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 $W$. 
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 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 $W$.
  - **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 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 \( \rightleftharpoons \)-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 $\ni$-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 \( \rightsquigarrow \)-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 \( \rightarrow \)-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 \( \rightarrow \)-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 \( \cdot \)-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

\[ \begin{align*}
AExp & ::= \text{Int} | \text{VarId} \\
& | AExp + AExp \quad \text{[strict]} \\
& | AExp / AExp \quad \text{[strict]} \\
BExp & ::= \text{Bool} \\
& | AExp \leq AExp \quad \text{[seqstrict]} \\
& | \text{not } BExp \quad \text{[strict]} \\
& | BExp \text{ and } BExp \quad \text{[strict(1)]} \\
Stmt & ::= \text{skip} | Stmt ; Stmt \\
& | \text{VarId} := AExp \quad \text{[strict(2)]} \\
& | \text{if } BExp \text{ then } Stmt \text{ else } Stmt \quad \text{[strict(1)]} \\
& | \text{while } BExp \text{ do } Stmt \\
& | \text{print } AExp \quad \text{[strict]} \\
& | \text{var } \text{VarId} ; Stmt
\end{align*} \]
Assignment Rule

Configuration

Variable Lookup Rule

\[
\left\langle X \cdots \right\rangle_k \left\langle \cdots X \mapsto L \cdots \right\rangle_{\text{env}} \left\langle \cdots L \mapsto V \cdots \right\rangle_{\text{store}}
\]
IMP Semantics

Assignment

\[
\langle X := V \cdots \rangle_k \quad \langle \cdots X \mapsto L \cdots \rangle_{\text{env}} \quad \langle \cdots L \mapsto - \cdots \rangle_{\text{store}}
\]

Print (Output)

\[
\langle \text{print } V \cdots \rangle_k \quad \langle \cdots \cdot \rangle_{\text{output}}
\]
The K Framework
Overview
Example

Semantics of C
Background
Results
C Semantics

- 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)

```c
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)

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

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

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

2147483648 ≠ 0x80000000
Expression evaluation order has few restrictions

\[(A + B++) + C\]

\[A, B, C, B^{++}\]

\[A, B, B^{++}, C\]

\[A, C, B, B^{++}\]

\[A, C, B, B^{++}\]

\[\ldots\]
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

- Tested against the GCC torture tests:
  - Of 1057 tests, 720 tests appear to be standards compliant. Of those 720, we pass about 95%.
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

Tested against the GCC torture tests:
- Of 1057 tests, 720 tests appear to be standards compliant. Of those 720, we pass about 95%.
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%.
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%. 