Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 155–172
www.elsevier.com/locate/entcs

Memory Representations in Rewriting Logic Semantics Definitions

Mark Hills[1,2]

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, US

Abstract

The executability of rewriting logic makes it a compelling environment for language design and experimentation; the ability to interpret programs directly in the language semantics develops assurance that definitions are correct and that language features work well together. However, this executability raises new questions about language semantics that don't necessarily make sense with non-executable definitions. For instance, suddenly the performance of the semantics, not just language interpreters or compilers based on the semantics, can be important, and representations must be chosen carefully to ensure that executing programs directly in language definitions is still feasible. Unfortunately, many obvious representations common in other semantic formalisms can lead to poor performance, including those used to represent program memory. This paper describes two different memory representations designed to improve performance: the first, which has been fully developed, is designed for use in imperative programs, while the second, still being developed, is intended for use in a variety of languages, with a special focus on pure object-oriented languages. Each representation is described and compared to the initial representation used in the language semantics, with thoughts on reuse also presented.

Keywords: Rewriting logic, programming language semantics, performance.

1 Introduction

The executability of rewriting logic makes it a compelling environment for language design and experimentation; the ability to interpret programs directly in the language semantics helps develop assurance that definitions are correct and that language features work well together. However, this executability raises new questions about definitions, questions which would not make sense in the context of non-executable language definitions. How fast can programs be executed in the semantics? What is the performance impact, at the semantic level, of various language features? Are there ways to define these features in rewriting logic that will

[1] Supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, by the Microsoft/Intel funded Universal Parallel Computing Research Center at UIUC, and by several Microsoft gifts.

[2] Email: mhills@cs.uiuc.edu

doi:10.1016/j.entcs.2009.05.018

make programs running using the definition faster or slower? Do semantic changes which improve the execution performance also improve analysis performance, or are there changes which speed up one to the detriment of the other? Do certain representations for items in the program configuration/state provide better performance than others?

In earlier work [14], the impact of design decisions on performance was explored in the context of KOOL [13], a pure, object-oriented language defined using rewriting logic semantics. That work focused on two particular facets of the language design: the representation of all values as objects, and the use of a global flat memory representation. Analysis performance was improved by making changes to each of these facets. For the first, scalar values were introduced alongside objects, with auto-boxing functionality used to automatically convert scalars to objects when needed. For the second, memory was divided into pools, one global pool for shared memory, and one local pool for memory locations visible in only one thread. These modifications improved analysis performance dramatically, with auto-boxing also improving execution performance.

However, these improvements came at a cost: both involved changes directly to the language semantics. Even though both changes appeared to preserve externally-visible behavior, they made the semantics much more complex, making the language definition harder to understand. These changes also introduced the burden of proving that they were behavior-preserving, something that was assumed but not proven in the prior work. Beyond this, changes to the semantics go to the heart of the definition; auto-boxing in some sense changed KOOL from a pure object-oriented language into a language more like Java, which has both scalars and objects. So, while changes to the semantics can improve performance, they can also force changes into the language semantics that may be unacceptable.

One promising direction is to change, not the rules defining the semantics of language features, but those defining the surrounding configuration. Since the configuration is seen as an abstraction inside the language rules, changes to the configuration can occur without requiring changes to these rules. Beyond this, configurations are often reused between languages, allowing improvements to be directly leveraged inside other definitions. This paper introduces changes to two languages, SILF and KOOL, both at the level of the memory representation in the configuration. After a brief introduction to rewriting logic semantics in Section 2, Section 3 shows the first memory representation change, introducing a stacked memory model to SILF; performance comparisons show the benefit of this model, while comparisons with the prior version show that few changes to the semantics are needed. After this, Section 4 shows the second change, the addition of a basic mark-sweep garbage collector to KOOL, including discussions of performance and reusability. Finally, Section 5 mentions some related work, while Section 6 concludes. All code referenced in the paper is available online [15].

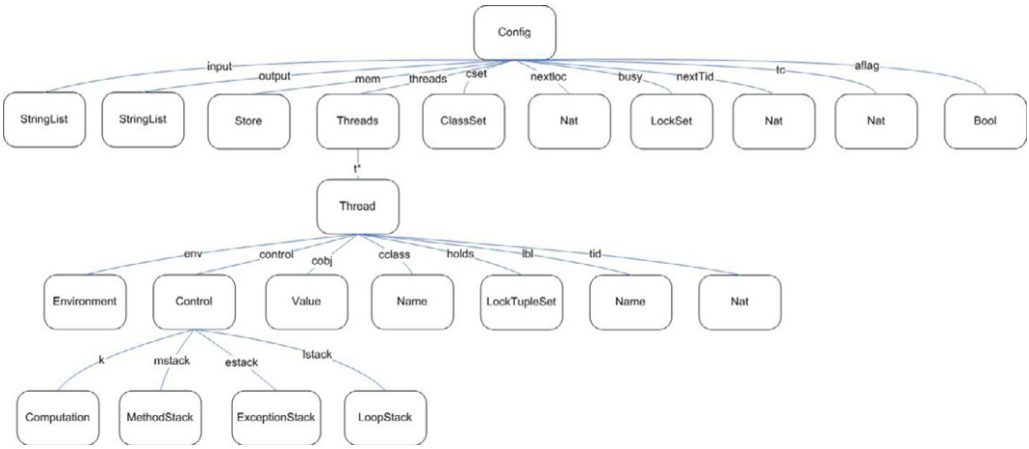

Fig. 1. KOOL State Infrastructure

2 Rewriting Logic Semantics

Rewriting logic semantics (RLS), proposed by Meseguer and Rosu [20,21], builds upon the observation that programming languages can be defined as rewriting logic theories. By doing so, one gets essentially "for free" not only an interpreter and an initial model semantics for the defined language, but also a series of formal analysis tools obtained as instances of existing tools for rewriting logic. It is possible to define languages using rewriting logic semantics in a wide variety of different styles. The style used here is often referred to as a *continuation-based* style, and is similar to the style being used in the newer K [23] technique for defining programming languages.

The semantics of SILF and KOOL are defined using Maude [6,7], a high-performance language and engine for rewriting logic. The current program is represented as a "soup" (multiset) of nested terms representing the current computation, memory, locks held, etc. A visual representation of this term, the state infrastructure, is shown in Figure 1 for the KOOL language. Here, each box represents a piece of information stored in the program state, with the text in each box indicating the information's *sort*, such as `Name` or `LockSet`. Information stored in the state can be nested, with the labels on edges indicating the operator name used to reference the nested information: each thread `t` contains control context stored inside `control`, for instance. The most important piece of information is the `Computation` `k`, nested inside `Control`, which is a first-order representation of the current computation made up of a list of instructions separated by `->`. The computation can be seen as a stack, with the current instruction at the left and the remainder (continuation) of the computation to the right.

Figure 2 shows examples of Maude equations included in the KOOL semantics. The first three equations (shown with `eq`) process a conditional. The first indicates the value of the guard expression `E` must be computed before a branch statement (`S` or `S'`) is evaluated; to do this, `E` is put before the branches on the computation, with the branches saved for later use by putting them into an `if` computation

```
eq stmt(if E then S else S' fi) = exp(E) -> if(S,S') .
eq val(primBool(true)) -> if(S,S') = stmt(S) .
eq val(primBool(false)) -> if(S,S') = stmt(S') .

ceq threads(t(control(k(llookup(L) -> K) CS) TS) KS) mem(Mem)
  = threads(t(control(k(val(getValue(Mem,L)) -> K) CS) TS) KS) mem(Mem)
 if not isShared(Mem,L) .
```

Fig. 2. Sample KOOL Rules

Integer Numbers	N ::=	$(+\|-)?(0..9)^+$
Declarations	D ::=	var I | var $I[N]$
Expressions	E ::=	N | E + E | E - E | E * E | E / E | E % E | - E | $E < E$ | $E <= E$ | $E > E$ | $E >= E$ | $E = E$ | E != E | E and E | E or E | not E | N | $I(El)$ | $I[E]$ | I | read
Expression Lists	El ::=	$E\ (,E)^*$ | *nil*
Statements	S ::=	$I := E$ | $I[E] := E$ | if E then S fi | if E then S else S fi | for $I := E$ to E do S od | while E do S od | S; S | D | $I(El)$ | return E | write E
Function Declarations	FD ::=	function $I(Il)$ begin S end
Identifiers	I ::=	$(a-zA-Z)(a-zA-Z0-9)^*$
Identifier Lists	Il ::=	$I\ (,I)^*$ | *void*
Programs	Pgm ::=	$S?\ FD^+$

Fig. 3. Syntax for SILF

item. The second and third execute the appropriate branch based on whether the guard evaluated to `true` or `false`. The fourth, a conditional equation (represented with `ceq`), represents the lookup of a memory location. The equation states that, if the next computation step in this thread is to look up the value at location `L`, and if that value is at a location that is not shared, then the result of the computation is the value found at location `L` in store `Mem`. `CS`, `TS`, and `KS` match the unreferenced parts of the control state, thread state, and threads container, respectively, while `K` represents the rest of the computation in this thread. Note that, since the fourth equation represents a side-effect, it can only be applied when it is the next computation step in the thread (it is at the head of the computation), while the first three, which don't involve side-effects, can be applied at any time.

3 SILF and Stacked Memory

SILF [12], the **S**imple **I**mperative **L**anguage with **F**unctions, is a basic imperative language with many core imperative features: functions, global variables, loops, conditionals, and arrays. Programs are made up of a series of variable and function declarations, with a designated function named `main` serving as the entry point to the program. The syntax of SILF is shown in Figure 3, while a sample program, which computes the factorial of 200 recursively, is shown in Figure 4.

3.1 The SILF Memory Model

In SILF, memory is allocated automatically for global and local variables and arrays, including for the formal parameters used in function calls. Users are not able to

```
function factorial(n)
begin
  if n = 0 then
    return 1
  else
    return n * factorial(n - 1)
  fi
end

function main(void)
begin
  write factorial(200)
end
```

Fig. 4. Recursive Factorial, SILF

allocate additional storage with operations like `new` or function calls like C's `malloc`, and are not able to create pointers/references or capture variable addresses. SILF includes a simple memory model, where memory is represented as a set of `Location` × `Value` pairs, referred to as `StoreCell`s; the entire set is just called the `Store` [3]. This, or something very similar, is the standard model used in a number of languages defined using K or the continuation-based definitional style [4]:

```
sorts StoreCell Store .
subsort StoreCell < Store .
op [_,_] : Location Value -> StoreCell .
op nil : -> Store .
op __ : Store Store -> Store [assoc comm id: nil] .
```

The main advantage of this model is that memory operations are simple to define; memory update and lookup can be performed just using matching within the `Store`, as shown here for variable lookup:

```
eq k(exp(X) -> K) env(Env [X,L])
 = k(lookupLoc(L) -> K) env(Env [X,L]) .
eq k(lookupLoc(L) -> K) store(Mem [L,V])
 = k(val(V) -> K) store(Mem [L,V]) .
```

Here, `exp(X)` means there is an expression `X`, a variable name; matching is used to find the location of `X`, `L`, in the current environment, a set of `Name` × `Location` pairs. This triggers the lookup of location `L` using operation `lookupLoc`. When this operation is processed, matching is performed against the store, returning the value `V` stored at location `L`.

This model has a major disadvantage, though: old locations are never removed from the store, even when they become unreachable, which happens quite often (formals become unreachable after each function return, for instance). As memory grows, it takes longer to match against the store, slowing execution performance.

[3] In SILF, a `StoreCell` is actually called a `<Location><Value>`, which has an associated `<Location><Value>Set` representing the `Store`. The terminology is changed here to make it simpler to read and type.

[4] Newer definitions often use the built-in `MAP` module instead; this is used by KOOL, for instance, as described in Section 4.

3.2 Stacked Memories

As mentioned above, it is not possible in SILF to dynamically allocate memory or take the addresses of variables. This prevents addresses from escaping a function, since there is no way to return a pointer to something inside the function; because of this, it should be possible to discard all memory allocated for the function call when the function returns [5]. A conceptually simple way to do this is to change from a flat memory to a *stack* of memories, with the memory for the current function on top and the global memory on the bottom. Memories can still be treated as sets of `StoreCell`s, but each set can be much smaller, containing just the cells allocated in the current function, and each set can easily be discarded simply by popping the stack at function return. Following this reasoning, the memory model for SILF can be changed appropriately:

```
sort StackFrame Stack .
subsort StackFrame < Stack .
op [_,_] : Nat Store -> StackFrame .
op nil : -> Stack .
op _,_ : Stack Stack -> Stack [assoc id: nil] .
```

Here, each element of the stack is referred to by the name `StackFrame`, a familiar term meant to show the intuition behind the technique. Each `StackFrame` is actually a pair, a `Store` and a natural number representing the first location in the frame; attempts to access a lower numbered location need to check in earlier frames, here the bottom frame, since SILF's scoping only allows access to local or global names and does not allow nested functions. These `StackFrame`s are assembled into `Stack`s, with the head of the list as the top element of the stack and the last element of the list the global frame.

Location lookup is now slightly more involved:

```
op stackLookup : Location Stack -> Value .
op lvsLookup : Location Store -> Value .

eq k(lookupLoc(L) -> K) store(ST)
 = k(val(stackLookup(L,ST)) -> K) store(ST) .

ceq stackLookup(loc(N),([Nb,Mem], ST))
  = lvsLookup(loc(N),Mem)
 if N >= Nb .

ceq stackLookup(loc(N),([Nb,Mem], ST, [Nb',Mem']))
  = lvsLookup(loc(N),Mem')
```

[5] This seems restrictive, but is actually standard for stack-allocated memory in imperative or object-oriented languages without address capture, such as Java or Pascal. Heap-allocated memory could not be similarly discarded.

	Standard (Flat) Memory Model			Stacked Memory Model		
Test Case	Time (sec)	Total Rewrites	Rewrites/sec	Time (sec)	Total Rewrites	Rewrites/sec
factorial	3.711	72158	20162	0.747	82173	135148
factorial2	1664.280	1321592	792	11.245	1505902	135593
ifactorial	1.047	65755	71780	0.978	83520	99422
ifactorial2	43.861	1204799	27676	15.441	1530809	100048
fibonacci	29.014	221932	7699	1.939	248870	138150
qsort	111.623	835552	7511	15.374	1087874	72071
ssort	21.557	751352	35114	14.657	1047118	72304

Single 3.40 GHz Pentium 4, 2 GB RAM, OpenSuSE 10.2, kernel 2.6.18.8-0.7-default, Maude 2.3. Times and rewrites per second averaged over three runs of each test.

Fig. 5. SILF: Comparing Memory Model Performance

```
if N < Nb .

eq lvsLookup(L,([L,V] Mem)) = V .
```

Now, location lookup just triggers `stackLookup`, which has two equations representing the two cases mentioned above. If the location number `N` is at least `Nb`, the smallest location in the stack, the location should be in the current stack frame. If the location number is smaller than `Nb`, it must be the location of a global variable, which should be in the frame at the bottom of the stack. Both cases then use a helper, `lvsLookup`, to find location `L` inside the store in the appropriate frame, using matching to find the matching `StoreCell` and retrieve the value.

3.3 Evaluation

To evaluate the effectiveness of the stacked memory model versus the standard flat memory model, seven test cases were executed in SILF under both models. The test cases implemented several standard recursive and iterative algorithms, with the intent being to not bias the tests in favor of either recursive or iterative styles of programming. The test cases were:

- `factorial`, recursively calculating the factorial for 20, 40, ..., 180, 200;
- `factorial2`, same as `factorial`, but for 1 ... 200;
- `ifactorial`, an iterative version of `factorial`;
- `ifactorial2`, an iterative version of `factorial2`;
- `fibonacci`, a recursive algorithm computing the fibonacci numbers from 1 to 15;
- `qsort`, a quick sort of two arrays of 100 elements;
- `ssort`, a selection sort of two arrays of 100 elements.

In all cases, the total execution time, total number of rewrites, and rewrites per second were recorded. The performance results are shown in Figure 5.

The results indicate that the stacked memory model provides improved performance over the flat memory model in many different programs, including all those tested here. Based on the total rewrites it is clear that the stacked model in some

Program	$P ::=$	$C^*\ E$
Class	$C ::=$	class X is $D^*\ M^*$ end | class X extends X' is $D^*\ M^*$ end
Decl	$D ::=$	var $\{X,\}^+$;
Method	$M ::=$	method X is $D^*\ S$ end | method X ($\{X',\}^+$) is $D^*\ S$ end
Expression	$E ::=$	X | I | F | B | Ch | Str | (E) | new X | new X ($\{E,\}^+$) | self | $E\ X_{op}\ E'$ | $E.X(())^?$ | $E.X(\{E,\}^+$) | super() | super.$X(())^?$ | super.$X(\{E,\}^+$) | super($\{E,\}^+$) | $E\ ==\ E$ | $E\ /=\ E$
Statement	$S ::=$	E <- E'; | begin $D^*\ S$ end | if E then S else S' fi | if E then S fi | try S catch $X\ S$ end | throw E ; | for X <- E to E' do S od | while E do S od | break; | continue; | return; | return E; | $S\ S'$ | E; | assert E; | X: | spawn E ; | acquire E ; | release E ; | typecase E of Cs^+ (else $S)^?$ end
Case	$Cs ::=$	case X of S

$X \in Name, I \in Integer, F \in Float, B \in Boolean, Ch \in Char, Str \in String, X_{op} \in Operator\ Names$

Fig. 6. KOOL Syntax

sense does more work, which is needed to maintain the stack and look up memory locations at different levels. It is also clear, though, that it does the work much more quickly, shown in the Rewrites/sec column, illustrating the benefit to matching performance of keeping the store small. The cost for making the change is fairly low, as well, since changing to the stacked memory model required few changes to SILF. Beyond adding new sorts and operations to model having stacks of memory frames, it was only necessary to change 6 existing SILF equations – specifically, those equations already dealing with memory, or with function call and return.

4 KOOL and Garbage Collection

KOOL [5,13] is a concurrent, dynamic, object-oriented language with support for many features found in common object-oriented languages. This includes standard imperative features such as assignments, conditionals, and loops, as well as object-oriented features including single inheritance, dynamic dispatch, and run-time type inspection. KOOL is a pure object-oriented language, meaning that all values are objects; operations such as addition are actually carried out via message sends. KOOL is currently untyped, with runtime exceptions thrown when invalid operations are attempted, such as message sends with the wrong number of parameters or sends to targets that do not define the target method. Concurrency follows a simple model, with multiple threads of execution accessing a shared memory and locks acquired on objects (similar to the lock model present in Java). The syntax for KOOL is shown in Figure 6, while a sample program is shown in Figure 7. The configuration for KOOL was shown in Section 2 in Figure 1.

```
class Factorial is
  method Fact(n) is
    if n = 0 then return 1;
    else return n * self.Fact(n-1);
    fi
  end
end

console << (new Factorial).Fact(200)
```

Fig. 7. Recursive Factorial, KOOL

4.1 The KOOL Memory Model

The KOOL memory representation is structured similarly to the default representation used by SILF, with two differences. First, instead of defining the store explicitly, it is defined using the built-in `MAP` module. Second, instead of just mapping locations to values, the stores maps locations to a sort `ValueTuple`, which contains the value as one of its projections:

```
protecting MAP{Location,ValueTuple} *
  (sort Map{Location,ValueTuple} to Store) .
op [_,_,_] : Value Nat Nat -> ValueTuple .
```

Lookups and updates then use the `MAP`-provided functionality, supplemented with some additional operations for adding more than one mapping to the `Store` at once and for extracting the value from the tuple.

Since KOOL is multi-threaded, memory accesses to shared locations can compete. In earlier work done on analysis performance [14], memory was segregated into *memory pools*, with a shared pool for locations accessible from multiple threads and a non-shared pool for locations accessible from only one thread. Currently, this is represented instead as one memory pool, with the first `Nat` flag in the `ValueTuple` indicating whether the location is shared. Based on the setting of this flag, rules or equations are used to access or update values in the store. The logic for location lookup is shown below; similar equations and rules for location assignment are not shown. Here, `L` is a location in memory, `N` and `M` are natural numbers, `V` is a value, `Mem` is the store, and `CS`, `TS`, and `KS` represent other parts of the state that are not needed directly in the equations and rules:

```
  op llookup : Location -> ComputationItem .
  op slookup : Location -> ComputationItem .
  op isShared : Store Location -> Bool .
  op getValue : Store Location -> Value .

  eq isShared(_‘,_(L |-> [V,1,N], Mem), L) = true .
  eq isShared(Mem, L) = false [owise] .

 ceq getValue(Mem,L) = V if [V,N,M] := Mem[L] .

 ceq threads(t(control(k(llookup(L) ->           K) CS) TS) KS) mem(Mem)
   = threads(t(control(k(val(getValue(Mem,L)) -> K) CS) TS) KS) mem(Mem)
  if not isShared(Mem,L) .

 ceq threads(t(control(k(llookup(L) -> K) CS) TS) KS) mem(Mem)
   = threads(t(control(k(slookup(L) -> K) CS) TS) KS) mem(Mem)
  if isShared(Mem,L) .

  rl threads(t(control(k(slookup(L) ->           K) CS) TS) KS) mem(Mem)
  => threads(t(control(k(val(getValue(Mem,L)) -> K) CS) TS) KS) mem(Mem) .
```

The first two equations define the `isShared` operation, which returns true when `L` is marked as shared (i.e., accessible by multiple threads) in `Mem`. The third defines `getValue`, used for extracting the value at location `L` from the value tuple stored in `Mem`. Following these definitions, the remaining two equations and one rule define the actual process of retrieving the value at location `L` from the store. In the first equation, `L` is not shared, so the value can be retrieved from `Mem` directly using `getValue`. In the second equation, `L` is shared; this causes lookup to switch over to a shared lookup operation. The rule then defines this shared lookup; the definition is identical to that for unshared locations, except in this case a rule is used, indicating that this could represent a race condition.

This model shares the same disadvantage as the original SILF model – old locations are never removed from the store, even when they become unreachable. And, since KOOL is a pure object-oriented language (boxing is not used here), locations become unreachable constantly. An expression such as $1 + 2 + 3$ is syntactic sugar for $(1. + (2)). + (3)$. New objects are created for the numbers 1, 2, 3, 3 again, and 6, with all but the last just temporaries that immediately become garbage. Unlike in SILF, a simple solution like stack frames cannot be used to remove unreachable objects, since often references to objects will be returned as method results. Without more sophisticated analysis, such as escape analysis [22,1], it must be assumed that any objects created in a method could escape, meaning they cannot just be discarded on method exit.

Overall, the constant expansion of memory, the lack of obvious ways to reduce the memory size, and the performance decrease related to using a larger memory can make it difficult to run even some fairly small programs just using the semantics-based interpreter. Since one of the goals of defining KOOL is to allow for quick, easy experimentation with language features, a way to decrease the memory size and increase performance, without having to change language features in unwanted ways, is crucial.

4.2 Defining Garbage Collection

A solution common to object-oriented languages is to use garbage collection. Garbage collection fits well with KOOL's allocation model, which uses `new` to create new objects but does not provide for explicit deallocation; it also accommodates the regular use of intermediate objects, which often quickly become garbage, in computations. If done properly, a GC-based solution also has the advantage that it can be defined at the level of the KOOL configuration, leaving the rules used to define language features unchanged.

The garbage collector defined below is a simple mark-sweep collector [17]. Mark-sweep collectors work by first finding a set of *roots*, which are references into the store. All locations transitively reachable from the roots are marked as being reachable (the marking phase); all unmarked locations are then removed from memory (the sweeping phase). GC equations can be divided into language-dependent equations, which need to be aware of language constructs, and language-independent equations, which just work over the structure of the memory and could be used in

any language with the same `Store` definition.

4.2.1 Language-Independent Rules

The rules to mark and sweep memory locations during collection are separated into four phases. In the first phase, `gcClearMem` (seen in state item `ingc`), a flag on each memory location (the third element of the `ValueTuple`) is set to 0. By default, then, all memory locations are assumed to be unreachable at the start of collection. The state component used to hold the memory is also renamed, from `mem` to `gcmem`. This has the benefit of blocking other memory operations during collection without requiring the other operations to even be aware of the collector:

```
op gcClearMem : -> GCState .
op unmarkAll : Store -> Store .

eq mem(Mem) ingc(gcClearMem)
 = gcmem(unmarkAll(Mem)) ingc(gcMarkRoots) .

eq unmarkAll(_‘,_(L |-> [V,N,M], Mem))
 = _‘,_((L |-> [V,N,0]), unmarkAll(Mem)) .
eq unmarkAll(Mem) = Mem [owise] .
```

In the second phase, `gcMarkRoots`, all locations directly referenced in computation portions of the KOOL state (inside the computation and in the stacks, for instance, but not in the memory) are found using `KStateLocs`, one of the language-dependent portions of the collector. Each of these root locations, stored in `LS`, is then marked by setting the third element of the `ValueTuple` at that location to 1:

```
op gcMarkRoots : -> GCState .
op markLocsInSet : Store LocationSet -> Store .
op mark : Store Location -> Store .

ceq threads(KS) ingc(gcMarkRoots    ) gcmem(Mem                  )
  = threads(KS) ingc(gcMarkTrans(LS)) gcmem(markLocsInSet(Mem,LS))
 if LS := KStateLocs(KS) .

eq markLocsInSet(Mem, (L LS)) = markLocsInSet(mark(Mem,L), LS) .
eq markLocsInSet(Mem, emptyLS) = Mem .

eq mark(_‘,_(L |-> [V,N,M], Mem), L) = _‘,_(L |-> [V,N,1], Mem) .
```

Next, the third phase, `gcMarkTrans`, determines the locations reachable transitively through the root locations. It works using both the `iterate` and `unmarkedOnly` operations; the first determines the set of locations reachable in one step from a given set of locations (if an object at location `L` holds references to

objects at locations L1 and L2, L1 and L2 would be reachable in one step, but not any locations referenced by the objects at L1 or L2), while the second filters this to only include locations that have not already been marked. At each iteration the locations found are marked and the process continues from just these newly-marked locations, ensuring that the traversal eventually terminates when no new, unmarked locations are found:

```
op gcMarkTrans : LocationSet -> GCState .
op iterate : LocationSet Store -> LocationSet .
op unmarkedOnly : LocationSet Store -> LocationSet .

ceq threads(KS) ingc(gcMarkTrans(LS) ) gcmem(Mem )
  = threads(KS) ingc(gcMarkTrans(LS')) gcmem(Mem')
 if LS' := iterate(LS,Mem) /\ LS' =/= emptyLS /\
    Mem' := markLocsInSet(Mem,LS') .

 eq threads(KS) ingc(gcMarkTrans(LS)) gcmem(Mem)
  = threads(KS) ingc(gcSweep        ) gcmem(Mem) [owise] .

 eq iterate(L LS, Mem)
  = unmarkedOnly(ListToSet(valLocs(getValue(Mem,L))),Mem)
    iterate(LS, Mem) .
 eq iterate(emptyLS,Mem) = emptyLS .
```

Finally, the fourth phase, gcSweep, uses the removeUnmarked operation to discard all memory locations not marked during the sweep performed in steps two and three. It also moves the store back into mem, so other parts of the semantics can again see the store:

```
op gcSweep : -> GCState .

eq ingc(gcSweep) gcmem(Mem               )
 = ingc(noGC(0)) mem(removeUnmarked(Mem)) .
```

4.2.2 Language-Dependent Equations

Language-dependent equations are used to gather the set of roots from the computation and any other parts of the state (such as stacks) that may contain them. Traversal of the state is initiated using KStateLocs, which returns a set of all locations found in a given state. KStateLocs is defined inductively over the various state components, with other operations specifically designed to deal with computations, computation items, stacks, and other state components. Examples of the equations used to find the locations inside the method stack and the computation are shown below:

```
op KStateLocs : KState -> LocationSet .
eq KStateLocs(mstack(MSTL) CS) = MStackLocs(MSTL) KStateLocs(CS) .

op MStackLocs : MStackTupleList -> LocationSet .
eq MStackLocs ([K,CS,Env,oref(L),Xc], MSTL)
 = KLocs(K) KStateLocs(CS) ListToSet(envLocs(Env))
   ListToSet(valLocs(oref(L))) MStackLocs(MSTL) .
eq MStackLocs(empty) = emptyLS .

op KLocs : Computation -> LocationSet .
eq KStateLocs(k(K) CS) = KLocs(K) KStateLocs(CS) .
eq KLocs(llookup(L) -> K) = L KLocs(K) .
```

`KLocs` deserves special comment, since it is the main operation that needs to be modified to account for new language features. `KLocs` is defined for each computation item in the language that may hold locations. This means that, when new computation items which can contain locations are added, the collector must be updated properly. A method of automatically transforming a theory into one with garbage collection would eliminate this potential source of errors.

Along with the equations used to find the roots, additional equations are used to find any locations referenced by a value (for instance, the locations referenced in the fields of an object). These equations are then used when finding the set of locations reachable transitively from the root locations. In this case, it was possible to reuse equations developed in earlier work [14] that were used to find all locations reachable from a starting location so they could be marked as shared.

4.2.3 Triggering Garbage Collection

Garbage collection is triggered using the `triggerGC` computation item:

```
op triggerGC : Nat -> ComputationItem .
```

This allows the language designer to decide how aggressive the collection policy should be. Currently, `triggerGC` has been added to the three equations in the memory operations that are used to allocate storage; no equations used to define KOOL language features have been modified. The `Nat` included in `triggerGC` contains the number of allocated locations. This is then used by the collector to decide when to begin collecting:

```
ceq threads(t(control(k(triggerGC(N) -> K) CS) TS) KS) ingc(GC)
  = threads(t(control(k(K) CS) TS) KS) ingc(GC)
 if runningGC(GC) .

ceq threads(t(control(k(triggerGC(N) -> K) CS) TS) KS)
      ingc(noGC(N')) gccount(GN)
```

```
    = threads(t(control(k(K) CS) TS) KS)
        ingc(gcClearMem) gccount(s(GN))
   if (N + N') >= 1000 .

  ceq threads(t(control(k(triggerGC(N) -> K) CS) TS) KS)
        ingc(noGC(N'))
    = threads(t(control(k(K) CS) TS) KS) ingc(noGC(N + N'))
   if (N + N') < 1000 .
```

The first equation just discards the trigger if the collector is already active. The second initiates collection when 1000 or more allocations have occurred [6] – `N` being the number of new allocations, `N'` being the number already reported with prior `triggerGC`s. The last equation increments the number of reported allocations stored in `noGC` by the number of new allocations when the sum is less than 1000.

4.3 Evaluation

To evaluate the effectiveness of the garbage collector, five test cases were executed in KOOL, both with the collector enabled and disabled. Along with three numerical test cases, two test cases were added that were designed to generate a large amount of garbage. The test cases were:

- `factorial`, recursively calculating the factorial for 20, 40, ..., 180, 200;
- `ifactorial`, an iterative version of `factorial`;
- `fibonacci`, a recursive algorithm computing the fibonacci numbers from 1 to 15;
- `addnums`, which sums the numbers 1...100, 1...200, ..., 1...1000;
- `garbage`, which defines a class that holds an integer and then creates a temporary object of this class (which quickly becomes garbage) for the numbers 1...2000.

In all cases, the total execution time was recorded. Also recorded were the final size of the store and (in the cases where garbage collection was enabled) the number of collections that occurred. The performance results are shown in Figure 8.

At this point, results are mixed. The `factorial` and `ifactorial` tests do not appear to benefit from garbage collection – in both cases the collector slows performance down, even though it obviously shrinks the size of the store. The result for `fibonacci` shows little difference in execution time, although again the store is much smaller. In these three test cases, the cost of collecting either is higher than the benefit (`factorial`, `ifactorial`) or roughly equal to the benefit (`fibonacci`). However, in the final two test cases, `addnums` and `garbage`, garbage collection obviously helps. Without GC, `addnums` crashes; `garbage` completes in both, but is much faster with collection enabled.

[6] 1000 was chosen after some experimentation, but further experimentation could show that a different number would be better. It may also be the case that there is no ideal number – hence the prevalence of collectors with generational policies, with each generation collected at different intervals.

	GC Disabled		GC Enabled		
Test Case	Time (sec)	Final Store Size	Time (sec)	Final Store Size	# of Collections
factorial	103.060	22193	119.987	300	22
ifactorial	97.100	21103	116.811	106	21
fibonacci	401.334	76915	399.785	935	76
addnums	NA	NA	516.023	946	93
garbage	259.500	32013	147.211	20	32

Single 3.40 GHz Pentium 4, 2 GB RAM, OpenSuSE 10.2, kernel 2.6.18.8-0.7-default, Maude 2.3. Times averaged over three runs of each test.

Fig. 8. KOOL: GC Performance

One goal in developing the collector is to be able to reuse it in other languages. Based on the current design in KOOL, this should be straight-forward. The only part of the collector that is language-specific is the operations and equations used to determine the set of root locations. The other parts of the collector definition are language-independent, and can be reused directly in any language that uses a similar (continuation-based) definitional style.

4.4 Correctness

One point which has not yet been addressed is correctness. Adding a garbage collector to a language should not actually change the results of any computation. The proof that this holds true is sketched here, and will be included in full in a technical report.

First, by definition, if the collector works correctly program behavior will not change. This is because only unreachable locations will be collected; since the values stored at these locations cannot be used in the computation, removing them will not alter the results of the computation. Only if the collector either alters the value at a location or collects a reachable location will program behavior potentially change. Showing that the collector is correct then involves showing that the following facts hold: 1) When the root locations are gathered from the state, no root locations are missed; 2) locations reachable from the root locations are always marked; 3) values at marked locations are not changed by the collector; 4) marked locations are not collected.

The first point can be proved by induction on the depth of the term, showing that each state component and each computation item defined in the semantics is traversed properly by the collector. The second point can be shown using a combination of structural induction, to show that locations stored inside values reachable in memory are properly discovered, and a proof by contradiction: assuming that all roots are properly identified (point 1), assume there is a location, transitively reachable from one of the roots, that is misidentified as unreachable and not marked. Select the closest such location to the roots, `L`. For it to be truly reachable, it must be referenced by a value `V` stored at a reachable location `L'`. However, based on the proof of the first part of this point, locations stored inside reachable values such as `V` are properly discovered, so they will be marked, a contradiction. Alternatively, `L'` was also improperly marked as being unreachable, but `L` was the closest unreach-

able location from the roots, again a contradiction. So, the assumption that `L` is reachable but not identified as such must be incorrect. The third point is easy to show, since no equations defined in the collector modify any values in memory, as is the fourth point, since the equations that define the `removeUnmarked` operation only discard value tuples with a GC flag of 0, while marked locations would have a GC flag of 1.

5 Related Work

There is a large volume of related work on rewriting logic semantics. This includes work on rewriting logic semantics in general [20,21], as well as work on specific languages, such as CML [4], Scheme [8,19], BC [2], CCS [30,2], CIAO [26], Creol [16], ELOTOS [29], MSR [3,24], PLAN [25,26], ABEL [18], FUN [23], and SIMPLE [21]. The work in this paper is related to prior work on SILF [12] and KOOL [14,13].

Research in rewriting logic semantics focused on performance has mainly been aimed at analysis performance instead of execution performance. Earlier work on Java focused on producing a high-performance analysis tool for both the Java language and Java bytecode [11,9], while earlier work on KOOL focused on improving analysis performance, even potentially at the expense of reducing execution performance [14]. There has also been some work on techniques, such as partial-order reduction, for improving analysis speed across multiple languages [10].

Outside the realm of rewriting logic semantics, research on memory management, including garbage collection, has been extensive. The collector presented here is novel only in that it has been formally defined in rewriting logic, but beyond that is a fairly basic mark-sweep collector. Resources on garbage collection [17], including research focused on pure object-oriented languages [27,28], provide additional information on this topic.

6 Conclusions

Having an executable language semantics raises new questions about a language definition that don't make sense in non-executable frameworks. Performance is one of these questions, and is key to making execution of programs during language design and exploration feasible. This paper has focused on performance-related changes to the memory representations of two languages, the imperative language SILF and the pure object-oriented language KOOL.

In SILF, the standard flat memory model was replaced with a stacked memory model, with each function call pushing a new memory layer onto the stack and each return popping the memory layer off. Performance results, presented in Section 3, show that this stacked model consistently outperforms the flat model. Although stack maintenance causes the number of total rewrites to increase, execution time for programs is significantly shorter. The number of rewrites per second is also much higher, showing that matching performance improves dramatically with smaller stores. It should be possible to move this model to other similar languages and

achieve similar improvements.

In KOOL, the existing KOOL memory model was augmented with a basic mark-sweep garbage collector, motivated by an interest in improving performance in a pure OO language definition without requiring changes to language features. The second goal seems successful: the collector is triggered in the part of the semantics used to define memory operations, but requires no changes to the definitions of language features. This should make it easy to move the collector to other languages defined using the same definitional style. Performance results at this point are mixed, though: in some cases performance results are almost the same or slower, while in other cases performance improves dramatically.

There are several interesting areas for further study. One is the impact of garbage collection on analysis. The ability to shrink the memory, as part of an effort to canonicalize state representations, should lead to fewer distinct states in the state space and faster performance. Another is the use of rewriting logic to investigate other garbage collection methods and various analyses, like the escape analysis mentioned in Section 4. Third is the use of alternative methods to improve memory performance, such as adding support directly to Maude for representing arrays. While this may improve performance, depending on how it is implemented it could make debugging language definitions more challenging. Finally, a way to automatically transform a language theory that does not include GC into one that does would be very useful and, because of the uniform nature of the language-dependent equations used in the GC definition, should be straight-forward.

Acknowledgement

I extend my thanks to the anonymous reviewers; this paper has greatly benefited from their comments and suggestions.

References

[1] B. Blanchet. Escape Analysis for Object-Oriented Languages: Application to Java. In *Proceedings of OOPSLA'99*, pages 20–34. ACM, 1999.

[2] C. Braga and J. Meseguer. Modular Rewriting Semantics in Practice. In *Proceedings of WRLA'04*, volume 117 of *ENTCS*, pages 393–416. Elsevier, 2005.

[3] I. Cervesato and M.-O. Stehr. Representing the MSR Cryptoprotocol Specification Language in an Extension of Rewriting Logic with Dependent Types. In *Proceedings of WRLA'04*, volume 117 of *ENTCS*. Elsevier, 2004.

[4] F. Chalub and C. Braga. A Modular Rewriting Semantics for CML. *Journal of Universal Computer Science*, 10(7):789–807, July 2004.

[5] F. Chen, M. Hills, and G. Roşu. A Rewrite Logic Approach to Semantic Definition, Design and Analysis of Object-Oriented Languages. Technical Report UIUCDCS-R-2006-2702, University of Illinois at Urbana-Champaign, 2006.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *Proceedings of RTA'03*, volume 2706 of *LNCS*, pages 76–87. Springer, 2003.

[8] M. d'Amorim and G. Roşu. An Equational Specification for the Scheme Language. *Journal of Universal Computer Science*, 11(7):1327–1348, 2005.

[9] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal Analysis of Java Programs in JavaFAN. In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.

[10] A. Farzan and J. Meseguer. Partial Order Reduction for Rewriting Semantics of Programming Languages. In *Proceedings of WRLA'06*, volume 176 of *ENTCS*, pages 61–78. Elsevier, 2007.

[11] A. Farzan, J. Meseguer, and G. Roşu. Formal JVM Code Analysis in JavaFAN. In *Proceedings of AMAST'04*, volume 3116 of *LNCS*, pages 132–147. Springer, 2004.

[12] M. Hills, T. F. Şerbănuţă, and G. Roşu. A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters. In *Proceedings of WRLA'06*, volume 176 of *ENTCS*, pages 215–231. Elsevier, 2007.

[13] M. Hills and G. Roşu. KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis. In *Proceedings of RTA'07*, volume 4533 of *LNCS*, pages 246–256. Springer, 2007.

[14] M. Hills and G. Roşu. On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance. In *Proceedings of FMOODS'07*, volume 4468 of *LNCS*, pages 107–121. Springer, 2007.

[15] M. Hills and G. Rosu. FSL Semantics Research Homepage. http://fsl.cs.uiuc.edu/semantics.

[16] E. B. Johnsen, O. Owe, and E. W. Axelsen. A runtime environment for concurrent objects with asynchronous method calls. In *Proceedings of WRLA'04*, volume 117 of *ENTCS*. Elsevier, 2004.

[17] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[18] M. Katelman and J. Meseguer. A Rewriting Semantics for ABEL with Applications to Hardware/Software Co-Design and Analysis. In *Proceedings of WRLA'06*, ENTCS. Elsevier, 2006. To appear.

[19] P. Meredith, M. Hills, and G. Roşu. An Executable Rewriting Logic Semantics of K-Scheme. In D. Dube, editor, *Proceedings of the 2007 Workshop on Scheme and Functional Programming*. Laval University, 2007.

[20] J. Meseguer and G. Roşu. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools . In *Proceedings of IJCAR'04*, volume 3097 of *LNAI*, pages 1–44. Springer, 2004.

[21] J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007. Also appeared in *SOS '05*, volume 156(1) of *ENTCS*, pages 27–56, 2006.

[22] Y. G. Park and B. Goldberg. Escape Analysis on Lists. In *Proceedings of PLDI'92*, pages 116–127. ACM, 1992.

[23] G. Rosu. K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2006-2802, Computer Science Department, University of Illinois at Urbana-Champaign, 2006.

[24] M.-O. Stehr, I. Cervesato, and S. Reich. An Execution Environment for the MSR Cryptoprotocol Specification Language. http://formal.cs.uiuc.edu/stehr/msr.html.

[25] M.-O. Stehr and C. Talcott. PLAN in Maude: Specifying an active network programming language. In *Proceedings of WRLA'02*, volume 117 of *ENTCS*. Elsevier, 2002.

[26] M.-O. Stehr and C. L. Talcott. Practical Techniques for Language Design and Prototyping. In *Abstracts Collection of the Dagstuhl Seminar 05081 on Foundations of Global Computing.*, 2005.

[27] D. Ungar and F. Jackson. Tenuring Policies for Generation-Based Storage Reclamation. In *Proceedings of OOPSLA'88*, pages 1–17, 1988.

[28] D. Ungar and F. Jackson. An Adaptive Tenuring Policy for Generation Scavengers. *ACM TOPLAS*, 14(1):1–27, 1992.

[29] A. Verdejo. *Maude como marco semántico ejecutable*. PhD thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain, 2003.

[30] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In *Proceedings of WRLA'02*, volume 117 of *ENTCS*. Elsevier, 2002.