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
1 Motivation
2 CPF
3 Unit Safety
4 Related Work
5 Conclusion
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...)
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 Roşu, 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
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 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 are domain-specific extensions to CPF:

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

```cpp
//@ 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

Outline
Motivation
CPF
Unit Safety
Related Work
Conclusion

Overview
Pre-processing
Core Semantics

Diagram:
- Annotated C Source
- Annotation Processor/Parser
- Verification Tasks
- Maude
- Policy Checking Results
- Core Framework Semantics
- Pluggable Policies
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

1. Motivation
2. CPF
3. Unit Safety
4. Related Work
5. Conclusion
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
- CombinationCPF + UNITS = CPF[UNITS]
Unit Representation

\[
\begin{align*}
\text{op } \_\_\_ & : \text{Unit } \text{Rat} \rightarrow \text{Unit} . \\
\text{op } \_\_ & : \text{Unit } \text{Unit} \rightarrow \text{Unit} \text{ [assoc comm] .} \\
\text{eq } U ^ 0 & = \text{noUnit} . \\
\text{eq } U ^ 1 & = U . \\
\text{eq } U U & = U ^ 2 . \\
\text{eq } U (U ^ Q) & = U ^ (Q + 1) . \\
\text{eq } (U ^ Q) (U ^ P) & = U ^ (Q + P) . \\
\text{eq } (U U') ^ Q & = (U ^ Q) (U' ^ Q) . \\
\text{eq } (U ^ Q) ^ P & = U ^ (Q * P) . \\
\text{ops } \text{noUnit} \text{ any fail cons : } & \rightarrow \text{Unit} . \\
\text{ops } \text{meter m feet f : } & \rightarrow \text{Unit} .
\end{align*}
\]
Unit Annotations

Unit
\[ U ::= \text{unit}(E) \mid \text{unit}(E) \land Q \mid BU \mid U \ U \]

UnitExp
\[ UE ::= U \mid U = UE \mid UE \text{ and } UE \mid UE \text{ or } UE \mid UE \text{ implies } UE \mid \text{not } UE \]

Annotations allowed in preconditions, postconditions, assert statements, assume statements
UNITS Abstract Values

\[
\begin{align*}
\text{op } & _\wedge_ : \text{Unit } \to \text{CInt } \to \text{Unit} . \\
\text{op } & u : \text{Unit } \to \text{Value} . \\
\text{op } & \text{ptr} : \text{Location } \to \text{Value} . \\
\text{op } & \text{arr} : \text{Location } \to \text{Value} . \\
\text{op } & \text{struct} : \text{Identifier } \times \text{SFieldSet } \to \text{Value} . \\
\text{op } & \text{union} : \text{Identifier } \times \text{SFieldSet } \to \text{Value} .
\end{align*}
\]
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] \( U \ast U' = U U' \)

[2] \( U + U' = \text{mergeUnits}(U,U') \rightarrow \text{checkForFail}("+")) \)

[3] \( U > U' = \text{mergeUnits}(U,U') \rightarrow \text{checkForFail}(">") \rightarrow \text{discard} \rightarrow \text{noUnit} \)

[4] \( (\text{lvp}(L,V) = V') = V' \rightarrow \text{assign}(L) \)

[5] \( (\text{lvp}(L,U) += U') = \text{mergeUnits}(U,U') \rightarrow \text{checkForFail}("=-") \rightarrow \text{assign}(L) \)

[6] \( *(\text{lvp}(L,\text{ptr}(L'))) = \text{llookup}(L') \)

[7] \( \text{lvp}(L,\text{struct}(X', (\text{sfield}(X,L') _))) . X = \text{llookup}(L') \)
## Performance

<table>
<thead>
<tr>
<th>Test</th>
<th>LOC</th>
<th>Total Time</th>
<th>Average Per Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>x100</td>
<td>x400</td>
</tr>
<tr>
<td>straight</td>
<td>25</td>
<td>6.39</td>
<td>23.00</td>
</tr>
<tr>
<td>ann</td>
<td>27</td>
<td>8.62</td>
<td>31.27</td>
</tr>
<tr>
<td>nosplit</td>
<td>69</td>
<td>12.71</td>
<td>46.08</td>
</tr>
<tr>
<td>split</td>
<td>69</td>
<td>27.40</td>
<td>106.55</td>
</tr>
</tbody>
</table>

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

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

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.
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
Annotation-based systems widely used: Spec# (Barnett, Leino, and Schulte, CASSIS’04), JML (Burdy et.al. FMICS’03)

Precursor C-UNITS system (Feng and Roşu, ASE’03) inspiration for current work, but extremely limited
Outline

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