An Executable Formal Semantics of C with Applications

Chucky Ellison    Grigore Roșu

Department of Computer Science
University of Illinois

POPL’12    January 27, 2012
1 Introduction
   - Introduction
   - Motivation

2 Semantics of C

3 Semantics-Based Analysis Tools
   - Interpreter
   - State-space Search
   - Model Checker
There is no formal semantics for C.
There is no formal semantics for C.

was
There are partial semantics

- *Gurevich and Huggins* (1993) [ASM]
- *Cook, Cohen, and Redmond* (1994) [Denotational]
- *Cook and Subramanian* (1994) [Denotational]
- *Black* (1998) [Axiomatic]
- *Papaspyrou* (2001) [Denotational]
- *Blazy and Leroy* (2009) [Big-step SOS]

But, they simplify or leave out large parts of the language: Nondeterminism, casts, bitfields, unions, struct values, variadic functions, memory alignment, goto, dynamic memory allocation (`malloc()`), ...
But, Previous Definitions Leave out 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>
</tr>
</thead>
<tbody>
<tr>
<td>Bitfields</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>
</tr>
<tr>
<td>Floats</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>
</tr>
<tr>
<td>Struct as Value</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>
</tr>
<tr>
<td>Bitwise</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>
</tr>
<tr>
<td>Functions</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>
</tr>
<tr>
<td>Variadic Funcs.</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>
</tr>
<tr>
<td>Concurrency</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>
</tr>
<tr>
<td>Goto</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>
</tr>
<tr>
<td>Longjmp</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>
</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), and Le is Leroy (unpublished, 2010).
No Semantics-Based Tools Either

There are many useful C analysis/verification tools, including:

- Lint/Purify/Coverity/Valgrind
- Blast
- Havoc
- Slam
- VCC
- Frama-C/Caduceus
- ...
No Semantics-Based Tools Either

There are many **useful** C analysis/verification tools, including:
- Lint/Purify/Coverity/Valgrind
- Blast
- Havoc
- Slam
- VCC
- Frama-C/Caduceus
- ...

These tools are based on **approximative models** of C.
- Most tools are not even based on an *incomplete* semantics
- Hard to argue for the soundness of the tools
Our Contribution

1. A complete formal semantics for C;

POPL'12, ACM, pp 533-544. 2012
Our Contribution

1. A complete formal semantics for C;
2. Semantics-based analysis tools for C;
Our Contribution

1. A complete formal semantics for C;
2. Semantics-based analysis tools for C;
3. Constructive evidence that rewriting-based semantics scale.
Outline

1 Introduction
   - Introduction
   - Motivation

2 Semantics of C

3 Semantics-Based Analysis Tools
   - Interpreter
   - State-space Search
   - Model Checker

POPL'12, ACM, pp 533-544. 2012

An Executable Formal Semantics of C with Applications
C Specifications

- The C Programming Language (K&R) (1978)
- ANSI C (1989)
  - 540 pp.
  - 62 person-years of work (from 1995–1999)
  - Work continued until 2007
  - About 50 new features over C90, and many fixes
  - 683 pp.
  - Adds first support for concurrency
Do We Really Need Formal Analysis Tools?

**Question.**
What happens when the approximative models of C fall short?

**Answer.**
Bad programs get proved correct, or behaviors go missing.
undefined behavior  Behavior, upon use of a non-portable or erroneous program construct or of erroneous data, [with] no requirements. [C11, §3.4.3:1]

- In essence, this refers to problematic situations that are hard to identify statically or expensive to identify dynamically
- Implementations can do *anything* for undefined behavior, including failing to compile, crashing, or appearing to work
C has over 200 explicitly undefined kinds of behaviors.

- Division by zero
- Referring to an object outside its lifetime
- Signed overflow
- ...

Undefined Behaviors are Fundamental to C
Two Unsequenced Writes to ‘x’

```c
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

Both Frama-C (Jessie plugin) and Havoc “prove” it returns 4
int main(void) {
    "foo"[0] = 'x';
    return "foo"[0];
}

**Undefined** according to C standard

- GCC: doesn’t compile
- ICC, Clang: segmentation fault
- MSVC: returns ‘f’

Frama-C (Jessie plugin) “proves” it returns ‘x’
Valid Nondeterminism

```c
int r;

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

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

Defined (Could return 1 or 2)

- GCC, ICC, MSVC, Clang: returns 2
- Both Frama-C (Jessie plugin) and Havoc “prove” it can only return 2
We are \emph{not} saying that these analysis tools are bad!

However, it is hard to argue for soundness without a semantics.

Instead of embedding different models of C in every tool, we need:

- An explicit and testable definition of C
- To build tools that conform to this semantics
Outline

1. Introduction
   - Introduction
   - Motivation

2. Semantics of C

3. Semantics-Based Analysis Tools
   - Interpreter
   - State-space Search
   - Model Checker
A Complete Definition of C

We have the first arguably complete formal definition of a conforming freestanding implementation of C.
A Complete Definition of C

We have the first arguably complete formal definition of a conforming freestanding implementation of C.

Conforming Must accept all portable programs, but can also accept non-portable programs.

[C11, §4:6]
We have the first arguably complete formal definition of a conforming freestanding implementation of C.

**Conforming** Must accept all portable programs, but can also accept non-portable programs.

**Freestanding** All language features except complex (i.e., imaginary) numbers, and only a subset of the standard library.

[C11, §4:6]
A Complete Definition of C

We have the first arguably complete formal definition of a conforming freestanding implementation of C.

**Conforming** Must accept all portable programs, but can also accept non-portable programs.

**Freestanding** All language features except complex (i.e., imaginary) numbers, and only a subset of the standard library. It includes only `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, and `<stdint.h>`.

[C11, §4:6]
Tested against the GCC torture tests:
- Of 1093 test programs, 776 appear to be standards compliant.
  Of those, we pass 770 (>99%).
- Better results than Clang or GCC itself; one fewer than ICC.

Tested against test suites of other compilers (Clang, LCC, etc.)

Tested against thousands of programs generated by Csmith
## Our Work is More Complete

<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 Roșu* (our work).
Some Information about Our Semantics

Mechanized in the \(\mathbb{K}\) Framework (http://k-framework.org/)

- Rewriting-style semantics
- Syntax, configuration, rewrite rules
Example Rules

\( V : T \) is a value \( V \) with type \( T \).

\([V] : T\) is an l-value \( V \) with type \( T \).

\[\text{lookup} \]

\[
\begin{array}{c}
\text{k} \\
X \\
\hline
L : T
\end{array}
\]

\( \text{env} \) \( X \rightarrow L \)

\( \text{types} \) \( X \rightarrow T \)

\[\text{ref} \]

\[
\begin{array}{c}
\text{k} \\
&([L] : T) \\
\hline
L : \text{pointerType}(T)
\end{array}
\]

\[\text{deref} \]

\[
\begin{array}{c}
\text{k} \\
*(L : \text{pointerType}(T)) \\
\hline
[L] : T
\end{array}
\]

(This rule is unsafe; see paper for details.)
Some Information about Our Semantics

- 150 syntactic operators
- 5900 source lines of semantics
- 1200 different \( K \) rules
  - Only 80 rules for statements
  - Only 160 for expressions
  - 500 rules for declarations and types!
Outline

1. Introduction
   - Introduction
   - Motivation

2. Semantics of C

3. Semantics-Based Analysis Tools
   - Interpreter
   - State-space Search
   - Model Checker
These tools are provided “for free” by rewriting logic and $\mathbb{K}$:

- Interpreter
- State-space explorer
- LTL Model-checker
- Debugger
- Program verifier (via Matching Logic)
Outline

1 Introduction
   - Introduction
   - Motivation

2 Semantics of C

3 Semantics-Based Analysis Tools
   - Interpreter
   - State-space Search
   - Model Checker
$ cat hello_world.c

#include <stdio.h>
int main(void) {
    printf("Hello world!\n");
}

$ kcc hello_world.c
$ ./a.out
Hello world!
Normal Interpretation

$ cat hello_world.c

#include <stdio.h>
int main(void) {
    printf("Hello world!\n");
}

$ kcc hello_world.c
$ ./a.out

Hello world!
$ cat buggy_strcpy.c

#include <string.h>

int main(void) {
    char dest[5], src[5] = "hello";
    strcpy(dest, src);
}

$ kcc buggy_strcpy.c
$ ./a.out
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
Interpretation to Find Bugs

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

```
$ kcc buggy_strcpy.c
$ ./a.out
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
```
This generated interpreter has been used in automated testcase reduction (Regehr, et al.)

- It’s fast enough to be useful
- Catches bugs that other tools (e.g., Valgrind) do not
- No spurious errors
Search to Find Bugs

```c
$ cat eval_order.c

int denominator = 5;

int setDenominator(int d) {
    return denominator = d;
}

int main(void) {
    return setDenominator(0) + (7 / denominator);
}
```
$ cat eval_order.c

int denominator = 5;

int setDenominator(int d) {
    return denominator = d;
}

int main(void) {
    return setDenominator(0) + (7 / denominator);
}

$ kcc eval_order.c
$ SEARCH=1 ./a.out
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
Search to Find Bugs (Cont.)
Search to Find Bugs (Cont.)

$ clang -O0 eval_order.c && ./a.out
Floating point exception
$ clang -O2 eval_order.c && ./a.out
$
Search to Explore Nondeterminism

```c
$ cat nondet.c

int r;

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

int main(void) {
    return f(1) + f(2), r;
}
```
$ cat nondet.c

```c
int r;

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

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

$ kcc nondet.c

$ SEARCH=1 ./a.out
Search to Explore Nondeterminism (Cont.)

2 solutions found

------------------------------------------------------------------------------------------------------------------
Solution 1
Program completed successfully
Return value: 1
------------------------------------------------------------------------------------------------------------------
Solution 2
Program completed successfully
Return value: 2
Outline

1 Introduction
   • Introduction
   • Motivation

2 Semantics of C

3 Semantics-Based Analysis Tools
   • Interpreter
   • State-space Search
   • Model Checker

POPL'12, ACM, pp 533-544. 2012
LTL-Based Model Checking

$ cat lights.c

typedef enum {green, yellow, red} state;
state lightNS = green; state lightEW = red;
int changeNS() {
    switch (lightNS) {
        case(green): lightNS = yellow; return 0;
        case(yellow): lightNS = red; return 0;
        case(red):
            if (lightEW == red) { lightNS = green; } return 0;
    }
}

... 
int main(void) { while(1) { changeNS() + changeEW(); } }
$ cat lights.c

typedef enum {green, yellow, red} state;
state lightNS = green; state lightEW = red;
int changeNS() {
    switch (lightNS) {
        case(green): lightNS = yellow; return 0;
        case(yellow): lightNS = red; return 0;
        case(red):
            if (lightEW == red) { lightNS = green; } return 0;
    }
}
...
int main(void) { while(1) { changeNS() + changeEW(); } }

#pragma __ltl safety: [] (lightNS == red \ lightEW == red)
#pragma __ltl progressNS: [] <> (lightNS == green)
LTL-Based Model Checking (Cont.)

$ kcc lights.c
$ MODELCHECK=safety ./a.out
LTL-Based Model Checking (Cont.)

$ kcc lights.c
$ MODELCHECK=safety ./a.out

False! The `safety' property does not hold.
LTL-Based Model Checking (Cont.)

$ kcc lights.c
$ MODELCHECK=safety ./a.out

False! The `safety' property does not hold.

# change "changeNS() + changeEW()" to "changeNS(); changeEW()"
LTL-Based Model Checking (Cont.)

```
$ kcc lights.c
$ MODELCHECK=safety ./a.out

False! The `safety' property does not hold.

# change "changeNS() + changeEW()" to "changeNS(); changeEW()"

$ kcc lights.c
$ MODELCHECK=safety ./a.out
```
LTL-Based Model Checking (Cont.)

$ kcc lights.c
$ MODELCHECK=safety ./a.out

False! The `safety' property does not hold.

# change "changeNS() + changeEW()" to "changeNS(); changeEW()"

$ kcc lights.c
$ MODELCHECK=safety ./a.out

True! The `safety' property holds.
$ kcc lights.c
$ MODELCHECK=safety ./a.out

False! The `safety' property does not hold.

# change "changeNS() + changeEW()" to "changeNS(); changeEW()"

$ kcc lights.c
$ MODELCHECK=safety ./a.out

True! The `safety' property holds.

$ MODELCHECK=progressNS ./a.out
LTL-Based Model Checking (Cont.)

```bash
$ kcc lights.c
$ MODELCHECK=safety ./a.out

False! The `safety' property does not hold.

# change "changeNS() + changeEW()" to "changeNS(); changeEW()"

$ kcc lights.c
$ MODELCHECK=safety ./a.out

True! The `safety' property holds.

$ MODELCHECK=progressNS ./a.out

True! The `progressNS' property holds.
```
We have the first arguably complete formal semantics of C

- Is executable, and has been thoroughly tested against the GCC torture test suite
- Can be used to generate analysis tools
- Demonstrates that rewriting-based semantics can handle large languages and all their gritty details
- Available at http://c-semantics.googlecode.com/