Memory Representations in Rewriting Logic Semantics Definitions

Mark Hills(Presented by Grigore Roșu)
mhills@cs.uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign

30 March 2008

1. Overview and Motivation
2. SILF and Stacked Memory
3. KOOL and Garbage Collection
4. Conclusions

Outline

1. Overview and Motivation
2. SILF and Stacked Memory
3. KOOL and Garbage Collection
4. Conclusions

Rewriting Logic Semantics

Rewriting logic semantics provides a powerful environment for language prototyping and design:

- Rewriting logic provides formalism for specifying language semantics
- K definitional style provides methods to model complex language features, such as jumps (goto, break, continue, exceptions) and dynamic method dispatch
- Tool support provides ability to try language features quickly by executing and analyzing programs directly in semantics

Executability

Executability raises new questions that don't make sense with non-executable semantics, such as:

- How fast can programs be executed using the semantics?
- In what ways can features be modified to maintain equivalent behavior but improve performance?
- What changes to the semantics will improve analysis performance? Will these improve execution performance as well?

The Importance of Performance

Execution performance can be critical for experimenting with language features – programs can quickly become too large (in code size, memory usage, etc) to evaluate in the semantics otherwise.

Semantics Changes and Analysis Performance

Prior work focused on improving analysis performance by modifying language semantics:

- KOOL language modified to use *auto-boxing* (automatic conversion of scalars to objects where needed) and *memory pools* (shared and unshared)
- Auto-boxing improved execution and analysis performance: reduced number of method calls, use of memory, complexity of language operations
- Memory pools improved analysis performance: non-shared memory operations used equations, reduced state space size

More details in [HillsRosu07].

Costs of Semantics Changes

Improving performance by modifying language semantics can come at a cost:

- Changes *believed* to be equivalent, but would need to do formal proof to show this to be the case
- Changes to language features break modularity, making it harder to reuse features in other languages
- "Performance-aware" definitions become more complicated
- Semantics of language features may not be best place for performance improvements – auto-boxing is an optimization, but moving it into semantics of message dispatch makes it part of the language feature

Memory-Representation Optimization

- Memory representation part of language-independent definition, provides *interface* containing operations that allow features to interact with memory
- Often reused in other languages, so improvements could be leveraged in multiple definitions
- Use of interface to memory maintains modularity of language features, minimizing number of changes needed outside memory modules – i.e., improvements to memory representation need not change individual language features

Outline

1. Overview and Motivation
2. SILF and Stacked Memory
3. KOOL and Garbage Collection
4. Conclusions

SILF

- SILF is a basic imperative language – **S**imple **I**mperative **L**anguage with **F**unctions
- Includes many features found in imperative languages, including global variables and arrays
- Does not include some features which make memory management hard – no address capture, no dynamic memory allocation

SILF: Flat Memory Model

- Original memory model was "flat" – memory represented as a single set of location/value pairs
- Memory never "cleaned up" – continues growing as new locations allocated

```
1  sorts StoreCell Store .
2  subsort StoreCell < Store .
3  op [_,_] : Location Value -> StoreCell .
4  op nil : -> Store .
5  op __ : Store Store -> Store [assoc comm id: nil] .
6
7  eq k(lookupLoc(L) -> K) store(Mem [L,V])
8   = k(val(V) -> K) store(Mem [L,V]) .
```

Improving SILF Memory Performance

- Improving ACI matching performance during lookup should improve evaluation performance
- Key Idea 1: Only global locations and locations in current function (i.e., stack frame) are visible – restricting lookups to just these should improve performance
- Key Idea 2: Without address capture and pointers, addresses are not visible on function return, so it should be possible to discard allocated memory on returning

Improving SILF Memory Performance

- Improving ACI matching performance during lookup should improve evaluation performance
- Key Idea 1: Only global locations and locations in current function (i.e., stack frame) are visible – restricting lookups to just these should improve performance
- Key Idea 2: Without address capture and pointers, addresses are not visible on function return, so it should be possible to discard allocated memory on returning

Key ideas lead to use of "memory stacks", similar to stack frames used in standard imperative and OO language implementations.

Stacked Memory Model

- Memory represented as a stack of smaller stores
- Top of stack contains store for current function, bottom contains global store
- Stack frame contains location number of smallest location in its store, with visible locations $\geq$ this in function store, locations $<$ this in global store
- Function call pushes new frame, return pops top frame

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

Stacked Memory Model: Example

Stack frame for f
...
Globals

```
1  function f(x) begin
2    var y; // 1
3    y = g(3); // 2
4    return y
5  end
6
7  function g(x) begin
8    ... return x // 3
9  end
```

- Example shows memory layout at code location 1
- Global variables located in bottom frame
- Variables local to current function `f` in top frame

Stacked Memory Model: Example

Stack frame for g
Stack frame for f
...
Globals

```
1  function f(x) begin
2    var y; // 1
3    y = g(3); // 2
4    return y
5  end
6
7  function g(x) begin
8    ... return x // 3
9  end
```

- Code location 2 contains a function call
- New memory frame for g pushed onto top of stack
- Items in `f` no longer visible

Stacked Memory Model: Example

Stack frame for f
...
Globals

```
1  function f(x) begin
2    var y; // 1
3    y = g(3); // 2
4    return y
5  end
6
7  function g(x) begin
8    ... return x // 3
9  end
```

- Code location 3 contains a function return
- Memory frame for function g popped from stack
- Memory stack now structured the same as before the call

Stacked Memory Model Interface

Lookups triggered by language features redirected to use stacks, maintaining same interface to memory:

```
1   eq k(lookupLoc(L) -> K) store(ST)
2    = k(val(stackLookup(L,ST)) -> K) store(ST) .
3
4  ceq stackLookup(loc(N),([Nb,Mem] , ST)) = lvsLookup(loc(N),Mem)
5   if N >= Nb .
6
7  ceq stackLookup(loc(N),([Nb,Mem] , ST, [Nb',Mem']))
8    = lvsLookup(loc(N),Mem')
9   if N < Nb .
10
11  eq lvsLookup(L,([L,V] Mem)) = V .
```

Performance

	Standard (Flat) Memory Model	Stacked Memory Model
Test Case	Time (sec)	Time (sec)
factorial	3.711	0.747
factorial2	1664.280	11.245
ifactorial	1.047	0.978
ifactorial2	43.861	15.441
fibonacci	29.014	1.939
qsort	111.623	15.374
ssort	21.557	14.657

- factorial recursively calculates $20!, 40!, \ldots, 200!$; factorial2 recursively calculates $1!, 2!, \ldots, 200!$; ifactorial and ifactorial2 are iterative versions of factorial and factorial2, respectively; fibonacci recursively calculates fib of $1, 2, \ldots, 15$; qsort and ssort each sort 2 arrays of 100 elements each

Evaluation

- Clear improvement in performance in all cases
- Limited changes to put new model in place: 6 equations, dealing with function call/return or memory, were modified
- Similar changes should be useful for other languages with similarly constrained memory models
- Lack of support for languages that can return allocated memory limits usefulness – not applicable to all imperative languages

Outline

1 Overview and Motivation

2 SILF and Stacked Memory

3 KOOL and Garbage Collection

4 Conclusions

KOOL

- KOOL is a standard object-oriented language: **K**-based **O**bject-**O**riented **L**anguage
- Provides standard OO features: classes, methods, inheritance, dynamic dispatch, etc
- Includes memory features of languages like Java: references, dynamic allocation, no explicit memory deallocation

KOOL: Flat Memory Model

- Like SILF, memory model is "flat", using a finite map and never performing clean-up
- Unlike SILF, uses Maude `MAP` module versus custom set definition
- Memory usage in KOOL much higher than in SILF – KOOL is a "pure" OO language, i.e., all values are objects, all operations (even arithmetic) involve object creation and method invocation (call by value)

Improving KOOL Memory Performance

- Use of references, ability to return newly-allocated objects rules out a solution similar to that in SILF

Improving KOOL Memory Performance

- Use of references, ability to return newly-allocated objects rules out a solution similar to that in SILF
- So, use a classic memory management solution: Garbage Collection (GC)

Improving KOOL Memory Performance

- Use of references, ability to return newly-allocated objects rules out a solution similar to that in SILF
- So, use a classic memory management solution: Garbage Collection (GC)

Goal: Add garbage collection to KOOL with minimal changes to the existing language definition.

Garbage Collection in KOOL

- Mark/sweep collector: works well with circular structures, easy to define and reason about
- Runs in two steps:
  - Mark: Identify all reachable locations in memory
  - Sweep: Discard all unreachable (unmarked) locations

Defining the Store for GC

```
1 protecting MAP{Location,ValueTuple} *
2   (sort Map{Location,ValueTuple} to Store) .
3
4 op [_,_,_] : Value Nat Nat -> ValueTuple .
```

- Before, Store defined as a Map from Location to Value; now, defined as Map from Location to ValueTuple
- ValueTuple includes two flags, one marking the memory location as shared/unshared, the other used as the GC mark bit (we're interested in the second here)

Triggering GC

- Triggering GC done using `triggerGC` computation item
- GC triggered after a predefined number of allocations
- Triggering sets the state component `ingc` to indicate GC has started
- In KOOL `triggerGC` used in allocation operations inside memory modules – no definitions of language features (method dispatch, etc) need to be changed

Step 1: Initialization

- When GC triggered, all mark flags on memory set to 0 (not marked) to start
- `mem` state component renamed to `gcmem` – keeps rules/equations that modify memory from firing, since they will fail to match

Step 2: Mark Root Locations

- First step in marking will find *root locations* (locations found directly in computation parts of state) and mark them (using `markLocsInSet`)
- Operations defined recursively to find roots in different state components, like:
  - `KStateLocs` finds root locations in state components
  - `MStackLocs` finds root locations in method stack
  - `KLocs` finds root locations in computation

```
1  ceq threads(KS) ingc(gcMarkRoots    ) gcmem(Mem                   )
2    = threads(KS) ingc(gcMarkTrans(LS)) gcmem(markLocsInSet(Mem,LS))
3   if LS := KStateLocs(KS) .
4
5  eq KStateLocs(env(Env) TS) = ListToSet(envLocs(Env)) KStateLocs(TS) .
6  eq KStateLocs(k(K) CS) = KLocs(K) KStateLocs(CS) .
7  eq KStateLocs(mstack(MSTL) CS) = MStackLocs(MSTL) KStateLocs(CS) .
```

Step 3: Mark Locations Reachable from Roots

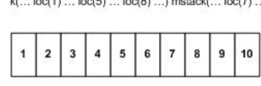
- Once root locations have been marked, need to find all other reachable locations
- Example: computation may hold reference to an object; object may then reference other objects
- Iterative process: keep marking reachable locations until fixed point reached
- Uses `valLocs` operation to get locations held in a value (like locations of object references held in object fields)

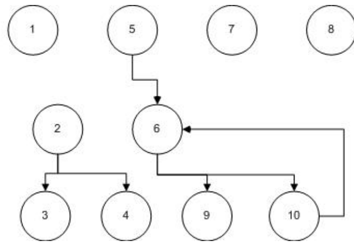
Step 4: Sweep

- Once iterative marking reaches a fixpoint, switch GC mode to `sweep`
- Sweeping is simple: all unmarked locations discarded
- When sweeping is complete, `gcmem` renamed to `mem` and allocation counter reset to 0 – GC is complete

```
1   eq ingc(gcSweep) gcmem(Mem                  ) =
2      ingc(noGC(0)) mem(removeUnmarked(Mem)   ) .
3
4   eq removeUnmarked(_‘,_(L |-> [V,N,0], Mem))
5    = removeUnmarked(Mem) .
6   eq removeUnmarked(Mem) = Mem [owise] .
```

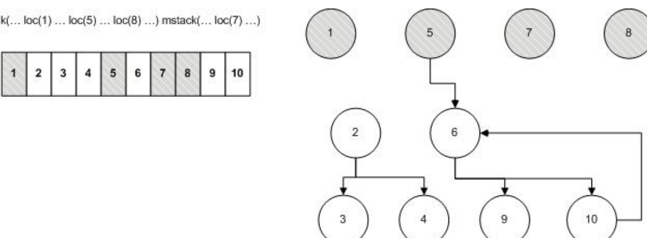
GC Example





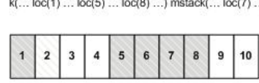
- Graphic shows representation of K configuration
- `k` contains current computation, `mstack` contains the method stack, with most of the configuration elided
- Boxes numbered 1 through 10 represent memory locations
- Circles with numbers show which objects at which locations hold references to other objects

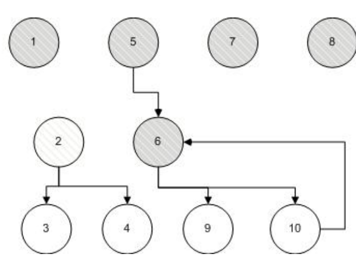
GC Example



- First part of mark step in GC marks roots
- Locations 1, 5 and 8 are present in `k`
- Location 7 is referenced in the method stack (so it would be visible on method return)

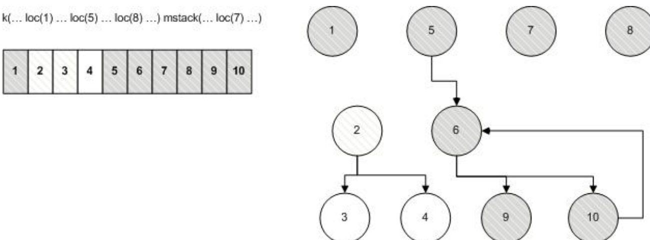
GC Example





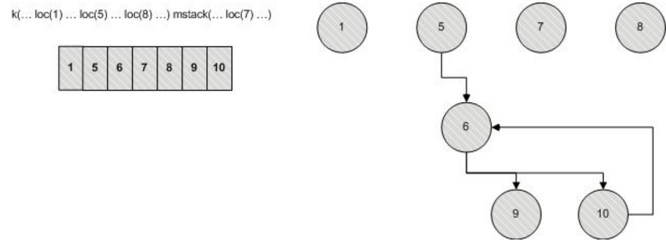
- Next, see which locations are reachable from the roots
- First step discovers location 6 reachable from location 5

GC Example



- Continue marking reachable locations
- Locations 9 and 10 both reachable from location 6
- No more reachable locations – 6 is already marked, so we don't loop between 10 and 6

GC Example



- Finally, GC sweeps unmarked/unreachable locations
- Locations 2, 3, and 4 unreachable, so discarded

Performance

	GC Disabled		GC Enabled		
Test Case	Time (sec)	Store Size	Time (sec)	Store Size	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

- `addnums` and `garbage` both perform repeated calculations or allocations; other examples are identical to those for SILF
- Store Size is the final size of memory
- Collections is the number of GCs performed

Evaluation

- GC results mixed: some examples with GC enabled cause reduced performance, others run faster or even allow execution to finish (versus segfault)
- Adding GC required only changes in the memory operations, with no changes in feature definitions
- GC is modular: some operations are generic for all collectors, with specific operations used to interrogate current state and computation (only these are language-specific)

Outline

1. Overview and Motivation
2. SILF and Stacked Memory
3. KOOL and Garbage Collection
4. Conclusions

Related Work

- Presented work part of rewriting logic semantics project [MeseguerRosu04, MeseguerRosu07]
- Most prior work on performance of rewriting logic definitions has focused on analysis performance, including:
  - Java source and bytecode analysis [FarzanEtAl04b, FarzanEtAl04a]
  - Analysis of KOOL programs [HillsRosu07]
  - Partial order reduction techniques [FarzanMeseguer07]
- Extensive work has been done on garbage collection [JonesLins96], including in pure OO languages [UngarJackson88, UngarJackson92]

Future Work

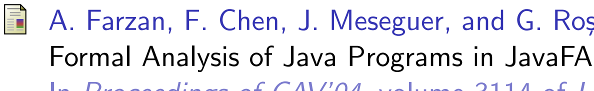
- Canonical representations of memory: would use GC, could improve performance
- Further experimentation with GC, other analyses (escape analysis, etc)
- Can Maude be extended with direct support for arrays (or other data structures) that would eliminate need for this work? Would this have drawbacks (in debugging definitions, for instance)?
- GC as a language definition transformation

Conclusions

- Improved performance of language definitions allows use of more realistic programs during language design
- Changes to shared language modules can improve performance with minimal alterations to feature definitions, can be leveraged across multiple languages
- A stacked memory model like that added to SILF can show significant improvements where it can be used
- Adding GC to languages like KOOL can be done with little additional effort, but has mixed results, improving performance for some programs but degrading it for others

Questions?

Citations



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

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.

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

Citations

M. Hills and G. Roșu.
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.

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

Citations

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.

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.

Citations

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

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