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.”

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...)

Thursday, June 27, 13
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)

```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

```c
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

```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 );
}
```
How do policies vary?

• Annotation languages

• Abstract value domains

• Memory layouts

• Rule definitions

```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 );
}
```
How do policies vary?

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

```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 );
}
```
How do policies vary?

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

```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 );
}
```
How do policies vary?

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

```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 );
}
```
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

Policy Framework Definition (Maude) → Extract Hooks (Maude) → Generate/Update Policy Template (Rascal)

Policy Value Definition (PV-DSL) → Policy Generator (Rascal) → Policy Rules Definition (PR-DSL)

Policy Definition (Maude)
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 $\text{int} \  \$\text{bool} : \rightarrow \text{PolicyVal}.
op $\text{notype} : \rightarrow \text{PolicyVal}.
op $\text{array} : \text{BaseType} \rightarrow \text{PolicyVal}.

eq \text{pv2pv}($('\text{int}$)) = $\text{int}$.
eq \text{pv2pv}($('\text{bool}$)) = $\text{bool}$.
eq \text{pv2pv}($('\text{array}(\ T \ )$)) = $\text{array}(\text{pv2pv}(T))$.

eq \text{ta2pv}($('\text{int}$)) = $\text{int}$.
eq \text{ta2pv}($('\text{bool}$)) = $\text{bool}$.
eq \text{ta2pv}($('\text{array}(\ T \ )$)) = $\text{array}(\text{ta2pv}(T))$.

eq \text{pretty-print}(\text{int}) = "$\text{int}$".
eq \text{pretty-print}(\text{bool}) = "$\text{bool}$".
eq \text{pretty-print}(\text{notype}) = "$\text{notype}$".
eq \text{pretty-print}(\text{array}(T)) = "$\text{array}(" + \text{pretty-print}(T) + ")$".
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

base $integer also $int;
based $boolean also $bool;
based $notype;
based $array(pv:PolicyVal) display as "$array[" + pv + "]";
based $map(dom:PolicyVal, rng:PolicyVal)
    display as "$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
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
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: Units Domain (partial)

Domain UNITS

# Length
base $meter also $m;
# Mass
base $kilogram also $kg;

# Builders
operator _^_ : PolicyVal Rat -> PolicyVal .
operator __ : PolicyVal PolicyVal -> PolicyVal .

# Equalities
eq U:PolicyVal U = U ^ 2 .
eq (U ^ N:Rat) ^ M:Rat = U ^ (N * M) .

# Derived Units
derived $hertz also $Hz = $s ^ -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?
op defaultIntVal : -> Value [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

op defaultIntVal : -> Value [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

op defaultIntVal : -> Value [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: Creating Policy Rule Skeletons

Policy SILF-TYPES

\[
\text{prule}[\text{GENERIC-ARITH-SEMANTICS}, + : \text{Exp Exp} \rightarrow \text{Exp}]: \\
\quad k(\text{val}(V1,V2) \rightarrow +(E1,E2) \rightarrow K) = k(K) \\
\quad \text{if notfail}(V1) \text{ and notfail}(V2) .
\]

\[
\text{prule}[\text{GENERIC-ARITH-SEMANTICS}, + : \text{Exp Exp} \rightarrow \text{Exp}]: \\
\quad k(\text{val}(V1,V2) \rightarrow +(E1,E2) \rightarrow K) = \\
\quad k(\text{mergefail}(V1,V2) \rightarrow K) \text{ if fail}(V1) \text{ or fail}(V2) .
\]

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]:
  \[ k(\text{val}(V1,V2) -> +(E1,E2) -> K) = k(K) \]
  if notfail(V1) and notfail(V2) .

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

End Policy

Extraction generates default equations that do nothing

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

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?
• Rascal: http://www.rascal-mpl.org

• SWAT: http://www.cwi.nl/sen1

• Me: http://www.cwi.nl/~hills