Cambridge, Aug 2016

A **New** Verified Compiler Backend for CakeML

Main contributors to date: Anthony Fox, Ramana Kumar, Magnus Myreen, Michael Norrish, Scott Owens, Yong Kiam Tan

CHALMERS UNIVERSITY OF TECHNOLOGY · UNIVERSITY OF CAMBRIDGE · NICTA · DATA 61 · CSIRO · University of Kent

CakeML

What?

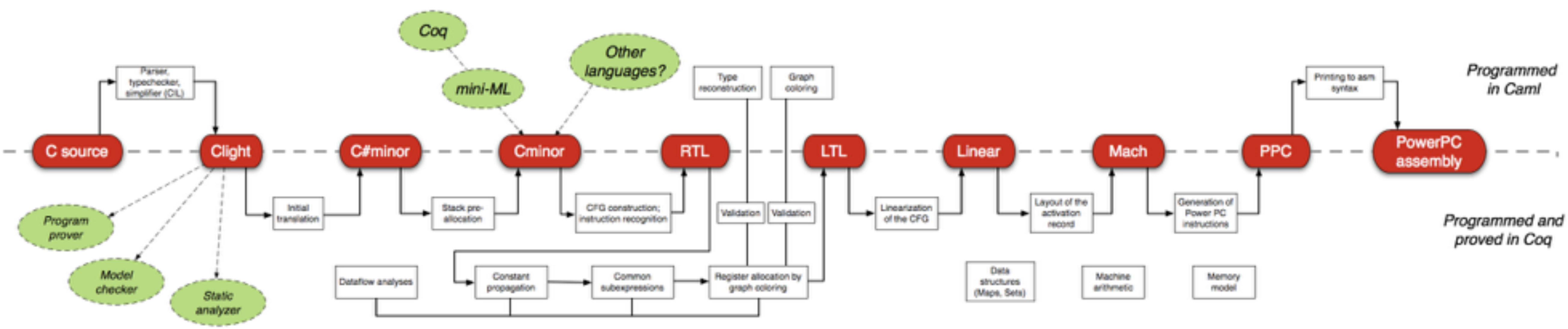
strict evaluation, stateful

1. A *programming language* in the style of Standard ML and OCaml.
2. An *ecosystem* of proofs and verification tools
3. A *verified, end-to-end development*

Verified compilation…

State of the art

CompCert



Leroy et al. ***Source:*** http://compcert.inria.fr/

Compiles C source code to assembly

Good performance numbers (between gcc -O1 and -O2)

Ecosystem: ***Verified Software Toolchain - Princeton University***

Verified compilation…

…for functional languages?

Answer:* *Many, but all are ‘toy’.

Attempt: CakeML first ‘realistic’ verified ML compiler (plus ecosystem).

The CakeML language

was originally

Design: *"The CakeML language ~~is~~ designed to be both easy to program in and easy to reason about formally"*

It is still clean, but not always simple.

CakeML, the language
≈ Standard ML without functors

i.e. with almost everything else:

- ✓ higher-order functions
- ✓ mutual recursion and polymorphism
- ✓ datatypes and (nested) pattern matching
- ✓ references and (user-defined) exceptions
- ✓ arbitrary-precision integers
- ✓ modules, signatures, abstract types

Update! New since POPL'14:

- ✓ foreign-function interface
- ✓ mutable arrays, byte arrays, bytes
- ✓ vectors strings, chars
- ✓ type abbreviations

e

Design:

th easy

ally"

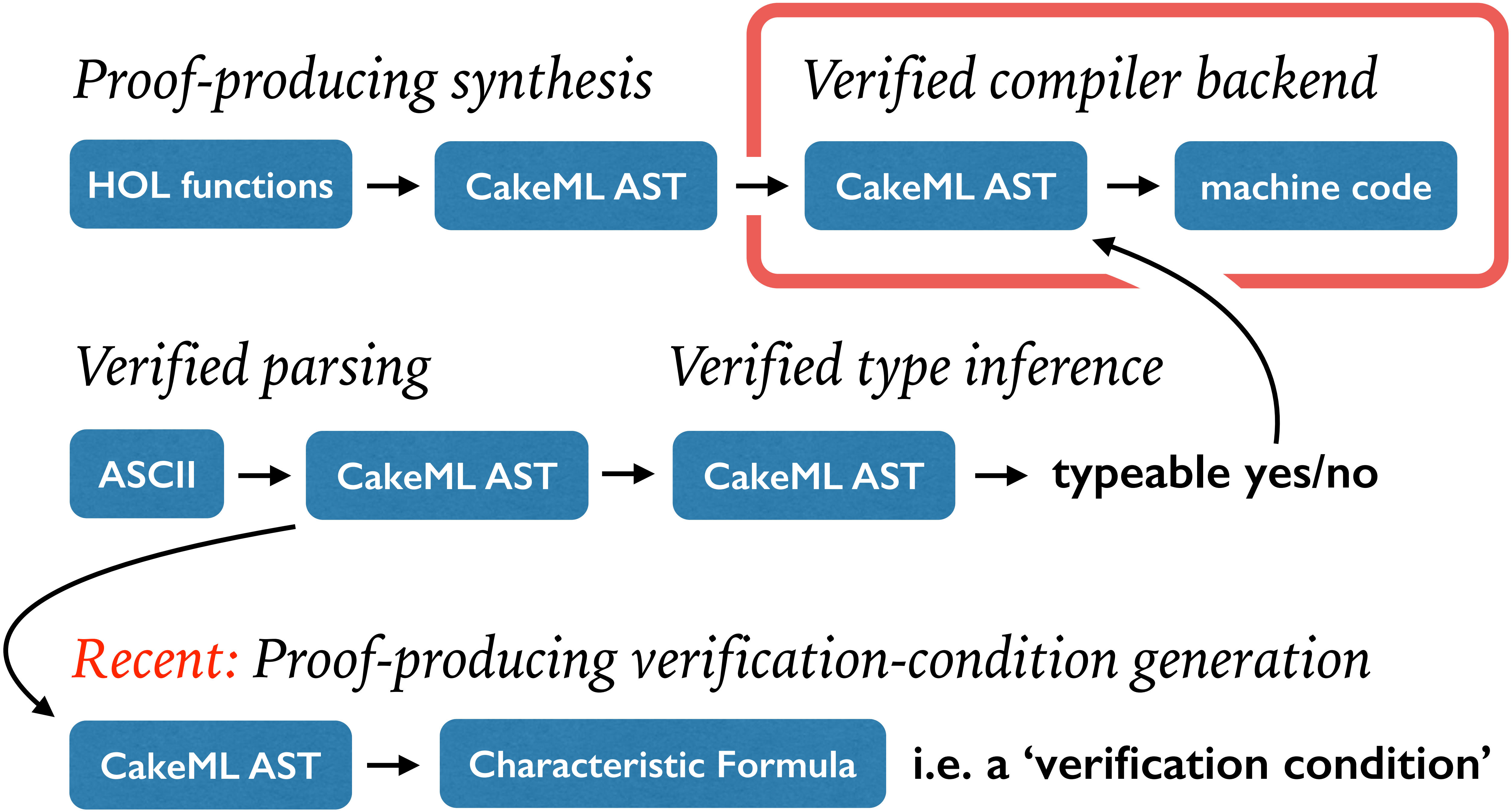
ot always simple.

CakeML, the language
≈ Standard ML without functors

i.e. with almost everything else:

- ✓ higher-order functions
- ✓ mutual recursion and polymorphism
- ✓ datatypes and (nested) pattern matching
- ✓ references and (user-defined) exceptions
- ✓ arbitrary-precision integers
- ✓ modules, signatures, abstract types

Ecosystem



Also: x86 implementation with read-eval-print-loop

This talk: **Compiler verification**

user expectations

gap

observational behaviour of source code

proved connection

modelled behaviour of generated machine code

gap

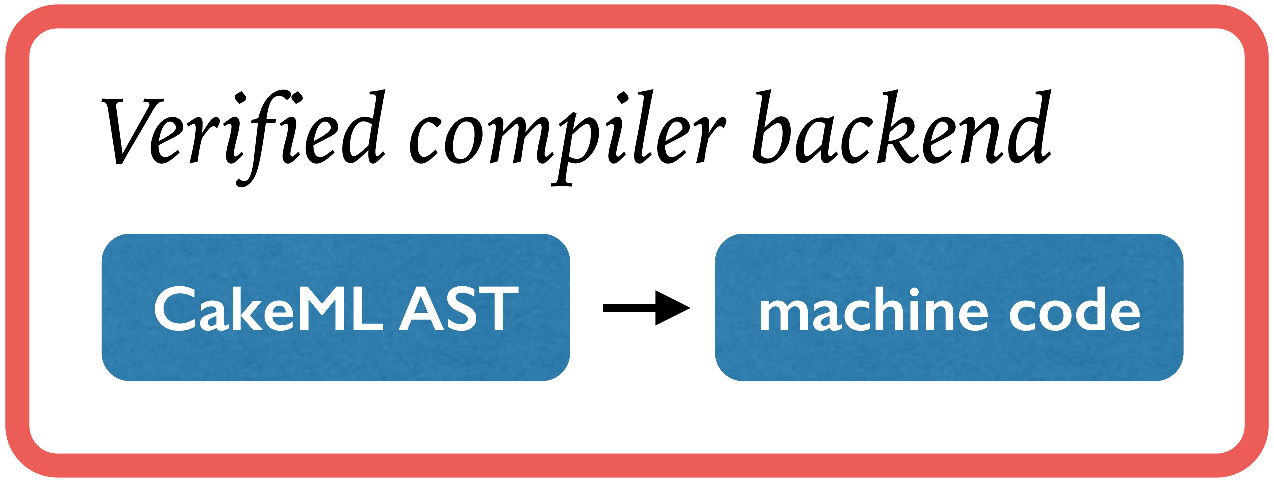
real behaviour of hardware

Verified compiler backend

CakeML AST → machine code

The entire development is in the HOL4 theorem prover.

The CakeML compiler

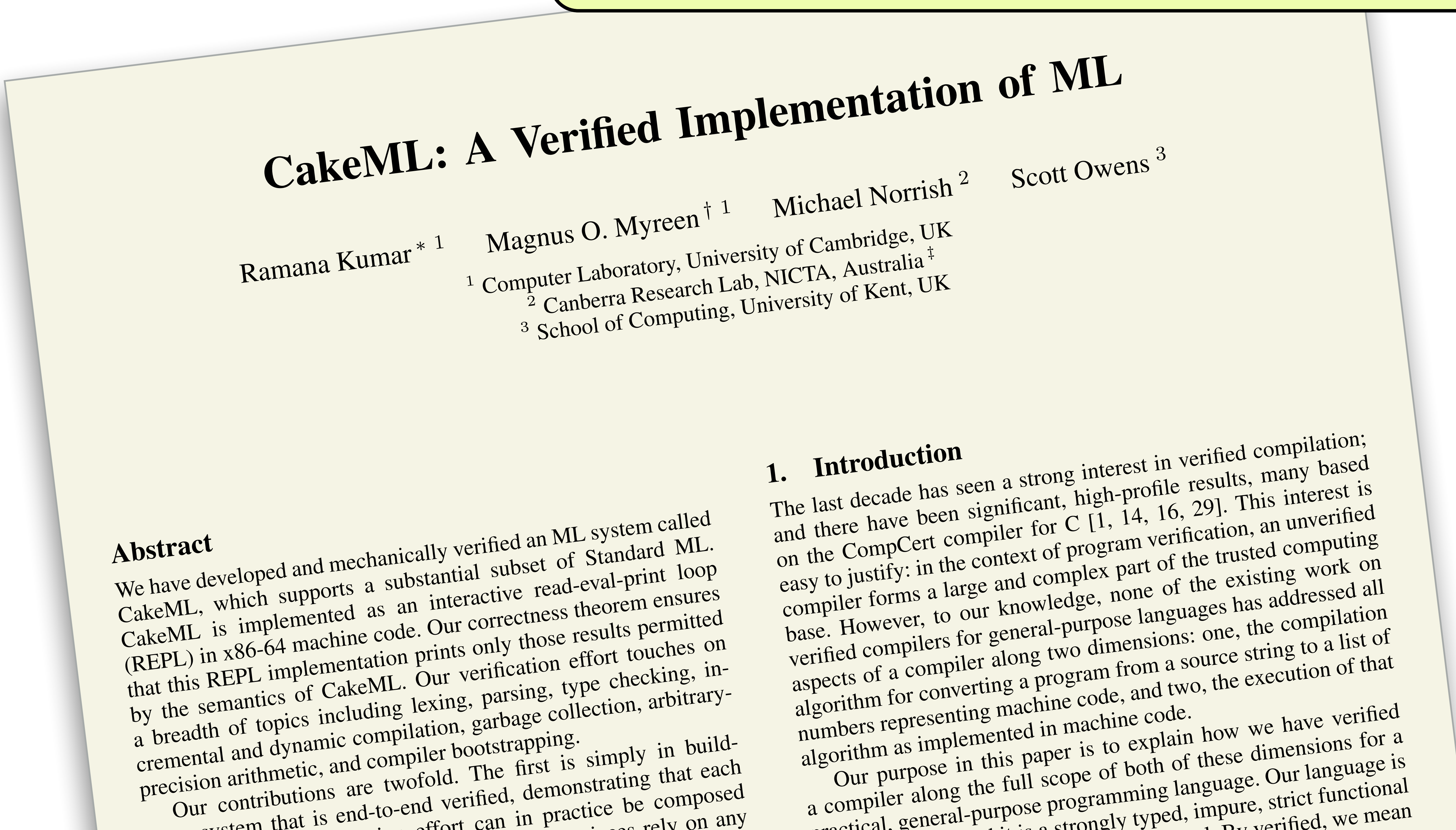


Version 1 & 2

Version 1

POPL'14

First bootstrapping of a verified compiler.



CakeML: A Verified Implementation of ML

Ramana Kumar * [1] Magnus O. Myreen † [1] Michael Norrish [2] Scott Owens [3]

[1] Computer Laboratory, University of Cambridge, UK
[2] Canberra Research Lab, NICTA, Australia ‡
[3] School of Computing, University of Kent, UK

Abstract

We have developed and mechanically verified an ML system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop (REPL) in x86-64 machine code. Our correctness theorem ensures that this REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on a breadth of topics including lexing, parsing, type checking, incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping.

Our contributions are twofold. The first is simply in building a system that is end-to-end verified, demonstrating that each ... effort can in practice be composed ... rely on any

1. Introduction

The last decade has seen a strong interest in verified compilation; and there have been significant, high-profile results, many based on the CompCert compiler for C [1, 14, 16, 29]. This interest is easy to justify: in the context of program verification, an unverified compiler forms a large and complex part of the trusted computing base. However, to our knowledge, none of the existing work on verified compilers for general-purpose languages has addressed all aspects of a compiler along two dimensions: one, the compilation algorithm for converting a program from a source string to a list of numbers representing machine code, and two, the execution of that algorithm as implemented in machine code.

Our purpose in this paper is to explain how we have verified a compiler along the full scope of both of these dimensions for a practical, general-purpose programming language. Our language is ... it is a strongly typed, impure, strict functional ... By verified, we mean

Dimensions of Compiler Verification

source code

abstract syntax

intermediate language

VM bytecode

machine code

how far compiler goes

First verification to cover the full spectrum of **both** dimensions.

compiler algorithm

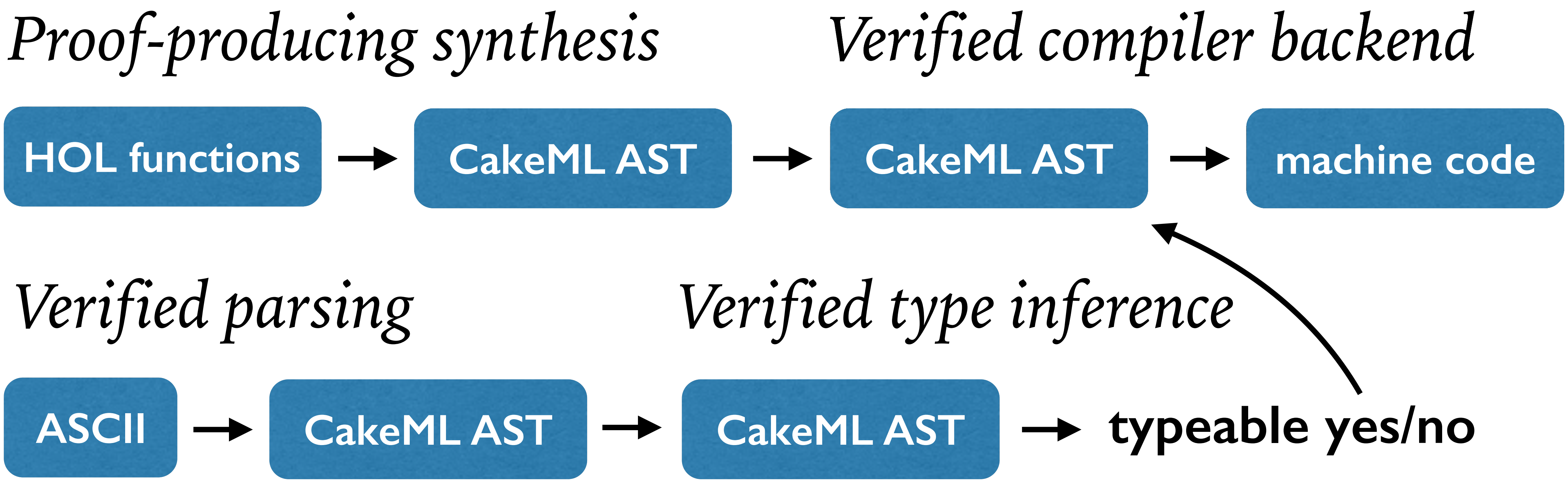
implementation in ML

implementation in machine code

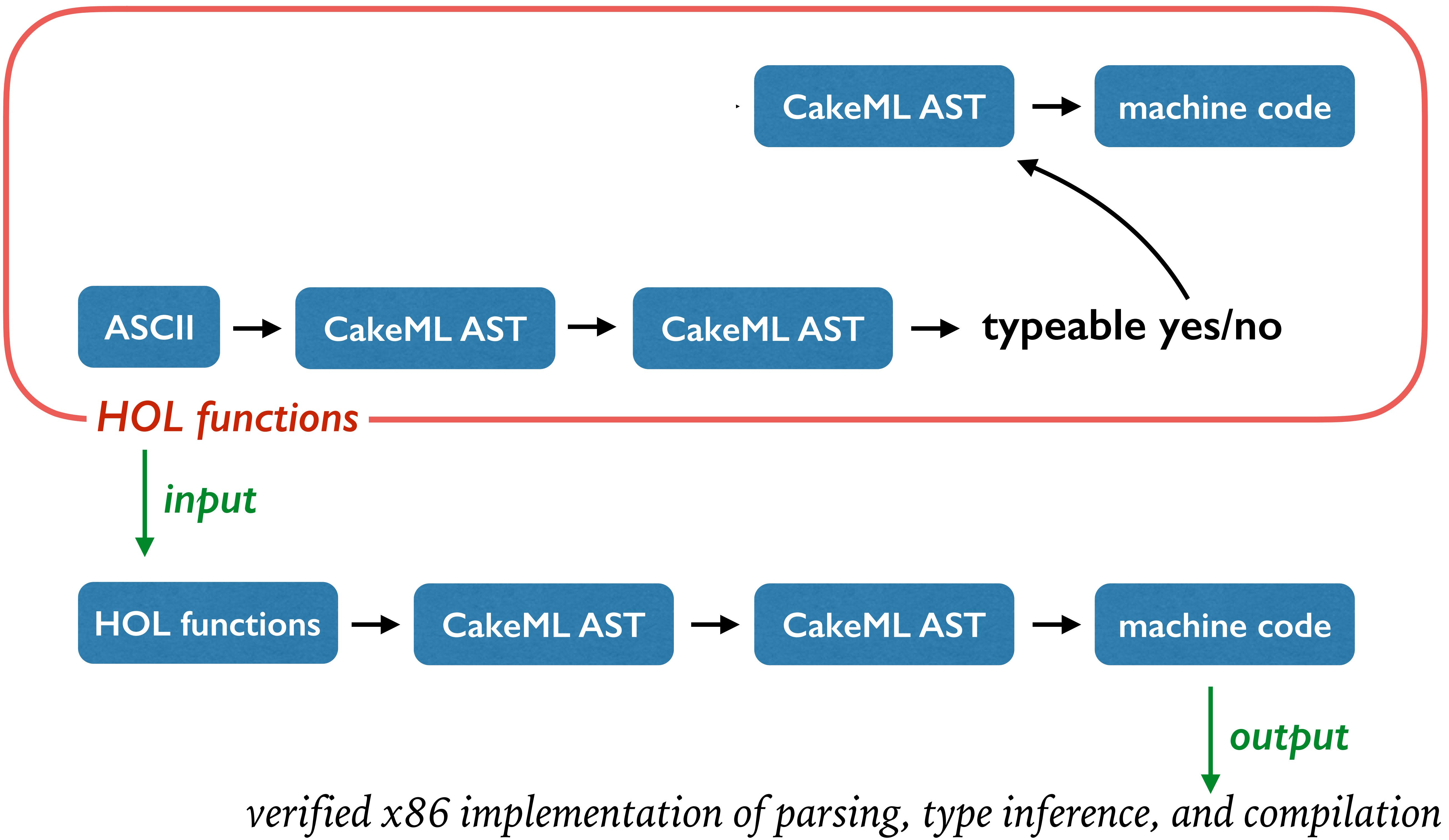
interactive call in read-eval-print loop runtime

the thing that is verified

Intuition for Bootstrapping



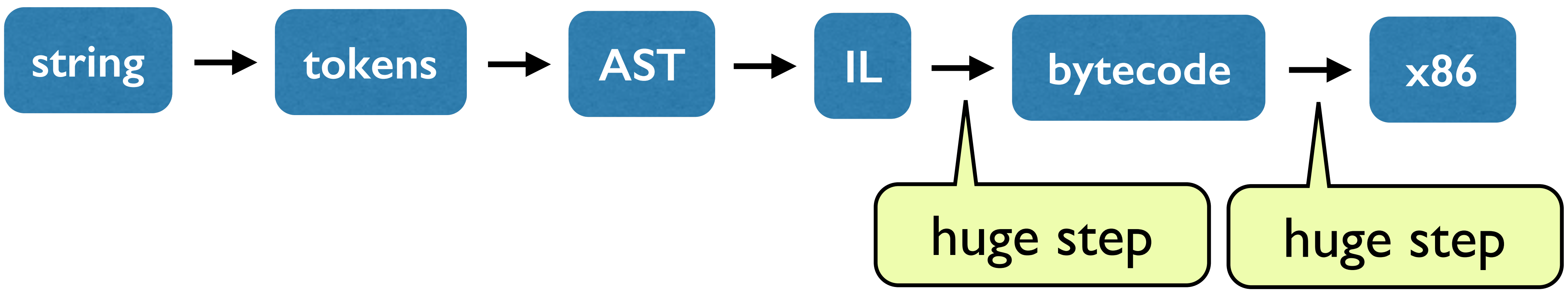
Intuition for Bootstrapping



a very short demo of version 1

Version 1 as in POPL'14

Compiler phases:



Bytecode simplified proofs of read-eval-print loop, but made optimisation impossible.

Almost no optimisations possible…

Poor design.

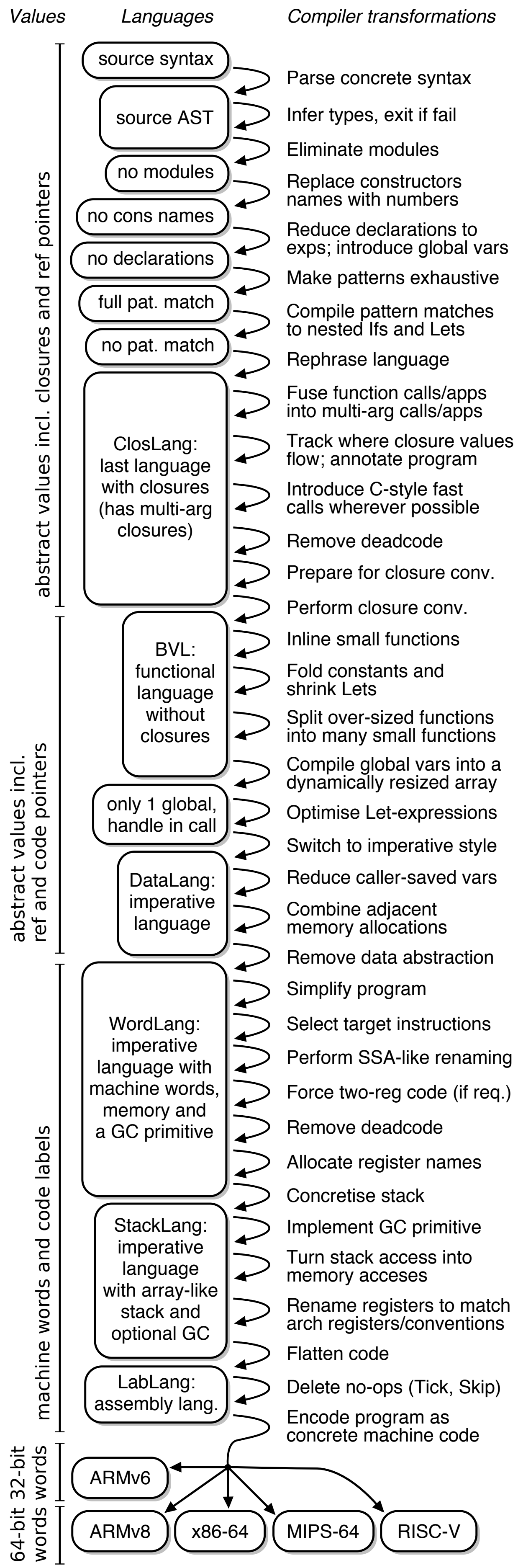
Version 2

Goals:

Design compatible with optimisations.

Acceptable performance.

Strategy: take inspiration from OCaml compiler (for some parts).



All languages communicate with the external world via a byte-array-based foreign-function interface.

(next slides will zoom in)

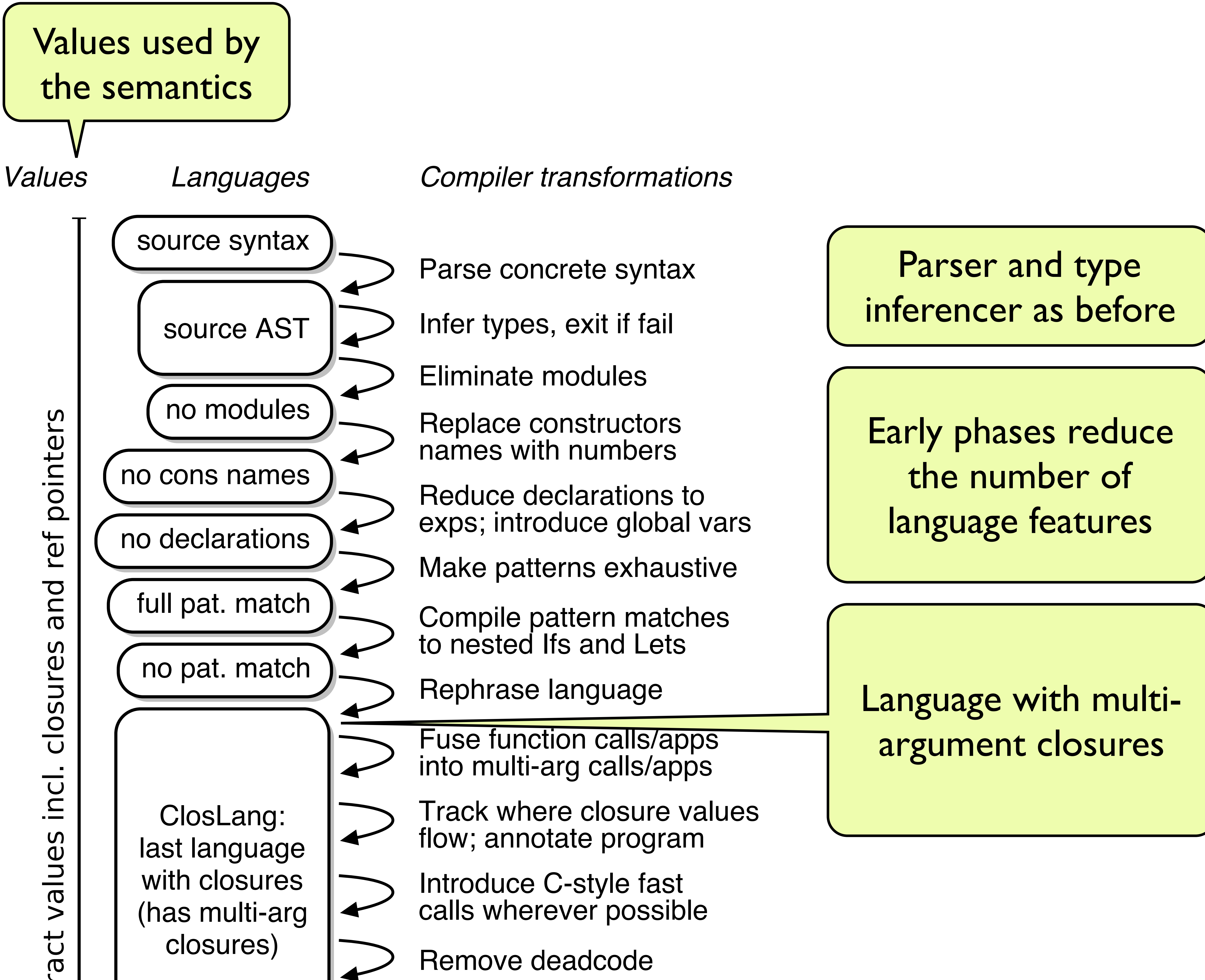
Result:

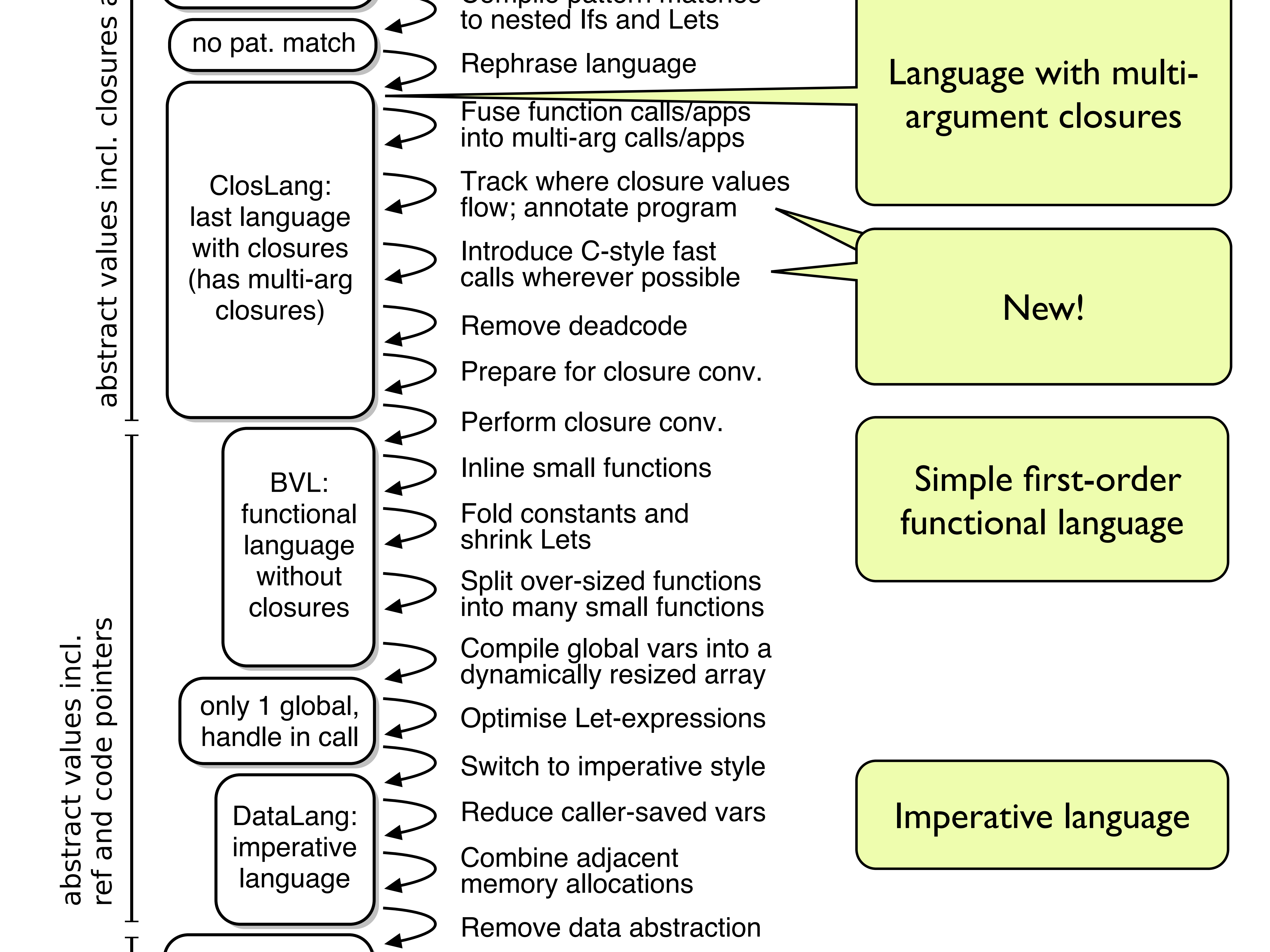
12 intermediate languages (ILs)

and many within-IL optimisations

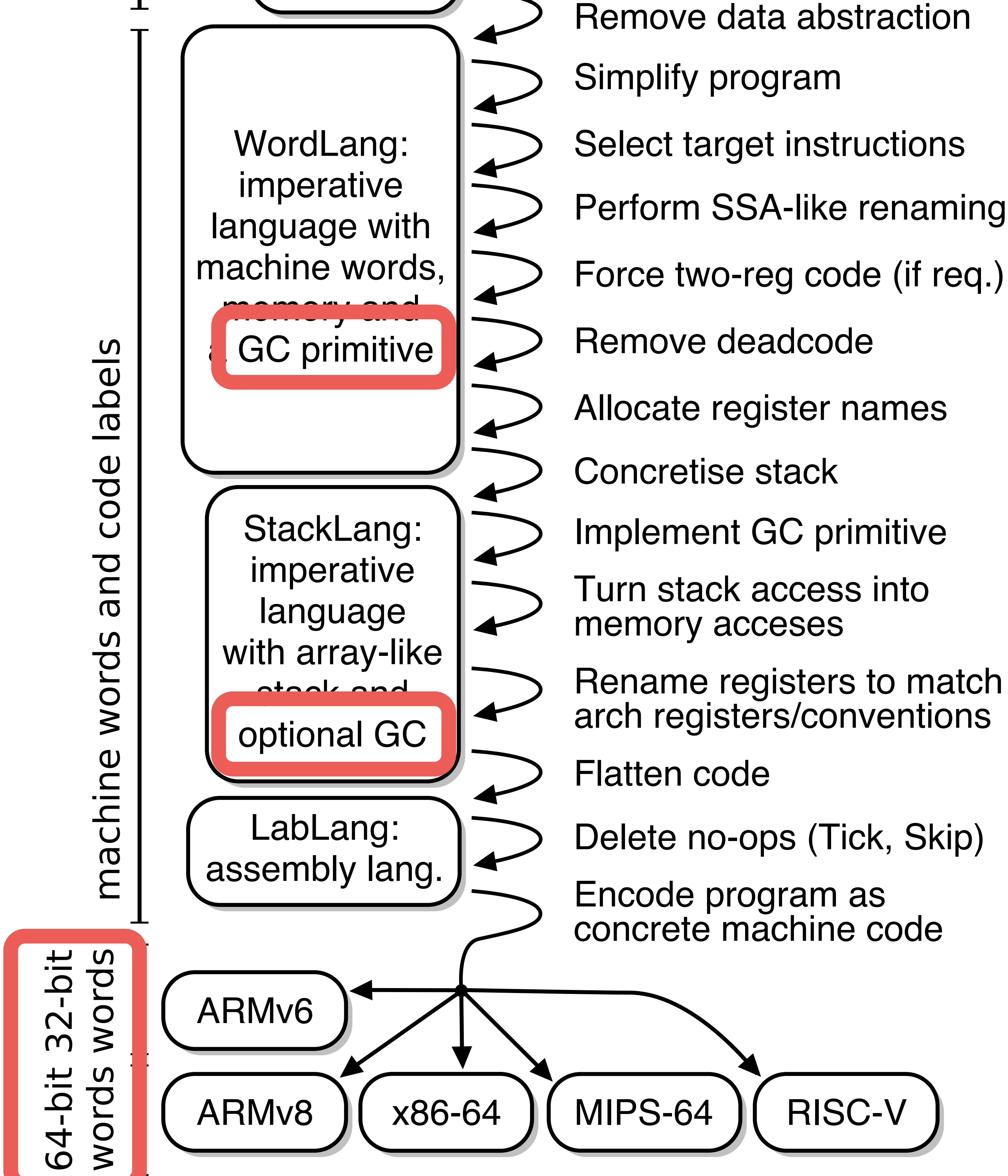
each IL at the right level of abstraction

for the benefit of proofs and compiler implementation


Values used by the semantics
Values
Languages
Compiler transformations
act values incl. closures and ref pointers
source syntax
Parse concrete syntax
source AST
Infer types, exit if fail
Eliminate modules
no modules
Replace constructors names with numbers
no cons names
Reduce declarations to exps; introduce global vars
no declarations
Make patterns exhaustive
full pat. match
Compile pattern matches to nested Ifs and Lets
no pat. match
Rephrase language
ClosLang: last language with closures (has multi-arg closures)
Fuse function calls/apps into multi-arg calls/apps
Track where closure values flow; annotate program
Introduce C-style fast calls wherever possible
Remove deadcode
Parser and type inferencer as before
Early phases reduce the number of language features
Language with multi-argument closures



abstract values incl. closures a
abstract values incl.
ref and code pointers
no pat. match
ClosLang:
last language
with closures
(has multi-arg
closures)
BVL:
functional
language
without
closures
only 1 global,
handle in call
DataLang:
imperative
language
to nested Ifs and Lets
Rephrase language
Fuse function calls/apps
into multi-arg calls/apps
Track where closure values
flow; annotate program
Introduce C-style fast
calls wherever possible
Remove deadcode
Prepare for closure conv.
Perform closure conv.
Inline small functions
Fold constants and
shrink Lets
Split over-sized functions
into many small functions
Compile global vars into a
dynamically resized array
Optimise Let-expressions
Switch to imperative style
Reduce caller-saved vars
Combine adjacent
memory allocations
Remove data abstraction
Language with multi-
argument closures
New!
Simple first-order
functional language
Imperative language



Machine-like types

Standard compiler for an imperative lang *with a few FP twists:* garbage collector, live-var annotations, fast exception mechanisms

Targets 5 architectures

All languages communicate with the external world via a byte-array-based foreign-function interface.

Latest developments

Optimisation of function calls

```
fun reverse xs = let
    fun append xs ys =
      case xs of [] => ys
      | (x::xs) => x :: append xs ys;
    fun rev xs =
      case xs of [] => xs
      | (x::xs) => append (rev xs) [x]
  in rev xs end;
val example = reverse [1,2,3];
```

Latest developments

Optimisation of function calls

```
set_global 0 (fn xs => let
     fun append xs = fn ys =>
       if xs = [] then ys else
         el 0 xs :: (append (el 1 xs)) ys
     fun rev xs =
       if xs = [] then xs else
         (append (rev (el 1 xs))) [el 0 xs]
   in rev xs end);
set_global 1 ((get_global 0) [1,2,3]);
```

Latest developments

Optimisation of function calls

```
set_global 0 (fn₄ xs => let
    fun append₀ ⟨xs,ys⟩ =
      if xs = [] then ys else
        el 0 xs :: append⁰ ⟨el 1 xs, ys⟩
    fun rev₂ xs =
      if xs = [] then xs else
        append⁰ ⟨rev² (el 1 xs), [el 0 xs]⟩
  in rev² xs end);
set_global 1 ((get_global 0)⁴ [1,2,3]);
```

Latest developments

Optimisation of function calls

```
set_global 0 (fn xs => Call 5 ⟨xs⟩);
set_global 1 (Call 5 [1,2,3]);
```

Code Table:

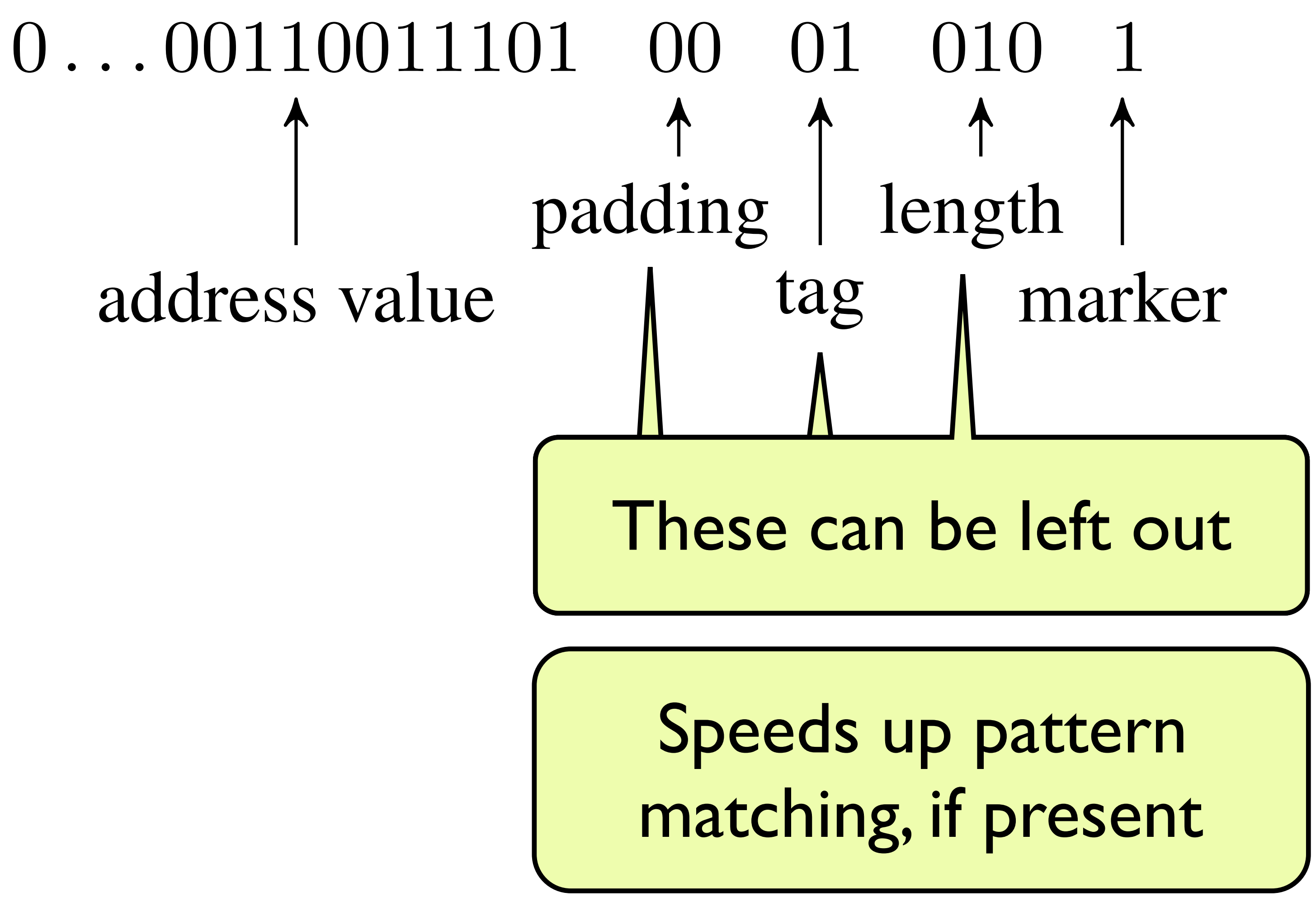
```
1 ⟨xs,ys⟩ => if xs = [] then ys else
                 el 0 xs :: Call 1 ⟨el 1 xs, ys⟩

3 ⟨xs⟩ => if xs = [] then xs else
           Call 1 ⟨Call 3 (el 1 xs), [el 0 xs]⟩

5 ⟨xs⟩ => let
     val append = 0
     val rev = 0
   in Call 3 ⟨xs⟩ end
```

C-like function calls

Pointers

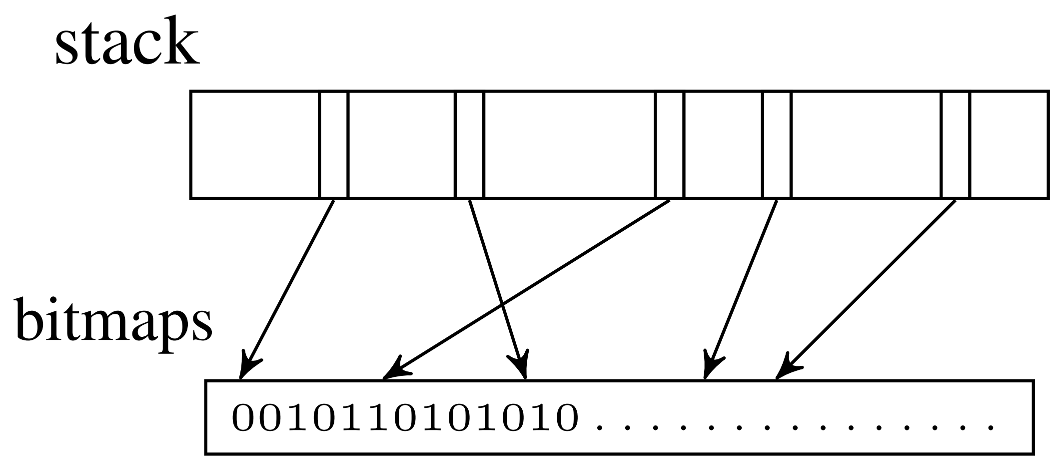
Configurable data representation

Example pointer value:

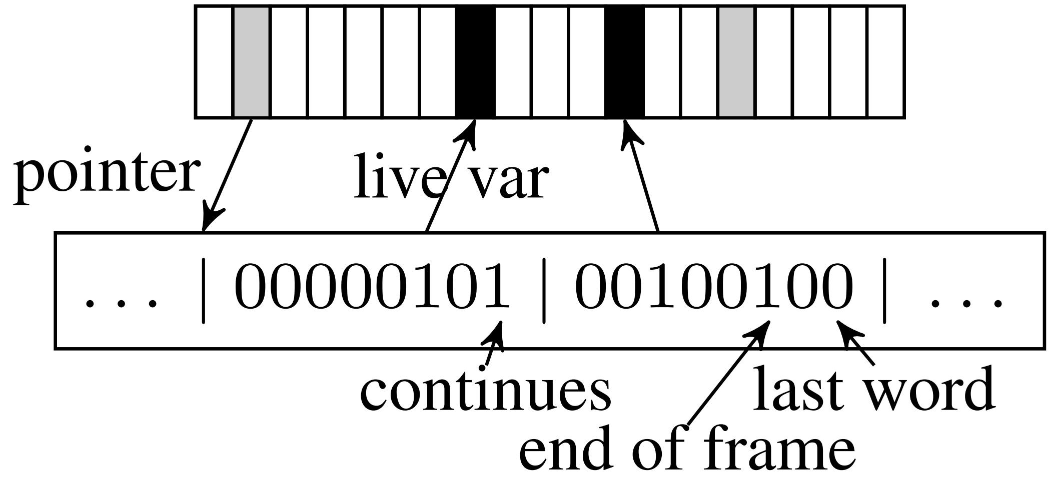


Stack

Stack contains information about live vars for the GC



Details of one stack frame:



Semantics & Proofs

Semantic values

Immediately before closure conversion:

$$
\begin{array}{ll}
\texttt{v} = & \\
& \textsf{Number}\ \texttt{int} \\
| & \textsf{Block}\ \texttt{num}\ (\texttt{v list}) \\
| & \textsf{RefPtr}\ \texttt{num} \\
| & \textsf{Closure}\ (\texttt{num option})\ (\texttt{v list})\ (\texttt{v list})\ \texttt{num}\ \texttt{exp} \\
| & \textsf{Recclosure}\ (\texttt{num option})\ (\texttt{v list})\ (\texttt{v list})\ \ldots
\end{array}
$$

Immediately after closure conversion:

$$\texttt{v} = \textsf{Number}\ \texttt{int} \mid \textsf{Block}\ \texttt{num}\ (\texttt{v list}) \mid \textsf{CodePtr}\ \texttt{num} \mid \textsf{RefPtr}\ \texttt{num}$$

Closures are values with a code pointer:

Block closure_tag
([CodePtr ptr; Number arg_count] @ $free\_var\_vals$)

For mutually recursive closures:

Block closure_tag
[CodePtr ptr; Number arg_count; RefPtr ref_ptr]

Semantics

Each intermediate language has a formal semantics.

We define these using a ***functional big-step style*** (ESOP'16) where the semantics is an evaluation function in logic

Extract of abstract first-order lang:

evaluate $([\text{Var } n], env, s) =$
 if $n <$ len env then (Rval [nth n env], s)
 else (Rerr (Rabort Rtype_error), s)

Observable semantics defined using evaluate (on later slide).

Semantics (cont.)

What about *infinite loops*?

evaluate ([Call $ticks$ $dest$ xs],env,s_1) =
  case evaluate (xs,env,s_1) of
    (Rval vs,s) $\Rightarrow$
      (case find_code $dest$ vs s.code of
        None $\Rightarrow$ (Rerr (Rabort Rtype_error),s)
      | Some ($args$,exp) $\Rightarrow$
          if s.clock $<$ $ticks$ $+ 1$ then
            (Rerr (Rabort Rtimeout_error),s with clock := 0)
          else evaluate ([exp],$args$,dec_clock ($ticks + 1$) s))
  | (Rerr e,s) $\Rightarrow$ (Rerr e,s)

A simple clock mechanism ensures evaluation always terminates.

Clock only decremented when necessary for termination.

Observational semantics

We define the obs. semantics using evaluate.

semantics_prog st env $prog$ (Terminate $outcome$ io_list) $\iff$
 $\exists\, k\ \mathit{ffi}\ r.$
  evaluate_prog_with_clock st env k $prog$ = (ffi,r) $\wedge$
  (if ffi.final_event = None then
   ($\forall\, a.\ r \neq$ Rerr (Rabort a)) $\wedge$ $outcome$ = Success
  else $outcome$ = FFI_outcome (THE ffi.final_event)) $\wedge$
  io_list = ffi.io_events
semantics_prog st env $prog$ (Diverge io_trace) $\iff$
 ($\forall\, k.$
  $\exists\, \mathit{ffi}.$
   evaluate_prog_with_clock st env k $prog$ =
    (ffi,Rerr (Rabort Rtimeout_error)) $\wedge$
   ffi.final_event = None) $\wedge$
 lprefix_lub
  (IMAGE
   ($\lambda\, k.$
    fromList
     (fst (evaluate_prog_with_clock st env k $prog$)).
     io_events) $\mathcal{U}$(: num)) io_trace
semantics_prog st env $prog$ Fail $\iff$
 $\exists\, k.$
  snd (evaluate_prog_with_clock st env k $prog$) =
   Rerr (Rabort Rtype_error)

Terminates if we can reach a result for some clock.

Diverges if evaluation times out for every initial clock value.

Fails on internal error. *This is ruled out by successful type inference.*

Proof

Proof styles:

Standard induction on evaluation function

✓ proofs in direction of compilation

✓ no co-induction needed for divergence pres. (ESOP'16)

Untyped logical relation (ind. on compile function)

Each part of the compiler preserves obs. semantics:

$\vdash$ compile $config\ prog = new\_prog\ \wedge$
syntactic_condition $prog\ \wedge$
Fail $\notin$ semantics $ffi\ prog \Rightarrow$
semantics $ffi\ new\_prog \subseteq$
extend_with_resource_limit (semantics $ffi\ prog$)

type-safe source implies this

due to out-of-memory error

Proof details

The obs. sem. theorems are proved using this about evaluate.

All evaluate *proofs are of the form:*

source IL

target IL

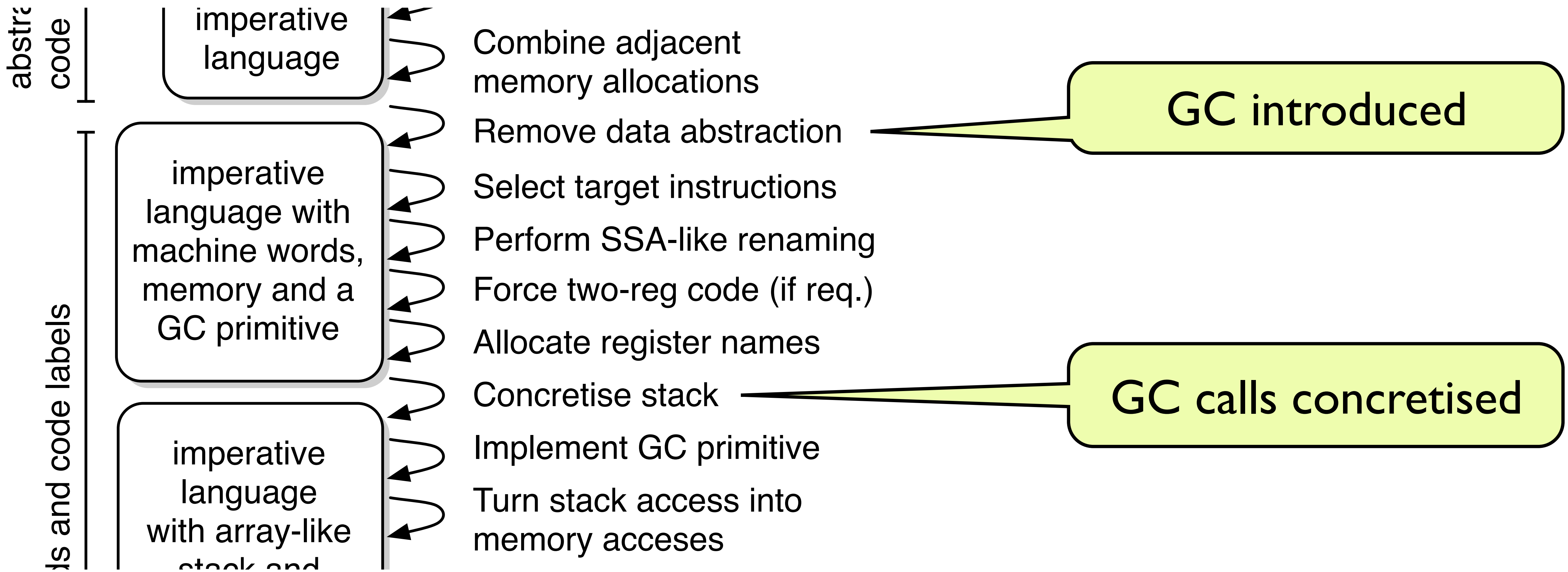
$$
\begin{array}{l}
\vdash \textsf{compile}\ \mathit{config}\ \mathit{exp} = \mathit{exp}_1 \wedge \\
\quad \textsf{evaluate}\ \mathit{exp}\ \mathit{state} = (\mathit{new\_state},\mathit{res}) \wedge \\
\quad \textsf{state\_rel}\ \mathit{state}\ \mathit{state}_1 \wedge \mathit{res} \neq \textsf{Error} \Rightarrow \\
\quad\quad \exists\, \mathit{new\_state}_1\ \mathit{res}_1 . \\
\quad\quad\quad \textsf{evaluate}\ \mathit{exp}_1\ \mathit{state}_1 = (\mathit{new\_state}_1,\mathit{res}_1) \wedge \\
\quad\quad\quad \textsf{state\_rel}\ \mathit{new\_state}\ \mathit{new\_state}_1 \wedge \textsf{res\_rel}\ \mathit{res}\ \mathit{res}_1
\end{array}
$$

Informally: the compiler produces code which *simulates* the original.

No co-induction required.

Difficult cases

GC and register allocator interaction



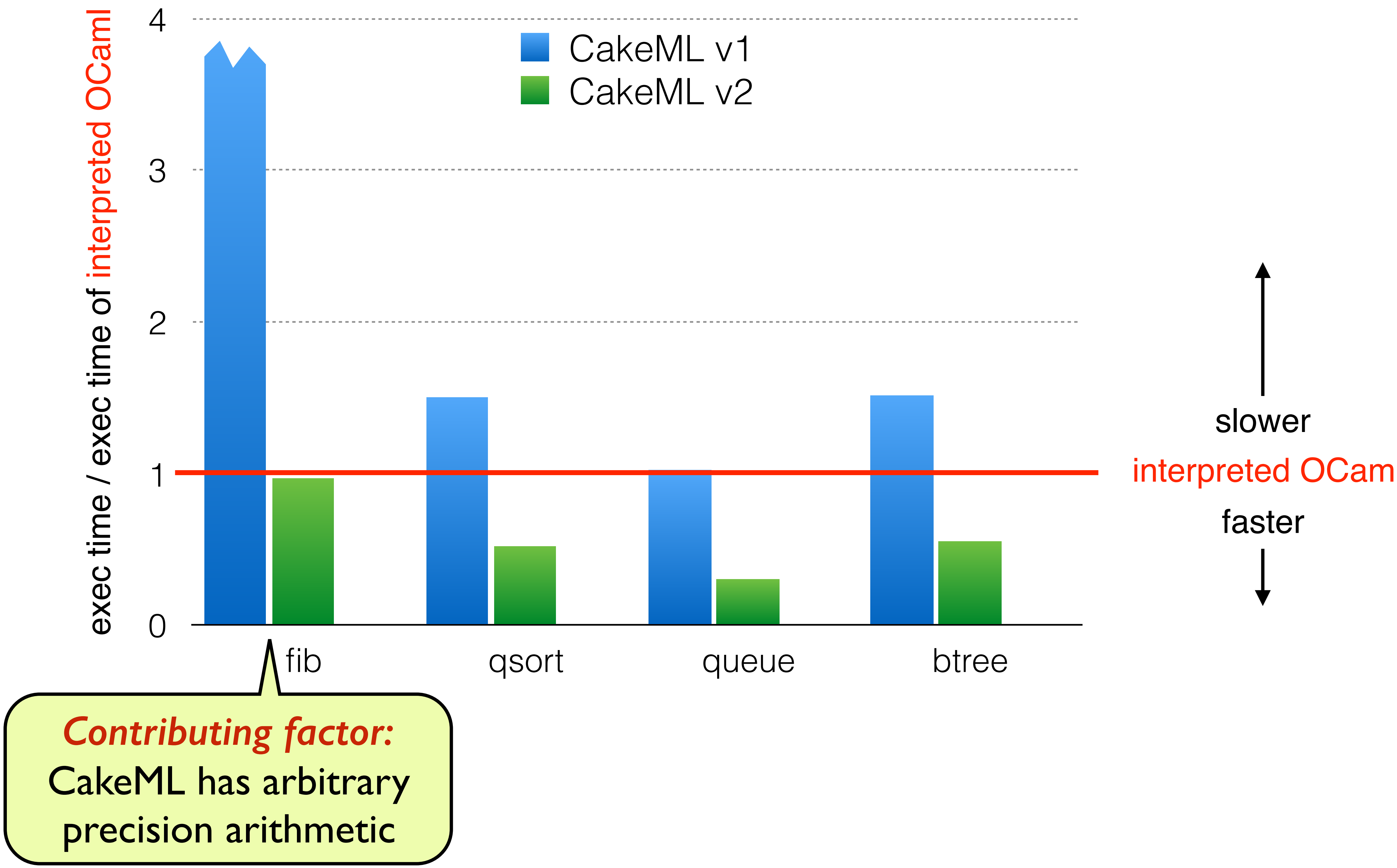
Solution: we use a semantics that allows reordering of stack variables.

Size, Effort, Speed

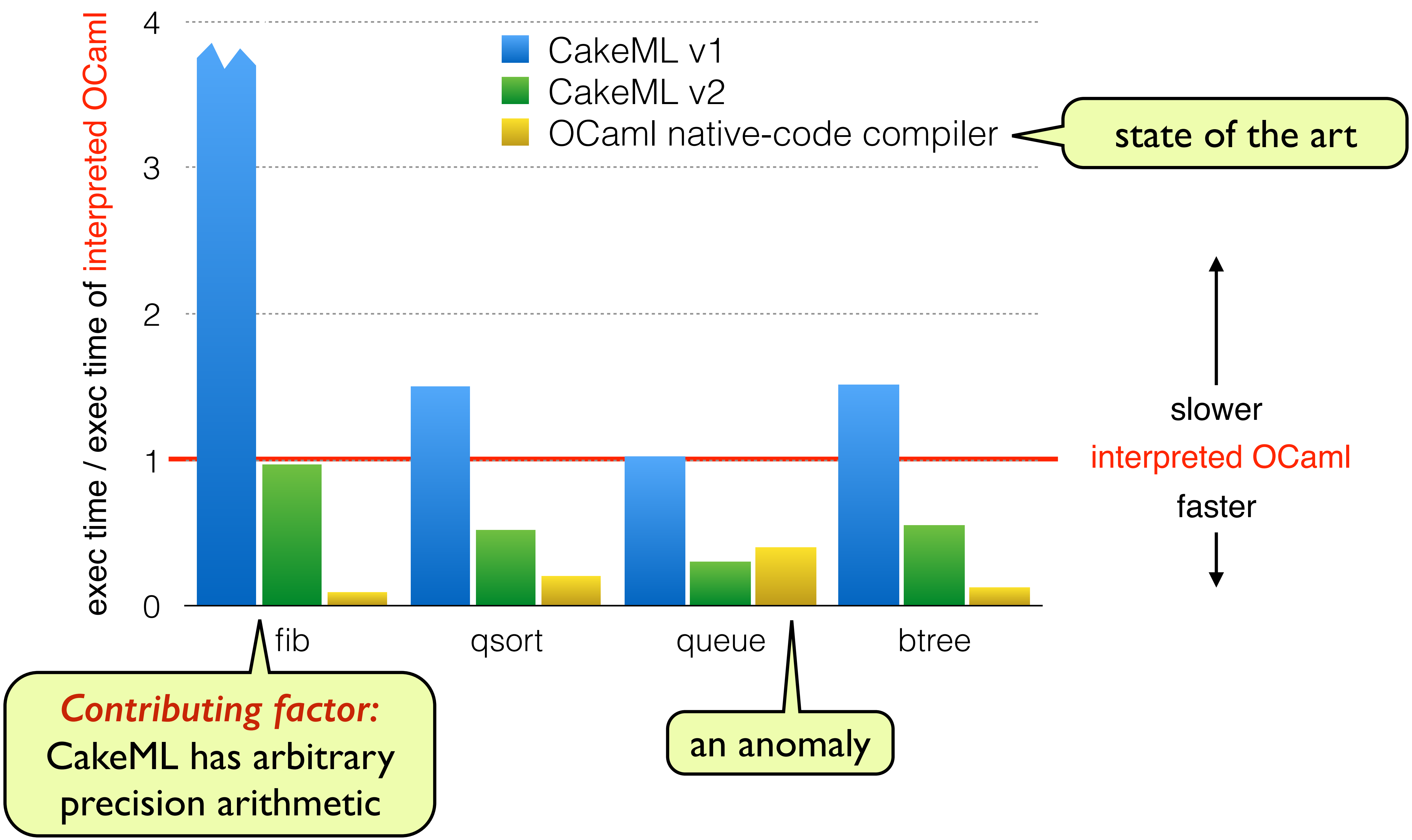
Compiler Size:	6 000 lines of function definitions (excludes target-specific instruction encoders & config.)
Proof Size:	100 000 lines of HOL4 proof scripts
Effort:	6 people, 3 years, but not full time
Speed:	next slide…

(Numbers up-to-date as of Aug 2016)

Simple Benchmarks



Simple Benchmarks



Simple Benchmarks

Why?

Version 1 *can compile big programs* (in-logic)

Version 2 *in-logic* evaluation is ***too slow*** for large examples

why not outside?

we are working to improve this

We will be able to compile large programs once v2 is bootstrapped.

CakeML

This talk: ***New compiler's design*** compatible with ***optimisations***

Big-picture: ***Ecosystem*** around a clean formalised ML language

Why? ***End-to-end verification,*** and end-to-end verified applications

Questions?

Magnus Myreen

Yong Kiam Tan

Ramana Kumar

Anthony Fox

Scott Owens

Michael Norrish