rethinking the specification, although many parts of the current specification could be reused (specifically, all parts not related to field or method access).
- Specify the byte-code verification conditions. The informal specification is rather unclear, especially concerning the verification conditions for local subroutines (the `jsr` and `ret` instructions), exceptions handlers, and their interaction, as discussed in [14].
- Find the 'early' premises for each JVM instruction: those that may be checked by a byte-code verifier at load-time. Prove that byte-code which passes such a verifier cannot fail the 'early' premises, and then remove those premises from the rules.

5 Related work

Börger and Schulte give a formal semantics of Java byte-code, factorized into a number of sub-languages [3]. Their JVM semantics serves as a basis for defining a compilation scheme from Java source programs to Java byte-code. In their specification, Börger and Schulte assume that the Java byte-code has already been verified by a byte-code verifier. Hence, they deal with fewer details in the JVM instruction set, although their approach resembles ours to some extent.

Also closely related to our work is Cohen's *Defensive Java Virtual Machine*, an executable specification expressed in ACL2, developed at Computational Logic Inc. [4]. Cohen's specification is fully formal and hence more precise than ours, but leaves out many aspects of the JVM, yet is far longer than ours (385 pages). It is probably better suited for machine manipulation and less suited for human readers.

Stata and Abadi show how to formalize some aspects of Java byte-code verification as a type system [14]. Thus while our work concerns the dynamic semantics of Java byte-code, their work concerns its static semantics.

We are also aware of other related work, e.g. the Alves-Foss book on Java semantics [6] and Diehl's formalization of Java compilation [5], but have not yet compared our work to theirs.

6 Conclusion

We have gained a thorough understanding of the JVM in a relatively short time. This has been an invaluable aid when subsequently implementing Java byte-code generators. Concretely, the present work has served as a basis for the design of the *SML-JVM Toolkit* [1], a toolkit for manipulating Java class file and Java bytecode.

Moreover, several ambiguities in the 'official' JVM specification [9] have been revealed, some of which are now being addressed by JavaSoft. These ambiguities in the informal specification (and more) are listed in the unofficial *Java Spec Report* [12], whose section on the JVM specification owes much to the present work.

The dynamic semantics of Java byte-code presented here has not yet been validated formally. It shows, however, that the JVM, although it is a fairly complicated virtual machine, can be given a precise yet comprehensible description using well-known mathematical concepts and notation.

We believe that the standardization of the JVM would benefit from using a specification style similar to that presented here.

References

1. P. Bertelsen. The SML-JVM toolkit, version 0.5. Web-pages at `http://www.dina.kvl.dk/~pmb`.
2. P. Bertelsen. Semantics of Java byte code. Technical report, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Copenhagen, Denmark, April 1997. Web pages at `http://www.dina.kvl.dk/~pmb`.
3. E. Börger and W. Schulte. Defining the Java Virtual Machine as platform for provably correct Java compilation. MFCS'98 Proceedings, 1998. To appear.
4. R.M. Cohen. The defensive Java virtual machine, version 0.5. Technical report, Computational Logic Inc., Austin, Texas, May 1997. Web pages at `http://www.cli.com`.
5. Stephan Diehl. A Formal Introduction to the Compilation of Java. *Software – Practice and Experience*, 28(3):297–327, 1998.
6. J. Alves-Foss (ed.). *Formal syntax and Semantics of Java*. Springer-Verlag, 1998. To be published.
7. M. J. C. Gordon and T. F. Melham (eds.). *Introduction to HOL: A theorem proving environment for higher order logic*. Cambrigde University Press, 1993. ISBN 0-521-44189-7.
8. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990. ISBN 0-13-880733-7.
9. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996. ISBN 0-201-63452-X.
10. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
11. L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science. Springer-Verlag, 1994. ISBN 3-540-58244-4.
12. R. Perera and P. Bertelsen. The unofficial Java Spec Report. Available at `http://www.nodule.demon.co.uk/java`.
13. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.

14. R. Stata and M. Abadi. A type system for Java bytecode subroutines. Technical report, Digital Equipment Corporation Systems Research Center, July 1997.

///