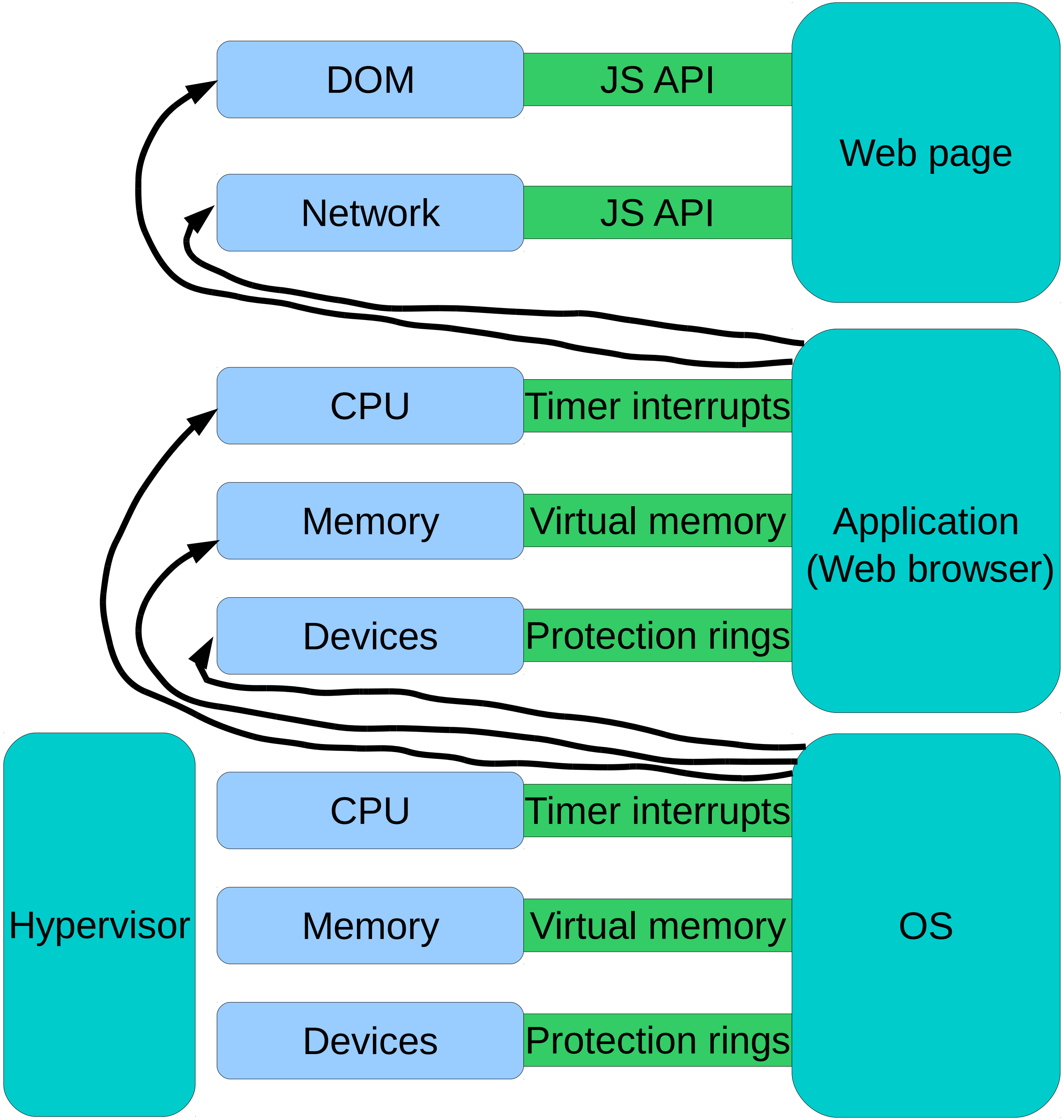
An Extensible Programming Language for Verified Systems Software

Adam Chlipala
MIT CSAIL
WG 2.16 meeting, 2012

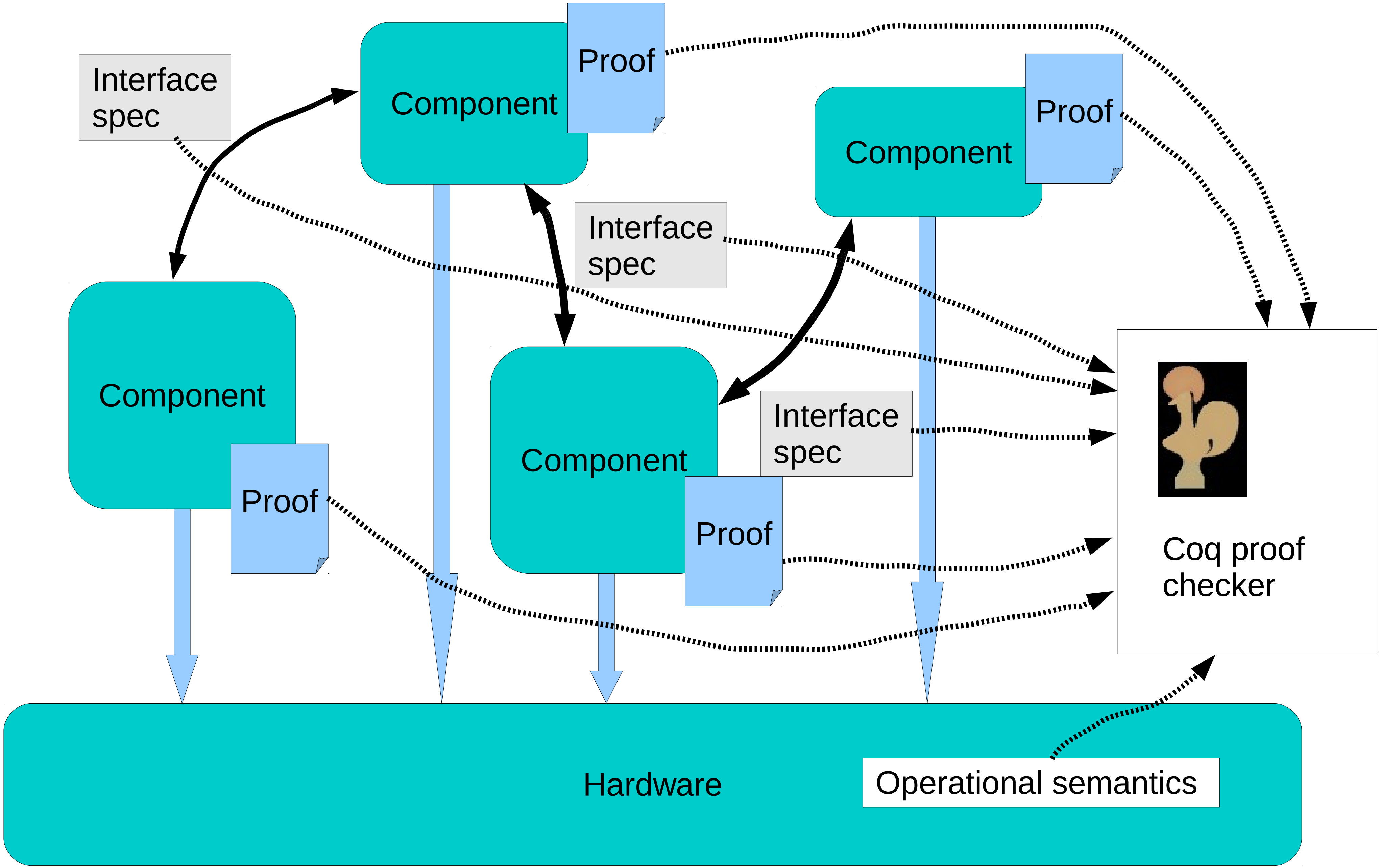
The status quo in computer system design



Nested sandbox architecture

We don't trust applications to do the right thing, so we spend lots of hardware & software resources monitoring their behavior.

The proof-carrying code approach



Step 1

What is the programming language underneath all this?
How do we formalize its semantics and convince ourselves we got it right?
What sorts of proof techniques and formal verification tools apply?

Too high-level!

Too low-level!

C?

Coq proof checker

Hardware

Operational semantics

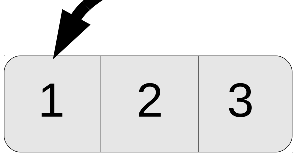
```
int *p = NULL;
*p;
```

Undefined behavior!

(nowhere)

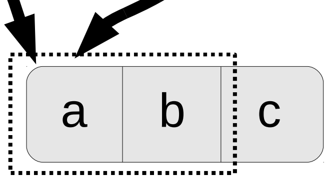
```
int a[] = {1, 2, 3};
a[3];
```

Undefined behavior!



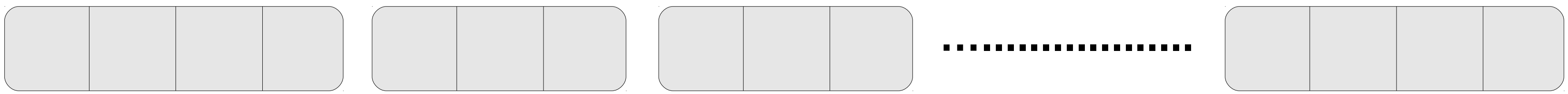
```
struct s1 { int a, b, c; };
struct s2 { int a, b; };

int foo(struct s1 *p1) {
   struct s2 *p2 = (struct s2 *) p1;
   return p2->a + p2->b;
}
```



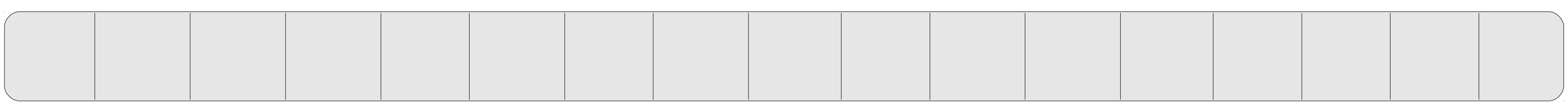
Undefined behavior?

C standard memory model: complex semantics of *objects*

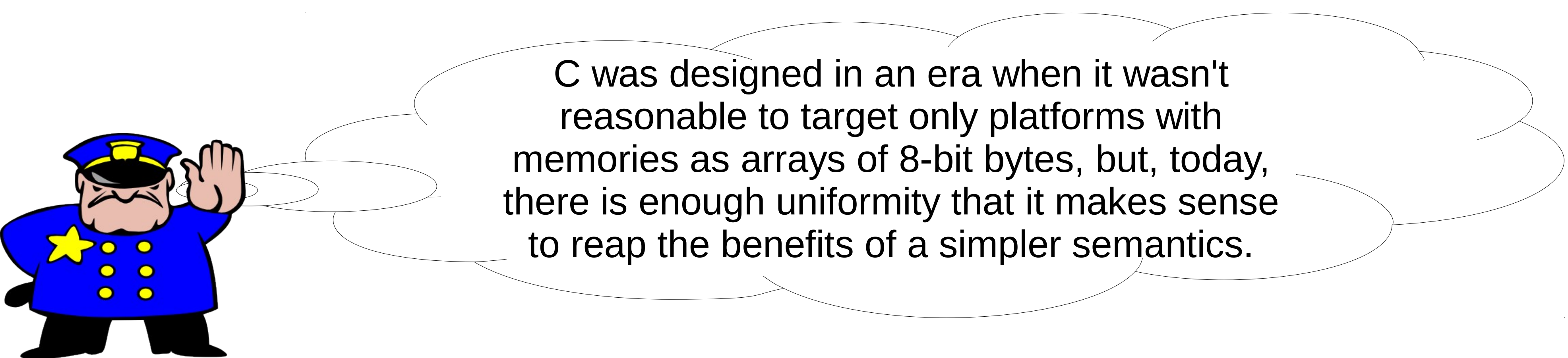


Plus a set of carefully chosen rules about when pointers within an object may be considered to denote other objects

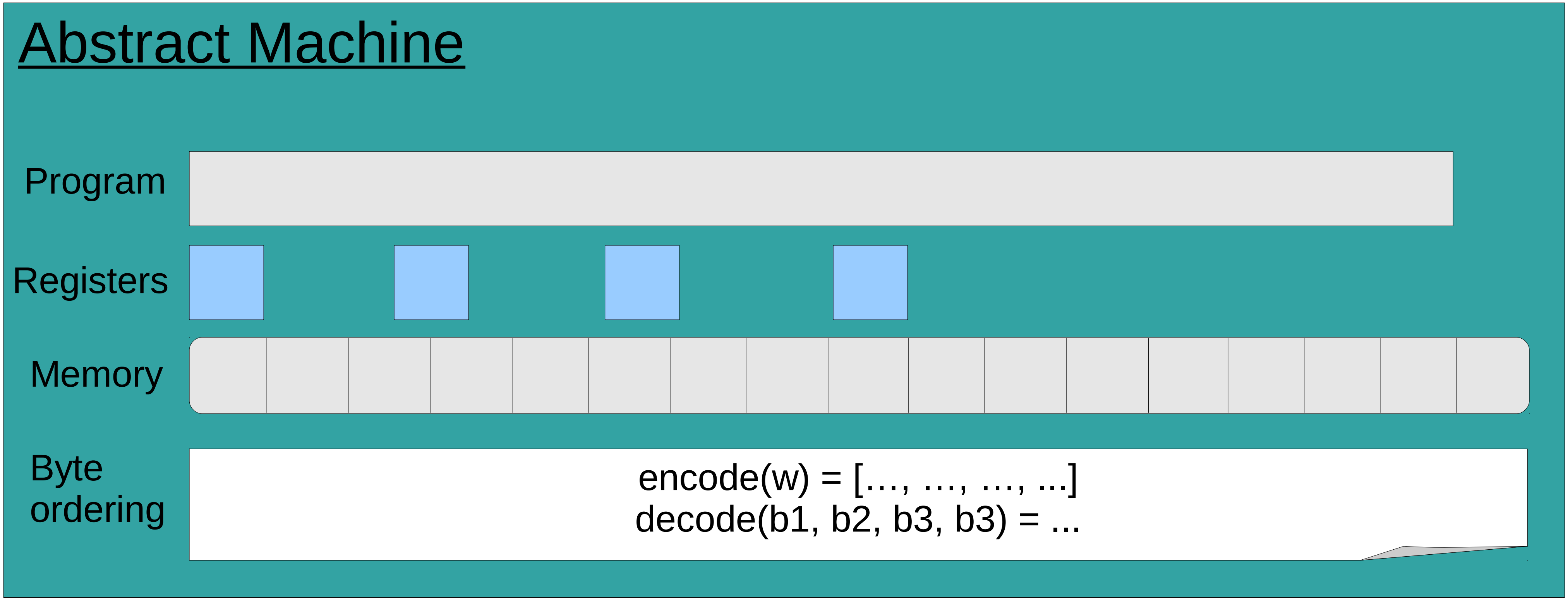
Alternative model: memory as an array of bytes



Cross-platform, lowest-common-denominator assembly language

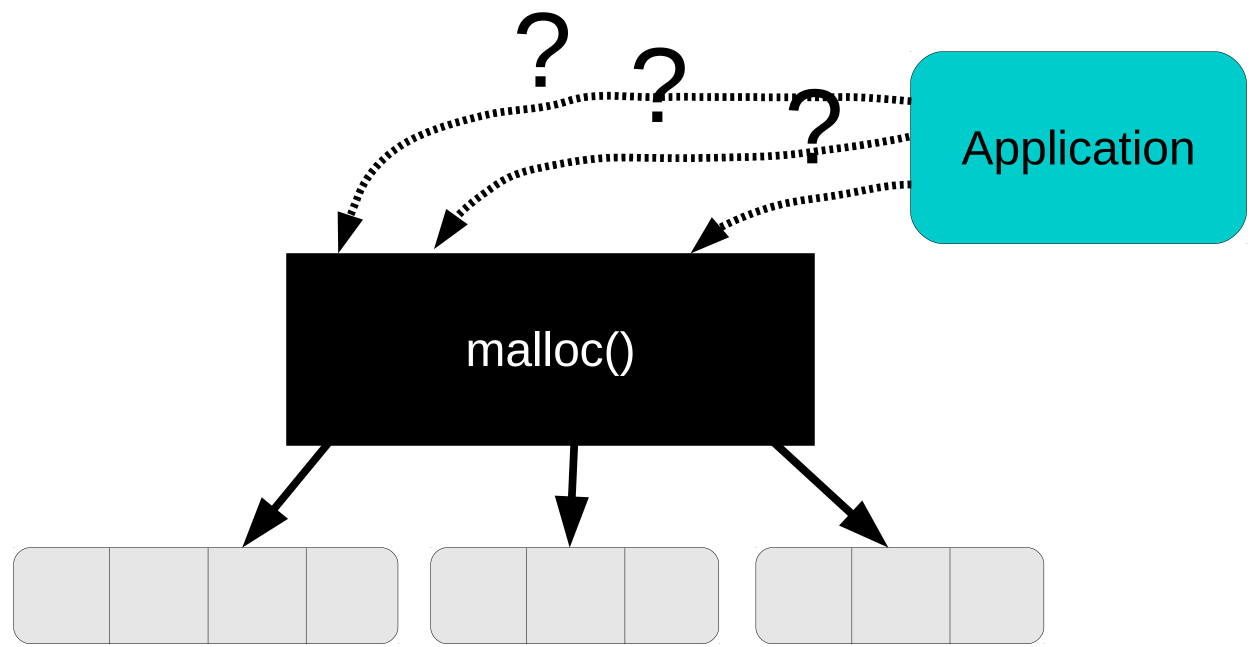


What about different byte orderings?



What about the interface of malloc() & co?

The C way:



When the language semantics makes memory an array of bytes, all this reasoning can be encapsulated portably in a well-specific library.

What about local variables & calling conventions?

Why not implement these at the library level, too?

Saves us some headaches specifying:

- Context management for process & thread schedulers
- Methods for garbage collectors to introspect call stack
- ...

The Bedrock IL

Bedrock

```
W ::= (* width-32 bitvectors *)
L ::= (* program code block labels *)

Reg ::= Sp | Rp | Rv
Loc ::= Reg | W | Reg + W
Lvalue ::= Reg | Loc
Rvalue ::= Lvalue | W | L
Binop ::= + | - | *
Test ::= = | != | < | <=

Instr ::= Lvalue := Rvalue | Lvalue := Rvalue Binop Rvalue

Jump ::= goto Rvalue | if Rvalue Test Rvalue then goto L else goto L

Block ::= L: Instr*; Jump
Module ::= Block*
```

Complex semantics,
with special case rules for many situations,
but still not enough for modern PL implementation.

C?

Poor support for **metaprogramming**:
we want good hygiene for macros,
and the possibility for macros to do complex compilation

Examples: Yacc and SQL via integrated use of macros, rather than ad-hoc external tools

C-like programming notation

Bedrock

Expressive **macro system** with **verification support**

Lowest-common-denominator, cross-platform “assembly language”

Bedrock version of linked-list length

```
Definition lengthS : spec := SPEC("x") reserving 1
  Al ls,
  PRE[V] sll ls (V "x")
  POST[R] [| R = length ls |] * sll ls (V "x").
```

← Specifications via functional programming

```
bfunction "length"("x", "n") [lengthS]
  "n" <- 0;;
  [Al ls,
    PRE[V] sll ls (V "x")
    POST[R] [| R = V "n" ^+ length ls |] * sll ls (V "x")]
  While ("x" <> 0) {
    "n" <- "n" + 1;;
    "x" <- "x" + 4;;
    "x" <-* "x"
  };;
  Return "n"
end.
```

Ignore for a moment....

Loop invariant

C-style syntax

This is all Coq code!
(so please excuse the slightly grungy concrete syntax)

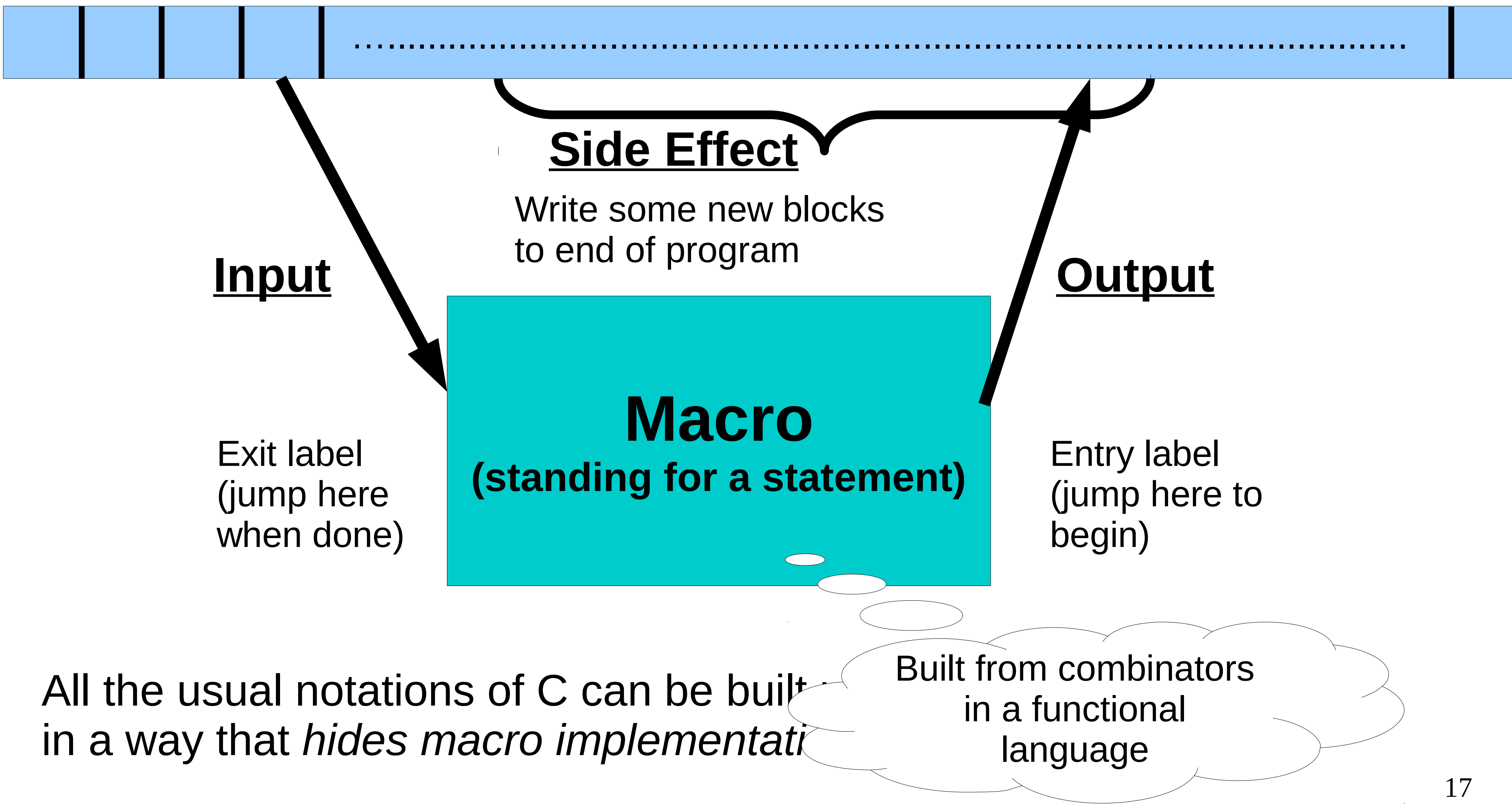
```
Theorem sllMOk : moduleOk sllM.
  vcgen; abstract (sep hints; finish).
Qed.
```

Mostly automated proofs

Challenge #1: Design a concept of macros that makes it possible to build up all the usual constructs of C and more, from first principles.

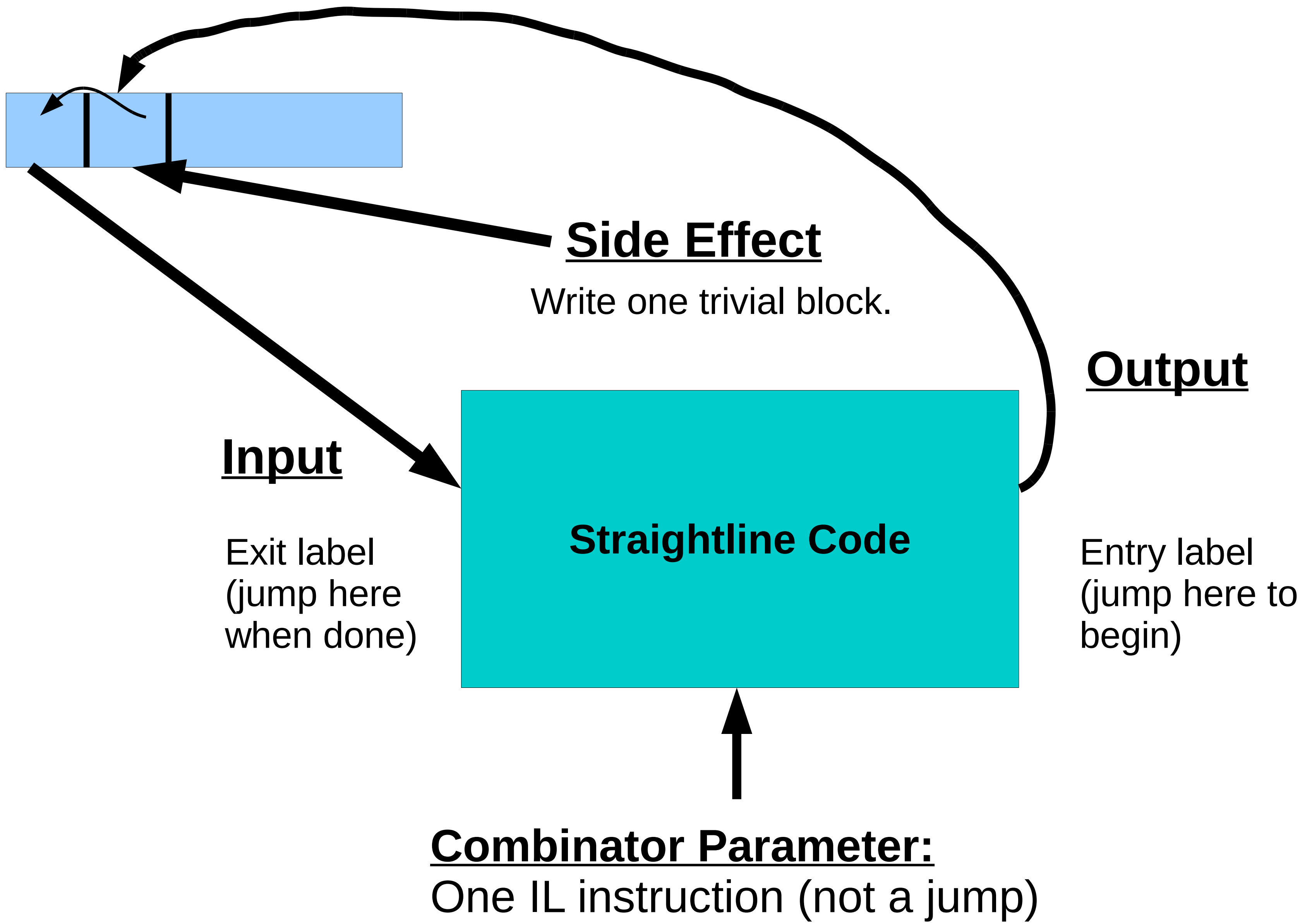
Anatomy of a macro

A macro appends to an array of program basic blocks.

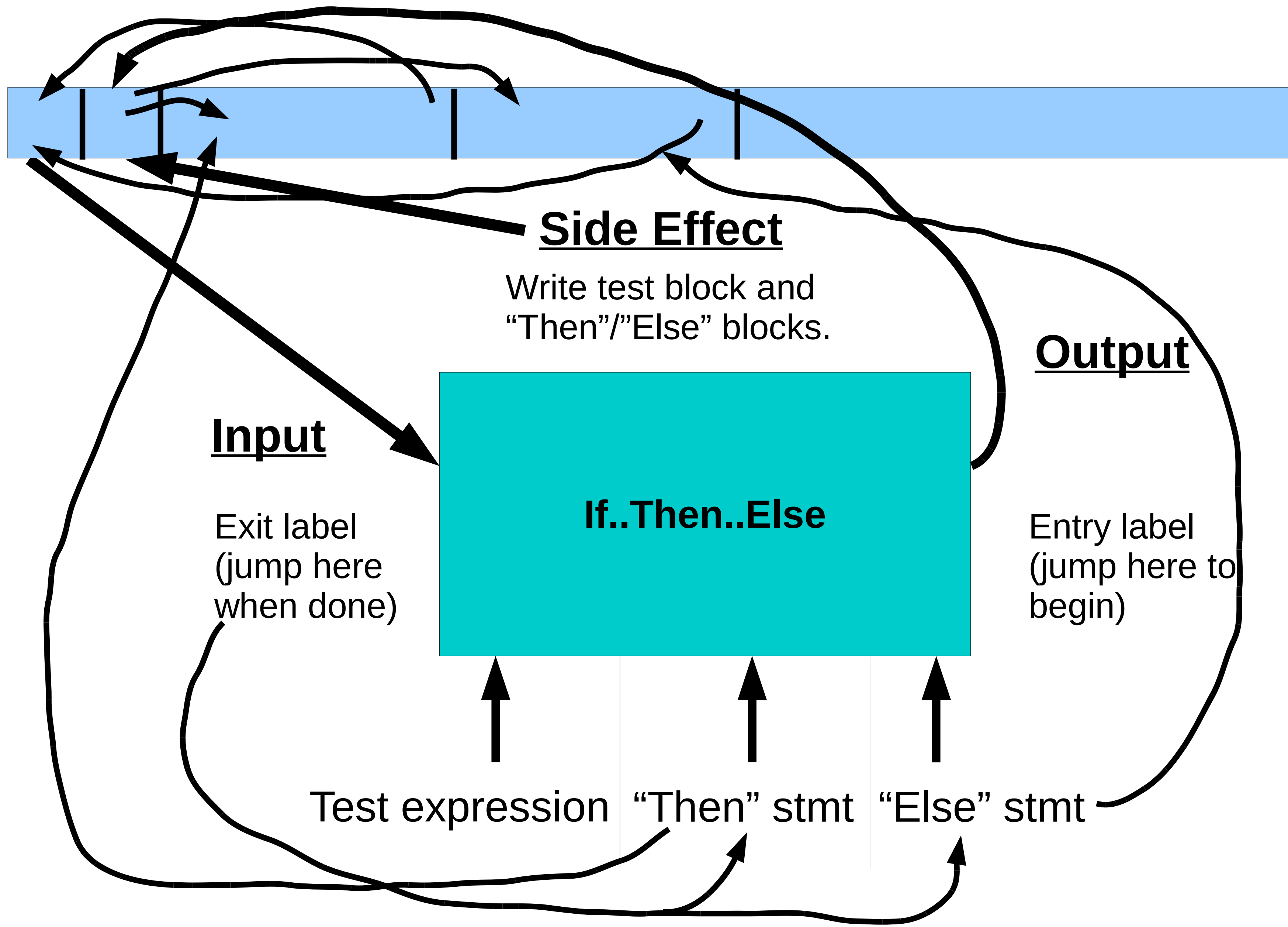


All the usual notations of C can be built
in a way that *hides macro implementati*

Anatomy of a macro

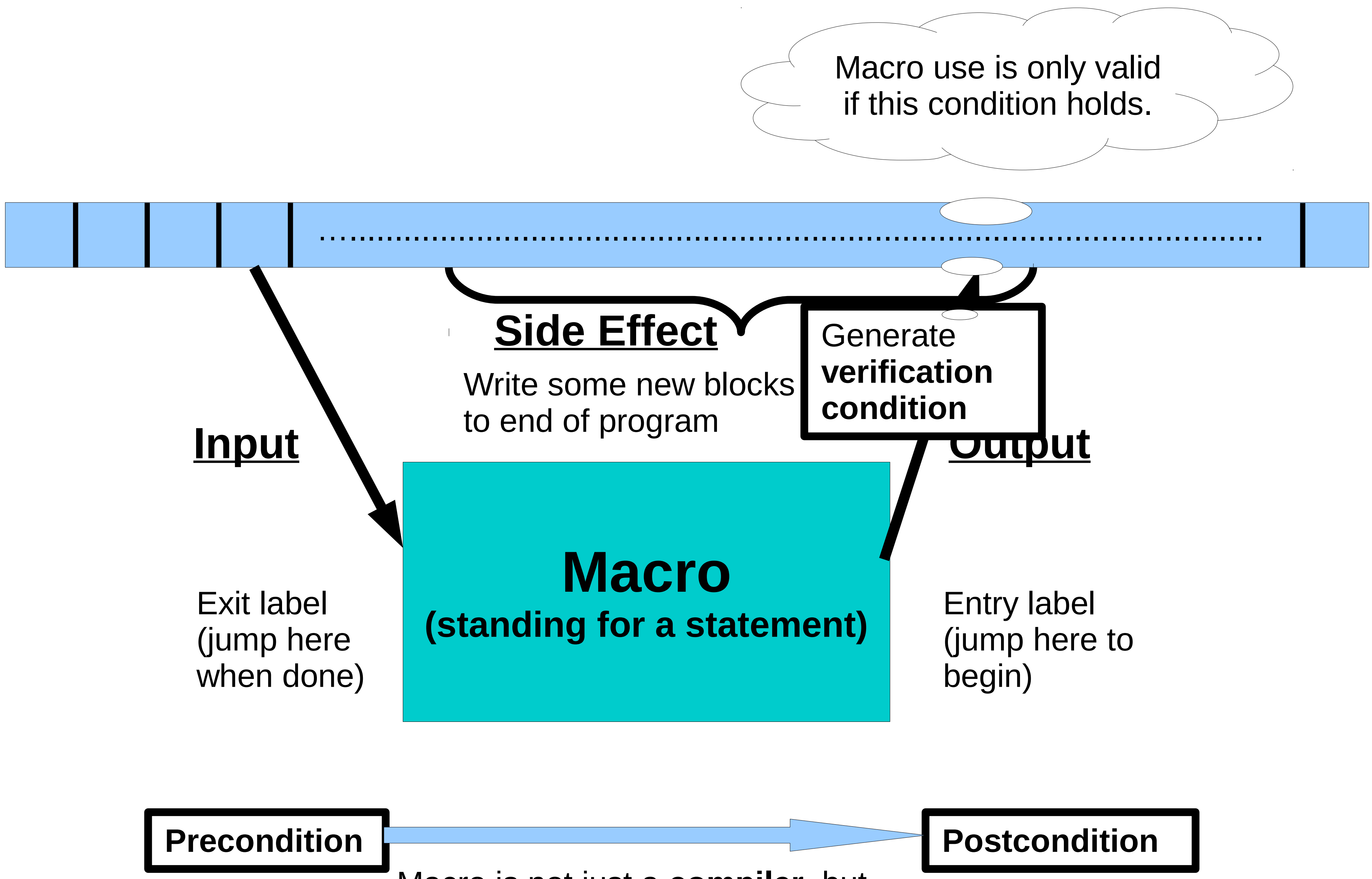


Anatomy of a macro



Challenge #2: Allow formal verification of macro-using programs, in a way that allows **reasoning about macros independently of their implementations**.

Anatomy of a macro



Macro is not just a **compiler**, but also a **predicate transformer**.

Example: Straightline code

Instruction: *i*
Precondition: *PRE*
Postcondition: $\lambda s.\ \exists s'.\ PRE(s') \wedge \mathrm{eval}(s', i, s)$
Verification condition: $\forall s.\ PRE(s) \Rightarrow \exists s'.\ \mathrm{eval}(s, i, s')$

One-instruction evaluation relation for Bedrock IL.

Check for no crash when *i* runs.

Conditions are predicates (functions) over machine states.

The boring part
Notations in Coq do what C macros do

```
Notation "[ p ] 'While' c { b }" :=
   (While_ (INV p) c b)
   (no associativity, at level 95, c at level 0)
   : SP_scope.
```

Pattern matching for network protocols

```
"pos" <- 0;;
Match "req" Size "len" Position "pos" {
  Case (0 ++ "x")
    Return "x"
  end;;
  Case (1 ++ "x" ++ "y")
    Return "x" + "y"
  end
} Default {
  Fail
}
```

Declarative querying of arrays

```
"acc" <- 0;;

[After prefix Approaching all
  PRE[V] [| V "acc" = countNonzero prefix |]
  POST[R] [| R = countNonzero all |] ]
For "index" Holding "value" in "arr" Size "len"
  Where (Value <> 0) {
  "acc" <- "acc" + 1
};;

Return "acc"
```

Loop has filter condition that the macro analyzes syntactically to decide on optimizations.

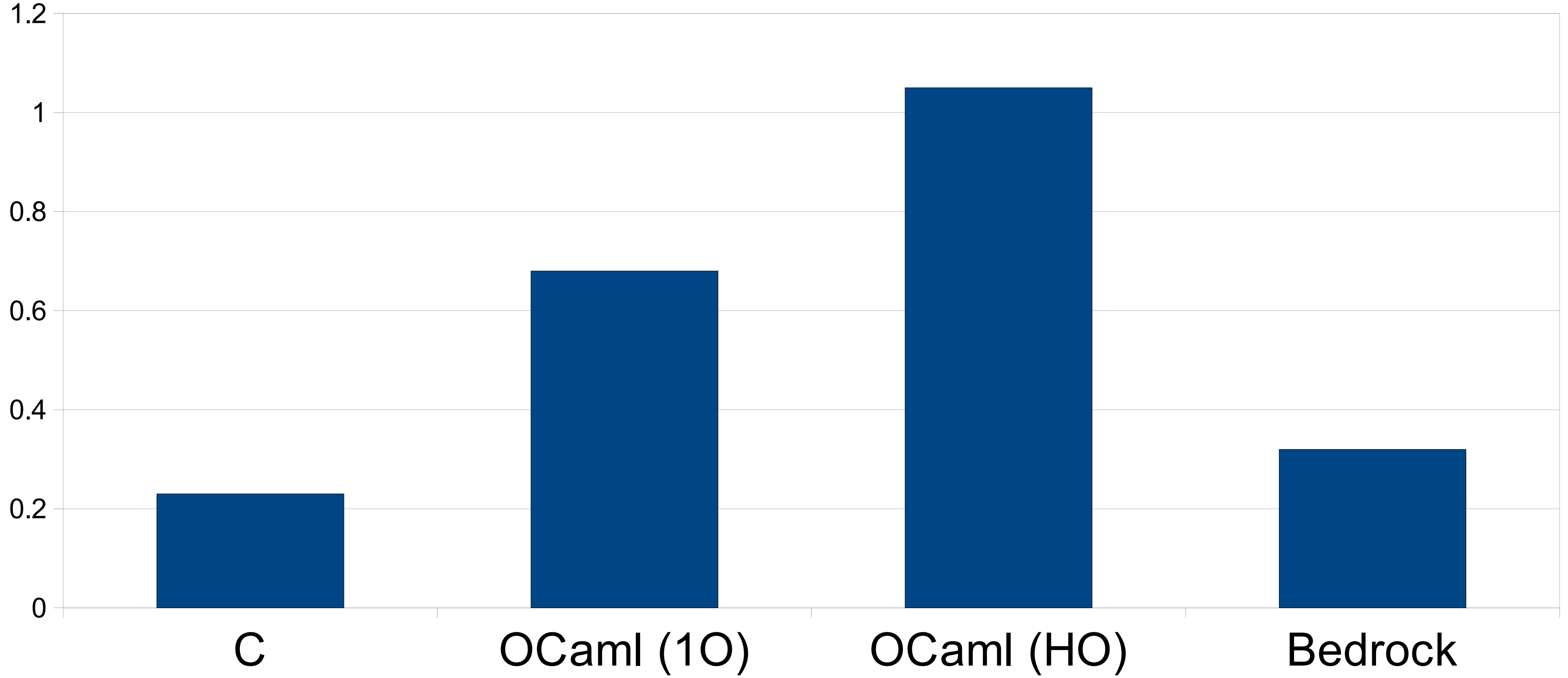
Parse byte sequences with a high-level pattern notation

For loop with “Where” condition; implementation analyzes condition to deduce that some loop iterations may be skipped

```
bfunction "main"("cmd", "cmdLen", "data", "dataLen", "output", "position", "posn",
                 "lower", "upper", "index", "value", "res", "node")
  "output" <- 0;;
  "position" <- 0;;
  While ("position" < "cmdLen") {
    Match "cmd" Size "cmdLen" Position "position" {
      Case (0 ++ "posn" ++ "lower")
        "res" <- 0;;
        For "index" Holding "value" in "data" Size "dataLen"
          Where ((Index = "posn") && (Value >= "lower")) {
          "res" <- 1
        };;
        "node" <-- Call "malloc"!"malloc"(0);;
        "node" *<- "res";; "node" + 4 *<- "output";; "output" <- "node"
      end;;
      Case (1 ++ "lower" ++ "upper")
        "res" <- 0;;
        For "index" Holding "value" in "data" Size "dataLen"
          Where (("lower" <= Value) && (Value <= "upper") && (Value >= "res")) {
          "res" <- "value"
        };;
        "node" <-- Call "malloc"!"malloc"(0);;
        "node" *<- "res";; "node" + 4 *<- "output";; "output" <- "node"
      end;;
      Case (2 ++ "lower" ++ "upper")
        For "index" Holding "value" in "data" Size "dataLen"
          Where ((Index >= "lower") && (Value <= "upper")) {
          "node" <-- Call "malloc"!"malloc"(0);;
          "node" *<- "value";; "node" + 4 *<- "output";; "output" <- "node"
        }
      end
    } Default {
      Fail
    }
  };;
  Return "output"
end
```

Not shown here: About 400 more lines to state & prove the correctness theorem

Running time (s) of 4 implementations of that program



(For a random workload of 200 queries to a database of 100,000 values)

Bedrock on the web

`http://plv.csail.mit.edu/bedrock/`

Backup

Example: If..Then..Else

Test expression: *e*
Then statement: *THEN*
Else statement: *ELSE*
Precondition: *PRE*
Postcondition: λs. Post(*THEN*)(λs'. *PRE*(s') $\wedge$ eval(s', *e*, 1))(s)
  $\vee$ Post(*ELSE*)(λs'. *PRE*(s') $\wedge$ eval(s', *e*, 0))(s)
Verification condition: ($\forall$s. *PRE*(s) $\Rightarrow$ $\exists$b. eval(s, *e*, b))
  $\wedge$ VC(*THEN*)(λs'. *PRE*(s') $\wedge$ eval(s', *e*, 1))
  $\wedge$ VC(*ELSE*)(λs'. *PRE*(s') $\wedge$ eval(s', *e*, 0))

Example: While

Test expression: e
Loop body statement: *BODY*
Loop invariant: *INV*
Precondition: *PRE*
Postcondition: $\lambda s.\ INV(s) \wedge \mathrm{eval}(s, e, 0)$
Verification condition: $(\forall s.\ INV(s) \Rightarrow \exists b.\ \mathrm{eval}(s, e, b))$
$\wedge\ (\forall s.\ PRE(s) \Rightarrow INV(s))$
$\wedge\ (\forall s.\ \mathrm{Post}(\mathrm{BODY})(\lambda s'.\ INV(s') \wedge \mathrm{eval}(s', e, 1))(s) \Rightarrow INV(s))$

Each macro is packaged with its **proof of correctness**, so programmers can use & reason about macros independently of their internals.

Once verification conditions are proved, the final theorem is **foundational**, independent of the macro approach.

Bedrock ...as a highly automated verification environment.