On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance

Mark Hills and Grigore Roşu
{mhills, grosu}@cs.uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign

6 June 2007

Outline

1. Rewriting Logic Semantics and KOOL
2. Analysis in KOOL with Rewriting Logic
3. Improving Performance
4. Conclusion

The KOOL Language

KOOL is

- *object-oriented*: classes, methods, dynamic dispatch, exceptions; all values objects
- *dynamic*: dynamically typed, adding extensions for modifying code at runtime
- *concurrent*: multiple threads of execution, shared memory, locks acquired on objects
- *extensible*, with various features “plugged in”: synchronized methods, semaphores, reflective capabilities

Design Motivations for KOOL

- Experiment with *optional* and *pluggable* type systems
- Investigate interaction of language features with verification and analysis
- Create a language suitable for languages courses, without some “confusing” features from other languages

A Sample KOOL Program

```
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)
```

Rewriting Logic Semantics of Programming Languages

- Rewriting logic is an extension of equational logic with support for concurrency
- Language semantics provides formal definitions of language features
- Rewriting logic semantics: formal language definitions using rewriting logic
- Definitions are executable with rewriting logic engines, like Maude

The Rewriting Logic Semantics Project

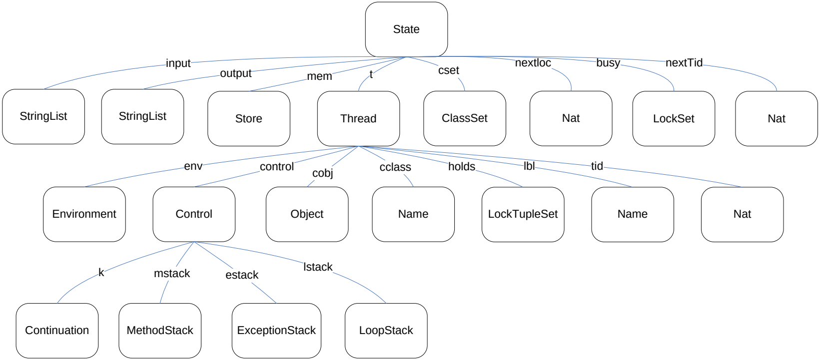
- KOOL is part of ongoing work on rewriting logic semantics
- Other work includes many languages and supporting tools, researchers at multiple universities
- Java, Beta, Scheme, Prolog, Haskell, PLAN, BC, CCS, MSR, ABEL, SILF, FUN, π-calculus, variants of λ-calculus, others

KOOL Program States

- States in KOOL represented as multisets of state components
- Multisets formed by putting components next to one another

```
op _ _ :  KState KState -> KState [assoc comm id:  empty]
```

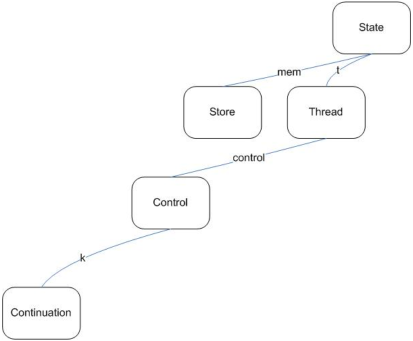
KOOL Program States



KOOL Program States: A Simple Term

```
stmt(if E then S else S' fi)
```

KOOL Program States: A More Complex Term



```
t(control(k(llookup(L) -> K) CS) TS) mem(Mem)
```

Sample KOOL Semantics

Equations represent non-competing transitions, and have the general form $eq\ l = r$ (unconditional) or $ceq\ l = r\ if\ c$ (conditional):

```
 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') .
```

Sample KOOL Semantics

Equations represent non-competing transitions, and have the general form $eq\ l = r$ (unconditional) or $ceq\ l = r\ if\ c$ (conditional):

```
 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’) .
```

Rules represent transitions which may compete, and have the general form $rl\ l => r$ (unconditional) or $crl\ l => r\ if\ c$ (conditional):

```
crl t(control(k(llookup(L) -> K) CS) TS) mem(Mem) =>
    t(control(k(val(V) -> K) CS) TS) mem(Mem)
 if V := Mem[L] /\ V =/= undefined .
```

Running KOOL Programs

- Programs parsed, converted to Maude, and executed, with results displayed to user
- KOOL programs execute directly in the language semantics, defined using rewriting logic
- Stats: 335 equations in semantics, 15 rules, 1406 lines
- No type checker; violations (message not understood, wrong number of arguments, etc) handled at runtime with exceptions

Outline

1. Rewriting Logic Semantics and KOOL
2. Analysis in KOOL with Rewriting Logic
3. Improving Performance
4. Conclusion

Analysis Overview

KOOL uses analysis capabilities of Maude to provide program analysis:

- **Search** allows a breadth-first search over the program state space
- **Model Checking** allows verification of finite-state systems using LTL formulae
- Rewriting logic *rules* determine size of state space/transitions between states

Breadth-First Search

- KOOL provides breadth-first search over output values "out-of-the-box"
- Can either find all output values or search for a specific value

Search Example: Output Interleavings

```
class Main is
  var p1, p2;

  method Test(id) is
    console << "ID is " << id;
  end

  method Run is
    spawn(self.Test(1));
    spawn(self.Test(2));
    console << "Done";
  end
end

(new Main).Run
```

Output Interleavings Results

```
> runkool -s Spawn5.kool

Solution 1 (state 16)
states: 38  rewrites: 8325 in 464ms cpu (471ms real) (17940
    rewrites/second)
SL:[StringList] --> "Done"

...

Solution 13 (state 455)
states: 456  rewrites: 70193 in 4944ms cpu (4994ms real) (14196
    rewrites/second)
SL:[StringList] --> "ID is ","2","ID is ","1","Done"

No more solutions.
```

Search Example: The Thread Game

KOOL version of a problem formulated by J. Moore

```
class ThreadGame is
  var x;

  method ThreadGame is
    x <- 1;
  end

  method Add is
    while true do x <- x + x; od
  end

  method Run is
    spawn(self.Add); spawn(self.Add);
    console << x;
  end
end
(new ThreadGame).Run
```

Thread Game Results

```
> runkool -t 5 ThreadGame.kool

Solution 1 (state 769)
SL:[StringList] --> "5"
```

Model Checking

- KOOL uses Maude to provide basic model checking capabilities
- Extended with labeled statements; labels can be used in LTL formulae
- Runtime allows custom Maude modules with new LTL properties to be loaded and used during verification

Dining Philosophers

```
class Philosopher is
  method Run(id,left,right) is
    while true do
      // thinking here...
      hungry:
        acquire left;
        acquire right;
      eating:
        release left;
        release right;
    od
  end
end
```

Model Checking the Dining Philosophers

```
> runkool DP.kool -m ... model checking arguments ...
```

- Model checking arguments generally include formula to check
- When formula doesn't hold, a counterexample is generated
- When formula holds, `true` is returned

A Problem Arises

Analysis is slow, we quickly hit maximum problem size.

Ph's	No Optimizations	
	Counterex	DeadFree
2	0.830	1.530
3	0.912	34.924
4	1.466	1226.323
5	6.465	NA
6	66.683	NA
7	805.278	NA
8	NA	NA

Figure: Dining Philosophers Model Checking Performance

A Problem Arises

But why?

- In KOOL, all operations are message sends, even addition

A Problem Arises

But why?

- In KOOL, all operations are message sends, even addition
- All operations will require memory lookups, since even numbers are objects

A Problem Arises

But why?

- In KOOL, all operations are message sends, even addition
- All operations will require memory lookups, since even numbers are objects
- All memory lookups are rules

A Problem Arises

But why?

- In KOOL, all operations are message sends, even addition
- All operations will require memory lookups, since even numbers are objects
- All memory lookups are rules
- Rules increase the size of the state space

A Problem Arises

But why?

- In KOOL, all operations are message sends, even addition
- All operations will require memory lookups, since even numbers are objects
- All memory lookups are rules
- Rules increase the size of the state space
- In addition, heap constantly changes, making many more programs infinite state (impossible to model check)

A Problem Arises

But why?

- In KOOL, all operations are message sends, even addition
- All operations will require memory lookups, since even numbers are objects
- All memory lookups are rules
- Rules increase the size of the state space
- In addition, heap constantly changes, making many more programs infinite state (impossible to model check)
- Shows that a reasonable definition for *execution* may not work well for analysis

Our Goal

Reduce the number of rule applications by changing the semantics of KOOL while still maintaining observable program behavior

Outline

1. Rewriting Logic Semantics and KOOL
2. Analysis in KOOL with Rewriting Logic
3. Improving Performance
4. Conclusion

Optimizing KOOL

Two approaches to optimizing KOOL programs for analysis:

- Change semantics to reduce usage of rules, focusing on changes that also speed up normal execution (e.g. reduce number of message sends)

Optimizing KOOL

Two approaches to optimizing KOOL programs for analysis:

- Change semantics to reduce usage of rules, focusing on changes that also speed up normal execution (e.g. reduce number of message sends)
- Change semantics to reduce usage of rules, even at the expense of slower program execution

Auto-Boxing in KOOL

```
(1 + 2) * 3 // desugars as (1.+(2)).*(3)
```

- In KOOL, this involves creation of 5 objects, 2 method calls, multiple primitive manipulation operations
- Heavy use of memory causes execution and analysis performance problems
- Familiar problem in OO languages (Smalltalk and SELF, for instance)
- Goal: use scalar values instead, automatically converting to objects (auto-boxing, as in C#) when needed
- With auto-boxing, 2 operations, neither requiring memory lookup

Adding Auto-Boxing

- Step 1: Allow scalar values, versus just objects (3 equations)

Adding Auto-Boxing

- Step 1: Allow scalar values, versus just objects (3 equations)
- Step 2: Modify method call semantics to handle scalar operations without performing a method call (42 equations)

Adding Auto-Boxing

- Step 1: Allow scalar values, versus just objects (3 equations)
- Step 2: Modify method call semantics to handle scalar operations without performing a method call (42 equations)
- Step 3: Return scalars from some operations that currently return objects (e.g. primitive integer addition) (50 equations)

Adding Auto-Boxing

- Step 1: Allow scalar values, versus just objects (3 equations)
- Step 2: Modify method call semantics to handle scalar operations without performing a method call (42 equations)
- Step 3: Return scalars from some operations that currently return objects (e.g. primitive integer addition) (50 equations)
- Step 4: Box scalars when needed (4 equations)

Adding Auto-Boxing

- Step 1: Allow scalar values, versus just objects (3 equations)
- Step 2: Modify method call semantics to handle scalar operations without performing a method call (42 equations)
- Step 3: Return scalars from some operations that currently return objects (e.g. primitive integer addition) (50 equations)
- Step 4: Box scalars when needed (4 equations)
- 8 more additional changes – most changes for auto-boxing mechanical

Auto-Boxing Results

Ph's	No Optimizations		Auto-boxing	
	Counterex	DeadFree	Counterex	DeadFree
2	0.830	1.530	0.799	0.878
3	0.912	34.924	0.899	2.901
4	1.466	1226.323	1.346	23.451
5	6.465	NA	5.226	237.714
6	66.683	NA	45.747	2501.498
7	805.278	NA	476.916	NA
8	NA	NA	NA	NA

Figure: Dining Philosophers Model Checking Performance

The KOOL Memory Model

- KOOL memory represented as finite map, $Location \rightarrow Value$
- Object references are *Location*s, objects are *Value*s
- Memory at toplevel, since all threads share same memory space
- Memory accesses use rules, since accesses in different threads can compete

Memory Pools

- **Idea:** Can we split memory into shared and unshared pools, only use rules for accessed to shared pool?

Memory Pools

- **Idea:** Can we split memory into shared and unshared pools, only use rules for accessed to shared pool?
- **Answer:** Yes, if we're careful...
  - Local variable accesses should never compete
  - Object-level variable accesses *may* compete
  - If a variable may be shared, anything reachable through it may be shared as well
  - Conservative assumption: if it can be shared, make it shared, else leave it unshared; once shared, never goes back (simple rule, room for improvement)

Implementing Memory Pools

- Add second memory pool (`smem`), with equations and rules to access it (4 equations, 2 rules)

Implementing Memory Pools

- Add second memory pool (`smem`), with equations and rules to access it (4 equations, 2 rules)
- Add equations to move items from unshared to shared memory, taking account of transitivity (3 equations)

Implementing Memory Pools

- Add second memory pool (`smem`), with equations and rules to access it (4 equations, 2 rules)
- Add equations to move items from unshared to shared memory, taking account of transitivity (3 equations)
- On spawn of a new method, all locations reachable through message target and actual arguments shared (1 rule)

Implementing Memory Pools

- Add second memory pool (`smem`), with equations and rules to access it (4 equations, 2 rules)
- Add equations to move items from unshared to shared memory, taking account of transitivity (3 equations)
- On spawn of a new method, all locations reachable through message target and actual arguments shared (1 rule)
- On spawn of arbitrary expression, contents of current environment (all names in scope) shared (1 rule)

Implementing Memory Pools

- Add second memory pool (`smem`), with equations and rules to access it (4 equations, 2 rules)
- Add equations to move items from unshared to shared memory, taking account of transitivity (3 equations)
- On spawn of a new method, all locations reachable through message target and actual arguments shared (1 rule)
- On spawn of arbitrary expression, contents of current environment (all names in scope) shared (1 rule)
- On assignment to shared location, share new reachable locations (included in above)

Implementing Memory Pools

- Add second memory pool (`smem`), with equations and rules to access it (4 equations, 2 rules)
- Add equations to move items from unshared to shared memory, taking account of transitivity (3 equations)
- On spawn of a new method, all locations reachable through message target and actual arguments shared (1 rule)
- On spawn of arbitrary expression, contents of current environment (all names in scope) shared (1 rule)
- On assignment to shared location, share new reachable locations (included in above)
- Overall, fewer changes compared to auto-boxing, but more complex

Shared Memory and Auto-Boxing Combined

Ph's	No Optimizations		Auto-boxing + Memory Pools	
	Counterex	DeadFree	Counterex	DeadFree
2	0.830	1.530	0.758	0.782
3	0.912	34.924	0.812	1.270
4	1.466	1226.323	1.070	4.192
5	6.465	NA	2.264	22.467
6	66.683	NA	9.236	124.818
7	805.278	NA	50.527	797.308
8	NA	NA	299.630	4744.427

Figure: Dining Philosophers Model Checking Performance

Shared Memory and Auto-Boxing Combined

Ph's	Auto-boxing		Auto-boxing + Memory Pools	
	Counterex	DeadFree	Counterex	DeadFree
2	0.799	0.878	0.758	0.782
3	0.899	2.901	0.812	1.270
4	1.346	23.451	1.070	4.192
5	5.226	237.714	2.264	22.467
6	45.747	2501.498	9.236	124.818
7	476.916	NA	50.527	797.308
8	NA	NA	299.630	4744.427

Figure: Dining Philosophers Model Checking Performance

Outline

Conclusions

- Methods to define languages using rewriting logic semantics fairly well understood
- Good definitions for *execution* can have poor analysis performance
- Optimizations from analysis and programming languages can be applied to improve analysis performance
- Two straight-forward implementations of optimizations shown here; both improve performance dramatically

Future Work

- Provide GC for KOOL, which should help improve memory performance
- Investigate ways to optimize definitions automatically, and/or prove changes preserve behavior
- Look for other optimizations that could further improve performance
- Investigate modularity of optimizations – can optimized memory model be applied to memory model for other languages, for instance?

Related Work

- Rewriting Logic Semantics: *The Rewriting Logic Semantics Project*, José Meseguer and Grigore Roşu, TCS, Volume 373(3), pp 217–237, 2007.
- Rewriting Logic Definition Performance: *On Modelling Sensor Networks in Maude*, Dilia E. Rodríguez, WRLA'06.
- Analysis Performance in Maude: *State Space Reduction of Rewrite Theories Using Invisible Transitions*, Azadeh Farzan and José Meseguer, AMAST 2006; *Partial Order Reduction for Rewriting Semantics of Programming Languages*, Azadeh Farzan and José Meseguer, WRLA 2006.