An Executable Formal Semantics of C with Applications

Chucky Ellison

University of Illinois

PhD Defense June 28, 2012

Thesis advisor: Grigore Roșu
Committee members: Gul Agha
José Meseguer
Wolfram Schulte

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]
- *Norrish* (1998) [Small- and big-step SOS]
- *Black* (1998) [Axiomatic]
- *Papaspyrou* (2001) [Denotational]
- *Blazy and Leroy* (2009) [Big-step SOS]
- *Leroy* (2010) [Small-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

	Definition						
Feature	GH	CCR	CR	No	Pa	BL	Le
Bitfields	●	◐	○	○	◐	○	○
Enums	◐	●	○	○	●	○	○
Floats	○	○	○	○	◐	●	●
Struct/Union	●	●	●	◐	●	●	●
Struct as Value	○	○	○	●	○	○	○
Arithmetic	◐	●	●	○	●	●	●
Bitwise	○	●	○	○	●	●	●
Casts	◐	◐	○	◐	◐	●	●
Functions	●	●	◐	●	●	●	●
Exp. Side Effects	●	●	○	●	●	○	●
Variadic Funcs.	○	○	○	○	○	○	○
Eval. Strategies	○	◐	○	●	●	○	●
Concurrency	○	○	○	○	○	○	○
Break/Continue	◐	●	◐	●	●	●	●
Goto	◐	○	○	○	●	○	●
Switch	◐	●	○	○	●	◐	◐
Longjmp	○	○	○	○	○	○	○
Malloc	○	○	○	○	○	○	○

●: 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Le is *Leroy* (unpublished, 2010).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 Contributions

1. A formal semantics for C in the $\mathbb{K}$ Framework;

Our Contributions

1. A formal semantics for C in the $\mathbb{K}$ Framework;
    1. Complete *positive* semantics (semantics of correct programs)

Our Contributions

1. A formal semantics for C in the $\mathbb{K}$ Framework;
   1. Complete *positive* semantics (semantics of correct programs)
   2. Large subset of *negative* semantics (being able to identify incorrect programs)

Our Contributions

1. A formal semantics for C in the $\mathbb{K}$ Framework;
    1. Complete *positive* semantics (semantics of correct programs)
    2. Large subset of *negative* semantics (being able to identify incorrect programs)
2. Semantics-based analysis tools for C;

Our Contributions

1. A formal semantics for C in the $\mathbb{K}$ Framework;
    1. Complete *positive* semantics (semantics of correct programs)
    2. Large subset of *negative* semantics (being able to identify incorrect programs)
2. Semantics-based analysis tools for C;
3. Test suite for analysis tools and compilers;

Our Contributions

1. A formal semantics for C in the $\mathbb{K}$ Framework;
    1. Complete *positive* semantics (semantics of correct programs)
    2. Large subset of *negative* semantics (being able to identify incorrect programs)
2. Semantics-based analysis tools for C;
3. Test suite for analysis tools and compilers;
4. Constructive evidence that rewriting-based semantics scale.

My Work

Work on $\mathbb{K}$:

- [Ellison, Șerbănuță, Roșu; WADT'08]
- [Ilseman, Ellison, Roșu; TR'10]
- [Șerbănuță, Arusoaie, Lazar, Ellison, Lucanu, Roșu; K'11]
- [Arusoaie, Șerbănuță, Ellison, Roșu; WRLA'12]
- [Lazar, Arusoaia, Șerbănuță, Ellison, Mereuta, Lucanu, Roșu; FM'12]

Work on C:

- [Roșu, Ellison, Schulte; AMAST'10]
- **[Ellison, Roșu; POPL'12]**
- **[Regehr, Chen, Cuoq, Eide, Ellison, Yang; PLDI'12]**
- **[Ellison, Roșu; Submitted]**

Outline

C Specifications

- The C Programming Language (K&R) (1978)
- ANSI C (1989)
- ISO/IEC 9899:1990 "C90"
- ISO/IEC 9899:1999 "C99"
    - 540 pp.
    - 62 person-years of work (from 1995–1999)
    - Work continued until 2007
    - About 50 new features over C90, and many fixes
- ISO/IEC 9899:2011 "C11"
    - 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.

What are “Bad” Programs?

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

Undefined Behaviors are Fundamental to C

C has over 200 explicitly undefined kinds of behaviors.

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

Two Unsequenced Writes to 'x'

```
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

Write to String Literal

```
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

```
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

Semantics-Based Analysis Tools

We are *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

Outline

Positive Semantics
[Ellison, Roșu, POPL'12]

Positive Semantics

When one thinks of formal semantics, one typically thinks of *positive* semantics. That is, the semantics of defined programs.

A positive semantics enables:

- Program interpretation
- Program debugging
- Program behavior exploration
- Deadlock/Livelock detection

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]

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.

[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]

Extensively Tested Definition

- 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

	Definition							
Feature	GH	CCR	CR	No	Pa	BL	Le	ER
Bitfields	●	◐	○	○	◐	○	○	●
Enums	◐	●	○	○	●	○	○	●
Floats	○	○	○	○	◐	●	●	●
Struct/Union	●	●	●	◐	●	●	●	●
Struct as Value	○	○	○	●	○	○	○	●
Arithmetic	◐	●	●	○	●	●	●	●
Bitwise	○	●	○	○	●	●	●	●
Casts	◐	◐	○	◐	◐	●	●	●
Functions	●	●	◐	●	●	●	●	●
Exp. Side Effects	●	●	○	●	●	○	●	●
Variadic Funcs.	○	○	○	○	○	○	○	●
Eval. Strategies	○	◐	○	●	●	○	●	●
Concurrency	○	○	○	○	○	○	○	◐
Break/Continue	◐	●	◐	●	●	●	●	●
Goto	◐	○	○	○	●	○	●	●
Switch	◐	●	○	○	●	◐	◐	●
Longjmp	○	○	○	○	○	○	○	●
Malloc	○	○	○	○	○	○	○	●

●: Fully Described
◐: Partially Described
○: Not Described

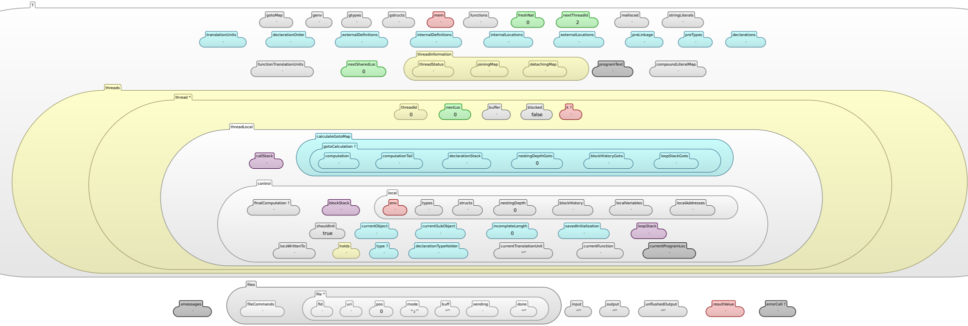
GH denotes *Gurevich and Huggins* (1993),
CCR is *Cook, Cohen, and Redmond* (1994),
CR is *Cook and Subramanian* (1994),
No is *Norrish* (1998),
Pa is *Papaspyrou* (2001),
BL is *Blazy and Leroy* (2009),
Le is *Leroy* (unpublished, 2010), and
ER is *Ellison and Roșu* (our work).

Some Information about Our Semantics

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

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

C's $\mathbb{K}$ Configuration



A $\mathbb{K}$ configuration is a nested tuple representing the state of a running program.

Some Information about Our Semantics

- 150 syntactic operators
- 5900 source lines of semantics
- 1200 different $\mathbb{K}$ rules
  - Only 80 rules for statements
  - Only 160 for expressions
  - 500 rules for declarations and types!

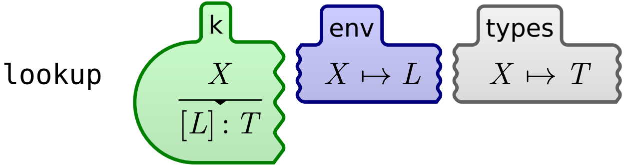
Variable Lookup

C11, §6.5.1:2

An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue)...

C11, §6.3.2.1:1

An lvalue is an expression (with an object type other than `void`) that potentially designates an object...



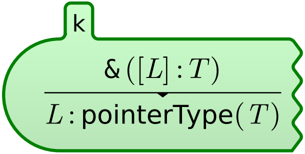
$[L] : T$ is an lvalue L with type T.

Reference

C11, §6.5.3.2:3

The unary & operator yields the address of its operand. If the operand has type “type”, the result has type “pointer to type”. . . [The] result is a pointer to the object or function designated by its operand.

ref



$L : T$ is a value L with type T.

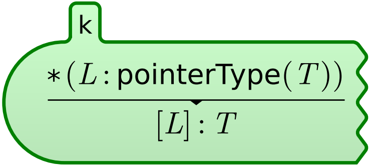
$[L] : T$ is an lvalue L with type T.

Dereference

C11, §6.5.3.2:4

The unary $*$ operator denotes indirection. If the operand . . . points to an object, the result is an lvalue designating the object. If the operand has type "pointer to type", the result has type "type".

deref



$L : T$ is a value L with type T.

$[L] : T$ is an lvalue L with type T.

Why does this work?

C99, §6.3.2.1:2

Except when it is the operand of the sizeof operator, the unary & operator, the ++ operator, the -- operator, or the left operand of the . operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called lvalue conversion. . . .

Outline

1. Introduction
   - Introduction
   - Motivation
2. Semantics of C
   - Positive Semantics
   - Negative Semantics
3. Semantics-Based Analysis Tools
   - Interpreter
   - State-space Search
   - Model Checker
4. Conclusion

Negative Semantics
[Ellison, Roșu; Submitted];
[Regehr, Chen, Cuoq, Eide, Ellison, Yang; PLDI'12]

Negative Semantics

The *negative* semantics of a language are the rules that can identify undefined behaviors. Such behaviors need attention in practice (as we shall see).

A negative semantics enables:

- Memory safety checker
- Race detector
- I.e., undefined behavior detection

Real World Application

Our tool has been used in automated testcase reduction [Regehr, Chen, Cuoq, Eide, Ellison, Yang; PLDI'12]

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

Techniques for Capturing Undefined Behavior

We use these techniques to identify undefined behavior:

- Side conditions
- Storing additional information
- Symbolic behavior

As we will see, they are strongly related to one another.

Side Conditions

A plain, unsafe rule...

deref

k

$$\frac{*(L : \text{pointerType}(T))}{[L] : T}$$

Side Conditions

A plain, unsafe rule. . .

deref — k: $\dfrac{*(L : \text{pointerType}(T))}{[L] : T}$

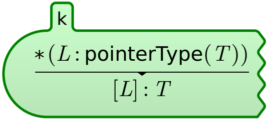
can be made safer with side conditions for type. . .

deref' — k: $\dfrac{*(L : \text{pointerType}(T))}{[L] : T}$ when $T \neq$ void

Side Conditions

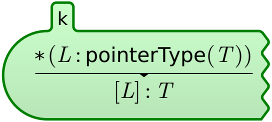
A plain, unsafe rule...

deref



can be made safer with side conditions for type...

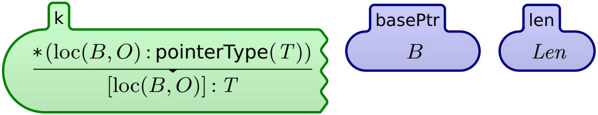
deref'



when $T \neq$ void

and for position...

deref''



when $O < Len \wedge T \neq$ void

Side Conditions (Cont.)

We used side conditions to avoid defining many kinds of undefined behavior:

- Safe dereferencing (previous example)
- Division by zero (when $denom \neq 0$)
- Zero length objects (when $n \geq 1$)
- Arithmetic overflow (when $sum \leq \texttt{INT\_MAX}$)
- Data races (when $\neg$overlaps($write_1$, $write_2$))
- ...

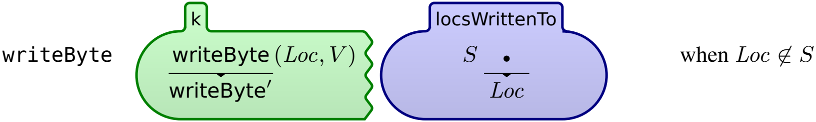
Storing Additional Information

```
int main(void) {
   int x = 0;
   return (x = 1) + (x = 2); // undefined!
}
```

Storing Additional Information

```
int main(void) {
   int x = 0;
   return (x = 1) + (x = 2); // undefined!
}
```

To detect unsequenced reads/writes, we start keeping track of writes:

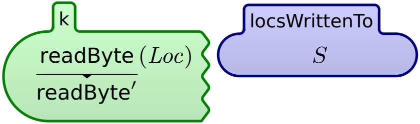


Storing Additional Information

```
int main(void) {
   int x = 0;
   return (x = 1) + (x = 2); // undefined!
}
```

We also check that reads are not of previously written to locations:

readByte

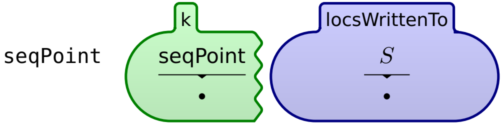


when $Loc \notin S$

Storing Additional Information

```
int main(void) {
   int x = 0;
   return (x = 1) + (x = 2); // undefined!
}
```

We empty the locsWrittenTo cell at every sequence point:



Storing Additional Information (Cont.)

We stored additional information to avoid defining many kinds of undefined behavior:

- Unsequenced reads and writes
- Modifying `const` objects
- Unlocking the mutexes of other threads
- Declaring a variable twice per scope
- ...

Symbolic Behavior

```
int main(void) {
   int a, b;
   if (&a < &b) { ... } // undefined!
}
```

Symbolic Behavior

```
int main(void) {
   int a, b;
   if (&a < &b) { ... } // undefined!
}
```

```
int main(void) {
   int a[4];
   int b[4];
   a[6] = 17; // undefined!
}
```

Symbolic Behavior

```
int main(void) {
   int a, b;
   if (&a < &b) { ... } // undefined!
}
```

```
int main(void) {
   int a[4];
   int b[4];
   a[6] = 17; // undefined!
}
```

- Memory allocation order is unspecified in C
- Any particular allocation scheme would allow programs to run that aren't portable
- Objects are self-contained and one must read/write in-bounds
- Therefore, we make memory symbolic.

Symbolic Behavior (Cont.)

We use $\mathrm{loc}(\mathit{Base}, \mathit{Offset})$ for a location: *Offset* byte of the *Base* object.

Symbolic Behavior (Cont.)

We use $\mathrm{loc}(\mathit{Base}, \mathit{Offset})$ for a location: *Offset* byte of the *Base* object.

This allows us to handle both equality and relational operators properly:

- All bytes in the same object share a base (and are comparable)
- Bytes of different objects have different bases (and are *not* comparable)

Symbolic Behavior (Cont.)

We use $\mathrm{loc}(\mathit{Base}, \mathit{Offset})$ for a location: *Offset* byte of the *Base* object.

This allows us to handle both equality and relational operators properly:

- All bytes in the same object share a base (and are comparable)
- Bytes of different objects have different bases (and are *not* comparable)

The same mechanism also ensures memory safety:

- Bytes can only be read from or written to a single object (cannot read or write out of bounds)

Symbolic Behavior (Cont.)

We used symbolic behavior to avoid defining many kinds of undefined behavior:

- Writing/reading out of bounds
- Comparisons between incomparable objects
- Improperly storing pointers in memory
- Using intedeterminate memory (e.g., uninitialized variables)
- ...

Undefined Behavior Test Suites

How do we evaluate our negative semantics?

- No existing test suite of undefined behavior
- We decided to make our own
    1. Take an existing static analysis tool test suite and extract only the undefined programs
    2. Write our own completely from scratch

Extraction from Juliet Test Suite

We extracted undefined tests from Juliet test suite (by NIST):

- Originally over 45,000 tests for secure programming
- We identified 4,113 undefined tests
- $\sim$96 SLOC per test (179 SLOC linking the helper-library)
- V. Analysis and our tool were improved with feedback

Extraction from Juliet Test Suite

We extracted undefined tests from Juliet test suite (by NIST):

- Originally over 45,000 tests for secure programming
- We identified 4,113 undefined tests
- $\sim$96 SLOC per test (179 SLOC linking the helper-library)
- V. Analysis and our tool were improved with feedback

		Tools (% passed)			
Undefined Behavior	No. Tests	Valgrind	CheckPointer	V. Analysis	Our Tool
Use of invalid pointer	3193	70.9	89.1	100.0	100.0
Division by zero	77	0.0	0.0	100.0	100.0
Bad argument to `free()`	334	100.0	99.7	100.0	100.0
Uninitialized memory	422	100.0	29.3	100.0	100.0
Bad function call	46	100.0	100.0	100.0	100.0
Integer overflow	41	0.0	0.0	100.0	100.0

Hand Written Test Suite

We also handwrote our own test suite:

- One undefined behavior per test
- Tests come in pairs: one undefined, one defined; makes sure tools are identifying real bugs
- 178 tests covering 70 undefined behaviors

Hand Written Test Suite

We also handwrote our own test suite:

- One undefined behavior per test
- Tests come in pairs: one undefined, one defined; makes sure tools are identifying real bugs
- 178 tests covering 70 undefined behaviors

Tools	Static (% Passed)	Dynamic (% Passed)
Valgrind	0.0	2.3
CheckPtr.	2.4	13.1
V. Analysis	1.6	45.3
Our Tool	44.8	64.0

Conclusions from Testing

- No tool was able to catch behaviors accurately unless they specifically focused on those behaviors

Conclusions from Testing

- No tool was able to catch behaviors accurately unless they specifically focused on those behaviors
- Undefinedness checking does not simply come for free

Conclusions from Testing

- No tool was able to catch behaviors accurately unless they specifically focused on those behaviors
- Undefinedness checking does not simply come for free
- Tools were able to improve performance by looking at concrete failing tests and adapting their techniques

Conclusions from Testing

- No tool was able to catch behaviors accurately unless they specifically focused on those behaviors
- Undefinedness checking does not simply come for free
- Tools were able to improve performance by looking at concrete failing tests and adapting their techniques
- Semantics-based analysis of undefinedness works at least as well or better than popular tools

Outline

Semantics-Based Analysis Tools

These tools are provided “for free” by rewriting logic and $\mathbb{K}$:

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

Outline

Normal Interpretation

```
$ cat hello_world.c

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

Normal Interpretation

```
$ 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!
```

```
$ gcc hello_world.c
$ ./a.out

Hello world!
```

Interpretation to Find Bugs

```
$ cat buggy_strcpy.c

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

Interpretation to Find Bugs

```
$ 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
```

Outline

Search to Find Bugs

```
$ cat eval_order.c

int denominator = 5;

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

Search to Find Bugs (Cont.)

```
$ clang -O0 eval_order.c && ./a.out
Floating point exception
$ clang -O2 eval_order.c && ./a.out
$
```

Search to Find Bugs (Cont.)

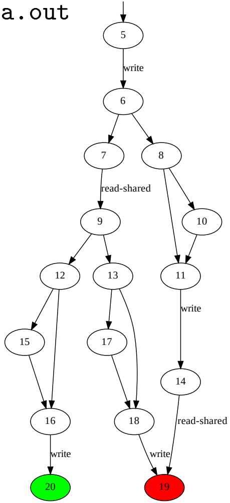
```
$ cat eval_order.c

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
```

Search to Find Bugs (Cont.)

```
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.)

```
$ GRAPH=1 SEARCH=1 ./a.out
```



Search to Explore Nondeterminism

```
$ cat nondet.c

int r;

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

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

Search to Explore Nondeterminism

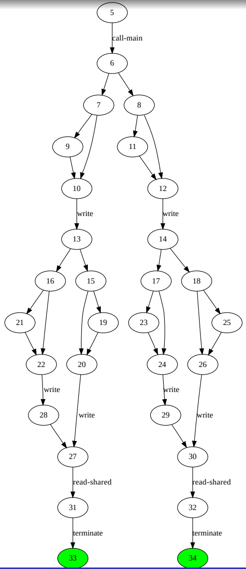
```
$ cat nondet.c

int r;

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

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

$ kcc nondet.c
$ GRAPH=1 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
```

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
   - Positive Semantics
   - Negative Semantics
3. Semantics-Based Analysis Tools
   - Interpreter
   - State-space Search
   - Model Checker
4. Conclusion

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(); } }
```

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(); } }

#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.
```

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.

$ MODELCHECK=progressNS ./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.

$ MODELCHECK=progressNS ./a.out

True! The `progressNS' property holds.
```

Outline

Future Work

Our work serves as a strong foundation for potential extensions:

- Semantics
  - C11 features (`_Alignas`, `_Atomic`, etc.)
  - Still some negative-semantics features left to address (e.g., strict aliasing)
  - Extensions to C (GCC, Apple blocks, CUDA, etc.)
- Tools
  - Extending MatchC to work with full C semantics
  - Abstracting state-space for search and model checking

Summary

We have the first arguably complete formal semantics of C

- Is executable, and has been thoroughly tested against the GCC torture test suite
- Focuses on both positive and negative semantics
- 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/`