

Computational Reflection, Theorem Provers and Verification

Lecture 4:

*Compiler Bootstrapping and
Verified HOL ITP*

Marktoberdorf Summer School MOD 2019

Magnus O. Myreen, Chalmers University of Technology

So far

Monday's lecture:

- ✓ Covered soundness-centric LCF-style design

Tuesday's lecture:

- ✓ Introduced Milawa with its alternative design
- ✓ Milawa's implementation in Lisp

Wednesday's lecture:

- ✓ Verification proof of Milawa

Thursday's exercises: hacking in Jitawa Lisp

At ITP conference

after presenting Milawa work



Freek Wiedijk
Radboud University Nijmegen

Please, do the same for
HOL Light!

My immediate response:

That would be difficult...

Later thought:

Maybe...

My thoughts on Milawa / Jitawa

they scale but are still only
proof-of-concept implementations

- ➔ A pity that Milawa and Jitawa are “toy” systems
- ➔ I prefer to program in typed languages — like SML
- ➔ Jitawa: *a technical solution bugged me* (next slide)

Looking back...

The bytecode in Jitawa includes:

Fail	signal a runtime error
Print	print an object to stdout
Compile	compile a function definition

Compile and install new code!

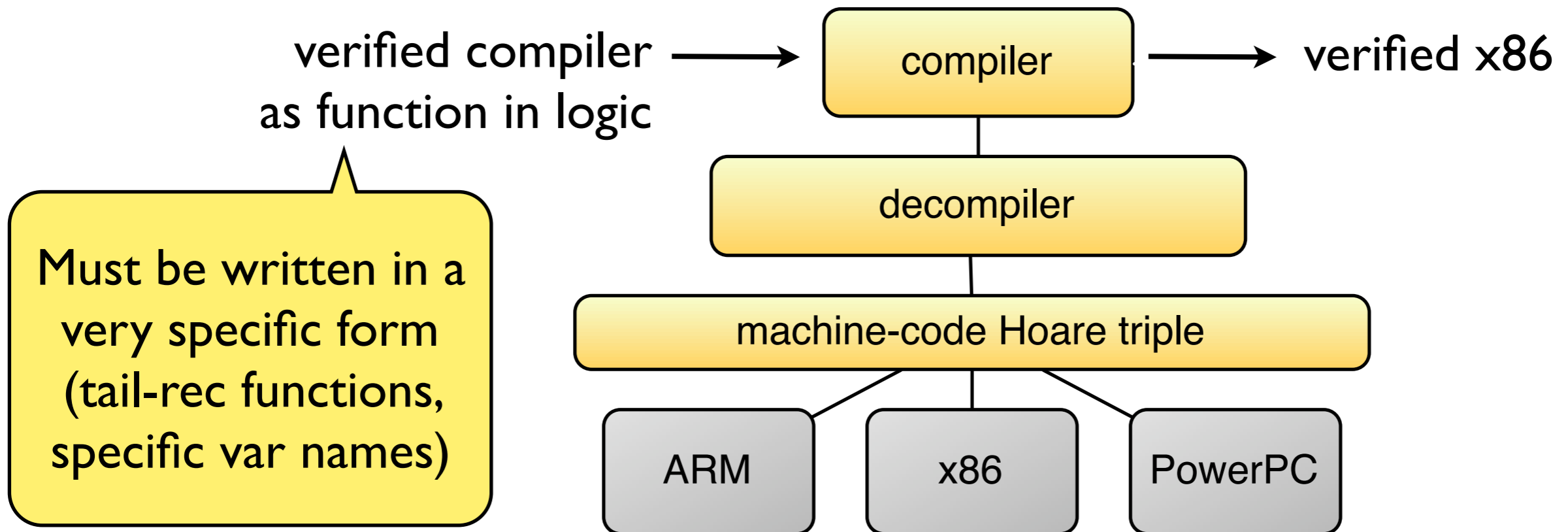
... and:

- bytecode is represented by numbers in memory that are x86 machine code
- we prove that jumping to the memory location of the bytecode executes it

Machine code must implement compiler

Looking back...

The x86 for the compile function was produced as follows:



Very cumbersome....

Looking back...

Two compilers. We should only need one!

verified compiler
as function in logic

compiler

verified x86

decompiler

machine-code Hoare triple

ARM

x86

PowerPC

Must be written in a very specific form (tail-rec functions, specific var names)

Compiler bootstrapping

Very cumbersome....

...should have compiled the verified compiler using itself!

Idea for:



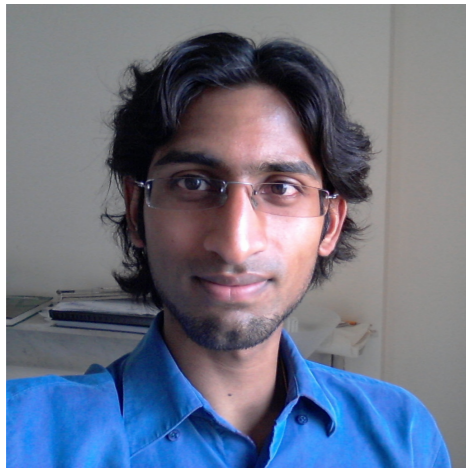
CakeML

A Verified Implementation of ML

Aim: a *realistic* verified implementation of ML, and
an ecosystem of proofs and tools built around the language

<https://cakeml.org/>

Initial effort (2012-2014)



Ramana Kumar
(Uni. Cambridge)



Michael Norrish
(NICTA, ANU)

operational *semantics*

verified *compilation* from
source to bytecode

verified *type* inference

verified *parsing* (syntax is
compatible with SML)

verified *x86* implementations

proof-producing *code*
generation from HOL



Scott Owens
(Uni. Kent)



Magnus Myreen
(Uni. Cambridge)

We also wanted to try out:

Compiler Bootstrapping inside ITP

Requires a change of program representation (i.e. reflection)

concrete syntax



SML parser



type inferencer



compiler backend



machine code

An ML compiler
in logic



= verified function in logic

Idea for Compiler Bootstrapping

a function in the logic

concrete syntax

SML parser

type inferencer



AST

compiler backend

machine code

function in the logic

synthesise AST

-  = verified function in logic
-  = proof-producing tool

Idea for Compiler Bootstrapping

If we input
the CakeML compiler ...

function in the logic

synthesise AST

AST

compiler backend

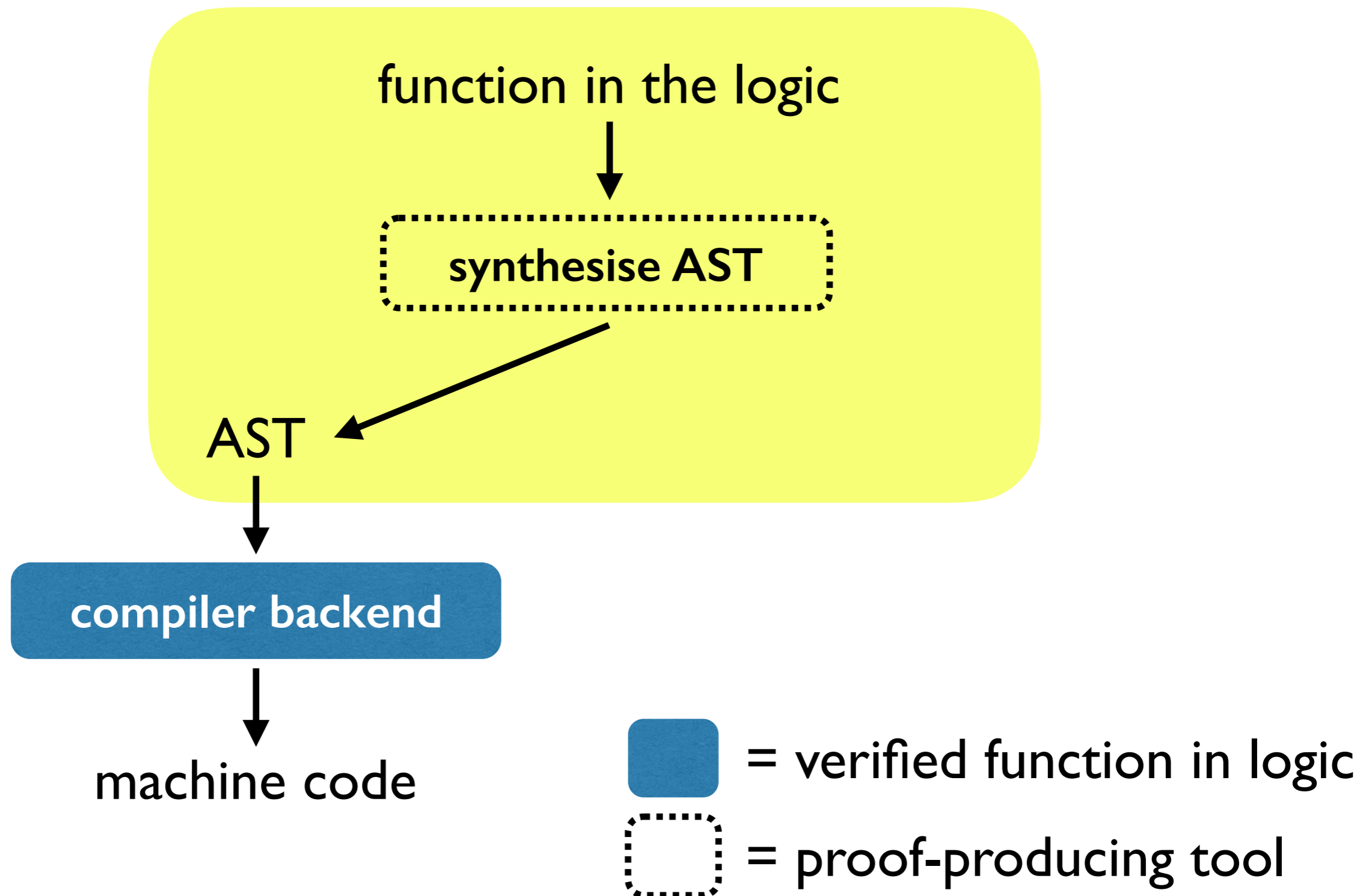
machine code

*Benefit: no need for code
extraction as used in CompCert*

*... then mach. code behaves like
the CakeML compiler function.*

■ = verified function in logic
□ = proof-producing tool

Change of representation



Proof-producing translation

Key definition:

CakeML exp evaluates to value v

$$\text{Eval } env \ exp \ post = \exists v. \langle env, exp \rangle \Downarrow v \wedge post \ v$$

big-step semantics, total-correctness

Example:

This states: CakeML expression 1 relates to ...

$$\text{Eval } env \ [1] \ (\text{int } 1) \ \dots \text{ integer } 1 \text{ in the logic.}$$

where $\text{int } i = (\lambda v. v = \text{Litv } (\text{IntLit } i))$

Automation

Each stage automation proves:

$assumptions \Rightarrow Eval\ env\ code\ (ref_inv\ t)$

Examples:

$\vdash T \Rightarrow Eval\ env\ [1]\ (int\ 1)$

$\vdash Eval\ env\ [x]\ (int\ x) \Rightarrow Eval\ env\ [x]\ (int\ x)$

Automation

Examples:

$$\vdash \top \Rightarrow \text{Eval env } [1] \text{ (int 1)}$$

$$\vdash \text{Eval env } [x] \text{ (int } x) \Rightarrow \text{Eval env } [x] \text{ (int } x)$$

Lemmas ...

$$\begin{aligned} &\vdash \text{Eval env } x_1 \text{ (int } n_1) \Rightarrow \\ &\quad \text{Eval env } x_2 \text{ (int } n_2) \Rightarrow \\ &\quad \text{Eval env } [x_1 + x_2] \text{ (int } (n_1 + n_2)) \end{aligned}$$

... are used to prove compound terms:

$$\vdash \text{Eval env } [x] \text{ (int } x) \Rightarrow \text{Eval env } [x + 1] \text{ (int } (x + 1))$$

Closures are similar

Lemma for lambda:

$$\vdash (\forall v x. a \ x \ v \Rightarrow \text{Eval } (env \ [n \mapsto v]) \ body \ (b \ (f \ x))) \Rightarrow \\ \text{Eval } env \ [\text{fn } n \Rightarrow body] \ ((a \longrightarrow b) \ f)$$

relation based on relations for input (a) and output (b)

Allows us to prove:

$$\vdash T \Rightarrow \text{Eval } env \ [\text{fn } x \Rightarrow x + 1] \ ((\text{int} \longrightarrow \text{int}) \ (\lambda x. x + 1))$$

... from result from previous slide, i.e.

$$\vdash \text{Eval } env \ [x] \ (\text{int } x) \Rightarrow \text{Eval } env \ [x + 1] \ (\text{int } (x + 1))$$

Tool

Implemented in HOL4 as a proof-producing tool

Easy to implement

Handles pure ML-like subset of higher-order logic

Scales to full CakeML compiler

Also effects and I/O:

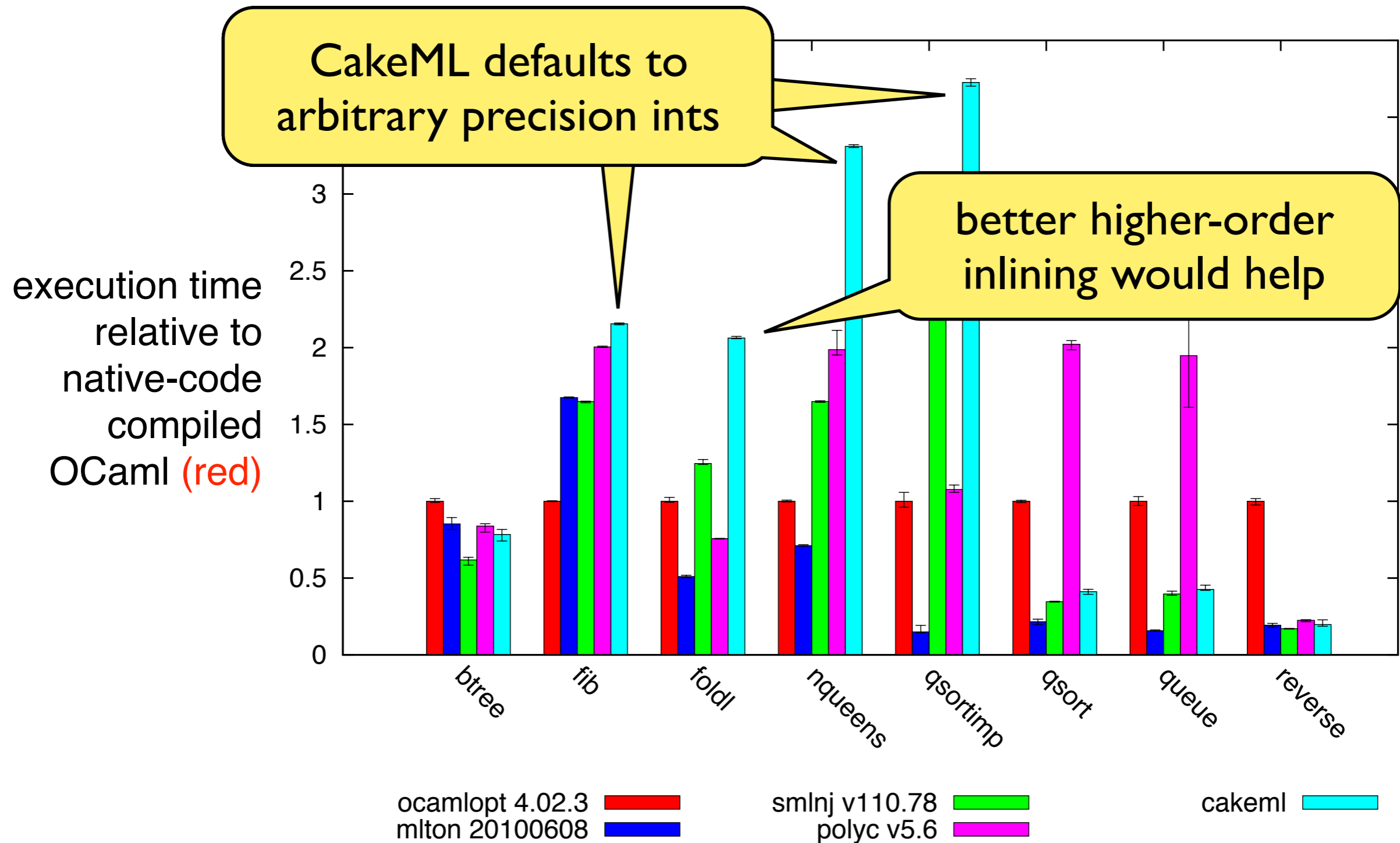
Support for monadic functions for I/O and stateful code

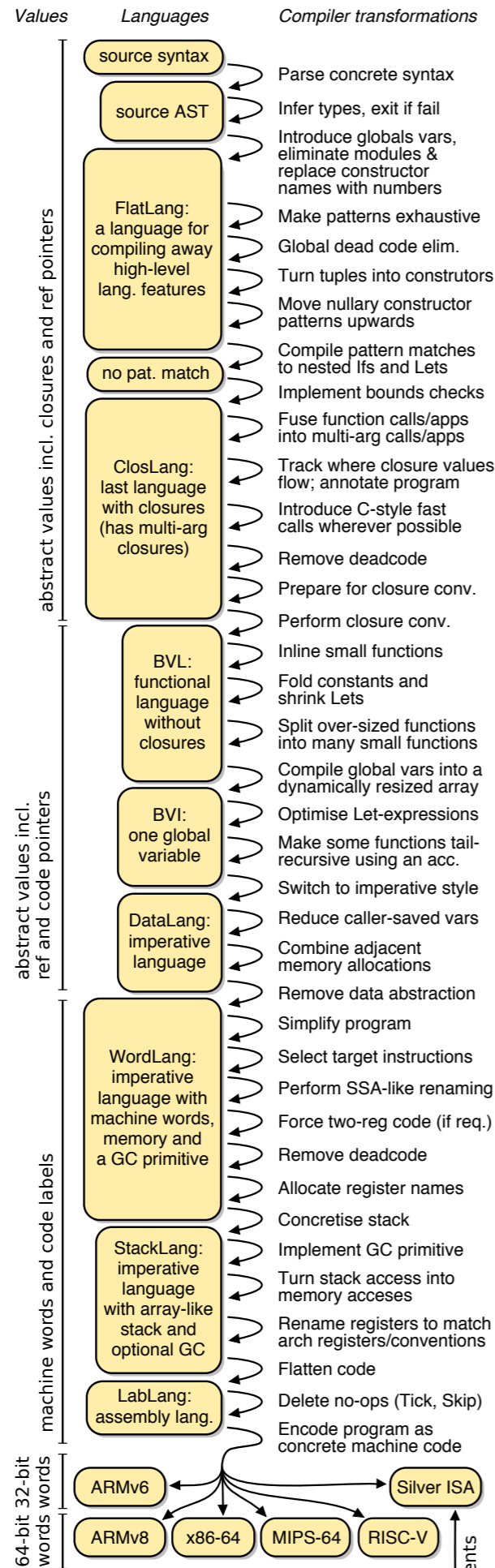
see IJCAR'18 preprint at cakeml.org

Using state significantly improved speed of register allocator.

So how realistic is it?

Performance numbers (x86)





Compiler backend:

9 intermediate languages (ILs)

and many within-IL optimisations

each IL at the right level of abstraction

for the benefit of proofs and compiler implementation

Next slide zooms in

Values used by the semantics

Values

Languages

Compiler transformations

source syntax

source AST

FlatLang:
a language for
compiling away
high-level
lang. features

no pat. match

ClosLang:
last language
with closures
(has multi-arg
closures)

Parse concrete syntax

Infer types, exit if fail

Introduce globals vars,
eliminate modules &
replace constructor
names with numbers

Make patterns exhaustive

Global dead code elim.

Turn tuples into constructors

Move nullary constructor
patterns upwards

Compile pattern matches
to nested ifs and Lets

Implement bounds checks

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

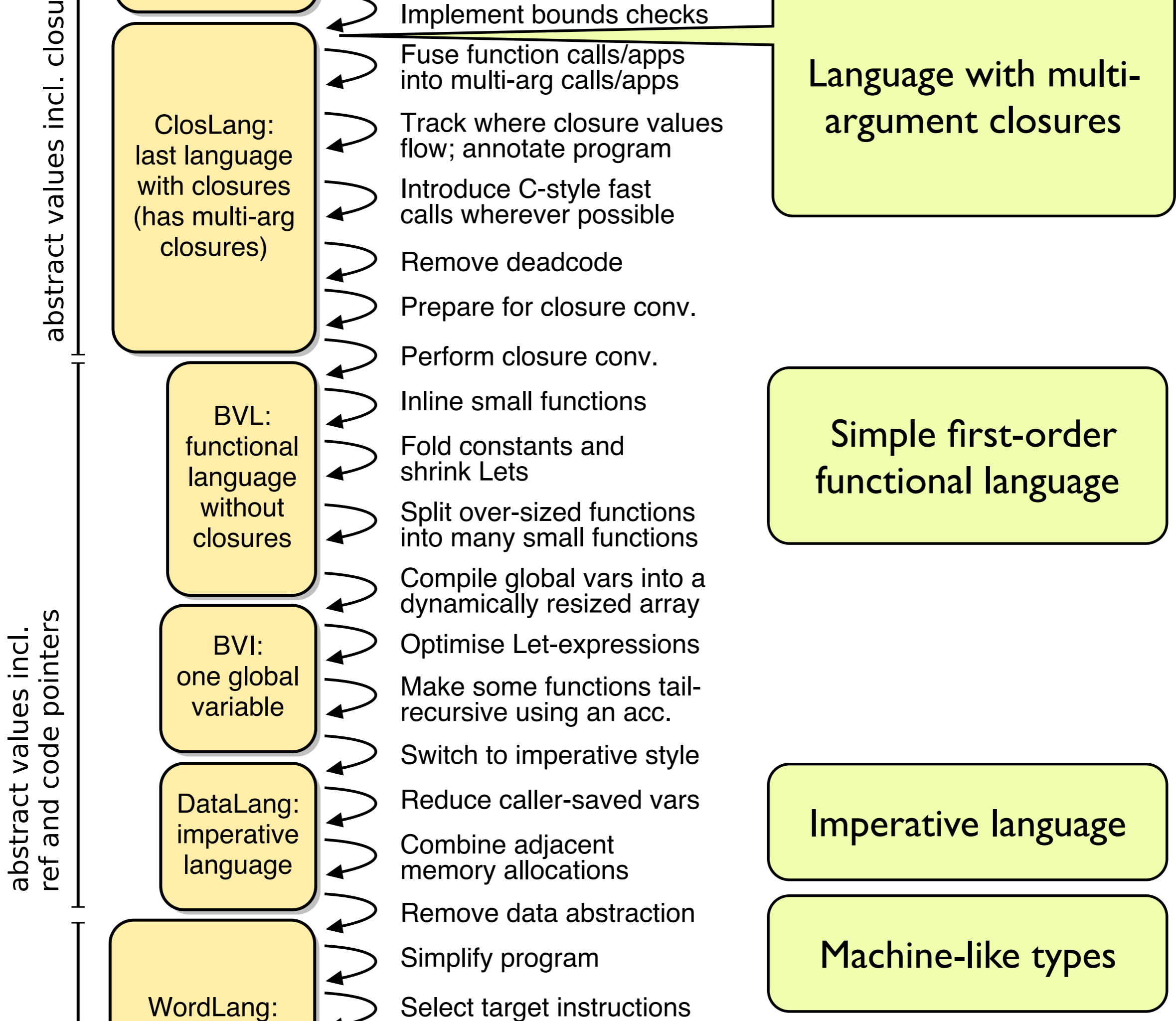
Parser and type
inferencer (proved
sound and complete)

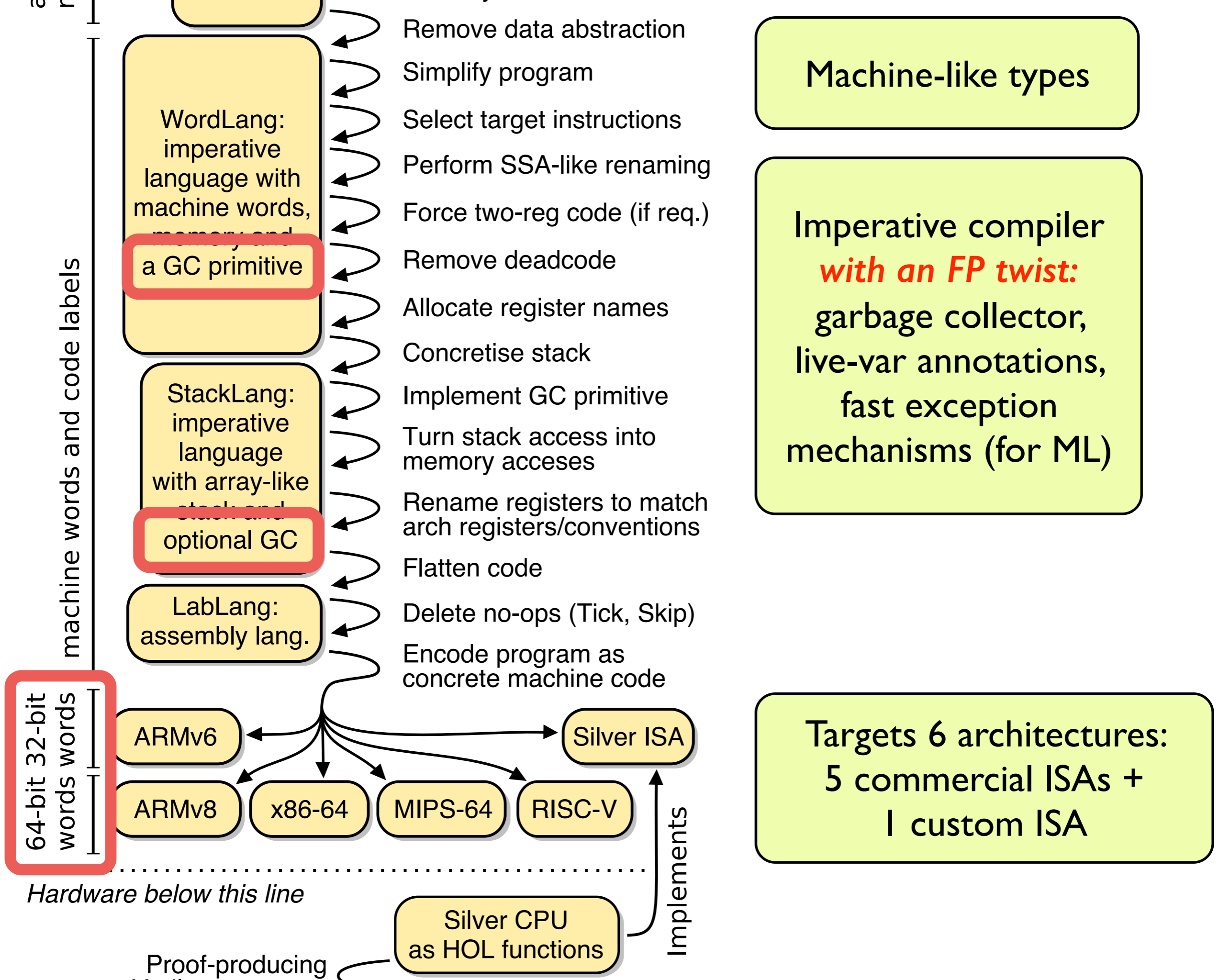
Early phases reduce
the number of
language features

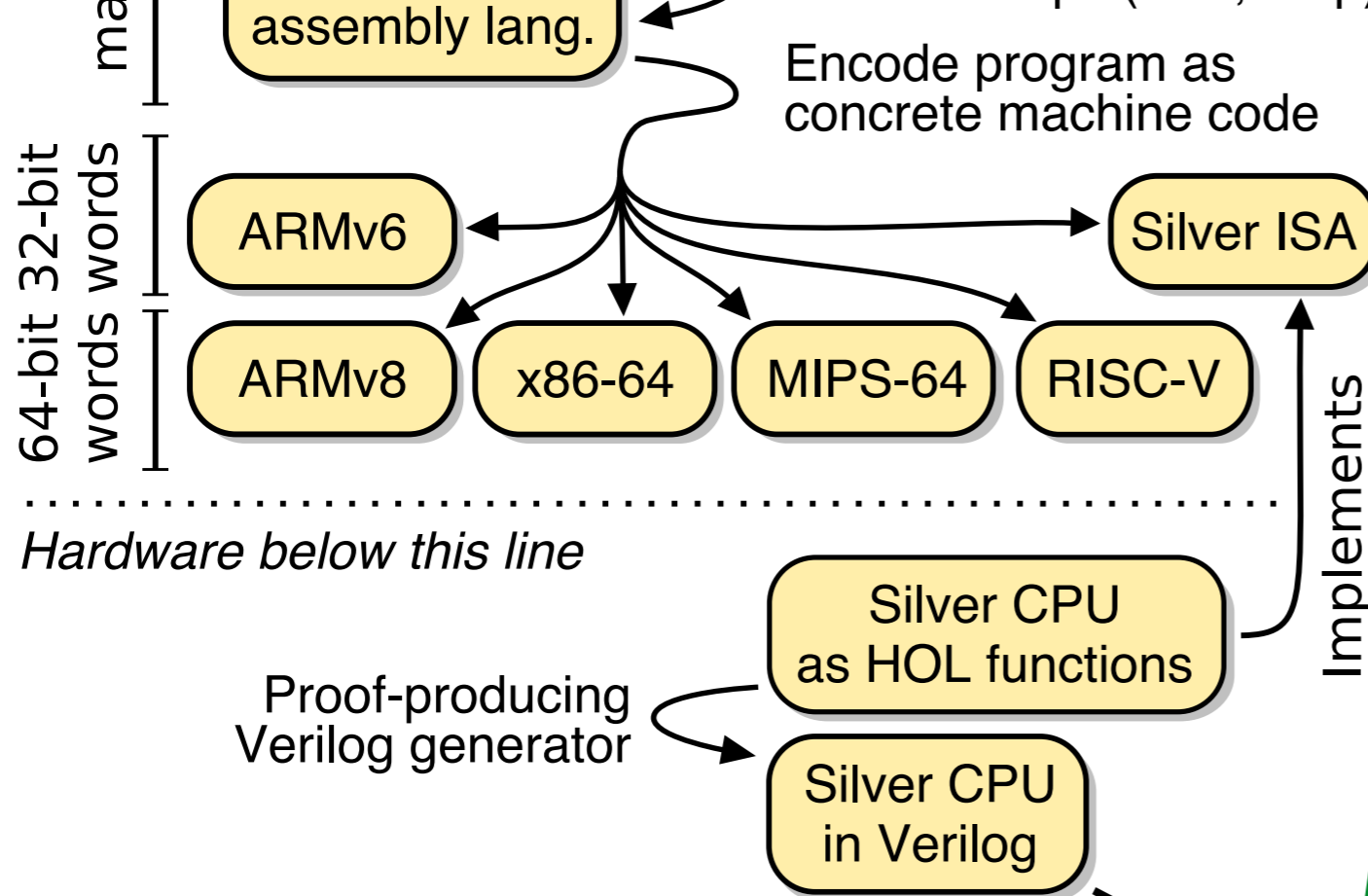
Language with multi-
argument closures

tract values incl. closures and ref pointers





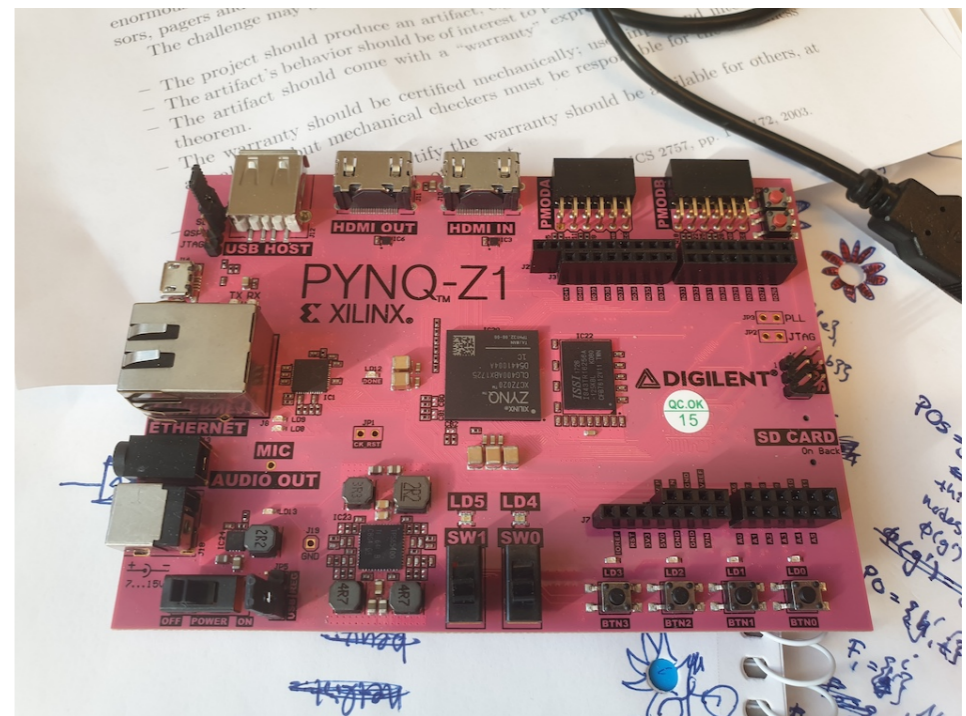




attaches to verified Silver assembly code implementing a simple filesystem that CakeML's standard library assumes.

assumption proved!

Runs on an FPGA:



We can run the bootstrapped compiler on the verified Silver processor on the FPGA.

Verified stack

A verified stack is a computer system that is demonstrably correct. Specifically, it is *a system with a formal proof of correctness that covers all layers of the implementation*, from the hardware through to the application code.

Examples: CLI stack, Verisoft, and soon DeepSpec
CakeML+Silver,



Verified HOL ITP ?

Reminder about Freek

Please, do the same for
HOL Light!



Freek Wiedijk
Radboud University Nijmegen

Nearly there...

semantics of HOL with defns

inferences of HOL with defns

HOL light kernel in CakeML
(module consisting of ~500 lines of CakeML)

CakeML op. semantics

CakeML implementation
(a read-eval-print loop in 64-bit x86 code)

semantics of x86-64 machine

A few pieces missing

... but only a matter of time before it is done.

Aim: a HOL prover people might actually like to use (e.g. for proof of Kepler Conjecture which was done in HOL Light)

... once we're there

With a verified HOL ITP

We will input more code into the LCF-style kernel to speed up common operations

e.g. speed up rewriting

fine because it no longer has to be minimal and “obviously correct”

Having users will force us to improve CakeML.

It is tempting to port the CakeML proofs to the new ITP.

Milawa design in HOL ITP?

Very interesting, but complicated:

Challenge 1:

Distance between ML and logic (HOL) is significant.

Challenge 2:

ML lacks way of turning data into code at runtime.

Challenge I

Challenge I:

Distance between ML and logic (HOL) is significant.

Milawa's logic is very close to Jitawa's Lisp

⇒ logic functions can be dropped into Jitawa's Lisp

For HOL and ML:

Option A: have ITP kernel compile ML-like HOL equations into ML code

Option B: provide CakeML semantics inside HOL & have user prove given AST is safe

both options not as neat as Milawa

Challenge 2

Challenge 2:

ML lacks way of turning data into code at runtime.

In Milawa, we use **define** to turn data into code.

```
> (defun some-code nil '(+ 4 5))
NIL
> (define 'foo 'nil (some-code))
NIL
> (funcall 'foo)
9
```

compiles
code at
runtime

code can be computed

Can we add **Eval** (or Define) to ML? (next slides)

Let's add **Eval** primitive to CakeML

Compiler version 1 (2014) has a verified read-eval-print loop.

ad hoc implementation and proof

For latest version, wouldn't it be nicer to compile:

```
fun loop n =  
  case read () of  
    NONE => ()  
  | SOME input => loop (eval n (parse_wrap_print input));  
  
loop basis_environment;
```

primitive in language

... and *eval* could be used to implement native-compute-style mechanism in (verified) theorem provers.

Eval primitive

The read-eval-print loop sketch from before:

```
fun loop n =  
  case read () of  
    NONE => ()  
  | SOME input =>  
    loop (eval n (parse_wrap_print input));  
  
loop basis_environment;;
```

Type of eval primitive:

eval : environment -> decs -> environment

AST of ML declarations

... is evaluated in a
given environment

Returns the input environment
extended with the new decls.

Communicating results

We propose that references are used:

```
val res = ref 0;
```

Declares an environment (incl. *res*)

```
environment n;
```

... which is used by *eval*

```
val _ = eval n (parse "val _ = (res := 1+2)");
```

```
print_int (!res);
```

This approach ensures that *res* has a type that is defined outside of *eval*

Interesting case

```
val res = ref (Bind:exception);
```

```
environment n;
```

```
eval n (parse "exception Foo of int;  
             res := Foo 4;");
```

```
eval n (parse "exception Foo of bool;  
             case !res of Foo b => (b = true)");
```

res contains Foo 4

Foo refers to local definition

Solution: semantics adds timestamp to each datatype.

How to compile Eval primitive?

Intuition: we want Eval = compile then run native code

We have the bootstrapped compiler...

... with which we can produce machine code at source level.

How do we use the machine code at the source level?

Compile Eval e to:

run (install (compile e))

compile to machine code

... write bytes into memory

... and jump to the new bytes.

If all this works, ...

Then we *can write read-eval-print-loops* in CakeML:

```
fun loop n =  
  case read () of  
    NONE => ()  
  | SOME input =>  
    loop (eval n (parse_wrap_print input));  
  
loop basis_environment;
```

If all this works, ...

Then we *can write read-eval-print-loops* in CakeML:

```
fun loop n =  
  case read () of  
    NONE => ()  
  | SOME input =>  
    loop (eval n (parse_wrap_print input)  
          handle NoType => (print ...; n)  
              | ParseErr => (print ...; n)  
              | other => (print ...; n));  
  
loop basis_environment;
```

... and *build verified reflection mechanism* in a verified theorem prover.

Summary

This lecture:

- ✓ CakeML — aims to be realistic verified ML implementation
- ✓ Compiler bootstrapping inside ITP
- ✓ Effort towards verified HOL ITP
- ✓ Adding Eval to CakeML language in order to have reflection mechanism in verified HOL ITP



Questions?