

# Modular Language Specifications for Program Analysis

---

Mark Hills

SLS 2013: Workshop on Scalable Language Specification

June 25 - 27, 2013

Cambridge, UK



<http://www.rascal-mpl.org>

# Overview

---

- Policy Frameworks
- Challenges
- DSLs for Program Analysis

# Overview

---

- Policy Frameworks
- Challenges
- DSLs for Program Analysis

# Initial Motivation: Units of Measurement

“NASA lost a \$125 million Mars orbiter because one engineering team used metric units while another used English units for a key spacecraft operation ... For that reason, information failed to transfer between the Mars Climate Orbiter spacecraft team at Lockheed Martin in Colorado and the mission navigation team in California.”



Picture and text from CNN.com, “NASA’s metric confusion caused Mars orbiter loss”, <http://www.cnn.com/TECH/space/9909/30/mars.metric>

# Why Units of Measurement?

---

- Tangible: unit safety violations have caused some well-known malfunctions; units used in many applications
- Interesting: has been the focus of much research, many different possible approaches
- Challenging: units have equational properties (not standard types); software in scientific domains can be hard to analyze (C, C++, Fortran, etc...)



# First Rewriting Logic Semantics Approaches

---

- Unit checker for BC [Chen et al, RTA'03]
- Unit checker for small subset of C [Rosu and Chen, ASE'03]
- Added annotations in comments for specifying unit properties
- Whole program analysis, abstract evaluation semantics

# What's Wrong? Early work was not scalable!

---

- Major rework needed to extend semantics
- New analysis == complete new semantics
- Could not share specification fragments between analyses
- Whole program analysis: not scalable for users
- New Goal: build a semantics-based, modular analysis framework

# Solution: Policy Frameworks!

---



- Modular static analysis framework
- Built in Maude with K-style rewriting logic semantics
- Language generic: analysis domains
- Language-specific, analysis-generic: base semantics, annotation-aware parser
- Analysis-specific: analysis semantics, annotation language



# CPF

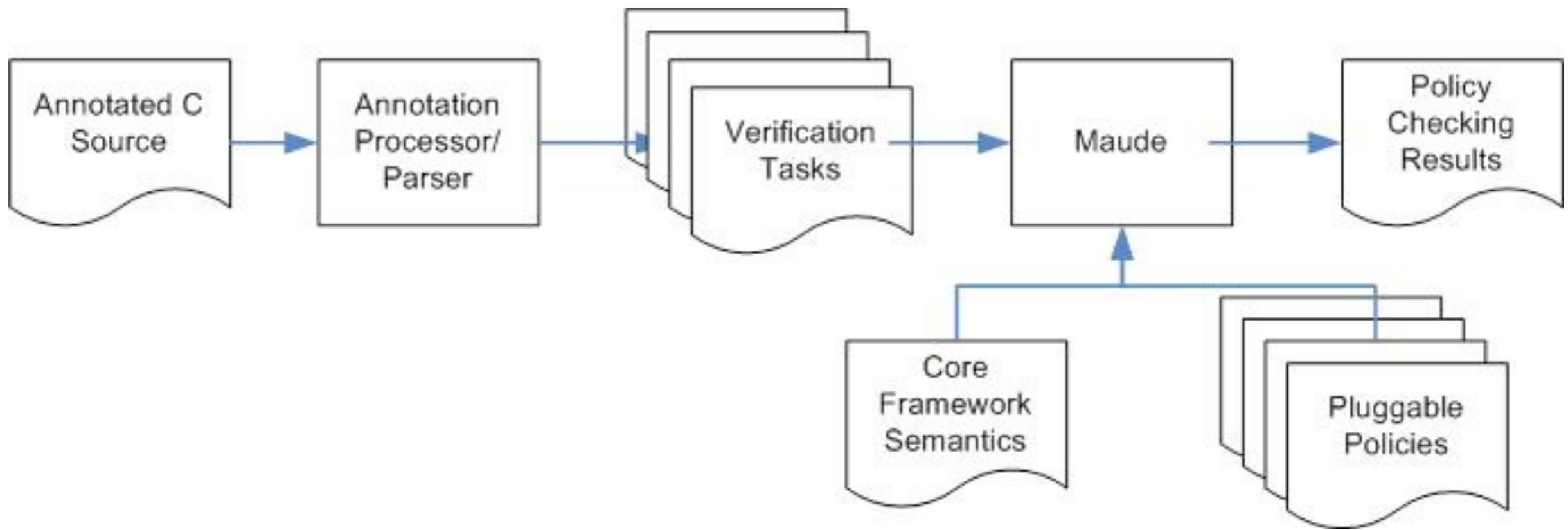
---

- CPF: C Policy Framework, analysis policies for units of measurement and pointer analysis [Hills et. al, RULE'08]
- Worked on real C code, found unit bugs seeded in NASA test code (C++ converted to C)

```
//@ pre(UNITS): unit(material->atomicWeight) = kg
//@ pre(UNITS): unit(material->atomicNumber) = noUnit
//@ 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 );
}
```

# CPF: Architectural View

---



# SILF-PF

---

- SILF-PF: SILF Policy Framework, policies for units and types [Hills and Rosu, RTA'10]
- Annotations added as language constructs and types
- Units domain shared between C and 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
```

# Did this Work?

---

- Reuse of modules achieved in both CPF and SILF-PF
- Reuse of annotation-aware frontends for both policy frameworks (CIL for CPF, custom for SILF-PF)
- UNITS analysis domain shared between frameworks for SILF and C

# Did this Work?

---

- Reuse of modules achieved in both CPF and SILF-PF
- Reuse of annotation-aware frontends for both policy frameworks (CIL for CPF, custom for SILF-PF)
- UNITS analysis domain shared between frameworks for SILF and C



# Overview

---

- Policy Frameworks
- Challenges
- DSLs for Program Analysis

# Modularity works, so what's wrong?

---



- Need to define “boilerplate” functionality to interact with existing framework
- Need to know which hooks are available for extension
- Need to know what modules can be extended
- Need to write lots of redundant cases for error propagation
- Need to define custom annotation languages and parsers

# Why is this a problem?

---



- CPF Core: 69 modules, 548 ops, 586 equations, 2016 lines
- CPF Units: 22 modules, 56 ops, 291 equations, 805 lines
- More than 100 “hooks” for policy-specific semantics



# What happened?

---

- We've transformed a specification challenge into a software engineering challenge -- more scalable in some ways, but not necessarily for the users
- Q: How do we make writing policies more abstract?
- Q: How do we provide support for people (other than me) to extend this?

# Overview

---

- Policy Frameworks
- Challenges
- DLSs for Program Analysis

# Why DSLs?

---

- Raise level of abstraction
- Provide reuse between language frameworks
- Provide clean separation of concerns between different tool aspects
- Generate complex parts of the specification

# How do policies vary?

---



- Annotation languages
- Abstract value domains
- Memory layouts
- Rule definitions

```
//@ pre(UNITS): unit(material->atomicWeight) = kg
//@ pre(UNITS): unit(material->atomicNumber) = noUnit
//@ 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 );
}
```

# How do policies vary?



- Annotation languages
- Abstract value domains
- Memory layouts
- Rule definitions

```
//@ pre(UNITS): unit(material->atomicWeight) = kg
//@ pre(UNITS): unit(material->atomicNumber) = noUnit
//@ 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 );
}
```

# How do policies vary?



- Annotation languages
- Abstract value domains
- Memory layouts
- Rule definitions

```
//@ pre(UNITS): unit(material->atomicWeight) = kg
//@ pre(UNITS): unit(material->atomicNumber) = noUnit
//@ 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 );
}
```

# How do policies vary?



- Annotation languages
- Abstract value domains
- Memory layouts
- Rule definitions

```
//@ pre(UNITS): unit(material->atomicWeight) = kg
//@ pre(UNITS): unit(material->atomicNumber) = noUnit
//@ 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 );
}
```

# How do policies vary?



- Annotation languages
- Abstract value domains
- Memory layouts
- Rule definitions

```
//@ pre(UNITS): unit(material->atomicWeight) = kg
//@ pre(UNITS): unit(material->atomicNumber) = noUnit
//@ 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 );
}
```



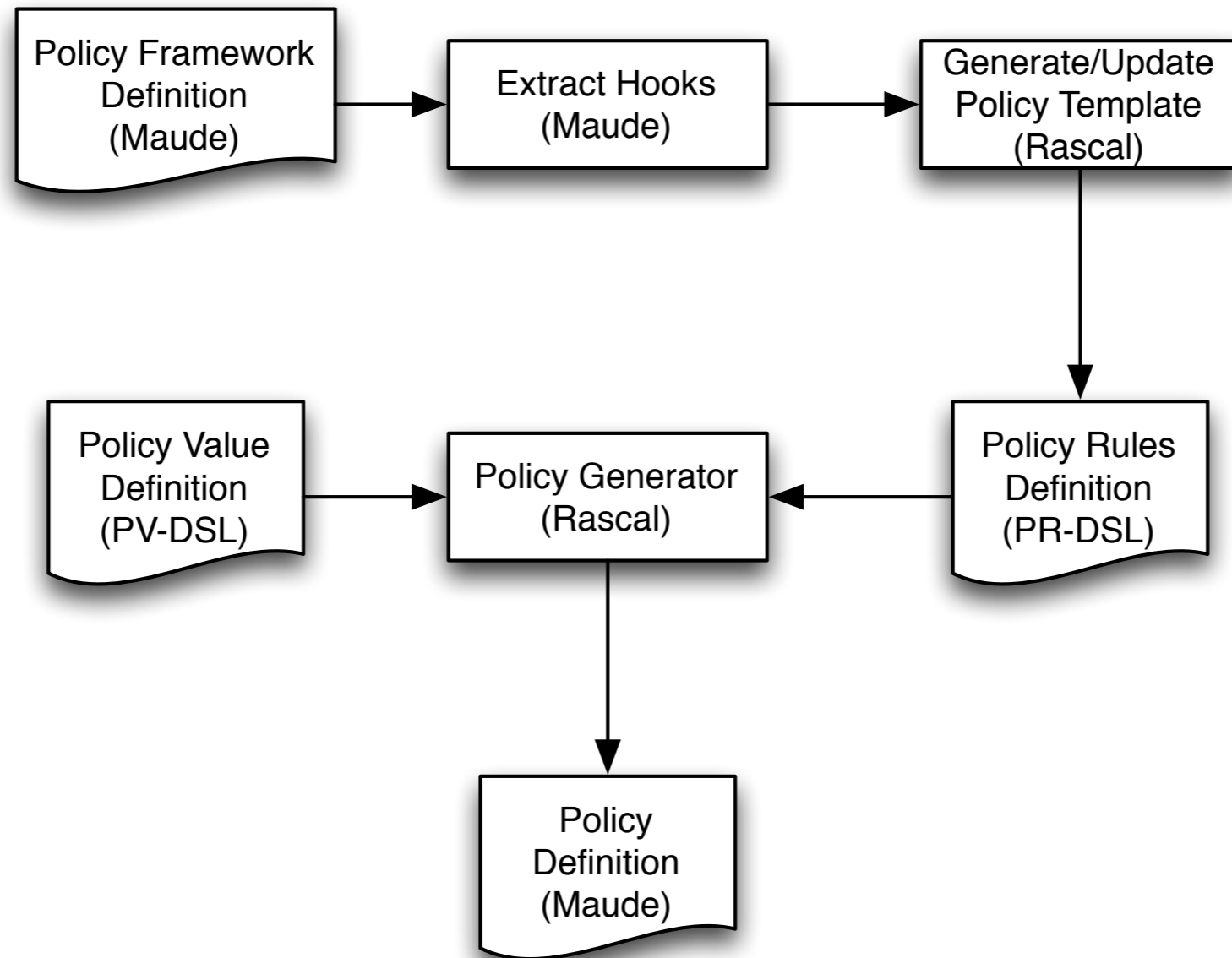
# Some opportunities for DSLs...

---

- Annotation languages
- Abstract value domains (PV-DSL)
- Memory layouts
- Rule definitions/skeletons (PR-DSL)
- Control Flow Graph construction
- Intermediate code generation (for program analysis)

# DSLs for Policy Frameworks: Architecture

---



# PV-DSL: Defining analysis domains

---

- Domains should be defined in declarative manner
- Need flexibility to handle complex domains like units
- Need ability to generate boring boilerplate specification

# Current Code: SILF Type Domain

---

```
ops $int $bool : -> PolicyVal .  
op $notype : -> PolicyVal .  
op $array : BaseType -> PolicyVal .
```

```
eq pv2pv($('int')) = $int .  
eq pv2pv($('bool')) = $bool .  
eq pv2pv($('array') ( T ) ) = $array(pv2pv(T)) .
```

```
eq ta2pv($('int')) = $int .  
eq ta2pv($('bool')) = $bool .  
eq ta2pv($('array') ( T ) ) = $array(ta2pv(T)) .
```

```
eq pretty-print($int) = "$int" .  
eq pretty-print($bool) = "$bool" .  
eq pretty-print($notype) = "$notype" .  
eq pretty-print($array(T)) = "$array(" + pretty-print(T) + ")" .
```

# PV-DSL: SILF Type Domain

---

Domain SILF-TYPES

base \$integer also \$int;

base \$boolean also \$bool;

base \$notype;

base \$array(pv:PolicyVal) display as "\$array[" + pv + "];"

base \$map(dom:PolicyVal, rng:PolicyVal)

display as "\$map[" + dom + ", " + rng + "];"

derived \$intArray = \$array(\$int);

End Domain

# PV-DSL: SILF Type Domain

Domain SILF-TYPES



Policy Name  
Provides Filtering

```
base $integer also $int;
```

```
base $boolean also $bool;
```

```
base $notype;
```

```
base $array(pv:PolicyVal) display as "$array[" + pv + "];"
```

```
base $map(dom:PolicyVal, rng:PolicyVal)
```

```
    display as "$map[" + dom + ", " + rng + "];"
```

```
derived $intArray = $array($int);
```

```
End Domain
```

# PV-DSL: SILF Type Domain

Domain SILF-TYPES

```
base $integer also $int;  
base $boolean also $bool;  
base $notype;  
base $array(pv:PolicyVal) display as "$array[" + pv + "];"  
base $map(dom:PolicyVal, rng:PolicyVal)  
    display as "$map[" + dom + "," + rng + "];"  
  
derived $intArray = $array($int);
```

End Domain

Policy Name  
Provides Filtering

Annotation  
Filtering Rules  
Generated

# PV-DSL: SILF Type Domain

Domain SILF-TYPES

```
base $integer also $int;  
base $boolean also $bool;  
base $notype;  
base $array(pv:PolicyVal) display as "$array[" + pv + "];"  
base $map(dom:PolicyVal, rng:PolicyVal)  
  display as "$map[" + dom + "," + rng + "];"  
  
derived $intArray = $array($int);
```

End Domain

Policy Name  
Provides Filtering

Annotation  
Filtering Rules  
Generated

Default Pretty  
Printing Rules



# PV-DSL: SILF Type Domain

Domain SILF-TYPES

```
base $integer also $int;  
base $boolean also $bool;  
base $notype;  
base $array(pv:PolicyVal) display as "$array[" + pv + "];"  
base $map(dom:PolicyVal, rng:PolicyVal)  
  display as "$map[" + dom + "," + rng + "];"  
  
derived $intArray = $array($int);
```

End Domain

Policy Name  
Provides Filtering

Annotation  
Filtering Rules  
Generated

Default Pretty  
Printing Rules

Custom Pretty  
Printing Rule

# PV-DSL: Units Domain (partial)

---

Domain UNITS

# Length

base \$meter also \$m;

# Mass

base \$kilogram also \$kg;

# Builders

operator  $\_ \wedge \_$  : PolicyVal Rat  $\rightarrow$  PolicyVal .

operator  $\_ \_$  : PolicyVal PolicyVal  $\rightarrow$  PolicyVal .

# Equalities

eq U:PolicyVal  $U = U \wedge 2$  .

eq  $(U \wedge N:\text{Rat}) \wedge M:\text{Rat} = U \wedge (N * M)$  .

# Derived Units

derived \$hertz also \$Hz = \$s  $\wedge -1$ ;

End Domain

# PR-DSL: Making specifications reflective

---

- Extension points, i.e. “hooks”, are operators with no defining equations
- New policies provide equations to add functionality
- How to find hooks? all ops in a module? all ops of a given sort or sorts?
- Rewriting logic is reflective: why not allow specifications to reason about where they can be extended?

# PR-DSL: Reflection in Action

---

```
op defaultIntVal : -> Value [metadata "hook"] .
```

```
ops + - * / % : Exp Exp -> ComputationItem [metadata "hook"] .
```

```
Maude> red hookRelToRascal(computeHookRel('GENERIC-ARITH-SEMANTICS)) .  
reduce in HOOK-OPS : hookRelToRascal(computeHookRel('GENERIC-ARITH-  
SEMANTICS))
```

```
rewrites: 201 in 0ms cpu (0ms real) (1 1823529 rewrites/second)
```

```
result String: "[hook(\"GENERIC-ARITH-SEMANTICS\", \"%\", [\"Exp\", \"Exp\"],  
\"ComputationItem\"), hook(\"GENERIC-ARITH-SEMANTICS\", \"*\", [\"Exp\", \"Exp\"],  
\"ComputationItem\"),  
hook(\"GENERIC-ARITH-SEMANTICS\", \"+\", [\"Exp\", \"Exp\"], \"ComputationItem\"),  
hook(\"GENERIC-ARITH-SEMANTICS\", \"-\", [\"Exp\", \"Exp\"], \"ComputationItem\"),  
hook(\"GENERIC-ARITH-SEMANTICS\", \"^\", [\"Exp\", \"Exp\"], \"ComputationItem\"),  
hook(\"GENERIC-ARITH-SEMANTICS\", \"u-\", [\"Exp\", \"ComputationItem\"]]"
```

# PR-DSL: Reflection in Action

Sample Hook  
Definitions in  
Maude

```
op defaultIntVal : -> Value [metadata "hook"] .
ops + - * / % : Exp Exp -> ComputationItem [metadata "hook"] .
```

```
Maude> red hookRelToRascal(computeHookRel('GENERIC-ARITH-SEMANTICS)) .
reduce in HOOK-OPS : hookRelToRascal(computeHookRel('GENERIC-ARITH-SEMANTICS))
```

```
rewrites: 201 in 0ms cpu (0ms real) (11823529 rewrites/second)
result String: "[hook(\"GENERIC-ARITH-SEMANTICS\", \"%\", [\"Exp\", \"Exp\"],
\"ComputationItem\"), hook(\"GENERIC-ARITH-SEMANTICS\", \"*\", [\"Exp\", \"Exp\"],
\"ComputationItem\"),
hook(\"GENERIC-ARITH-SEMANTICS\", \"+\", [\"Exp\", \"Exp\"], \"ComputationItem\"),
hook(\"GENERIC-ARITH-SEMANTICS\", \"-\", [\"Exp\", \"Exp\"], \"ComputationItem\"),
hook(\"GENERIC-ARITH-SEMANTICS\", \"^\", [\"Exp\", \"Exp\"], \"ComputationItem\"),
hook(\"GENERIC-ARITH-SEMANTICS\", \"u-\", [\"Exp\", \"ComputationItem\"]]"
```

# PR-DSL: Reflection in Action

Sample Hook  
Definitions in  
Maude

```
op defaultIntVal : -> Value [metadata "hook"] .  
ops + - * / % : Exp Exp -> ComputationItem [metadata "hook"] .
```

```
Maude> red hookRelToRascal(computeHookRel('GENERIC-ARITH-SEMANTICS)) .  
reduce in HOOK-OPS : hookRelToRascal(computeHookRel('GENERIC-ARITH-  
SEMANTICS))
```

```
.  
rewrites: 201 in 0ms cpu (0ms real) (11823529 rewrites/second)  
result String: "[hook(\"GENERIC-ARITH-SEMANTICS\", \"%\", [\"Exp\", \"Exp\"],  
\"ComputationItem\"), hook(\"GENERIC-ARITH-SEMANTICS\", \"*\", [\"Exp\", \"Exp\"],  
\"ComputationItem\"),  
hook(\"GENERIC-ARITH-SEMANTICS\", \"+\", [\"Exp\", \"Exp\"], \"ComputationItem\"),  
hook(\"GENERIC-ARITH-SEMANTICS\", \"-\", [\"Exp\", \"Exp\"], \"ComputationItem\"),  
hook(\"GENERIC-ARITH-SEMANTICS\", \"^\", [\"Exp\", \"Exp\"], \"ComputationItem\"),  
hook(\"GENERIC-ARITH-SEMANTICS\", \"u-\", [\"Exp\", \"Com
```

Extraction from  
Maude (in Rascal-  
friendly format)

# PR-DSL: Creating Policy Rule Skeletons

---

## Policy SILF-TYPES

prule[GENERIC-ARITH-SEMANTICS, + : Exp Exp -> Exp]:

$k(\text{val}(V1, V2) \rightarrow +(E1, E2) \rightarrow K) = k(K)$

if notfail(V1) and notfail(V2) .

prule[GENERIC-ARITH-SEMANTICS, + : Exp Exp -> Exp]:

$k(\text{val}(V1, V2) \rightarrow +(E1, E2) \rightarrow K) =$

$k(\text{mergefail}(V1, V2) \rightarrow K)$  if fail(V1) or fail(V2) .

End Policy

# PR-DSL: Creating Policy Rule Skeletons

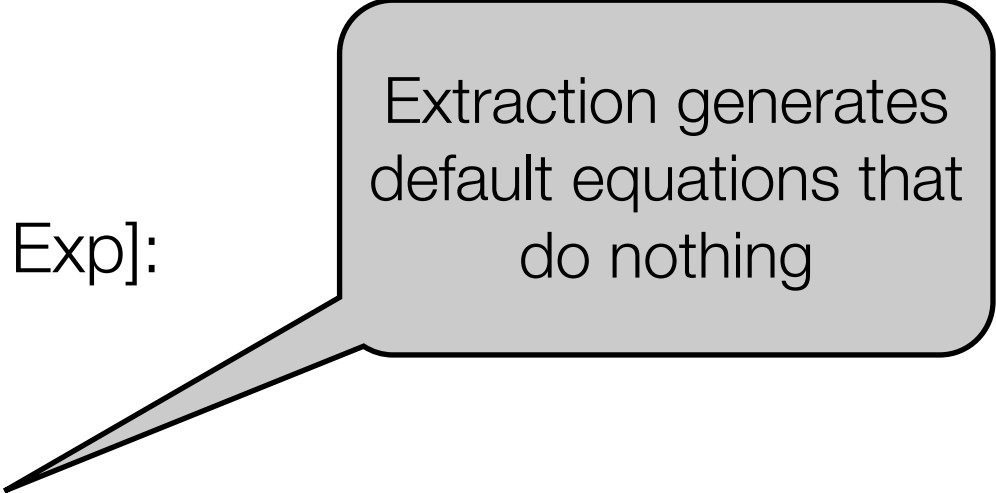
---

## Policy SILF-TYPES

```
prule[GENERIC-ARITH-SEMANTICS, + : Exp Exp -> Exp]:  
  k(val(V1,V2) -> +(E1,E2) -> K) = k(K)  
  if notfail(V1) and notfail(V2) .
```

```
prule[GENERIC-ARITH-SEMANTICS, + : Exp Exp -> Exp]:  
  k(val(V1,V2) -> +(E1,E2) -> K) =  
  k(mergefail(V1,V2) -> K) if fail(V1) or fail(V2) .
```

End Policy



Extraction generates default equations that do nothing



# PR-DSL: Creating Policy Rule Skeletons

---

## Policy SILF-TYPES

```
prule[GENERIC-ARITH-SEMANTICS, + : Exp Exp -> Exp]:  
  k(val(V1,V2) -> +(E1,E2) -> K) = k(K)  
  if notfail(V1) and notfail(V2) .
```

Extraction generates default equations that do nothing

```
prule[GENERIC-ARITH-SEMANTICS, + : Exp Exp -> Exp]:  
  k(val(V1,V2) -> +(E1,E2) -> K) =  
  k(mergefail(V1,V2) -> K) if fail(V1) or fail(V2) .
```

Need to add better notation for error propagation: still working on this (currently done by writing more equations)

End Policy

# PR-DSL: Creating Policy Rule Skeletons

---

## Policy SILF-TYPES

```
prule[GENERIC-ARITH-SEMANTICS, + : Exp Exp -> Exp]:  
  k(val(V1,V2) -> +(E1,E2) -> K) = k(K)  
  if notfail(V1) and notfail(V2) .
```

Extraction generates default equations that do nothing

```
prule[GENERIC-ARITH-SEMANTICS, + : Exp Exp -> Exp]:  
  k(val(V1,V2) -> +(E1,E2) -> K) =  
  k(mergefail(V1,V2) -> K) if fail(V1) or fail(V2) .
```

End Policy

Limitation: don't want to reparse Maude, so the body isn't checked...

Need to add better notation for error propagation: still working on this (currently done by writing more equations)

# Wrap-Up: Further Challenges

---



- Reflection: How can we extend this to other parts of the specification?
- How can we model memory in languages like C?
- How can we support developers in writing semantic rules (parsing/error reporting/etc)?
- How can we make all these DSLs work well across languages?


[MarkHills](#) (karma: 341 badges: ● 6 ● 10) [sign out](#) [help](#)

[tags](#) [people](#) [badges](#)

[ALL](#) [UNANSWERED](#) [FOLLOWED](#)


[ASK YOUR QUESTION](#)

**102 questions**

 Sort by » [date](#) [activity](#) ▼ [answers](#) [votes](#) [RSS](#)

Search tip: add tags and a query to focus your search

**Operator Overloading**

 no votes    2 answers    19 views

[operator](#) [overloading](#) [support](#)

Mar 07 Hosseln

**How to solve this MissingFormatArgumentException?**

 no votes    1 answer    11 views

[java](#) [exception](#)

Mar 07 Atze

Contributors



- Rascal: <http://www.rascal-mpl.org>
- SWAT: <http://www.cwi.nl/sen1>
- Me: <http://www.cwi.nl/~hills>