The Bedrock Structured Programming System

Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier

Adam Chlipala
MIT CSAIL

ICFP 2013
September 27, 2013

In the beginning, there was assembly language....

Low programmer effort

“High-Level” Languages (e.g., Haskell, ML, Scheme)

Harder to predict/control performance

implementation

Fine-grained control over performance

“Low-Level” Languages (e.g., C)

High programmer effort

implementation

```
movl 5+heap(%ebx),%edx
movl %edx,%edi
movl heap(%ebx),%edx
movl %edx,%esi
jmp bar
```

The high cost of abstraction

```
1 + 1
```

vs.

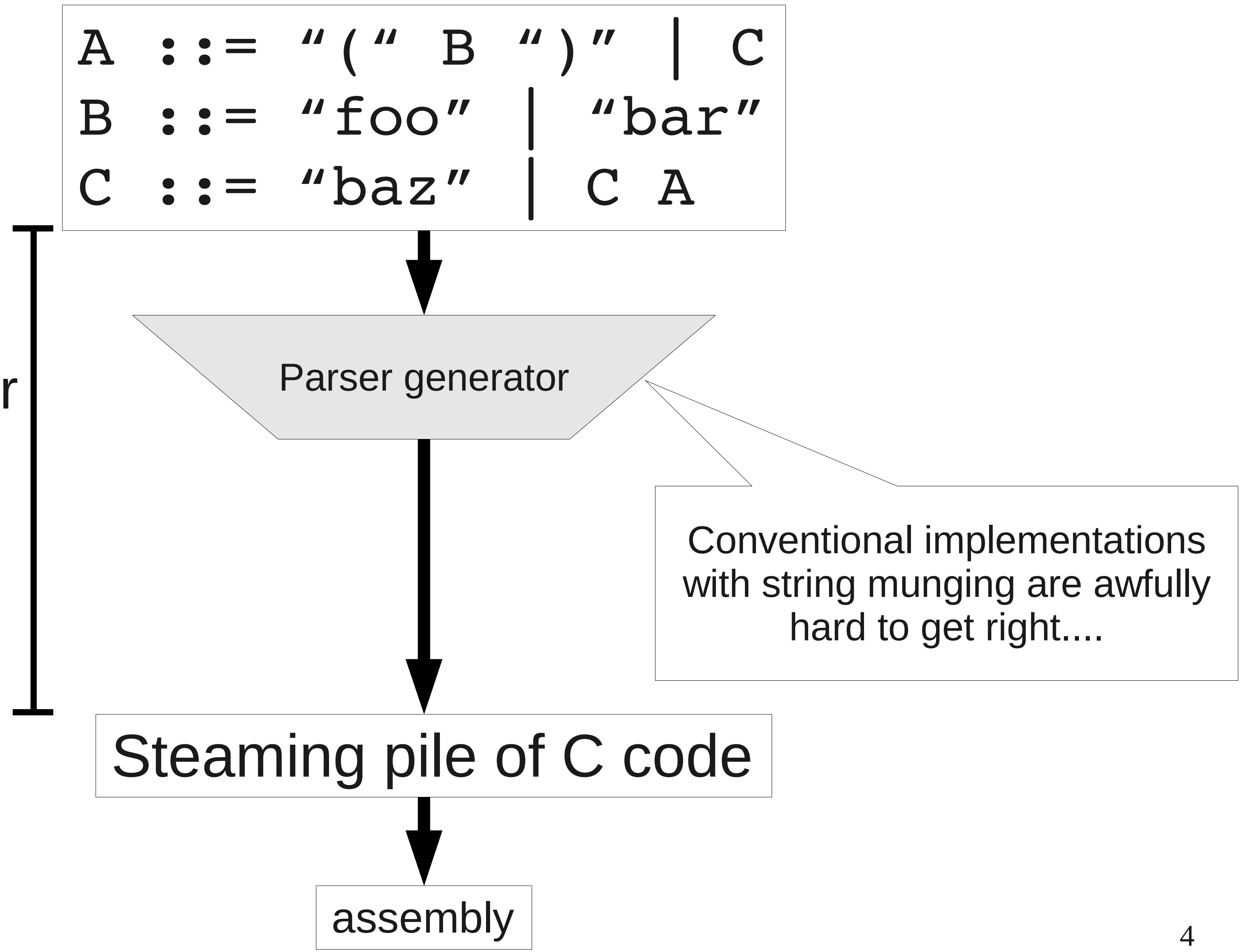
```
f x y = x + y

f 1 1
```

Is there a performance cost to functional abstraction?
higher-order functions?
modules?
laziness?
garbage collection?
exceptions?

Having your cake & eating it, too: code generators

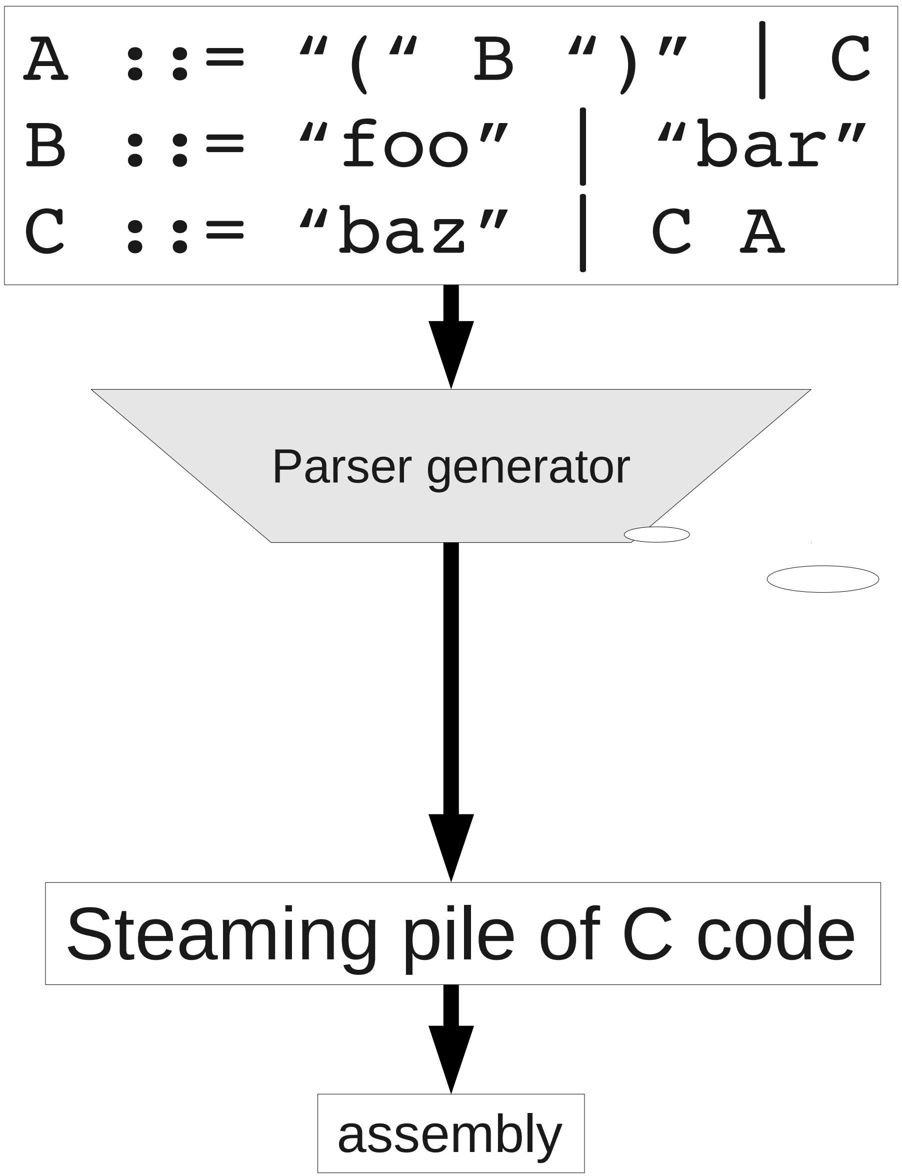
We pay in *performance* for this abstraction gap only at **compile time**!

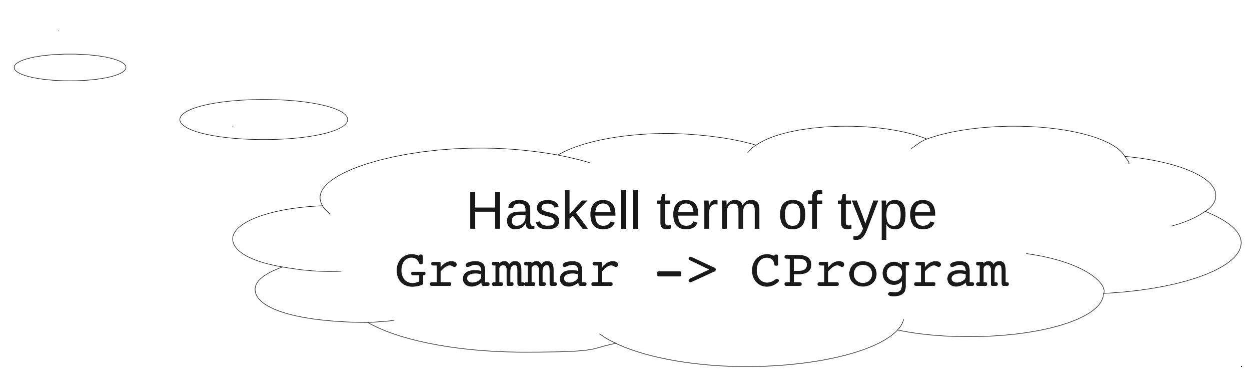


Enter embedded domain-specific languages!

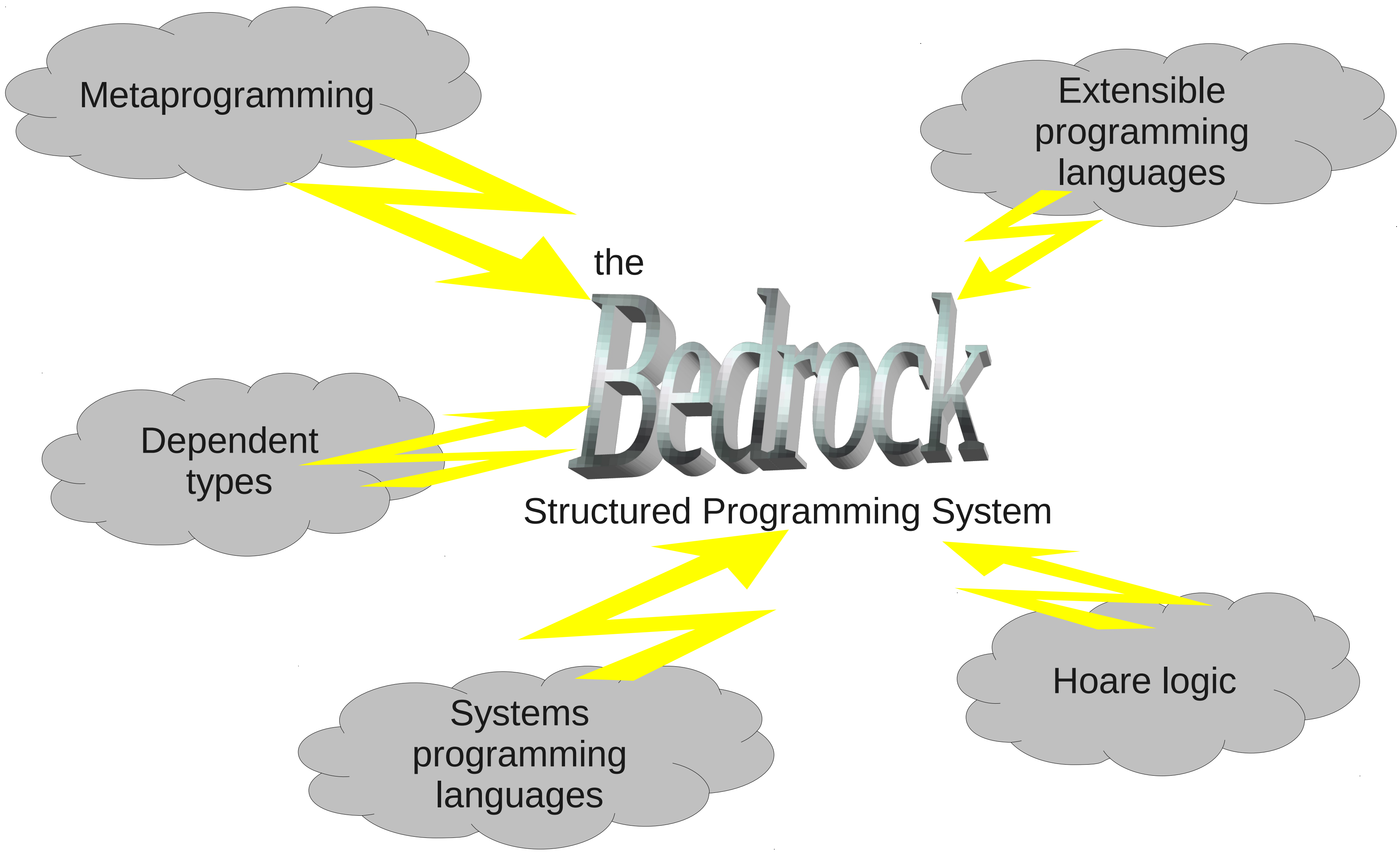
The type system of the **metalanguage** (e.g., Haskell) helps us guarantee that the translation always generates reasonable code in the **object language** (e.g., C)!

✔ Basic syntactic well-formedness
✔ Variable binding
✔ Type checking
✔ **Functional correctness...?**



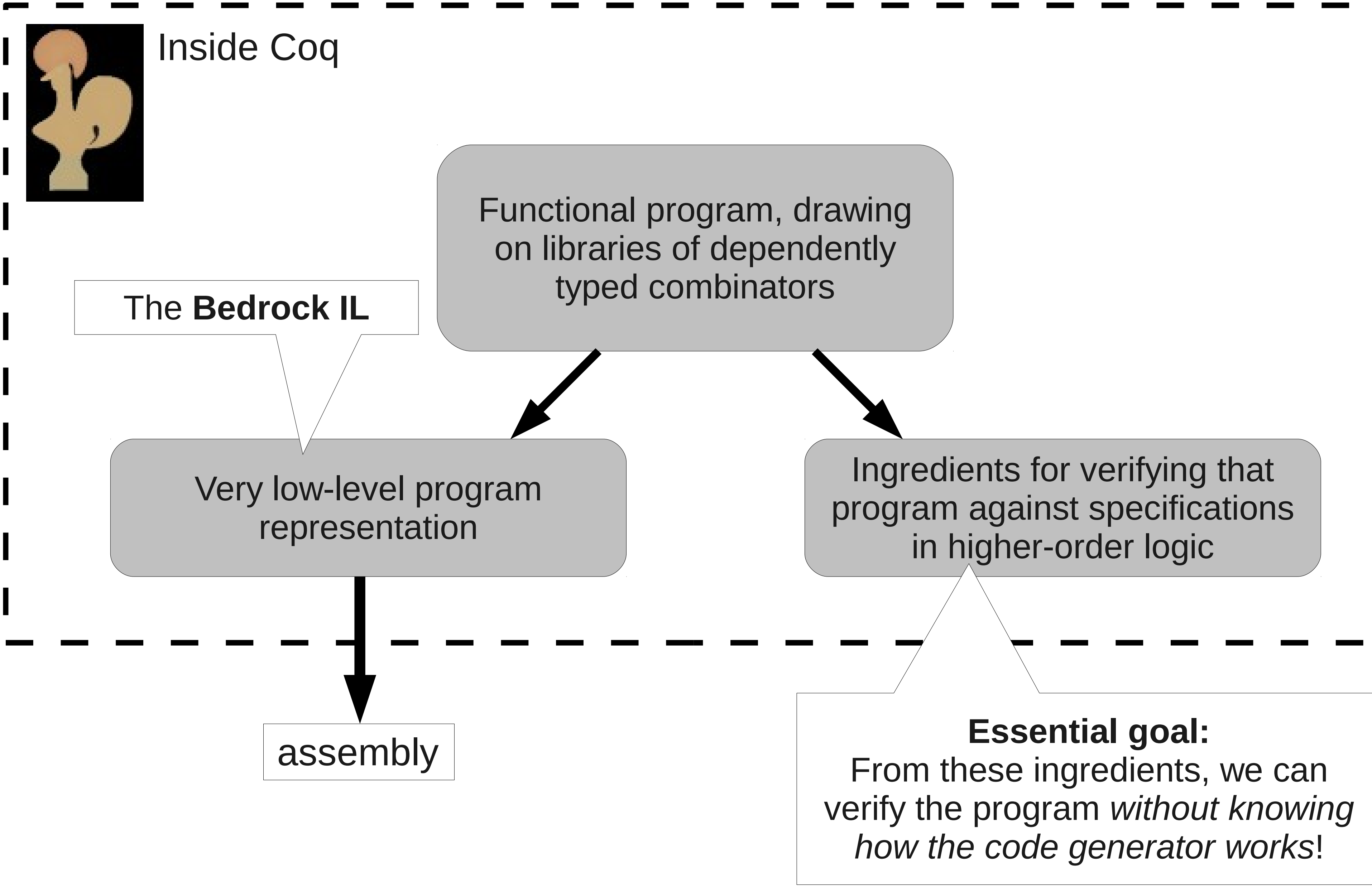


What this talk is probably about



Results in this talk use other parts of Bedrock contributed by Gregory Malecha, Thomas Braibant, Patrick Hulin, and Edward Z. Yang!

The big picture



The Bedrock IL

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

Reg ::= Sp | Rp | Rv
Loc ::= Reg | W | Reg + W
Lvalue ::= Reg | [Loc]_32 | [Loc]_8
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 ::= Instr*; Jump
Spec ::= (* assertion language of XCAP *)
Module ::= (L: {Spec} Block)*
```

Why not LLVM or a similar IR?

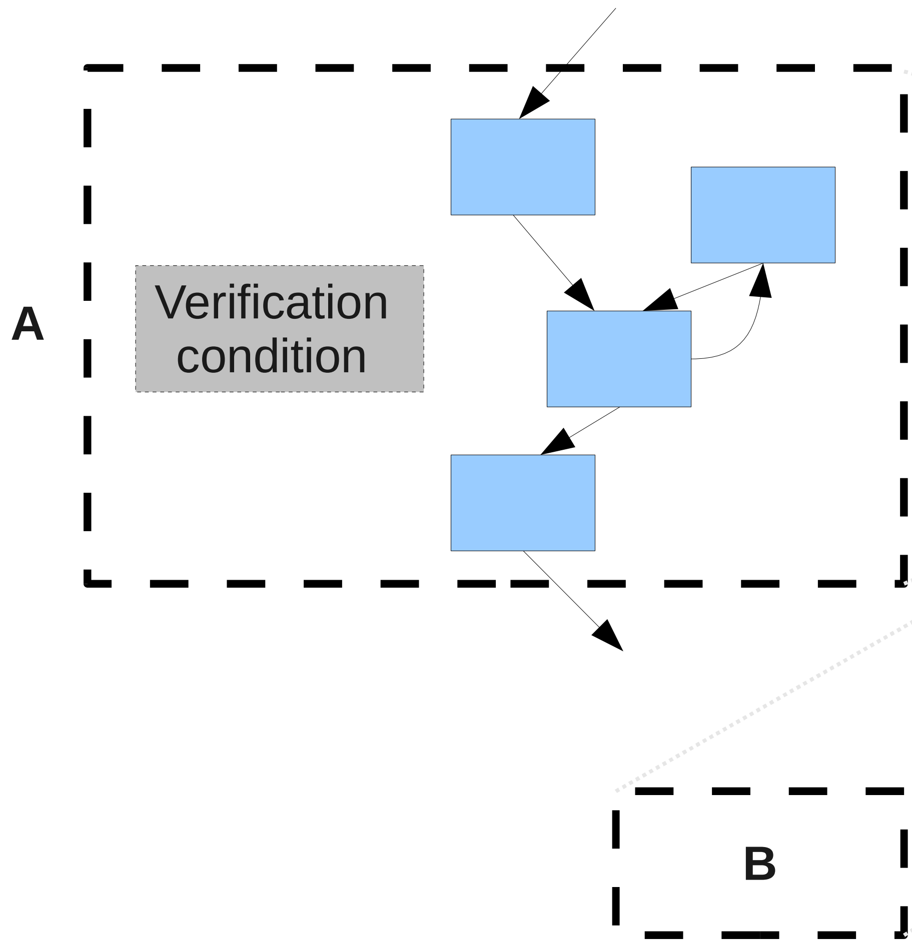
Answer: Builds in a host of features:
- Types
- Variables
- Functions

We will implement all of these as **libraries** in Bedrock!

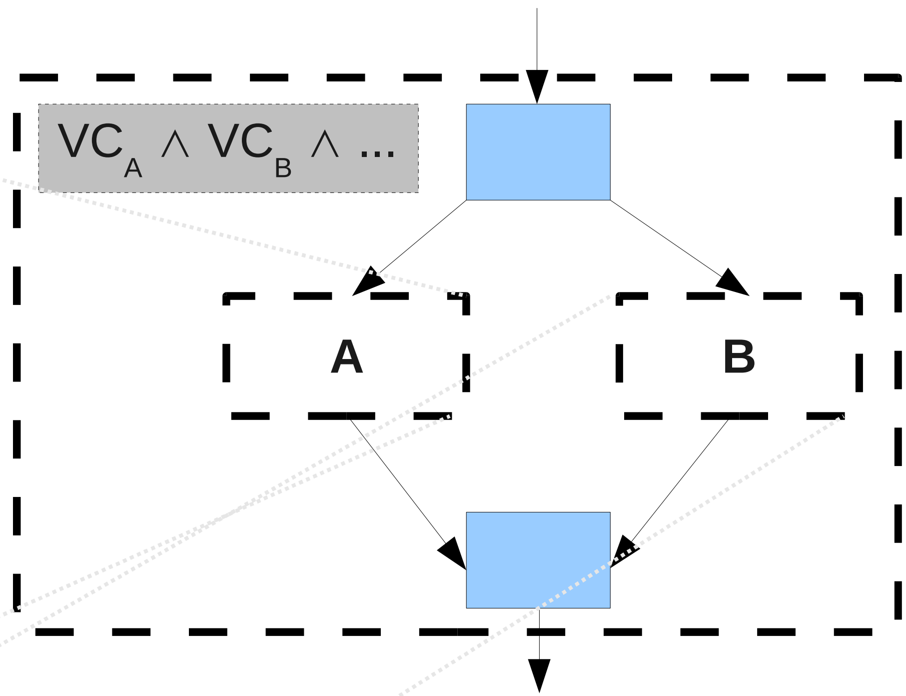
A simple language makes it easier to prove *foundational program correctness* theorems in Coq.

One-slide sketch of the formal details

An extensible C-like language based on *macros* that produce *chunks* encapsulating control-flow graphs:



if..then..else combinator (functional program):



Details omitted here:

Plumbing of connecting CFGs
Predicate transformers
Formal connection to Hoare logic

Bedrock version of linked list length

Specification

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

Program

```
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
  };;
  Return "n"
end.
```

Loop invariant

This is all Coq code, taking advantage of Coq's extensible parser!

Proof

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

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

Fancy macro-specific loop invariant form

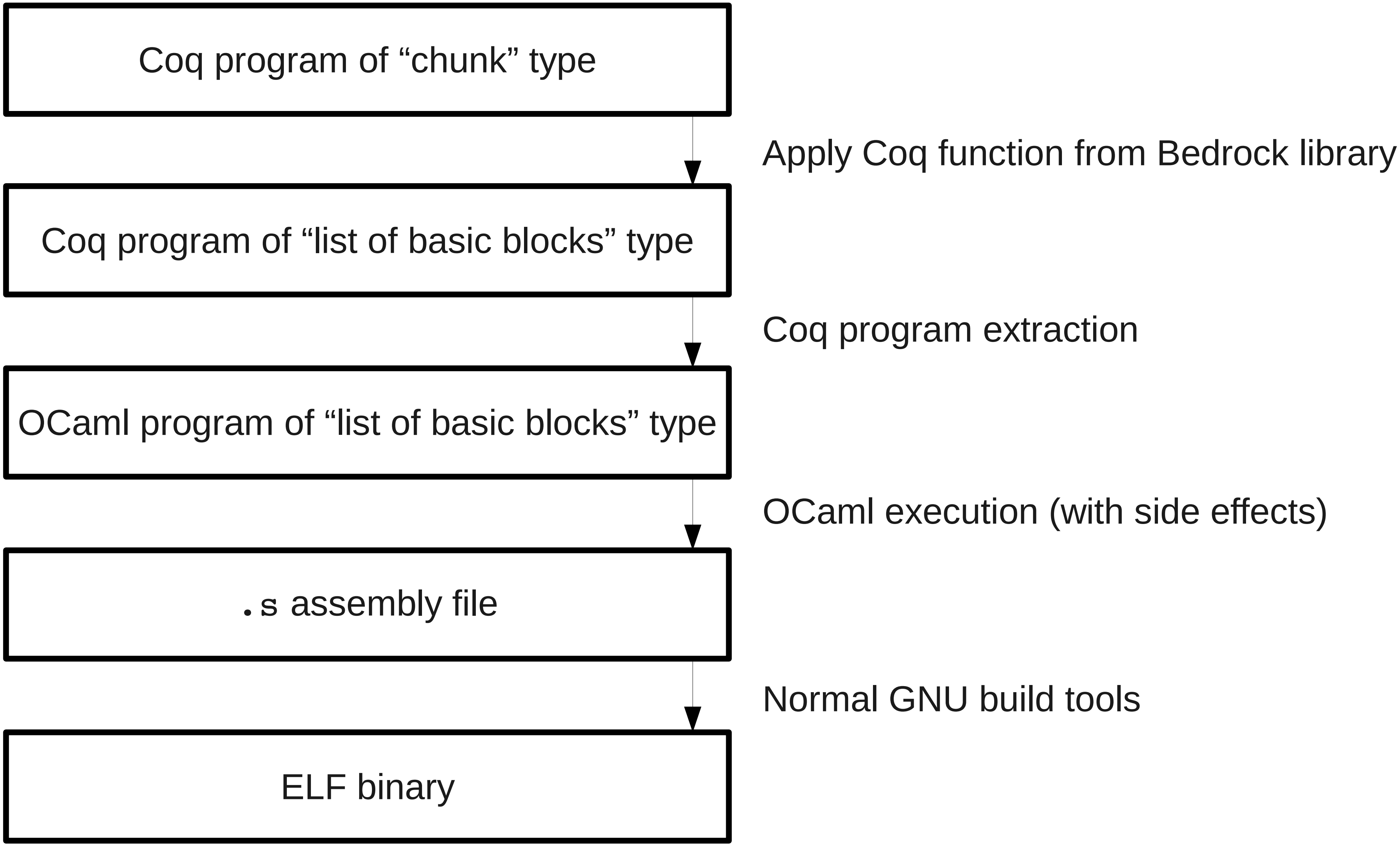
```
"acc" <- 0;;

[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"
```

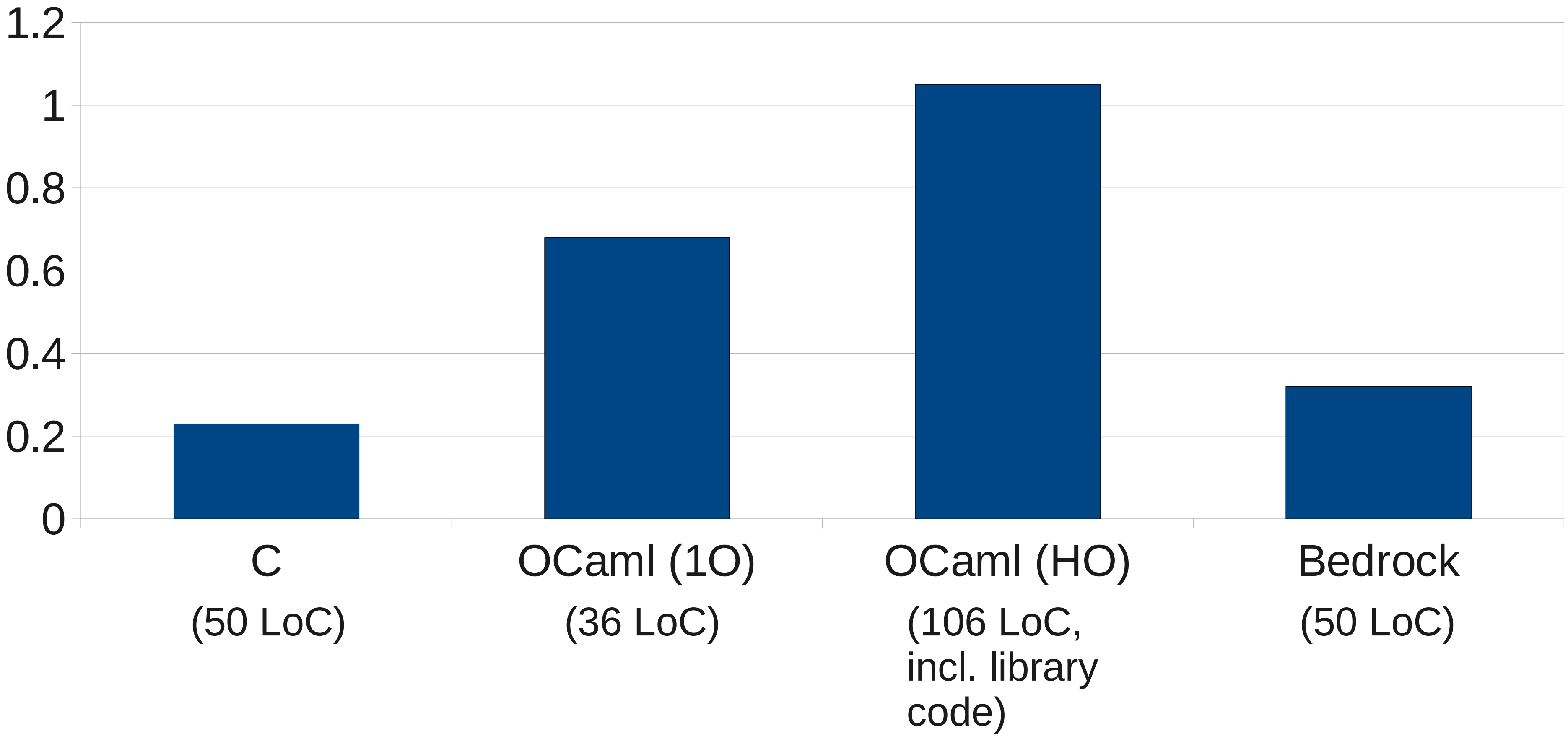
Loop has filter condition that the macro analyzes syntactically to decide on optimizations.

Build toolchain



Running time comparison on a database-inspired benchmark

All programs parse and execute the same set of 200 random queries over a random array of length 100,000.



Bedrock on the web

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