The K Framework and a Formal Semantics of C

Chucky Ellison Traian Florin Șerbănuță Grigore Roșu

University of Illinois

MVD'10 September 17, 2010

What is K?

- K is an executable semantic framework for defining programming languages, calculi, type systems, and formal analysis tools

What is K?

- K is an executable semantic framework for defining programming languages, calculi, type systems, and formal analysis tools
- It has been used to define:

What is K?

- K is an executable semantic framework for defining programming languages, calculi, type systems, and formal analysis tools
- It has been used to define:

  Languages Functional, Imperative, OO, Declarative; including Verilog, Scheme, Java, and now C

What is K?

- K is an executable semantic framework for defining programming languages, calculi, type systems, and formal analysis tools
- It has been used to define:

  Languages Functional, Imperative, OO, Declarative; including Verilog, Scheme, Java, and now C

  Type Systems Checkers, Inferencers; including $\mathscr{W}$

What is K?

- K is an executable semantic framework for defining programming languages, calculi, type systems, and formal analysis tools
- It has been used to define:

  Languages Functional, Imperative, OO, Declarative; including Verilog, Scheme, Java, and now C

  Type Systems Checkers, Inferencers; including $\mathscr{W}$

  Algorithms Sorting, Dijkstra's Algorithm, Sudoku Solving

What is K?

- K is an executable semantic framework for defining programming languages, calculi, type systems, and formal analysis tools
- It has been used to define:

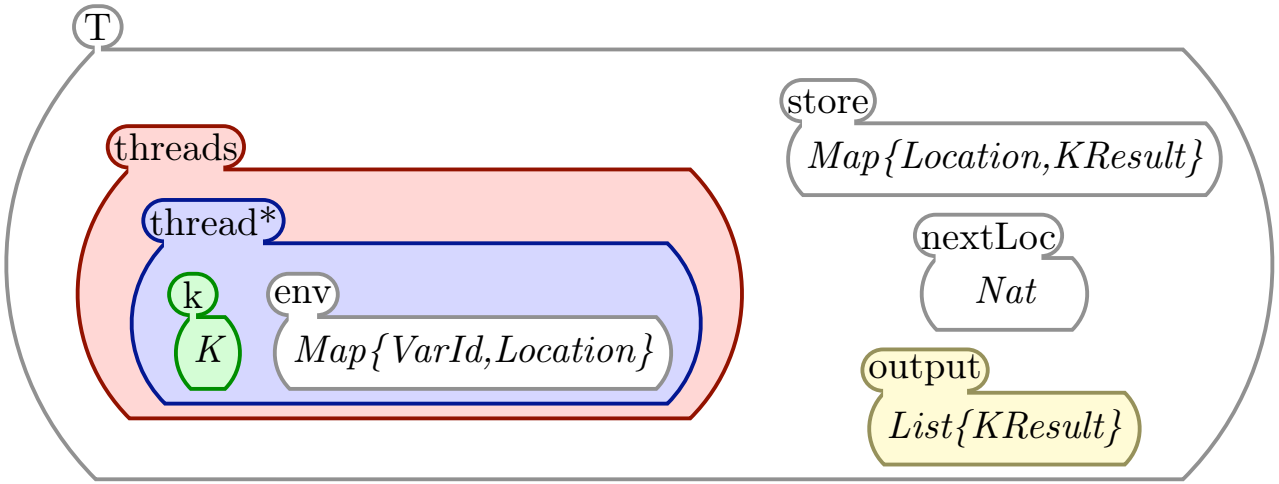
Languages	Functional, Imperative, OO, Declarative; including Verilog, Scheme, Java, and now C
Type Systems	Checkers, Inferencers; including $\mathscr{W}$
Algorithms	Sorting, Dijkstra's Algorithm, Sudoku Solving
Tools	Debugging, Race Detection

What is K?

- K is an executable semantic framework for defining programming languages, calculi, type systems, and formal analysis tools
- It has been used to define:

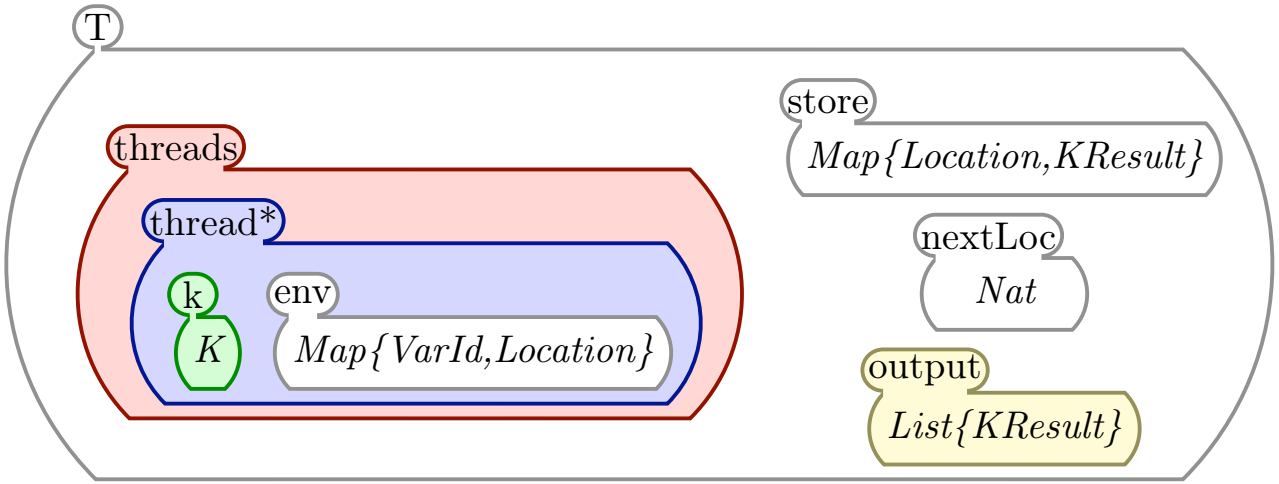
Languages	Functional, Imperative, OO, Declarative; including Verilog, Scheme, Java, and now C
Type Systems	Checkers, Inferencers; including $\mathscr{W}$
Algorithms	Sorting, Dijkstra's Algorithm, Sudoku Solving
Tools	Debugging, Race Detection
Logics	Matching Logic

The K Technique



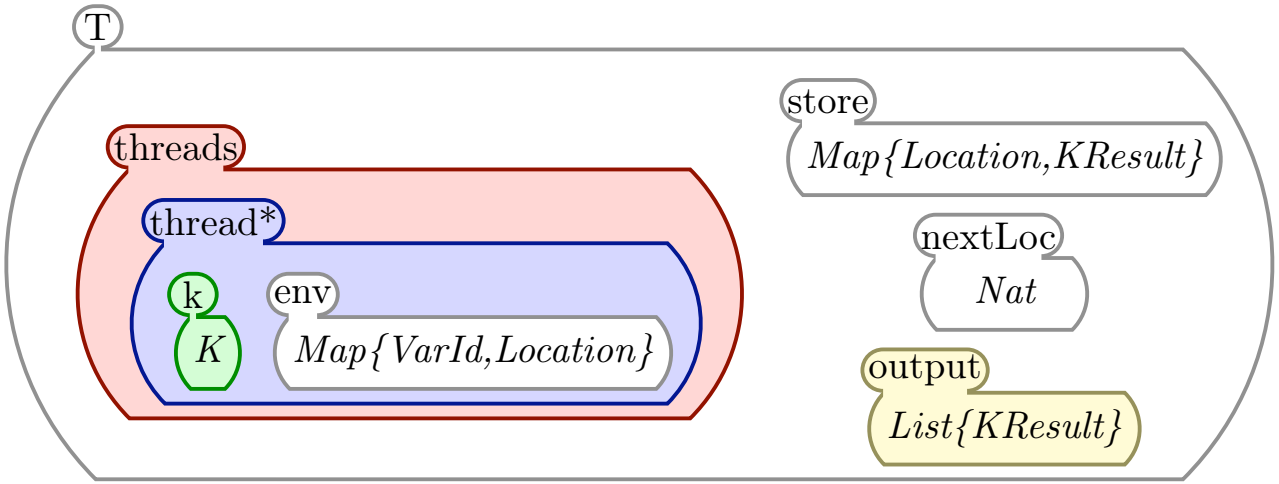
- Flexible, extensible, configurations as multi-sets of nested cells

The K Technique



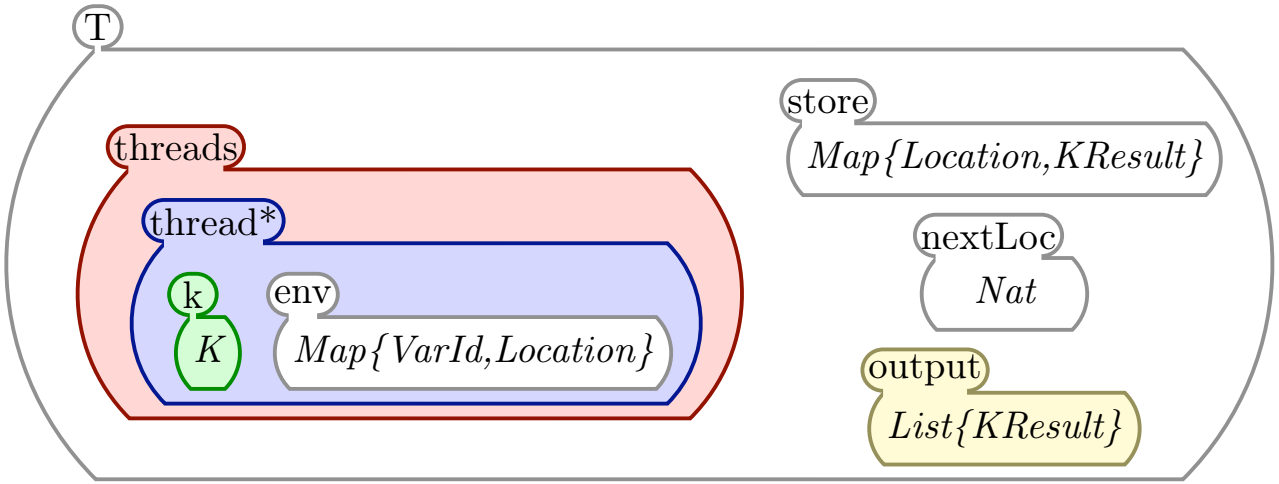
- Flexible, extensible, configurations as multi-sets of nested cells
  - Cells can contain (multi-)sets, lists, maps, or computations

The K Technique



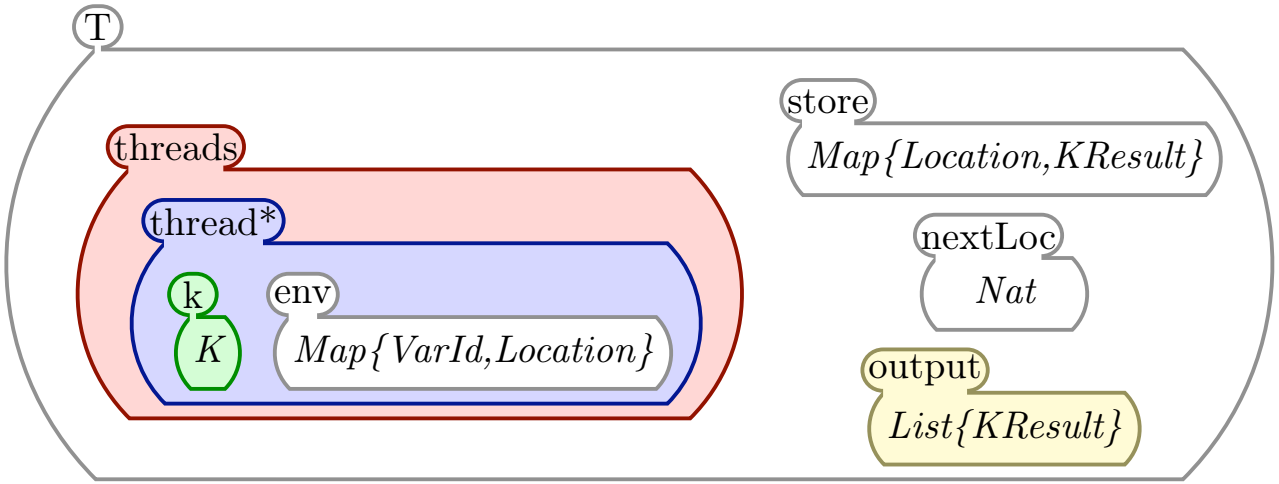
- Flexible, extensible, configurations as multi-sets of nested cells
  - Cells can contain (multi-)sets, lists, maps, or computations
- Computation (K) as a list of $\curvearrowright$-separated tasks

The K Technique



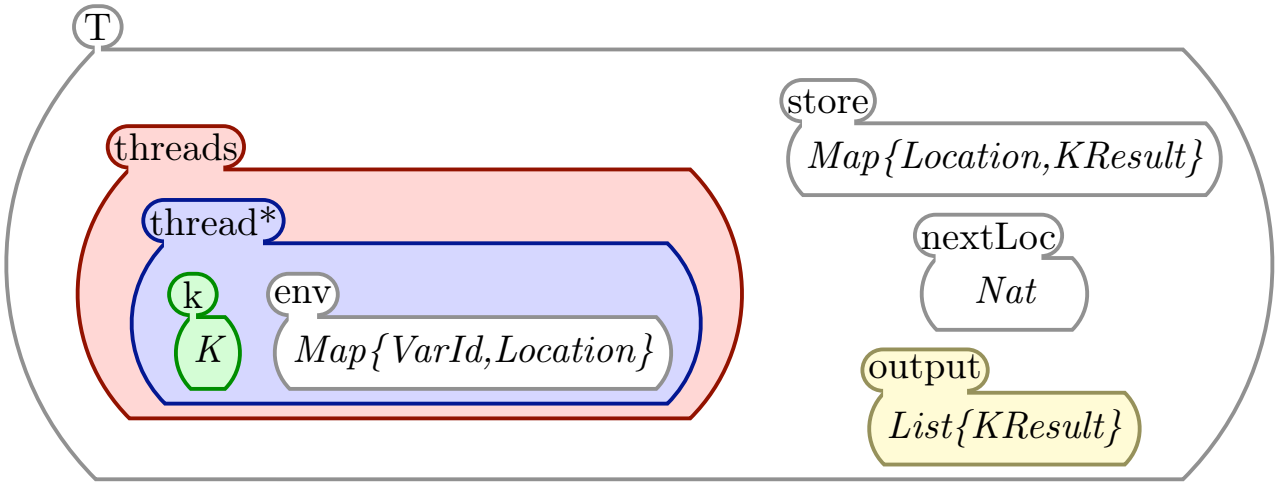
- Flexible, extensible, configurations as multi-sets of nested cells
  - Cells can contain (multi-)sets, lists, maps, or computations
- Computation (K) as a list of $\curvearrowright$-separated tasks
  - Next task always at top of the list

The K Technique



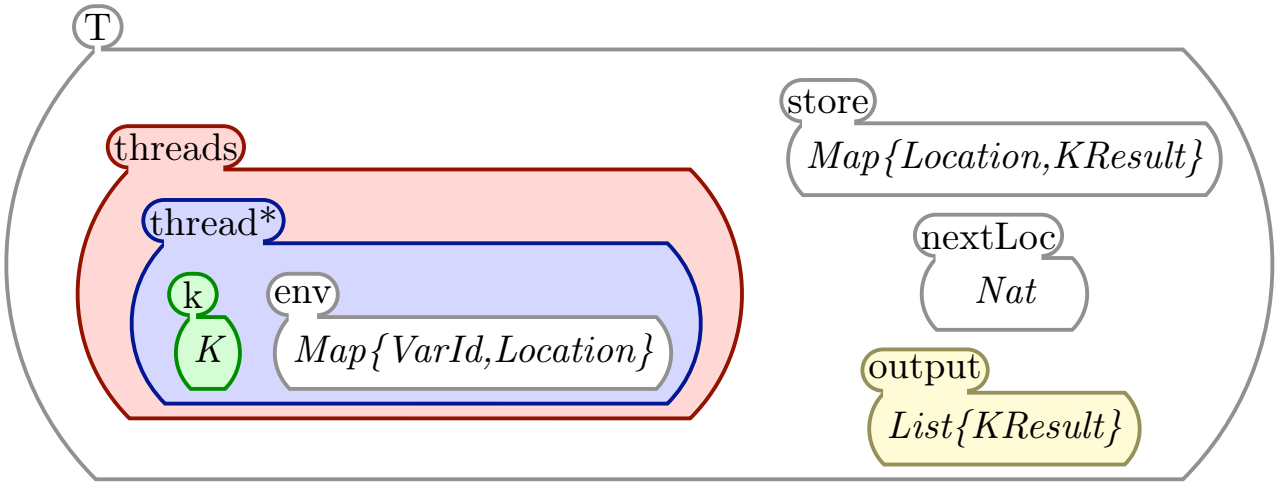
- Flexible, extensible, configurations as multi-sets of nested cells
  - Cells can contain (multi-)sets, lists, maps, or computations
- Computation (K) as a list of $\curvearrowright$-separated tasks
  - Next task always at top of the list
  - Easy to define control-intensive features like halt, call/cc

The K Technique



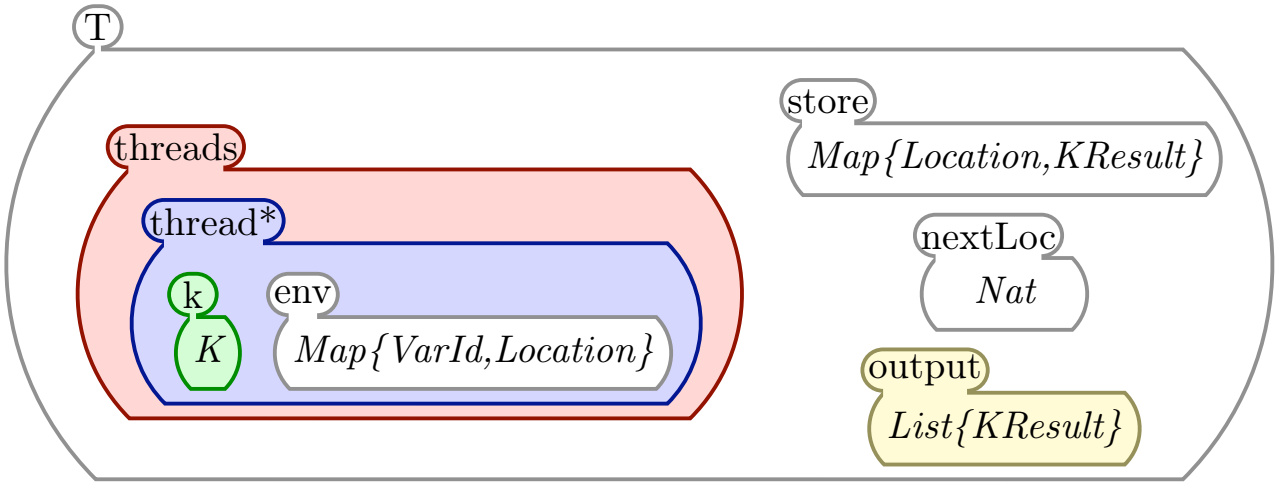
- Flexible, extensible, configurations as multi-sets of nested cells
  - Cells can contain (multi-)sets, lists, maps, or computations
- Computation (K) as a list of $\curvearrowright$-separated tasks
  - Next task always at top of the list
  - Easy to define control-intensive features like halt, call/cc
- Rewriting modulo ACI (associativity, commutivity, identity) to improve modularity

The K Technique



- Flexible, extensible, configurations as multi-sets of nested cells
  - Cells can contain (multi-)sets, lists, maps, or computations
- Computation (K) as a list of $\curvearrowright$-separated tasks
  - Next task always at top of the list
  - Easy to define control-intensive features like halt, call/cc
- Rewriting modulo ACI (associativity, commutivity, identity) to improve modularity
  - Specify only what is needed from a cell for a semantic rule

The K Technique



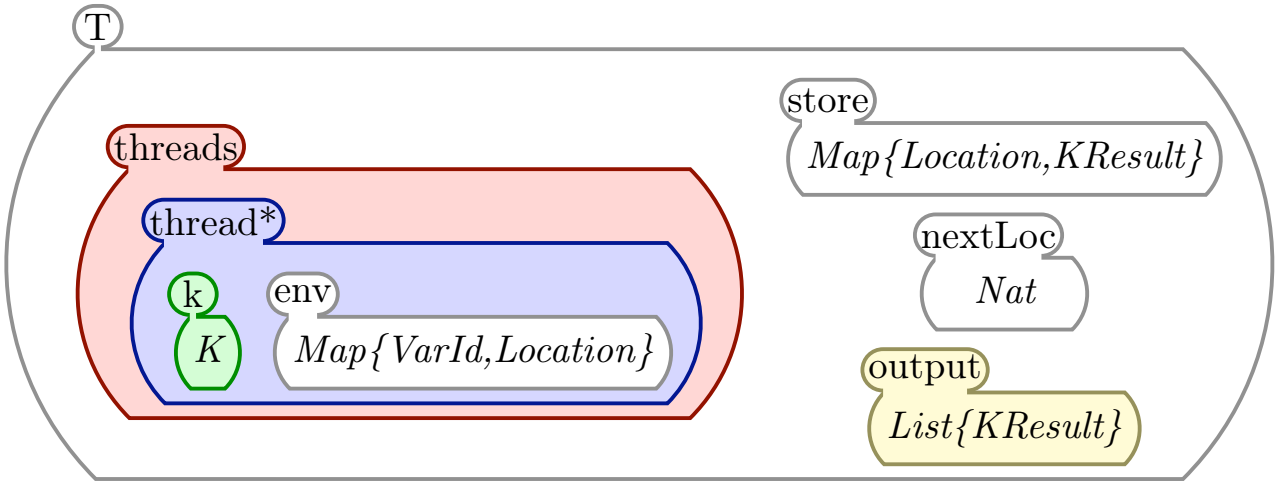
- Flexible, extensible, configurations as multi-sets of nested cells
  - Cells can contain (multi-)sets, lists, maps, or computations
- Computation (K) as a list of $\curvearrowright$-separated tasks
  - Next task always at top of the list
  - Easy to define control-intensive features like halt, call/cc
- Rewriting modulo ACI (associativity, commutivity, identity) to improve modularity
  - Specify only what is needed from a cell for a semantic rule
  - Abstract the remainder of the cell

IMP Syntax

AExp	::=	*Int* | *VarId*	
	|	*AExp* + *AExp*	[*strict*]
	|	*AExp* / *AExp*	[*strict*]
BExp	::=	*Bool*	
	|	*AExp* <= *AExp*	[*seqstrict*]
	|	`not` *BExp*	[*strict*]
	|	*BExp* `and` *BExp*	[*strict*(1)]
Stmt	::=	`skip`| *Stmt* `;` *Stmt*	
	|	*VarId* `:=` *AExp*	[*strict*(2)]
	|	`if` *BExp* `then` *Stmt* `else` *Stmt*	[*strict*(1)]
	|	`while` *BExp* `do` *Stmt*	
	|	`print` *AExp*	[*strict*]
	|	`var` *VarId* `;` *Stmt*	

Assignment Rule

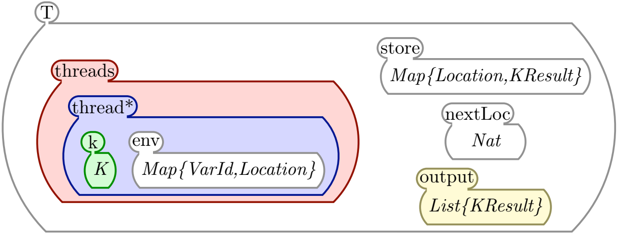
Configuration



Variable Lookup Rule

$$\langle \frac{X}{V} \cdots \rangle_{\text{k}} \ \langle \cdots \ X \mapsto L \ \cdots \rangle_{\text{env}} \ \langle \cdots \ L \mapsto V \ \cdots \rangle_{\text{store}}$$

IMP Semantics



Assignment

$$\langle \underset{\cdot}{\underline{X\ :=\ V}} \cdots\rangle_{\mathsf{k}}\ \ \langle\cdots X \mapsto L \cdots\rangle_{\mathsf{env}}\ \ \langle \cdots L \mapsto \underset{V}{\underline{\_}} \cdots\rangle_{\mathsf{store}}$$

Print (Output)

$$\langle \texttt{print}\ V \cdots\rangle_{\mathsf{k}}\ \ \langle \cdots \underset{V}{\underline{\cdot}}\rangle_{\mathsf{output}}$$

C Semantics

- Plenty of formal semantics for C already: [Gurevich and Huggins, 1993]; [Cook, Cohen, and Redmond, 1994]; [Cook, and Subramanian, 1994]; [Norrish, 1998]; [Papaspyrou, 1998]; [Blazy and Leroy, 2009]
- Hard to deal with:
  - Unstructured control flow (`goto`, `switch`)
  - Intricate typing rules
  - Expression evaluation order has few restrictions

Duff's Device

- Unstructured control flow (`goto`, `switch`)

```
int n = (count+7)/8;
switch(count%8) {
   case 0: do{ *dest++ = *src++;
   case 7:     *dest++ = *src++;
   case 6:     *dest++ = *src++;
   case 5:     *dest++ = *src++;
   case 4:     *dest++ = *src++;
   case 3:     *dest++ = *src++;
   case 2:     *dest++ = *src++;
   case 1:     *dest++ = *src++;
   } while(--n>0);
}
```

- Intricate typing rules

Signed chars: (−128 to 127) Ints: (−32768 to 32767)
Unsigned ints: (0 to 65535) Long ints: (−2M to 2M)

```
int a = 1000, b = 1000;
long int c = a * b;
```

```
unsigned int a = 1000, b = 1000;
long int c = a * b;
```

```
signed char a = 100, b = 100;
int c = a * b;
```

2147483648 ≠ 0x80000000

- Expression evaluation order has few restrictions

```
(A + B++) + C
```

A, B, C, B^{++}
A, B, B^{++}, C
A, C, B, B^{++}
A, C, B, B^{++}
...

Our Semantics

The most complete formal semantics for C to date

- Parameterizable on implementation-defined parts of the semantics, but given a default instantiation

Our Semantics

The most complete formal semantics for C to date

- Parameterizable on implementation-defined parts of the semantics, but given a default instantiation
- Covering every major feature including parts of the standard library: `goto`, `longjump`, `malloc`, variadic functions, enums, structs, unions, bitfields, typedefs...

Our Semantics

The most complete formal semantics for C to date

- Parameterizable on implementation-defined parts of the semantics, but given a default instantiation
- Covering every major feature including parts of the standard library: `goto`, `longjump`, `malloc`, variadic functions, enums, structs, unions, bitfields, typedefs...
- Yielding an interpreter, debugger, and state space search and model checker "for free"

Our Semantics (Results)

- 125 syntactic operators

Our Semantics (Results)

- 125 syntactic operators
- 200 auxiliary semantic operators

Our Semantics (Results)

- 125 syntactic operators
- 200 auxiliary semantic operators
- 620 different rules

Our Semantics (Results)

- 125 syntactic operators
- 200 auxiliary semantic operators
- 620 different rules
- 2800 source lines of code (SLOC)

Our Semantics (Results)

- 125 syntactic operators
- 200 auxiliary semantic operators
- 620 different rules
- 2800 source lines of code (SLOC)
- Only 50 rules for statements

Our Semantics (Results)

- 125 syntactic operators
- 200 auxiliary semantic operators
- 620 different rules
- 2800 source lines of code (SLOC)
- Only 50 rules for statements
- Only 130 for expressions

Our Semantics (Results)

- 125 syntactic operators
- 200 auxiliary semantic operators
- 620 different rules
- 2800 source lines of code (SLOC)
- Only 50 rules for statements
- Only 130 for expressions
- Tested against the GCC torture tests:

Our Semantics (Results)

- 125 syntactic operators
- 200 auxiliary semantic operators
- 620 different rules
- 2800 source lines of code (SLOC)
- Only 50 rules for statements
- Only 130 for expressions
- Tested against the GCC torture tests:
  - Of 1057 tests, 720 tests appear to be standards compliant. Of those 720, we pass about 95%.