A Rewriting Logic Approach to Static Checking of Units of Measurement in C

Mark Hills, Feng Chen, and Grigore Roșu
{mhills, fengchen, grosu}@cs.uiuc.edu

Formal Systems Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign

RULE'08, 18 July 2008

Outline

1. Motivation
2. CPF
3. Unit Safety
4. Related Work
5. Conclusion

Why 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, `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

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

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; software in scientific domains can be hard to analyze (C, C++, Fortran, etc...)

High Level Approach: Leverage Formal Language Definitions

- Our belief: having formal definitions of programming languages is important
- Without a formal definition, impossible to effectively reason about programs
- Research goal: increase usefulness of formal definitions, should lead to increased adoption
- Practical: leverage existing tools, language definition and analysis techniques, expertise

Contributions

- Extended earlier work on C-UNITS to provide coverage of complex language constructs
- Generalized domain-specific analysis framework, using rewriting logic semantics, to handle many domains, including units
- Provided a more modular, faster analysis capable of handling larger programs
- UNITS policy capable of extension to match other similar tools, while currently providing more flexibility

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
- Programs represented as first-class computations that can be stored, manipulated, executed

Outline

1. Motivation
2. CPF
3. Unit Safety
4. Related Work
5. Conclusion

The C Policy Framework

- Earlier work on C language in our group very focused on specific problem domains
- Wanted to extend this work to generalize it for many domains
- Also wanted to increase performance and flexibility, ensure we can handle realistic C programs
- Want to make sure it is formal, based on a (possibly domain specific) semantics of C
- Result: The C Policy Framework (CPF)

CPF Core

CPF provides generic functionality for C program analysis:

- Annotation processing
- C program parsing
- C abstract syntax
- Semantics for C statements
- Generic semantics for some expressions
- Extension hooks

CPF Policies

CPF Policies are domain-specific extensions to CPF:

- Abstract semantics for expressions and declarations
- Annotation language
- Annotation language processor
- Overrides of generic CPF functionality

CPF Policies

CPF Policies are domain-specific extensions to CPF:

- Abstract semantics for expressions and declarations
- Annotation language
- Annotation language processor
- Overrides of generic CPF functionality
- CPF Core + CPF Policy = Domain-Specific Abstract Semantics of C

Annotation Processing

- CPF allows information to be added in annotations
- Annotations provided in C comments
- Annotation processor moves these into C code, utilizing custom extension to C language (but not visible to user)

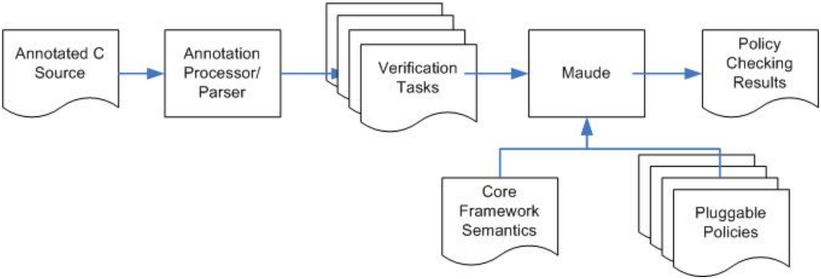
Example: Annotations

```
//@ 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 );
}
```

Parsing

- Parsing performed using customized CIL
- C programs with inlined annotations taken as input
- CPF-specific program transformations performed
  - pre- and post-condition inlining
  - simplification
  - limited alias analysis
- Maude code, using C abstract syntax, generated

CPF Processing



C Abstract Syntax/Generic State

- Abstract syntax provided for all C constructs not removed by CIL
- 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

- Currently support all C statements not removed by CIL (including goto)
- Statements executed in *environments*
  - Some statements can return different values along different paths
  - Environments capture path-sensitive information
  - Sets of environments used, with a statement executed once in each env in the set
  - Can cause problems: need to limit size of env set to prevent exponential explosion
  - Special logic to handle temporaries created by CIL
- Can be disabled in policies that do not need it

Outline

The UNITS Policy

- 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
- Combination CPF + UNITS = CPF[UNITS]

Unit Representation

```
op _^_ : Unit Rat -> Unit .
op __ : Unit Unit -> Unit [assoc comm] .
eq U ^ 0 = noUnit .
eq U ^ 1 = U .
eq U U = U ^ 2 .
eq U (U ^ Q) = U ^ (Q + 1) .
eq (U ^ Q) (U ^ P) = U ^ (Q + P) .
eq (U U') ^ Q = (U ^ Q) (U' ^ Q) .
eq (U ^ Q) ^ P = U ^ (Q * P) .
ops noUnit any fail cons : -> Unit .
ops meter m feet f : -> Unit .
```

Unit Annotations

$$
\begin{array}{lrl}
\textit{Unit} & U ::= & \texttt{unit(}E\texttt{)} \mid \texttt{unit(}E\texttt{)} \wedge\ Q \mid BU \mid U\ U \\
\textit{UnitExp} & UE ::= & U \mid U\ =\ UE \mid UE\ \texttt{and}\ UE \mid UE\ \texttt{or}\ UE \mid \\
 & & UE\ \texttt{implies}\ UE \mid \texttt{not}\ UE
\end{array}
$$

Annotations allowed in preconditions, postconditions, assert statements, assume statements

UNITS Abstract Values

```
op _^_ : Unit CInt -> Unit .
op u : Unit -> Value .
op ptr : Location -> Value .
op arr : Location -> Value .
op struct : Identifier SFieldSet -> Value .
op union : Identifier SFieldSet -> Value .
```

Declaration Semantics

- Declarations of non-unit values reusable in other policies
  - Structures, unions as maps
  - Pointers, arrays as references to other locations, eventually point to an abstract value
- Declarations of numeric values assigned abstract unit values
- "Fresh" unit values assigned as default to catch unit errors without preventing normal computations

Expression Semantics

- Expressions manipulate UNITS abstract values, including unit values and pointers
- Semantics ensures that attempts to combine units maintain unit safety
- Expressions working with structures build structure representation as needed during analysis
- Memory model handles allocations and casts
- Note: no function calls – removed by CIL

Expression Semantics

```
[1] U * U' = U U'

[2] U + U' = mergeUnits(U,U') -> checkForFail("+")

[3] U > U' = mergeUnits(U,U') -> checkForFail(">") ->
             discard -> noUnit

[4] (lvp(L,V) = V') = V' -> assign(L)

[5] (lvp(L,U) += U') = mergeUnits(U,U') -> checkForFail("=") ->
                       assign(L)

[6] *(lvp(L,ptr(L'))) = llookup(L')

[7] lvp(L,struct(X', (sfield(X,L') _))) . X = llookup(L')
```

Performance

		Total Time			Average Per Function		
Test	LOC	x100	x400	x4000	x100	x400	x4000
straight	25	6.39	23.00	229.80	0.06	0.06	0.06
ann	27	8.62	31.27	307.54	0.09	0.08	0.08
nosplit	69	12.71	46.08	467.89	0.13	0.12	0.12
split	69	27.40	106.55	1095.34	0.27	0.27	0.27

Times in seconds. All times averaged over three runs of each test. LOC (lines of code) are per function, with 100, 400, or 4000 identical functions in a source file.

Error Detection

Test	Prep Time	Check Time	LOC	Annotations	Errors	FP
ex18.c	0.083	0.754	18	10	3	0
fe.c	0.113	0.796	19	9	1	0
coil.c	0.113	59.870	299	14	3	3
projectile.c	0.122	0.882	31	16	0	0
projectile-bad.c	0.121	0.866	31	16	1	0
big0.c	0.273	5.223	2705	0	0	0
big1.c	0.998	22.853	11705	0	0	0
big2.c	33.144	381.367	96611	0	0	0

Times in seconds. All times averaged over three runs of each test. Function count includes annotated prototypes in parens. FP represents False Positives.

CPF Safety Restrictions

- Address capture
- Pointers
- Formal parameters
- Aliasing (the root of all evil)
- Precondition/post-condition requirement
- Fresh units

Relaxing restrictions can eliminate false positives, at the cost of potential missed errors.

Outline

Libraries

- Solutions involve using unit-specific libraries to enforce safety
- SIUNITS and C++ meta-programming (Brown, 2001)
- MDS JPL C++ library
- Others in Eiffel, Ada, probably more

Language and Type System Extensions

- MetaGen (Allen, Chase, Luchangco, Maessen, and Steele, OOPSLA'04)
- ML Dimensions/Type Inference (Kennedy, PhD Thesis)
- Older work on extensions to Pascal, Ada
- Newer work on Osprey (Jiang and Su, ICSE'06) also for C; fast, less flexible, checks at level of dimensions

Annotations

- Annotation-based systems widely used: Spec# (Barnett, Leino, and Schulte, CASSIS'04), JML (Burdy et.al. FMICS'03)
- Precursor C-UNITS system (Feng and Roșu, ASE'03) inspiration for current work, but extremely limited

Outline

Summary

- CPF[UNITS] extends C-UNITS with support for much larger portion of C language, more modular unit checking, improved parsing, easier to modify semantics
- Leverages formal techniques for defining (abstract) language semantics
- Initial tests show efficiency
- Annotation language, annotation burden compare well with Osprey – tradeoff between flexibility and performance

Thank You

http://fsl.cs.uiuc.edu/cpf