A Rewriting Logic Semantics Approach to Modular Program Analysis

Mark Hills [1] Grigore Roșu [2]

[1]Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
`Mark.Hills@cwi.nl`

[2]Formal Systems Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
`grosu@cs.uiuc.edu`

RTA'10, 11 July 2010

Outline

Overall Goals

- Leverage rewriting logic semantics for program analysis
- Focus on modularity at two levels
    - In the definition: definition should be modular, making it possible to create new analyses while leveraging large parts of the existing system
    - In the analysis itself: should not need to analyze the entire program, but should instead include support for analysis of program fragments: functions, etc.
- Support simpler languages for experimentation with concepts (SILF) while supporting more complex languages (C) to determine if concepts work in real life

Our Approach: Policy Frameworks

A policy framework is a framework for building individual program analyses (here called policies); a framework uses a combination of a front-end language parser and a language semantics created using rewriting logic.

Our Approach: Policy Frameworks

A policy framework is a framework for building individual program analyses (here called policies); a framework uses a combination of a front-end language parser and a language semantics created using rewriting logic.

Individual analysis policies provide a combination of an annotation language and an analysis semantics: analysis leverages term rewriting by evaluating a program in an abstract rewriting logic semantics.

Goals of The Work Presented Here

- Extended earlier work on CPF, a policy framework for C, to provide support for type annotations – CPF supported only annotations in code comments and in comments on function headers
- Provide a simpler environment for experimentation: earlier work on C made it hard to untangle complexity of the technique from the complexity of the language
- Provide examples of additional policies: in this case several variants on checking units of measurement plus a static type system

Motivation for This Approach

Why take this approach?

- Rewriting logic powerful enough to define abstract analysis semantics even for complex features of languages
- Modularity of rewriting logic definitions and K (the notation used here for the semantic rules) provides reuse, allowing a framework of reusable pieces to be built
- Annotation-driven approach taken here provides a natural mechanism for programmers to give the analysis needed information

Rewriting Logic Semantics

- Presented work in part of Rewriting Logic Semantics project (Meseguer and Roșu, TCS'07)
- Project encompasses many different languages, definitional formalisms, goals (analysis, execution, formal verification, etc.)
- Presented work falls into *continuation-based* style described in earlier published work, and is written using K notation
- Programs represented as first-class computations that can be stored, manipulated, and executed, with execution here equal to analysis

Outline

The SILF Language

- SILF is the **S**imple **I**mperative **L**anguage with **F**unctions
- Provides standard features of a paradigmatic imperative language: functions, globals, arrays, IO
- Introduced in earlier work (Hills, Serbanuta and Rosu, WRLA'07) (Hills, WRLA'08), so here we can just focus on the extensions

The SILF Language

Integer Numbers	$N ::=$	$(+\|\text{-})?(0..9)^+$
Declarations	$D ::=$	var I | var $I[N]$
Expressions	$E ::=$	N | $E + E$ | E - E | E * E | E / E | E % E | - E |
		$E < E$ | $E <= E$ | $E > E$ | $E >= E$ | $E = E$ | E != E |
		E and E | E or E | not E | N | $I(El)$ | $I[E]$ | I | read
Expression Lists	$El ::=$	$E\ (, E)^*$ | *nil*
Statements	$S ::=$	$I := E$ | $I[E] := E$ | if E then S fi | if E then S else S fi |
		for $I := E$ to E do S od | while E do S od | S; S | D |
		$I(El)$ | return E | write E
Function Declarations	$FD ::=$	function $I(Il)$ begin S end
Identifiers	$I ::=$	$(\mathrm{a}-\mathrm{zA}-\mathrm{Z})(\mathrm{a}-\mathrm{zA}-\mathrm{Z0}-9)^*$
Identifier Lists	$Il ::=$	$I\ (, I)^*$ | *void*
Programs	$Pgm ::=$	$S?\ FD^+$

Extension strategy

- Question 1: Add analysis extensions in comments, or directly extend language?
  - Add in comments, can add policy framework while not breaking existing implementations
  - Extend language, can better integrate analysis features
  - Here, go with #2 – our own language, no concerns over breaking implementations

Extension strategy

- Question 1: Add analysis extensions in comments, or directly extend language?
    - Add in comments, can add policy framework while not breaking existing implementations
    - Extend language, can better integrate analysis features
    - Here, go with #2 – our own language, no concerns over breaking implementations
- Question 2: Use just type annotations, just code annotations, or both?
    - Just code annotations make annotation language more verbose
    - Just type annotations can make some analysis information difficult to encode
    - Here, use both: allows user to use whichever feels most "natural" and can encode the information properly

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

Frameworks-Related SILF Extensions

Declarations	$D ::=$	... | var TI | var $TI[N]$
Statements	$S ::=$	... | for $I := E$ to E IVl do S od | while E IVl do S od | assert(I): *ann*; | assume(I): *ann*;
Function Declarations	$FD ::=$	function $TI(TIl)$ PPl begin S end
Typed Identifiers	$TI ::=$	I | *tann* I | *tvar* I
Typed Identifier Lists	$TIl ::=$	TI $(, TI)^*$ | *void*
Invariants	$IV ::=$	inv(I): *ann*; | invariant(I): *ann*;
Invariant Lists	$IVl ::=$	$IV*$
PrePosts	$PP ::=$	pre(I): *ann*; | precond(I): *ann*; | post(I): *ann*; | postcond(I): *ann*; | mod(I): *ann*; | modifies(I): *ann*;
PrePost Lists	$PPl ::=$	$PP*$

Defining Types in SILF

```
sort BaseType .
subsort BaseType < Type .
ops $int $bool : -> BaseType .
op $array : BaseType -> Type .
op $notype : -> Type .
```

Defining Type Checking Rules

$$i_1 + i_2 \rightarrow i,\ \text{ if } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{1}$$

$$(\$int, \$int) \curvearrowright plus \rightarrow \$int \tag{2}$$

$$\langle k \rangle \frac{(t, t') \curvearrowright plus}{issueWarning(1, msg) \curvearrowright \$int} ... \langle / k \rangle,\quad \text{if } t =/= \$int \text{ or } t' =/= \$int \tag{3}$$

$$\texttt{if } true \texttt{ then } Kt \texttt{ else } Kf \rightarrow Kt \tag{4}$$

$$\$bool \curvearrowright \texttt{if}(Kt, Kf) \rightarrow Kt \curvearrowright Kf \tag{5}$$

Checking 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: Identifier m is not defined.
```

Checking Types in SILF (2)

```
 1  function $int f($int x)
 2  begin
 3    return x + 1;
 4  end
 5  function $int main(void)
 6  begin
 7    var $int x;
 8    x := 3;
 9    x := f(x);
10    x := f(x,x);
11    if x then write 1; fi
12    if (x < 5) then write 1; else write false; fi
13  end
```

```
Type checking found errors:
  ERROR on line 10: Too many arguments provided in call to function f.
  ERROR on line 11: Expression x should have type $bool, but has type $int.
  ERROR on line 12: Write expression false has type $bool, expected type $int.
```

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.
```

Outline

Annotation-Based Approaches

- Osprey (Jiang and Su, ICSE'06) uses type annotations to check units of measurement safety for C programs: fast, less flexible than approach used here (limited polymorphism, unit of function result cannot be tied to input units, etc)
- Spec# (Barnett, Leino, and Schulte, CASSIS'04), JML (Burdy et.al. FMICS'03) provide annotation systems for (an extension to) C# and Java
- CQUAL (Foster, FA'99) provides type annotation system for C, seems to handle more limited annotations than we handle here
- Frama-C provides very similar support for C, but performs static analysis using plug-ins written in OCaml as extensions to the base framework

Earlier Approaches Using RLS

- Initial work on BC (Chen, Roșu, and Venkatesan, RTA'03) started this line of work
- Follow-up C-UNITS system (Feng and Roșu, ASE'03) applied this approach to C and units of measurement; used older semantic style, much harder to extend, didn't support many important C language features
- CPF (Hills, Chen and Roșu, RULE'08) reformulated this using a K-style RLS definition, making it much more modular, while also focusing on complex C language features (function pointers, gotos, etc)

Outline

Conclusion

- The SILF Policy Framework and CPF demonstrate the viability of using rewriting logic semantics as a platform for writing static analysis tools
- Experience with type and code annotations shows that both can be useful and should be supported, but with knowledge of the tradeoffs (e.g., use of type annotations would prevent standard compilers from compiling code)