Specifying Semantics of Programming Languages in K Framework

http://d3s.mff.cuni.cz

David Hauzar
hauzar@d3s.mff.cuni.cz
Overview

**K** k-framework

- University of Illinois at Urbana-Champaign
- Scheme, Verilog, Python, C, ...

**C** c-semantics

- An Executable Formal Semantics of C with Applications
- Chucky Ellison, Grigore Rosu
- POPL 2012

**Weverca**

- WEb VERifiCAtion for PHP
- GAUK project
K Framework - Overview

- Executable
  - Interpreter, Model checker, Deductive reasoning
  - Static analyser
- Rewriting-style semantics
  - Uses Maude engine (developed since 80’)
  - Use of OCAML (fast execution) and CoQ (verification) planned

http://k-framework.org/
### K Framework - Example

<table>
<thead>
<tr>
<th>Original language syntax</th>
<th>K Strictness</th>
<th>K Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pgm ::= var List{Id}; Stmt</td>
<td></td>
<td>(&lt;k&gt;\text{var } x_1:\text{ListItem}; s:\text{Stmt }\Rightarrow s&lt;/k&gt;)</td>
</tr>
<tr>
<td>Stmt ::= Id = AExp</td>
<td>[strict(2)]</td>
<td>(&lt;k&gt;x:\text{Id }= i:\text{Int }\Rightarrow ._&lt;/k&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(&lt;\text{state}&gt;x_1 \Rightarrow 0&lt;/text&gt;)</td>
</tr>
<tr>
<td></td>
<td>[strict(1)]</td>
<td>(&lt;k&gt;x:\text{Id }= i:\text{Int }\Rightarrow ._&lt;/k&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(&lt;\text{state}&gt;x \Rightarrow i_&lt;/text&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s_1:\text{Stmt}; s_2:\text{Stmt }\Rightarrow s_1 \Rightarrow s_2</td>
</tr>
</tbody>
</table>
|  |  | if true then \(s_1:\text{Stmt}\) else 
|  |  | else 
|  |  | if false then 
|  |  | else 
|  |  | s_2:\text{Stmt }\Rightarrow s_2  |
|  |  | \(<\text{state}>x \Rightarrow i_</text>\)  |
|  |  | \(<k>\text{spwan } s:\text{Stmt }\Rightarrow ._</k>\)  |
|  |  | \(<k>s</k>\)  |
|Stmt; Stmt |  | \(<k>x:\text{Id }= i:\text{Int }\Rightarrow ._</k>\)  |
|  |  | \(<\text{state}>x \Rightarrow i_</text>\)  |
| if BExp then Stmt else Stmt |  | \(<k>x:\text{Id }= i:\text{Int }\Rightarrow ._</k>\)  |
|  |  | \(<\text{state}>x \Rightarrow i_</text>\)  |
| while BExp do Stmt |  | \(<k>x:\text{Id }= i:\text{Int }\Rightarrow ._</k>\)  |
|  |  | \(<\text{state}>x \Rightarrow i_</text>\)  |
| spawn Stmt |  | \(<k>x:\text{Id }= i:\text{Int }\Rightarrow ._</k>\)  |
|  |  | \(<\text{state}>x \Rightarrow i_</text>\)  |
| BExp ::= Bool |  | \(<k>x:\text{Id }= i:\text{Int }\Rightarrow ._</k>\)  |
|  |  | \(<\text{state}>x \Rightarrow i_</text>\)  |
| AExp ::= Int |  | \(<k>x:\text{Id }= i:\text{Int }\Rightarrow ._</k>\)  |
|  | [strict] | \(<\text{state}>x \Rightarrow i_</text>\)  |
|  |  | i_1:\text{Int }+ i_1:\text{Int }\Rightarrow i_1 +\text{Int }i_2  |
| Id |  | \(<k>x:\text{Id }= i:\text{Int }\Rightarrow ._</k>\)  |
|  |  | \(<\text{state}>x \Rightarrow i_</text>\)  |
| AExp + AExp |  | \(<k>x:\text{Id }= i:\text{Int }\Rightarrow ._</k>\)  |
|  | [strict] | \(<\text{state}>x \Rightarrow i_</text>\)  |
|  |  | i_1:\text{Int }+ i_1:\text{Int }\Rightarrow i_1 +\text{Int }i_2  |
| KResult ::= Int |  | \(<k>x:\text{Id }= i:\text{Int }\Rightarrow ._</k>\)  |
|  |  | \(<\text{state}>x \Rightarrow i_</text>\)  |
|  |  | i_1:\text{Int }+ i_1:\text{Int }\Rightarrow i_1 +\text{Int }i_2  |
| Boolean |  | \(<k>x:\text{Id }= i:\text{Int }\Rightarrow ._</k>\)  |
|  |  | \(<\text{state}>x \Rightarrow i_</text>\)  |
|  |  | i_1:\text{Int }+ i_1:\text{Int }\Rightarrow i_1 +\text{Int }i_2  |

Configuration ≡ \(<k>*K</k>*\)

\(<\text{state}>\text{Map}\{\text{Id }\Rightarrow \text{Int}\}</text>\)
Conforming
- Must accept all portable programs, but can also accept non portable programs

Freestanding
- All language features except complex numbers
- Subset of the standard library

Extensively tested
- E.g. against GCC torture tests (1093 test programs, 776 standard compliant. Of those 770 passed)
- Better results than Clang or GCC

http://code.google.com/p/c-semantics/
# C Semantics – Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>GH</th>
<th>CCR</th>
<th>CR</th>
<th>No</th>
<th>Pa</th>
<th>BL</th>
<th>Le</th>
<th>ER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitfields</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Enums</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Floats</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Struct/Union</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Struct as Value</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Bitwise</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Casts</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Functions</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Exp. Side Effects</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Variadic Funcs.</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Eval. Strategies</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Concurrency</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Break/Continue</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Goto</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Switch</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Longjmp</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Malloc</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
</tbody>
</table>

- ●: Fully Described
- ○: Partially Described
- ○: Not Described

GH denotes Gurevich and Huggins (1993), CCR is Cook, Cohen, and Redmond (1994), CR is Cook and Subramanian (1994), No is Norrish (1998), Pa is Papaspyrou (2001), BL is Blazy and Leroy (2009), Le is Leroy (unpublished, 2010), and ER is Ellison and Roşu (our work).
C Semantics – Implementation

- 75 cells
- 150 syntactic operators
- 5900 source lines of semantics
- 1200 rules
  - 80 rules for statements
  - 160 rules for expressions
  - 500 rules for declarations and types
  - 115 rules for standard library
  - ...
- 6 person-months
Existing C analysis/verification tools
- Lint/Purify/Coverity/Valgrind
- Blast
- Havoc
- Slam
- VCC
- ...

Based on approximative models of C
- Hard to argue for the soundness of the tools
int main(void) {
    int x = 0;
    return (x = 1) + (x = 2);
}

Undefined according to C standard

GCC4, MSVC   returns 4
GCC3, ICC, Clang   returns 3

Frama-C „proves“ it returns 4
int main(void) {
   "foo"[0] = 'x';
   return "foo"[0];
}

Undefined according to C standard

GCC4 doesn’t compile
ICC, Clang segmentation fault
Frama-C „proves“ it returns ’x’
int r;

int f(int x) {
  return (r = x);
}

int main(void) {
  f(1) + f(2); return r;
}

Defined (Could return 1 or 2)

GCC, ICC, MSVC, Clang returns 2

Both Frama-C and Havoc „prove“ it can only return 2
C Semantics – Built tools

Use an explicit and testable definition to build tools that conform to this semantics

Semantics-Based Analysis Tools

- Interpreter
- State-space explorer
- LTL Model-checker
- Debugger
- Program verifier
C Semantics – Interpreter

```c
#include <string.h>
int main(void) {
    char dest[5], src[5] = "hello";
    strcpy(dest, src);
}
```

ERROR! KCC encountered an error while executing this program.
Description: Reading outside the bounds of an object.
File: buggy_strcpy.c
Function: strcpy
Line: 4
```c
int denominator = 5;
int setDenominator(int d) {
    return denominator = d;
}
int main(void) {
    return setDenominator(0) + (7 / denominator);
}
```

2 solutions found
----------------------------------
Solution 1
Program got stuck
File: eval_order.c
Line: 8
Description: Division by 0.
----------------------------------
Solution 2
Program completed successfully
Return value: 1
C Semantics – LTL-Based Model Checking

typedef enum {green, yellow, red} state;
state lightC = green; state lightW = red;
int changeC() {
    switch (lightC) {
        case(green): lightC = yellow; return 0;
        case(yellow): lightC = red; return 0;
        case(red):
            if (lightW == red) { lightC = green; } return 0;
    }
...
int main(void) { while(1) { changeC() + changeW(); } }

#pragma __ltl safety: [] (lightC == red \ lightW == red)
#pragma __ltl progressC: [] <> (lightC == green)
void listNode* reverse(struct listNode *x) {
    struct listNode *p;
    struct listNode *y;
    p = 0;
    while (x) {
        y = x->next;
        x->next = p;
        p = x;
        x = y;
    }
}

rule <k> $ => return p; </k>
<heap>... list(x,A) => list(p,rev(A)) ...

$ is the body of the function
Weverca
Security Analysis of PHP applications
  - Flow of sensitive data into critical commands
Weverca – Overview

For each program point:
The set of possible values for each variable
The set of possible types for each variable
The taint and sanitization status for each variable
The set of conditions defined on the program’s variables

Static analysis

For each program point:
The set of possible values for each variable
The set of possible types for each variable
The taint and sanitization status for each variable
The set of conditions defined on the program’s variables

Visualisation of static analysis results

Information for a developer

PHP source codes

PHP parser

Abstract syntax tree

Hints for static analysis

Vulnerability analysis

SQL injection
XSS
Sensitive information leakage
Semantic URL
Spoofed form submissions
Spoofed HTTP request
CSRF
Session fixation
Session hijacking

Visualisation of vulnerability analysis results

Information for a developer:
flow of vulnerable data to critical commands
vulnerable values of variables
vulnerable data dependencies

Other analyses:
Refactoring
Type-safety analysis
Weverca – Work in Progress

For each program point:
The set of possible values for each variable
The set of types for each variable
The taint and sanitization status for each variable
The set of conditions defined on the program's variables

Other analyses:
Refactoring
Type-safety analysis

Visualisation of static analysis results

Information for a developer:
Flow of vulnerable data to critical commands
Vulnerable values of variables
Vulnerable data dependencies

SQL injection
XSS
Sensitive information leakage
Semantic URL
Spoofed form submissions
Spoofed HTTP request
CSRF
Session fixation
Session hijacking

Information for a developer:

Vulnerability analysis

Visualisation of vulnerability analysis results

PHP source codes

PHP parser

Abstract syntax tree

Static analysis

Hints for static analysis
Summary

**K** k-framework
- Defining semantics of programming languages

**C** c-semantics
- Semantics-based analysis tools

**Weverca**
- Static analyzer in K