A Modular Rewriting Approach to Language Design, Evolution and Analysis

Mark Hills
mhills@cs.uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign

4 August 2009

Research Motivation

- Goal: increase use of formal semantics techniques for language definition
- Why: more complex languages, language features, programs – need formal techniques to reason about all these
- Techniques to increase adoption:
    - Flexible methods to define even complex features: should not be limited to simplified core of language
    - Improved ways to leverage language definitions: ensure effort invested in creation is not wasted
    - Improved tool support: make working with semantics more like working with programs

Research Focus

Thesis research has focused on three areas:

- Language Prototyping
- Modular Language Definition Frameworks
- Policy Frameworks for Program Analysis

Good synergy between areas:

- Work on modular language definitions for language prototyping influences work on policy frameworks
- Work on policy frameworks influences work on modular language definition frameworks

Special Focus: Modularity

One focus of research has been language *modularity*:

- Assemble a language from existing, tested feature definitions
- Extend a language with new features, without modifying existing features
- Try variants of features with different behaviors or performance characteristics
- Construct analyses as modular extensions to existing analysis frameworks, without requiring changes to existing framework

Secondary Focus: Performance

A second focus has been *performance*:

- Important for program execution during language prototyping, must be "fast enough"
- Also important for analysis and verification, where performance can be critical

Research in performance has focused on:

- Performance of feature variants for language execution and analysis
- Modular analysis and verification, improving performance by shrinking size of analysis task

K: High Level View

- K provides a rewrite-based method to formally define computation
- Focus here: formal definitions of programming languages
- Built on lessons learned from work on rewriting logic semantics (especially “computation-based” or “continuation-based” style)
- Some influences from other formalisms: chemical abstract machine (CHAM), reduction semantics

K: Equations

- Computations defined used equations and rules (discussed below) that transform terms
- Equations manipulate term structure, non-computational, reversible, can think of as equational logic equations
- One special type of equation (based on intuition from CHAMs): language constructs can *heat* (break apart into pieces for evaluation) and *cool* (form back together)
- Represented using $\rightleftharpoons$, like $a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \Box + a_2$
- Operators containing $\Box$ called *freezer*s

K: Sample Equations

$$A_1 + A_2 \rightleftharpoons A_1 \curvearrowright \Box + A_2 \tag{1}$$

$$A_1 + A_2 \rightleftharpoons A_2 \curvearrowright A_1 + \Box \tag{2}$$

$$\text{if } B \text{ then } S_1 \text{ else } S_2 \rightleftharpoons B \curvearrowright \text{if } \Box \text{ then } S_1 \text{ else } S_2 \tag{3}$$

Operations with freezers are boring to write, so we can mark operations `strict(`*natlist*`)`, with a freezer generated for each position in the list. To do so for all operands, just use `strict`.

```
_+_ : AExp AExp -> AExp [strict]
if_then_else : BExp Stmt Stmt -> Stmt [strict(1)]
_:=_ : Id Exp -> Stmt [strict(2)]
```

K: Rules

Rules: computational, not reversible, may be concurrent, could be either equations or rules in rewriting logic

$$I_1 + I_2 \longrightarrow I,\ \text{where } I \text{ is the sum of } I_1 \text{ and } I_2 \tag{4}$$

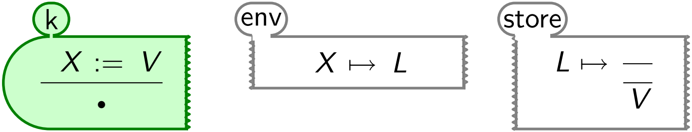
$$\text{if true then } S_1 \text{ else } S_2 \longrightarrow S_1 \tag{5}$$

$$\text{if false then } S_1 \text{ else } S_2 \longrightarrow S_2 \tag{6}$$

K: Configurations and Cells

- Computations take place in context of a *configuration*, made up of K *cells*
- Each cell holds specific piece of information: computation, environment, store, etc
- Cells can be arranged into a hierarchy, nested in other cells
- Cells can be repeated (e.g., multiple computations in a concurrent language)
- Two regularly used cells:
    - $\top$ (*top*), representing entire configuration
    - k, a list of computational tasks separated by $\curvearrowright$, like $t_1 \curvearrowright t_2 \curvearrowright \ldots \curvearrowright t_n$

K: Rules, Part 2



$$\langle k\rangle X := V \ \ldots\langle/k\rangle \ \langle env\rangle\ldots\ X \mapsto L \ \ldots\langle/env\rangle \ \langle store\rangle\ldots\ L \mapsto \_ \ \ldots\langle/store\rangle \rightarrow \langle k\rangle\cdot \ \ldots\langle/k\rangle \ \langle env\rangle\ldots\ X \mapsto L \ \ldots\langle/env\rangle \ \langle store\rangle\ldots\ L \mapsto V \ \ldots\langle/store\rangle \tag{7}$$

Outline

Language Prototyping

- Program design and development with prototyping provides flexible, interactive environment with rapid feedback
- Want to provide same environment for design and evolution of programming languages
- Rewriting-based tool support, flexibility of K provide firm foundation for prototyping support
- Back to goals: encourage use of formal definitions, here by providing interactive environment for language design

The KOOL Language

KOOL was designed using prototyping techniques; language goals include:

- Provide pure OO language as basis for prototyping new language features
- Experiment with impact of language design on program execution, analysis, verification
- 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)
```

Feature Prototyping in KOOL: Concurrency

```
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
```

Feature Prototyping in KOOL: Mutual Exclusion

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

Feature Prototyping in KOOL: Synchronized Methods

```
class WriteNums is
  var theNum;

  method WriteNums(n) is theNum <- n; end

  synchronized method AddTo(n) is theNum <- theNum + n; end

  synchronized method Sub(n) is theNum <- theNum - n; end

  synchronized method Write is
    console << "Starting value:" << theNum;
    self.AddTo(10); self.Sub(8);
    console << "Ending value:" << theNum;
  end
end
```

Other Feature Prototyping Examples

- New core language features (vectors, support for additional primitive types and operations)
- Class project extensions (including reflection)
- Annotation system

KOOL Semantics: Overview

- Designed using Continuation-based Style of Rewriting Logic Semantics, using a K-like definitional style
- Original language: standard single-inheritance OO model, pure OO language, dynamic dispatch, exceptions, no concurrency features
- New features added through prototyping (shown above)
- Uses first-order representation of computations heavily in features like method dispatch, exception handling, loop break and continue

KOOL Semantics: Modularity

- Features defined in separate modules, increase reuse
- Features defined with minimal commitment to language configuration, allows new features to be added without changing existing features
- Multiple versions of some features defined, allow different versions to be used to meet different needs
- Example: Synchronized methods impact method call semantics, require only 2 modifications to ops, 4 to equations (these can be split out into alternate module versions), 10 new ops and equations.
- Definition stats: 72 modules, 621 equations, 20 rules, 513 operators

KOOL Semantics: Performance

- Executability of semantics, for evaluation and analysis, makes performance important
- Modularity provides mechanism to experiment with different versions of same feature, again using prototyping techniques
- Several experiments to judge performance of language features and design decisions for evaluation and analysis:
    - Boxed versus unboxed scalar values (evaluation/analysis)
    - Tracking shared, unshared memory (analysis)
    - Mark/sweep garbage collection (evaluation)

Other Work on Language Prototyping

- Beta (including addition of `super` calls)
- ML (in progress)
- Many more by others, both as part of research projects and as class projects

Outline

Motivation

- Modularity important for building reusable language definitions
- Also important for adding new features, creating variants of existing features
- K provides some support using *context transformers* – automatically adjust K rules and equations to match current configuration
- Still need a way to effectively group K definitions into reusable pieces

The K Module System

The K Module System is designed to:

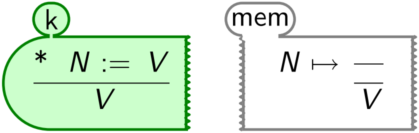
- Organize K features into reusable modules
- Provide keywords to identify standard parts of K definitions: sorts, operators, K rules and equations, K cells and configurations
- Allow shorthands for common definitional needs: sort aliases, variable prefixes, tags on module names identifying type of information in module

An Abstract Syntax Module

```
module KERNELC/EXP/MEM/DEREF[SYNTAX] is
  import KERNELC/EXP[SYNTAX] .
begin
  xop *_ : Exp -> Exp [strict prec 25] .
endmodule
```

A Semantics Module

```
module KERNELC/EXP/MEM/DEREF[DYNAMIC] is
  import KERNELC/EXP[DYNAMIC] .
  import KERNELC/CONFIG/MEM .
  import KCONFIG/CONFIG .
begin
  vars K1 K2 : K . var N : Nat . vars V V’ : KResult .
  kcxt * K1 := K2 [strict(K1)] .
  krl <k> * #(N) := V ...</k>  <mem>... #(N) |-> _ ...</mem> =>
      <k> V ...</k> <mem>... #(N) |-> V ...<mem> .
endmodule
```



Defining K Cells

```
module KERNELC/CONFIG/PTRMAP is
  import K/K .
begin
  sortalias PtrMap = KMap{K,K} .
  varprefix PM : PtrMap .
  cell ptr: PtrMap .
endmodule
```

Defining A Configuration

```
module KERNELC/STANDARD[LANGUAGE] is
  import[DYNAMIC]
         KERNELC/EXP/BOOL, KERNELC/EXP/MEM/DEREF, ...
  import[SYNTAX]
         KERNELC/EXP/TERNARY, KERNELC/EXP/BOOL/NOT, ...
begin
  var P : Pgm . var ST : Stream . var I : Item . var Mem : Mem .

  kconf <T> <thread*> <k> K </k> </thread*>
          <mem>  Mem </mem> <ptr> PM </ptr>
          <nextItem> I </nextItem> <out> ST </out>
        </T> .
endmodule
```

Current Tool Support

- Module lexer and parser provide basic syntax checking for modules
- Module system tool provides support for module file loading across multiple module paths, manipulation of module contents, translation of K modules into equivalent Maude modules
- The module tool command shell, `modtool`, provides an interactive frontend for the module system functionality

The Online Module Repository

Goal: provide a way to share semantics module *online*, using a standards-based mechanism to exchange modules.

The Online Module Repository

Goal: provide a way to share semantics module *online*, using a standards-based mechanism to exchange modules.

- Online Module Repository (OMR) provides online database used to store information about modules
- Initial XML schema defined as a starting point for developing a standard for semantic module interchange
- Web-services based support for retrieving modules (lists of modules, module bodies) from repository

Outline

Motivation: Programs Use Implicit Data

Programs use implicit data, but there is no way to check that implicit values are used correctly:

```
typedef struct intpair { int x; int y; } IntPair;

int product(IntPair *p) {
  return p->x * p->y;
}
```

Question: does *p* point to `NULL`? If so, this program has a bug...

Partial Solution: Annotations

```
typedef struct intpair { int x; int y; } IntPair;

int product($notnull IntPair *p) {
  return p->x * p->y;
}
```

- Annotations provide a way to make implicit assumptions about the program explicit
- Provides way to check properties with tools
- Necessary to scale to large programs – comments are not enough

Challenge: Multiple Analysis Domains

```
typedef struct intpair { $m int x; $kg int y; } IntPair;

$m $kg int product($notnull IntPair *p) {
  return p->x * p->y;
}
```

- Sometimes want to reason about multiple analysis domains
- Many systems focused on just one domain, not extensible

Challenge: Type Annotations Can Be Inflexible

```
//@ post(UNITS): @unit(@return) ^ 2 = @unit(x)
double sqrt(double x) {
  return ...
}
```

- May not want to restrict values to specific types
- Having code annotations can be more expressive
- But, many systems support one annotation style or the other, not both

Challenge: Non-Type Behavior

```
typedef struct intpair { int x; int y; } IntPair;

//@ post(UNITS): @unit(p->x) = @old(@unit(p->y))
//@ post(UNITS): @unit(p->y) = @old(@unit(p->x))
void flippair(IntPair *p) {
  int t = p->x;
  p->x = p->y;
  p->y = t;
}
```

- Some programs don't treat values like types
- Want to make sure analysis can handle these domains as well

Challenge: Reusable Analysis Domains

```
function product(x,y)
  post(UNITS): @unit(@return) = @unit(x) @unit(y);
begin
  return x * y ;
end
```

- Most analysis domains formulated for a specific language
- If possible, want to *reuse* analysis domains between languages

Policy Frameworks

Policy frameworks provide support for adding different analysis *policies*, with policy-specific type and code annotations, to a language.

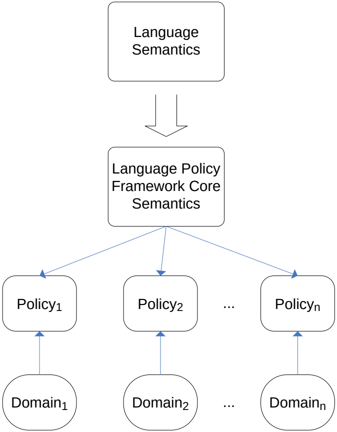
A policy framework is made up of four parts:

- Policy-generic language front-end
- Policy-generic, reusable core language semantics (abstract syntax, shared semantics, generic policy support)
- Analysis domains (types, units, etc)
- Policy-specific analysis semantics, defining semantics of language features and annotations

The Importance of Modularity

- Need to be able to reuse core functionality with few to no changes while building extensions
- Also need to be able to easily use different semantics for the same feature to account for differences in analysis policies

Reuse of Core Framework


Language Semantics

Language Policy Framework Core Semantics

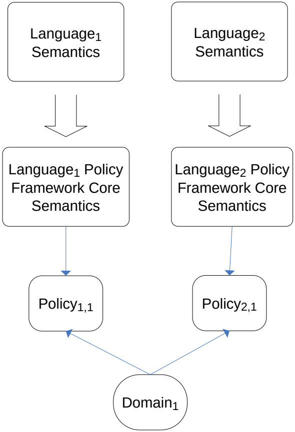
Policy$_1$ Policy$_2$ ... Policy$_n$

Domain$_1$ Domain$_2$ ... Domain$_n$

Key Requirement: Modularity

- Need to be able to reuse core functionality with few to no changes while building extensions
- Also need to be able to easily use different semantics for the same feature to account for differences in analysis policies
- Ideally, should be able to reuse policy infrastructure and analysis domains across languages

Reuse of Analysis Domains



Language_1 Semantics

Language_2 Semantics

Language_1 Policy Framework Core Semantics

Language_2 Policy Framework Core Semantics

$\text{Policy}_{1,1}$

$\text{Policy}_{2,1}$

Domain_1

Key Requirement: Modularity

- Need to be able to reuse core functionality with few to no changes while building extensions
- Also need to be able to easily use different semantics for the same feature to account for differences in analysis policies
- Ideally, should be able to reuse policy infrastructure and analysis domains across languages
- Techniques currently supported in two policy frameworks: one for SILF, one for C

The SILF Policy Framework

- An extension of the SILF language to support policies
- Front-end modified to provide direct language support for type and code annotations
- Policy-generic core semantics created based on SILF dynamic semantics
- Individual policies for types, units as types, and units with code annotations

Types in SILF

```
1 function $int factorial($int n)
2 begin
3   if n = 0 then
4     return 1;
5   else
6     return n * factorial(m - 1);
7   fi
8 end
```

```
Type checking found errors:
  ERROR on line 6(1): Identifier m is not defined.
```

Units in SILF

```
function main(void)
begin
  var x; var y; var n;
  assume(UNITS): @unit(x) = $m;
  assume(UNITS): @unit(y) = $kg;
  for n := 1 to 10
    invariant(UNITS): @unit(x) = @unit(y);
  do
    x := x * x;
    y := y * y;
  od
  write x + y;
end
```

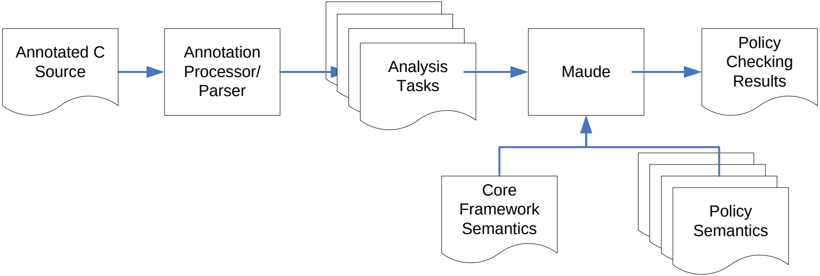
```
Unit checking successful.
```

The C Policy Framework

CPF provides a policy framework for the C language:

- Front-end parsing performed using customized version of CIL, generates per-function analysis tasks
- Other parts of policy framework implemented using Maude
- Shares parts of policy support and some analysis domains with SILF Policy Framework

CPF Processing



C Abstract Syntax/Generic State

- Abstract syntax provided for all C constructs not removed by CIL (which simplifies and transforms some C constructs)
- Includes support for C declarations, operations to deconstruct name and type information (used in policy semantics)
- Generic definitions of CPF policies, values, configurations provided

Statement Handling

- Defines abstract semantics for all C statements not removed by CIL, including `while` loops and `goto` statements
- Individual statements processed in an *environment*, tracking analysis information (like assignments of values to C objects)
- Statement execution engine uses *sets* of environments to model path-sensitive information, can be disabled if not used by policy
- Most expression semantics left to individual policies

A Sample Policy: UNITS

- CPF UNITS policy extends CPF to handle units of measurement
- Adds unit-specific support to C expressions and declarations: units treated as abstract values
- Adds support for unit-specific annotations

Example: CPF UNITS Annotations

```
typedef struct {
  $kg double atomicWeight;
  $noUnit double atomicNumber;
} Element;

//@ post(UNITS): @unit(@result) = $m ^ 2 $kg ^ -1
double radiationLength(Element * material) {
  double A = material->atomicWeight;
  double Z = material->atomicNumber;
  double L = log( 184.15 / pow(Z, 1.0/3.0) );
  double Lp = log( 1194.0 / pow(Z, 2.0/3.0) );
  return ( 4.0 * alpha * re * re) * ( NA / A ) *
         ( Z * Z * L + Z * Lp );
}
```

Example: SC-ROVER Scenario 1

> *Defect 2: Rover is turning too much. The problem is that in the PositionAndHeadingController algorithm, I assumed the atan function returned values in degrees when it actually returned values in radians. Thus, the rover's target turn angle is higher than it should be. (HDCP_DEFECT2)*

```
typedef struct { $noUnit double x; $noUnit double y;
                                   $noUnit double z; } RoverHeading;
$radian double atan2($noUnit double x, $noUnit double y);
$degree double getDeltaH(void);

void test1(RoverHeading *h) {
  double yaw = atan2(h->y,h->x);
  double m_targetyaw = yaw + getDeltaH();
}
```

```
ERROR on line 11(1): Unit violation detected in addition operation,
                     incompatible units.
```

Example: SC-ROVER Scenario 2

> *During the execution of a turn, the rover does a core dump. The problem is that we are trying to compare a number vs a number with a SI unit. (HDCP_DEFECT3)*

```
void test2($meter double deltaX) {
  $noUnit double val = 0;
  $noUnit double c_sign = 1.0;

  if (deltaX < val) {
    c_sign = -1.0;
  }
}
```

```
ERROR on line 18(1): Unit violation detected in less than operation,
                     incompatible units.
```

Performance

- Individual functions generally checked quickly – usually under 1 second
- Extremely large functions take longer, still reasonable amount of time (5.223s for 2705 LOC, 22.853s for 11705 LOC)
- Two largest impacts on performance are function size and number of environments (in policies that need sets of environments)
- Annotation burden similar to other popular tools, like Osprey: use of just type annotations would give same number of annotations at most

CPF Statistics

- CPF Core: 525 operators, 563 equations, 64 modules
- CPF UNITS Policy: 22 modules, 56 operators, 291 equations
- Shared Units Domain: 7 modules, 42 operators, 242 equations
- CPF NOTNULL Policy: 26 modules, 81 operators, 348 equations

Outline

1. Research Motivation
2. K
3. Language Prototyping
4. Modular Language Definition Frameworks
5. Policy Frameworks for Program Analysis
6. Related Work

Related Work

Various parts of related work, organized in each category alphabetically; semantics and tools focuses on modularity, language prototyping, and executability.

- Modular Semantics: Action Semantics, ASMs (Montages), Component-Based Semantics, Monads (Modular Monadic Semantics), (I)MSOS, Other RLS styles, etc.
- Tools: Action Semantics, ASF+SDF Meta-Environment, Centaur, Montages, PLT Redex, RML, Semantic Lego, etc.
- Program Analysis with Annotations: Caduceus, CQual, Frama-C (ACSL), Havoc, JML, Osprey, Spec#, Splint, etc.