The C Policy Framework

Mark Hills

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

5 March 2009
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 CPF?

- Many analysis tools specific to at most several domains
- Tools that use annotations often have fixed vocabularies
Why CPF?

- Many analysis tools specific to at most several domains
- Tools that use annotations often have fixed vocabularies
- Idea: take advantage of flexibility offered by rewriting logic semantics and K to make more general-purpose, semantics-based tools
Why CPF?

- Many analysis tools specific to at most several domains
- Tools that use annotations often have fixed vocabularies
- Idea: take advantage of flexibility offered by rewriting logic semantics and K to make more general-purpose, semantics-based tools
- Starting proof of concept: Units of Measurement
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...)
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 and K
- 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, with definitions transitioning to K
- 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

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
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

```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);
}
```
Example: Annotations

typedef struct {
    double $kg atomicWeight;
    double $noUnit atomicNumber;
} Element;

//@ post(UNITS): @unit(@result) = $m ^ 2 $kg ^ -1
double radiationLength(Element * material) {
    // same as before
}

Example: Annotations

```c
$U$ double distance($U$ double x1, $U$ double y1,
              $U$ double x2, $U$ double y2) {
    return sqrt(pow(x2-x1, 2) + pow(y2-y1, 2));
}
```
Example: Annotations

```c
//@ pre(UNITS): @unit(*p) = $lb
//@ modifies: p
void lb2kg(double *p) {
    *p = ($kg)(10 * *p / 22);
    return;
}

void test(void) {
    int *p; //@ assume(UNITS): @unit(*p) = $lb
    int q = 5; //@ assume(UNITS) @unit(q) = $kg
    int r = 5; //@ assume(UNITS) @unit(r) = $lb
    lb2kg(p); q += *p; r += *p;
    return;
}
```
Example: Annotations

```c
int main(int argc, char* argv[]) {
    int x; //@ assume(UNITS): @unit(x) = $m
    int y; //@ assume(UNITS): @unit(y) = $m
    int n = 0;
    //@ invariant(UNITS): @unit(x) = @unit(y)
    while (n < 10) {
        x *= x;
        y *= y;
    }
    return 1;
}
```
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

1. Annotated C Source
2. Annotation Processor/Parser
3. Verification Tasks
4. Maude
5. Policy Checking Results
6. Core Framework Semantics
7. 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
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 \( CPF + \text{UNITS} = \text{CPF[UNITS]} \)
Unit Representation

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

Unit
\[ U ::= \@unit(E) | \@unit(E) \land Q | BU | U \land U \]

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

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

\[
\begin{align*}
\text{op } _\wedge_ &: \text{Unit } 
\text{CInt } \rightarrow \text{Unit} \\
\text{op } u &: \text{Unit } \rightarrow \text{Value} \\
\text{op } ptr &: \text{Location } \rightarrow \text{Value} \\
\text{op } arr &: \text{Location } \rightarrow \text{Value} \\
\text{op } struct &: \text{Identifier } \text{SFieldSet } \rightarrow \text{Value} \\
\text{op } union &: \text{Identifier } \text{SFieldSet } \rightarrow \text{Value} 
\end{align*}
\]
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] $(\text{lvp}(L,U) += U') = \text{mergeUnits}(U,U') \rightarrow \text{checkForFail}('=',L') \rightarrow \text{assign}(L)$

[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>x100</th>
<th>x400</th>
<th>x4000</th>
<th>x100</th>
<th>x400</th>
<th>x4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>straight</td>
<td>25</td>
<td>6.39</td>
<td>23.00</td>
<td>229.80</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>ann</td>
<td>27</td>
<td>8.62</td>
<td>31.27</td>
<td>307.54</td>
<td>0.09</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>nosplit</td>
<td>69</td>
<td>12.71</td>
<td>46.08</td>
<td>467.89</td>
<td>0.13</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td>split</td>
<td>69</td>
<td>27.40</td>
<td>106.55</td>
<td>1095.34</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</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

- Pointers considered type safe
- Formal parameters and globals (modifies/havoc)
- Aliasing (the root of all evil)
- “Full annotation” requirement
- Fresh units

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

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

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