Inria, Paris, Oct 2023

The CakeML Project

Verified Compilation, Verified Bootstrapping, Just-In-Time Compilation, and Applications

Magnus O. Myreen

Chalmers, Sweden

Mentions work by: Ramana Kumar, Scott Owens, Yong Kiam Tan, Andreas Lööw, Oskar Abrahamsson, Michael Norrish, Anthony Fox, Samuel Vivien, …

Mentions work by: Ramana Kumar, Scott Owens, Yong Kiam Tan, Andreas Lööw, Oskar Abrahamsson, Michael Norrish, Anthony Fox, Samuel Vivien, . . .

A more accurate list of contributors:

Oskar Abrahamsson, Johannes Åman Pohjola, Rob Arthan, Heiko Becker, Matthew Brecknell, Fanny Canivet, Connor Cashman, Mauricio Chimento, Nicholas Coughlin, Gregorio Curello, Eva Darulova, Hugo Férée, Anthony Fox, Arve Gengelbach, Sofia Giljegård, Armaël Guéneau, Rikard Hjort, Son Ho, Jakob Holmgren, Lars Hupel, Felix Kam, Hrutvik Kanabar, Stephen Kell, Ramana Kumar, Quentin Ladeveze, Théo Laurent, John Lind, Alejandro Gómez-Londoño, Andreas Lööw, Alexander Mihajlovic, Nebojsa Mihajlovic, Dominic Mulligan, Prashanth Mundkur, Michael Norrish, Oskar Nyberg, Stefan O'Rear, Scott Owens, Christian Persson, Christopher Pulte, Henrik Rostedt, Adam Sandberg Eriksson, Thomas Sewell, Konrad Slind, Michael Sproul, Partha Susarla, Hira Syeda, Yong Kiam Tan, Timotej Tomandl, Trần Tiến Dũng, Théophile Wallez, Johan Wennerbeck, Freek Wiedijk, James Wood.

What is CakeML?

The name
“CakeML”
comes from
“Cambridge and
Kent ML”

CakeML is:

→ a functional programming language (SML/OCaml like)

What is CakeML?

CakeML is:

- a functional programming language (SML/OCaml like)
- an ecosystem of proofs and tools built around the language (including a verified compiler)
- a “verified stack” extending down to hardware (Verilog)

CakeML is developed in the HOL4 interactive theorem prover.

This talk

Part 1: The core of the CakeML project

research questions, main ideas,

verified compilation, end-to-end correctness

Part 2: Extensions and collaborations

hardware, HOL light, other compilers,

proof checkers, collaborations

Research questions

Is it possible to have a ***clean high-level programming language*** formalised?

… with a compiler that generates code with good performance?

Can we have everything properly ***connected with proofs***?

… even ***transport proved properties*** down to actual machine code / hardware?

(some goals shared with the DeepSpec project)

Going back to 2012 …

Original motivation

Around 2012: it had become common to use code generators (e.g. Coq's code extraction) to generate code from ITPs.

Example:

Given the logic definition of list append (++),

```
[] ++ ys = ys  ∧  (x::xs) ++ ys = x::(xs ++ ys)
```

the ITP's code generator might produce SML code:

```
fun append []      ys = ys
  | append (x::xs) ys = x :: append xs ys
```

Non-trivial cases

Underspecification:

```
hd (x::xs) = x
```

nil case [] is left unspecified

```
... map hd xs ...
```

underspecification propagates

Semantic mismatches:

```
if x = y then ... else ...
```

SML's = is different from logic =

```
fac n = if n < 2 then 1 else fac (n-1) * n
```

how should numbers be translated?

This is not a good state of affairs.

Magnus Myreen

Well, automated translation is better than manual ad hoc transcription…

Scott Owens

Yes

However, ITPs should do more! ITPs should prove that the generated SML/OCaml/Scala code is behaviourally equivalent to the original functions.

I completely agree.

Result: we formalised subset of SML (i.e. CakeML) and developed a proof-producing code-generation tool for HOL4.

What exactly is that?

Result: we formalised subset of SML (i.e. CakeML) and developed a proof-producing code-generation tool for HOL4.

Interactive theorem provers

Most developments look like this:

```
Definition fac_def:
  fac n =
    if n = 0 then 1 else fac (n-1) * n
End

Definition fac_it_def:
  fac_it n acc =
    if n = 0 then acc else fac_it (n-1) (acc * n)
End

Theorem fac_it_correct:
  ∀n acc. fac_it n acc = fac n * acc
Proof
  Induct
  \\ once_rewrite_tac [fac_def,fac_it_def]
  \\ rw []
QED
```

definition

definition

goal-directed tactic proof

Interactive theorem provers

Most developments look like this:

```
Theorem fac_it_correct:
  ∀n acc. fac_it n acc = fac n * acc
Proof
  Induct
  \\ once_rewrite_tac [fac_def,fac_it_def]
  \\ rw []
QED
```

goal-directed tactic proof

Prover responds:

Proved theorems are just values of type "thm" in SML.

```
val fac_it_correct = ⊢ ∀n acc. fac_it n acc = fac n * acc: thm
>
```

HOL4 prover is built on top of SML read-eval-print loop

HOL4 is highly programmable

Proof-producing tool

We developed a tool (called translate*) that generates CakeML AST and automatically proves correspondence:*

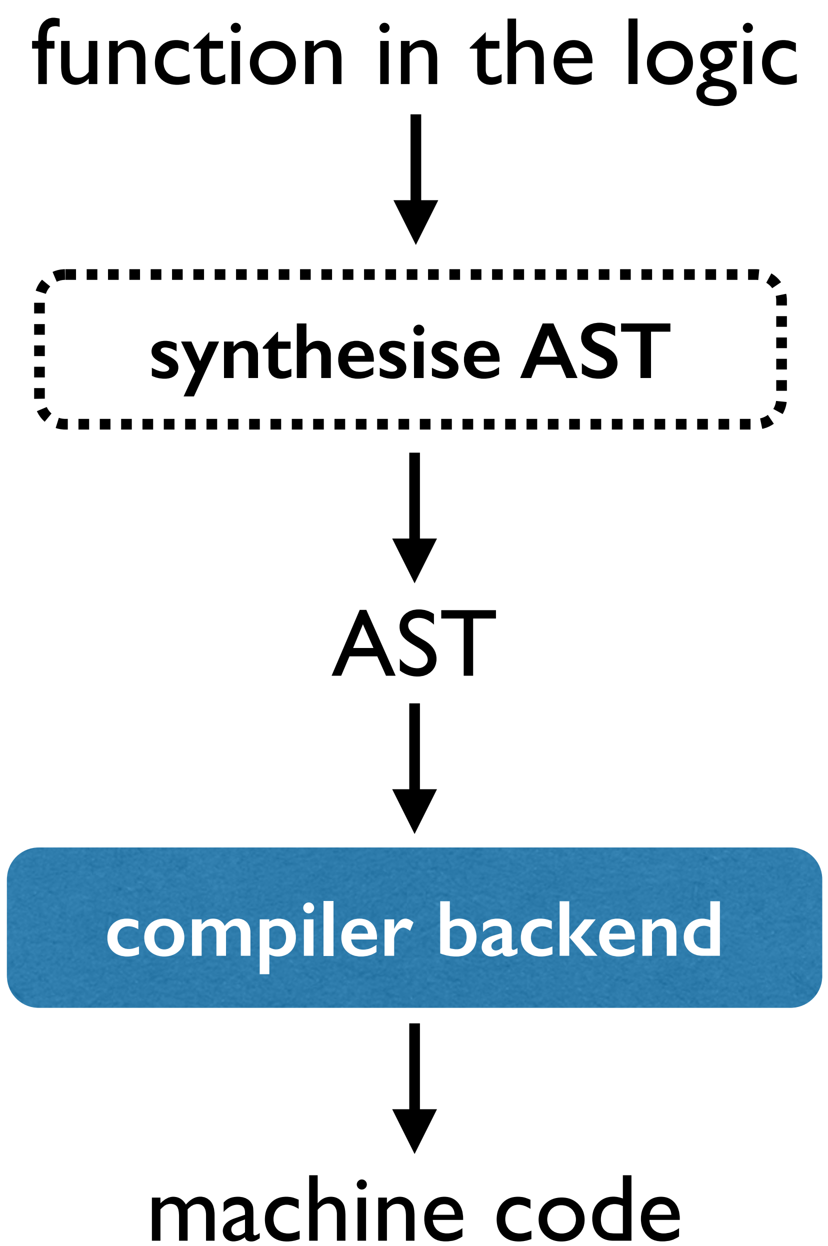
User requests translation of fac_def

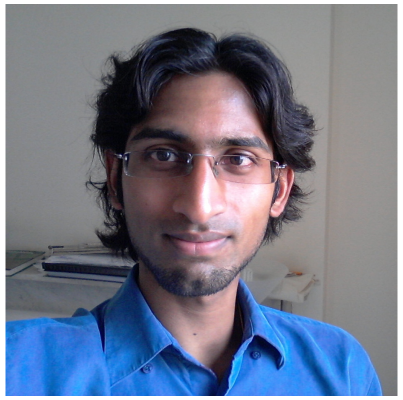
```
> val v_thm = translate fac_def;
Translating fac
val v_thm = ⊢ (NUM --> NUM) fac fac_v: thm
>
```

The tool defines a CakeML value (containing an AST)

… and proves a theorem asserting the correspondence between fac and fac_v.

Components of toolchain

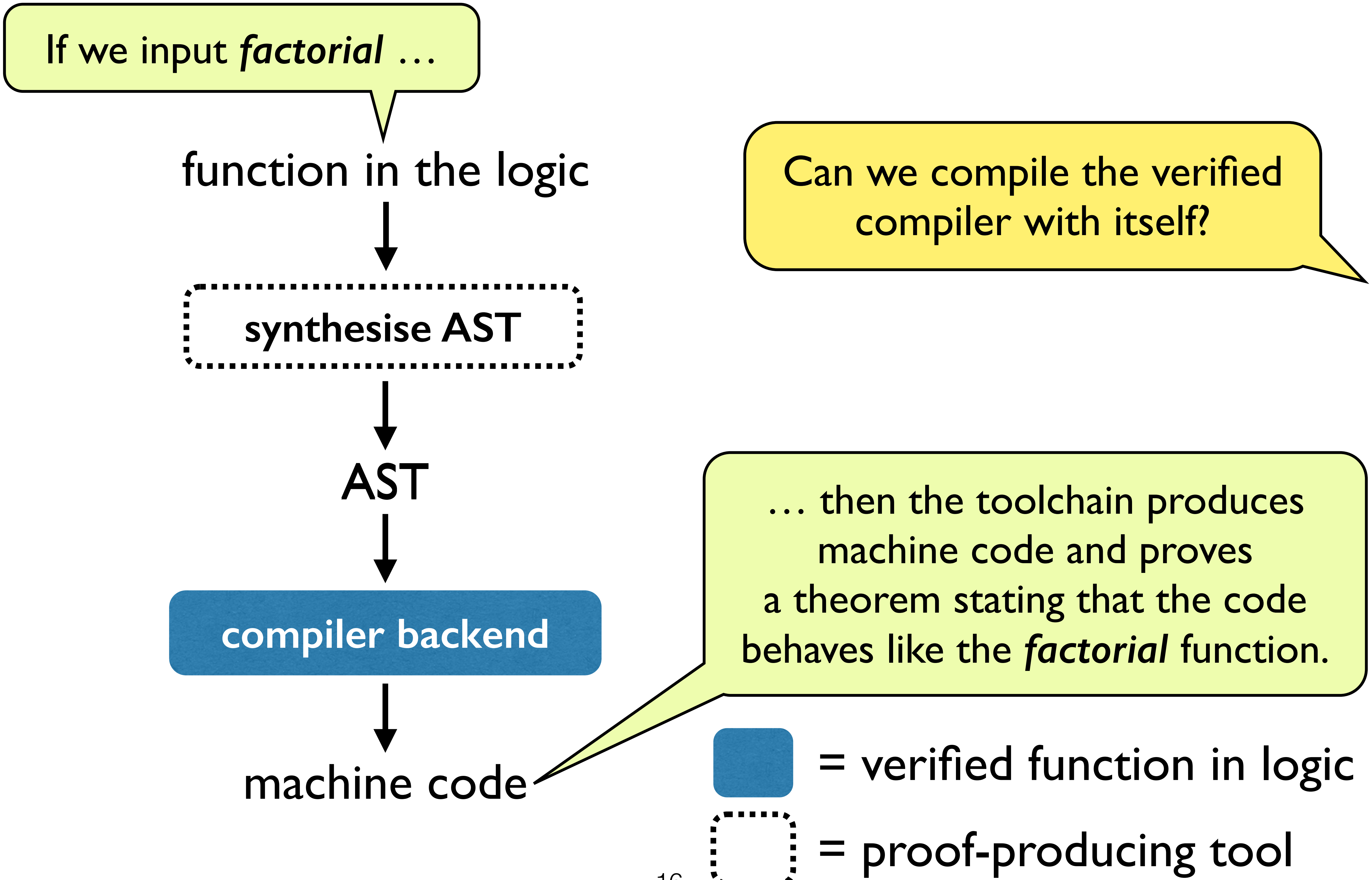



Ramana Kumar

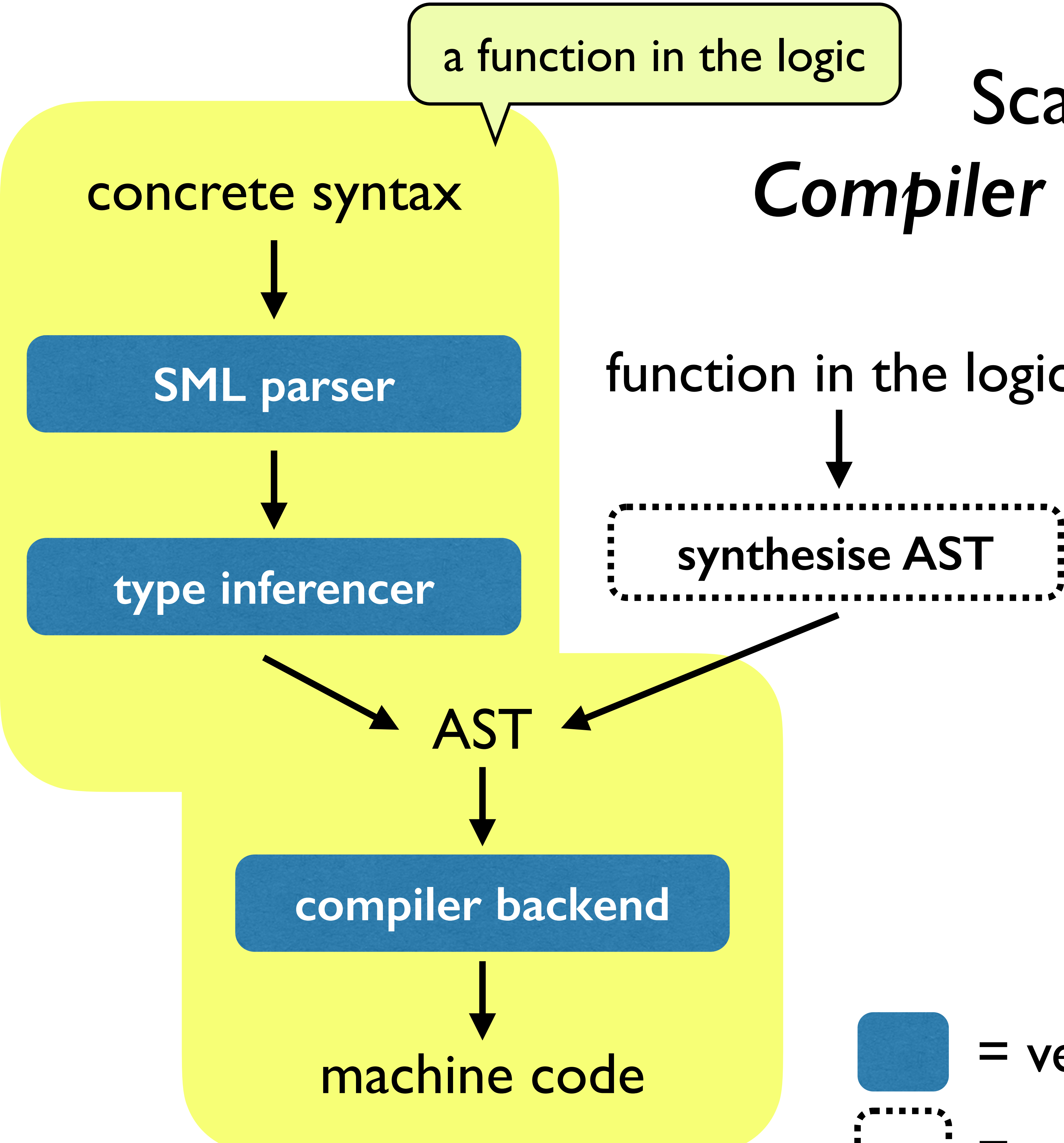
= verified function in logic

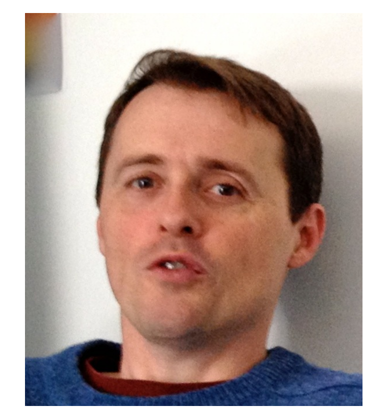
= proof-producing tool

Components of toolchain



Scaling up:
Compiler Bootstrapping



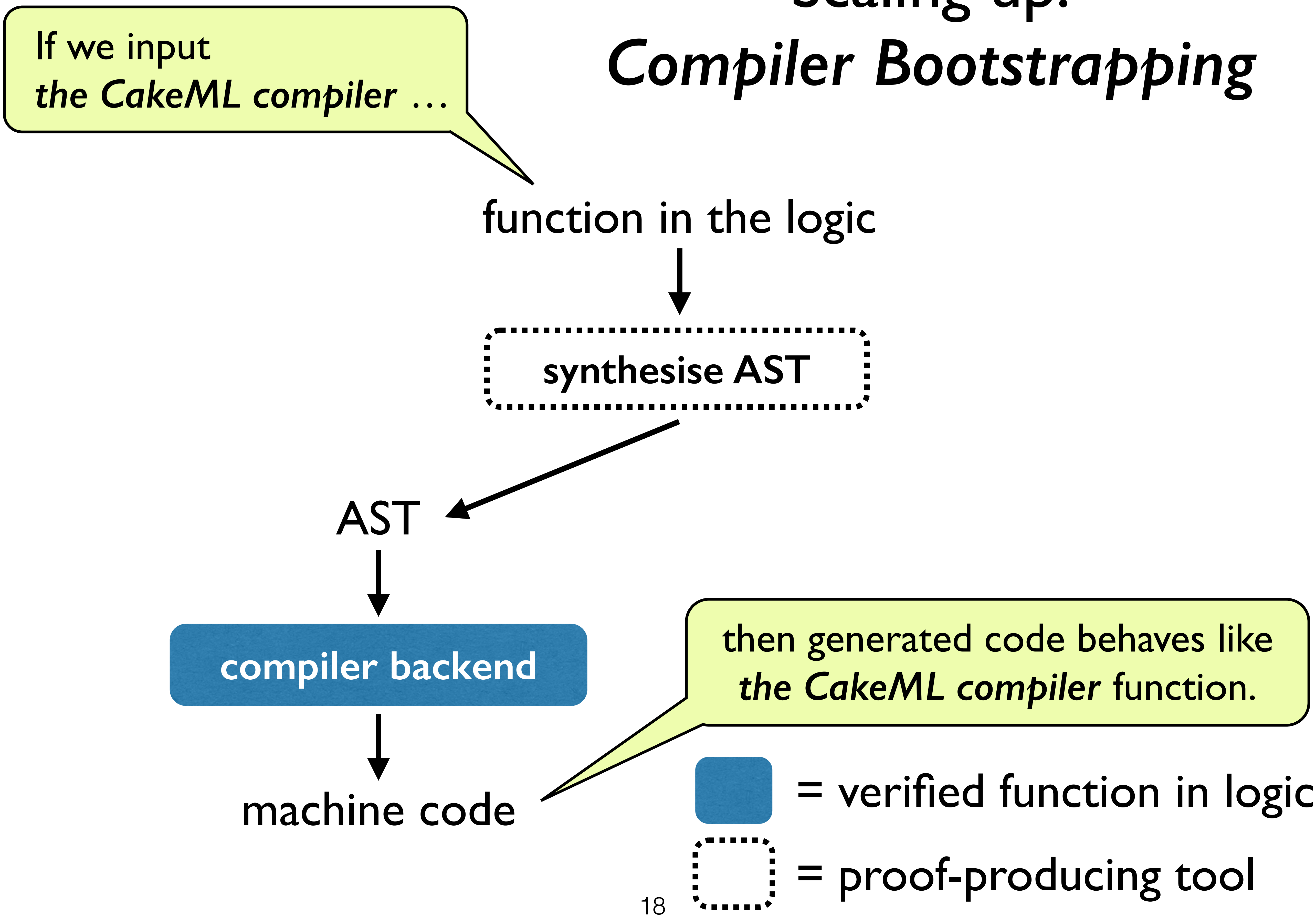


Michael Norrish

= verified function in logic

= proof-producing tool

Scaling up: *Compiler Bootstrapping*



Version 1

POPL'14

Compiler Bootstrapping. Everything fit together.

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

However, compiler had very few optimisations.

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-

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

Steps towards realism

How real can we make the CakeML compiler?

Would like: ***speed***, better ***I/O*** etc.

Settled on new methodology:

ESOP'16

Functional Big-step Semantics

Scott Owens[1], Magnus O. Myreen[2], Ramana Kumar[3], and Yong Kiam Tan[4]

[1] School of Computing, University of Kent, UK
[2] CSE Department, Chalmers University of Technology, Sweden
[3] NICTA, Australia
[4] IHPC, A*STAR, Singapore

Abstract. When doing an interactive proof about a piece of software, it is important that the underlying programming language's semantics does not make the proof unnecessarily difficult or unwieldy. Both small- ... semantics are commonly used, and the latter is typi- ...

Functional big-step

Sample:

$$\text{eval } env \text{ (Const } n) \stackrel{\text{def}}{=} \text{return (Num } n)$$

$$\text{eval } env \text{ (Var } n) \stackrel{\text{def}}{=}$$
$$\quad \text{case } env\ n \text{ of None} \Rightarrow \text{fail} \mid \text{Some } v \Rightarrow \text{return } v$$

$$\text{eval } env \text{ (Op } f\ xs) \stackrel{\text{def}}{=}$$
$$\quad \text{do } vs \leftarrow \text{evals } env\ xs;\ \text{eval\_op } f\ vs \text{ od}$$

$$\text{eval } env \text{ (Let } vname\ x\ y) \stackrel{\text{def}}{=}$$
$$\quad \text{do } v \leftarrow \text{eval } env\ x;$$
$$\qquad \text{eval } env\langle vname \mapsto \text{Some } v\rangle\ y \text{ od}$$

where:

$$\text{return } v\ s \stackrel{\text{def}}{=} (\text{Res } v, s)$$

$$\text{fail } s \stackrel{\text{def}}{=} (\text{Err Crash}, s)$$

$$\text{monad\_bind } f\ g\ s \stackrel{\text{def}}{=}$$
$$\quad \text{case } f\ s \text{ of (Res } v, s_1) \Rightarrow g\ v\ s_1 \mid (\text{Err } e, s_1) \Rightarrow (\text{Err } e, s_1)$$

Functional big-step

Sample:

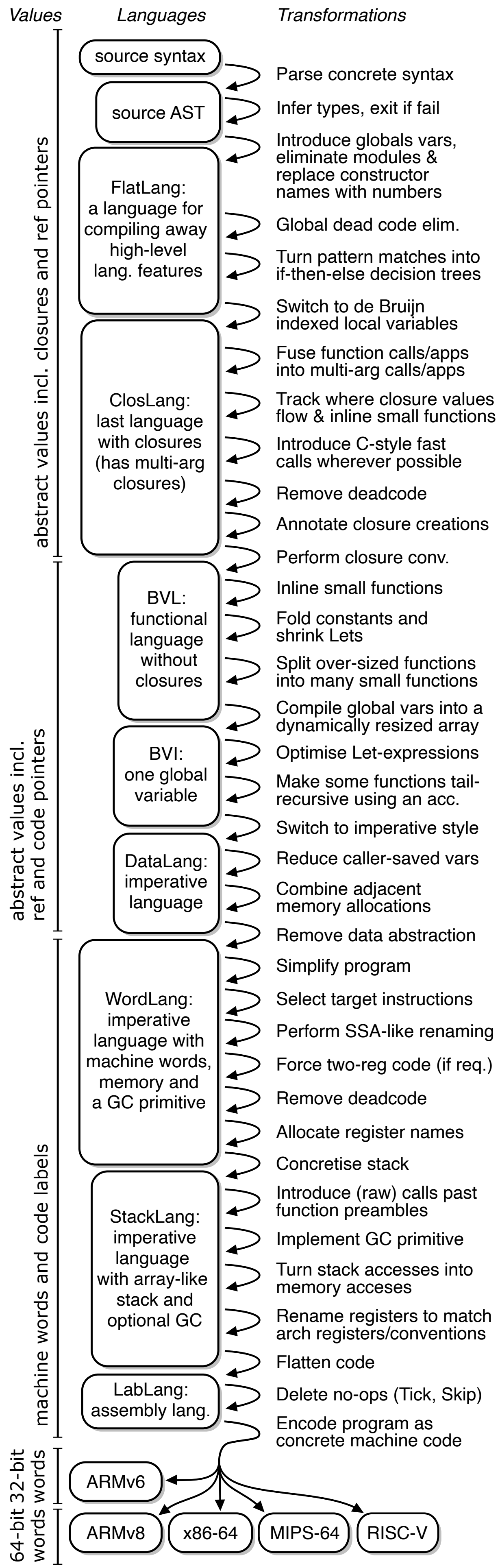
$$\text{eval}\ \mathit{env}\ (\text{Call}\ \mathit{fname}\ \mathit{xs}) \stackrel{\text{def}}{=}$$
$$\quad \text{do}\ \mathit{vs} \leftarrow \text{evals}\ \mathit{env}\ \mathit{xs};$$
$$\qquad (\mathit{fenv},\mathit{body}) \leftarrow \text{get\_env\_and\_body}\ \mathit{fname}\ \mathit{vs};$$
$$\qquad \text{eval}\ \mathit{fenv}\ \mathit{body}\ \ \text{od}$$

decrements a clock ("uses fuel") …

… that is passed around in state.

where:

$$\text{return}\ v\ s \stackrel{\text{def}}{=} (\text{Res}\ v,s)$$
$$\text{fail}\ s \stackrel{\text{def}}{=} (\text{Err Crash},s)$$
$$\text{monad\_bind}\ f\ g\ s \stackrel{\text{def}}{=}$$
$$\quad \text{case}\ f\ s\ \text{of}\ (\text{Res}\ v,s_1) \Rightarrow g\ v\ s_1 \mid (\text{Err}\ e,s_1) \Rightarrow (\text{Err}\ e,s_1)$$



New compiler backend:

8 intermediate languages (ILs)

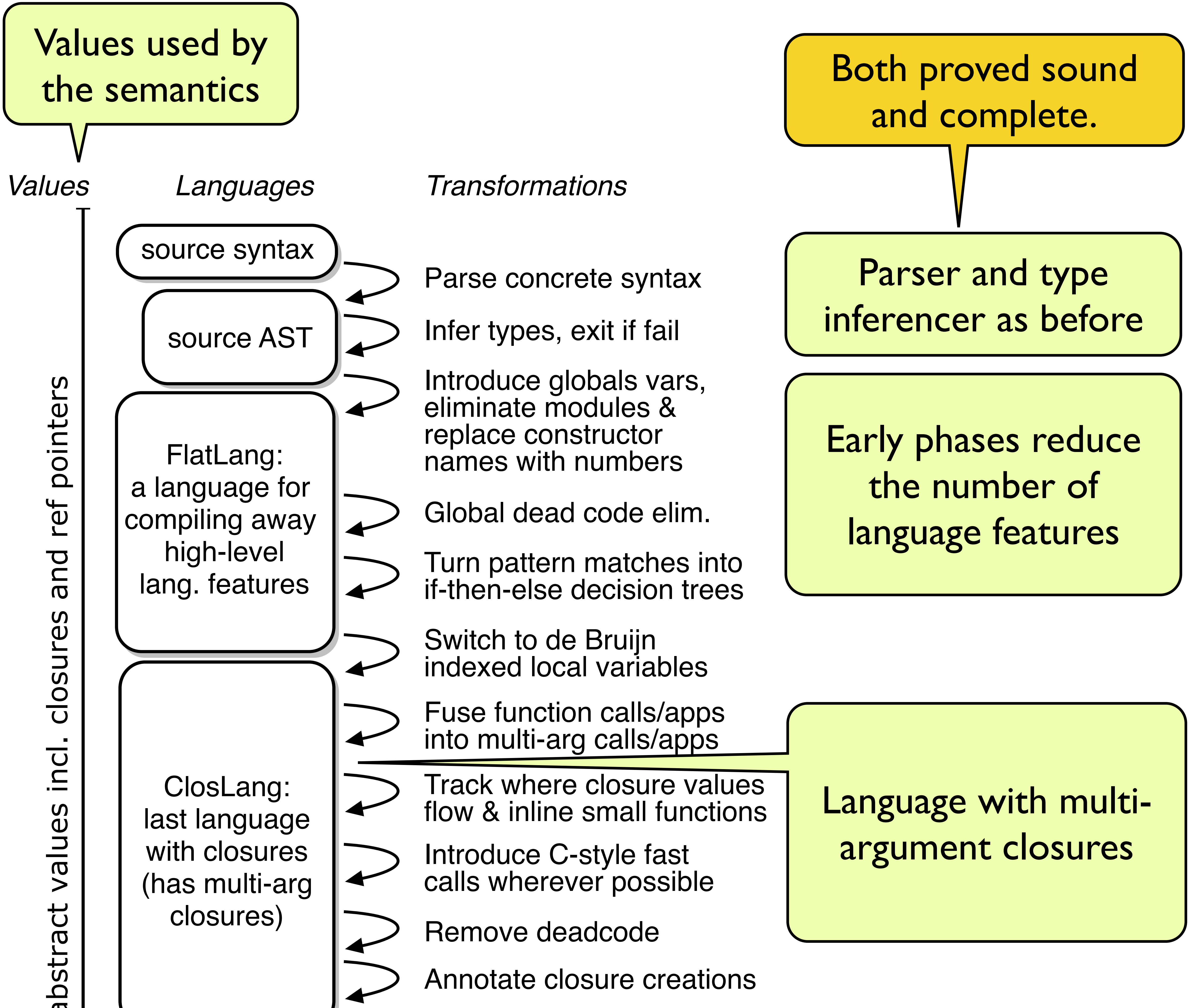
and many within-IL optimisations

each IL at the right level of abstraction

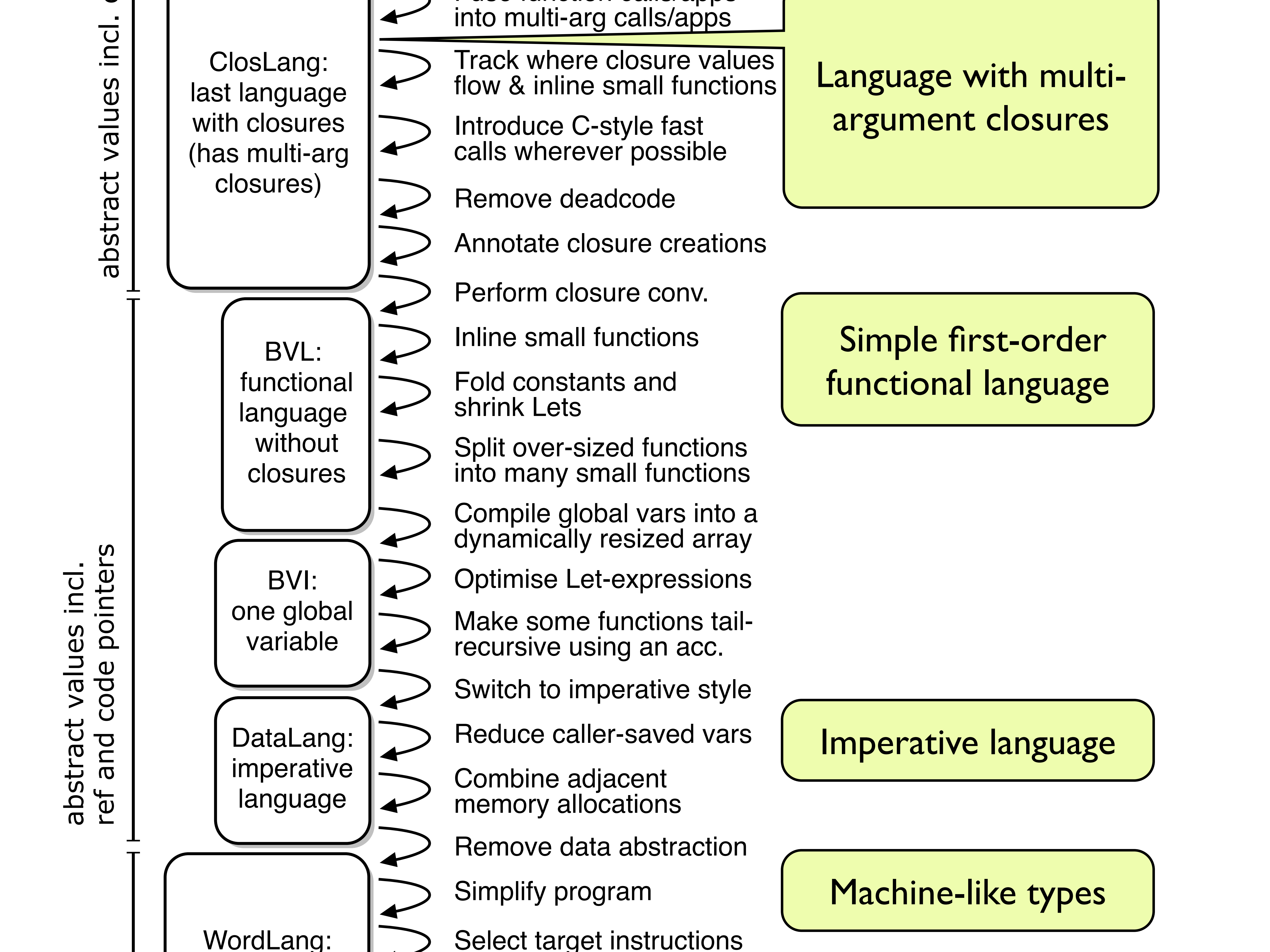
for the benefit of proofs and compiler implementation

Next slide zooms in

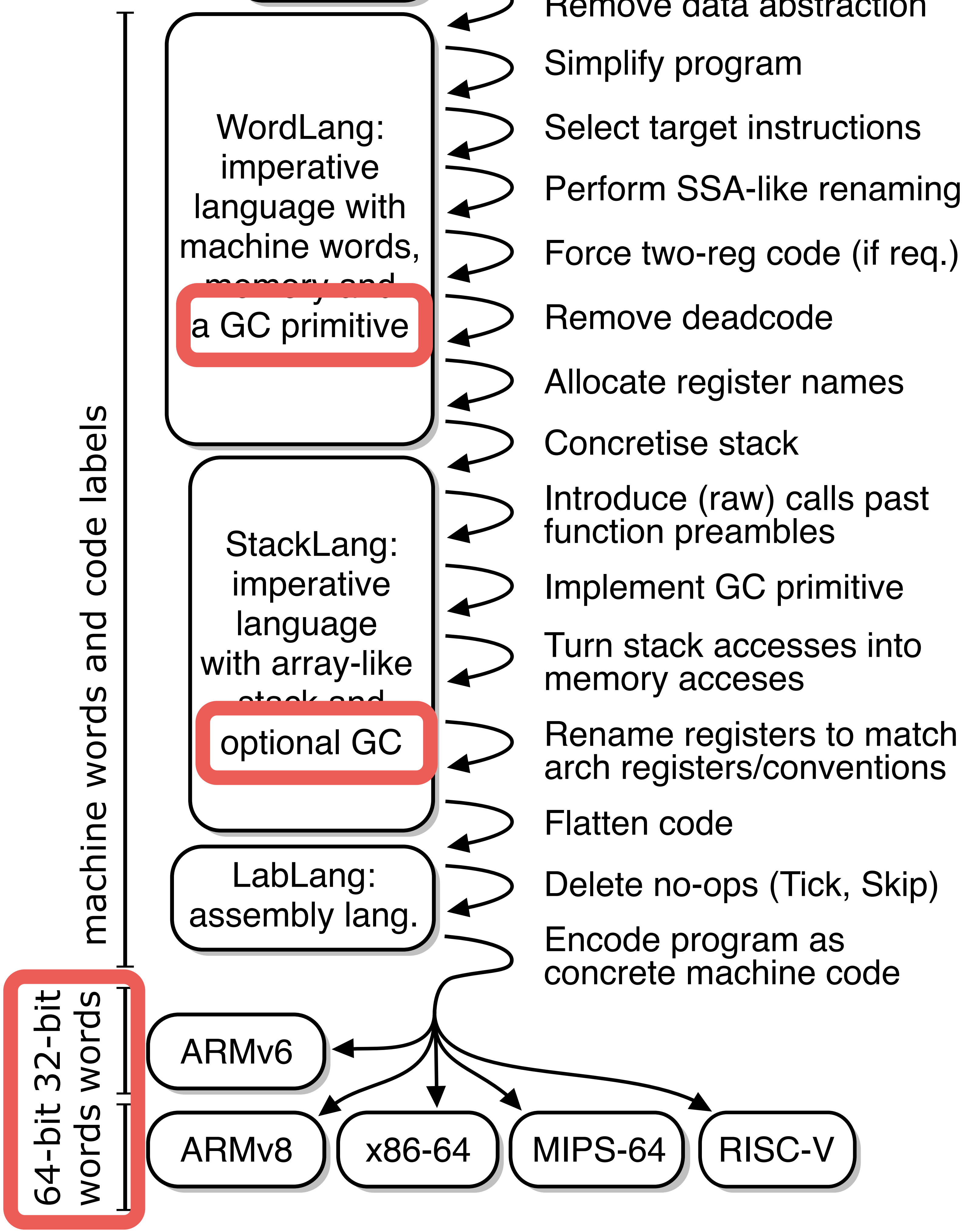
Yong Kiam Tan



Values used by the semantics
Both proved sound and complete.
Values
Languages
Transformations
abstract values incl. closures and ref pointers
source syntax
Parse concrete syntax
source AST
Infer types, exit if fail
Introduce globals vars, eliminate modules & replace constructor names with numbers
FlatLang: a language for compiling away high-level lang. features
Global dead code elim.
Turn pattern matches into if-then-else decision trees
Switch to de Bruijn indexed local variables
ClosLang: last language with closures (has multi-arg closures)
Fuse function calls/apps into multi-arg calls/apps
Track where closure values flow & inline small functions
Introduce C-style fast calls wherever possible
Remove deadcode
Annotate closure creations
Parser and type inferencer as before
Early phases reduce the number of language features
Language with multi-argument closures



abstract values incl.
into multi-arg calls/apps
ClosLang: last language with closures (has multi-arg closures)
Track where closure values flow & inline small functions
Introduce C-style fast calls wherever possible
Remove deadcode
Annotate closure creations
Perform closure conv.
Language with multi-argument closures
BVL: functional language without closures
Inline small functions
Fold constants and shrink Lets
Split over-sized functions into many small functions
Simple first-order functional language
Compile global vars into a dynamically resized array
BVI: one global variable
Optimise Let-expressions
Make some functions tail-recursive using an acc.
Switch to imperative style
abstract values incl. ref and code pointers
DataLang: imperative language
Reduce caller-saved vars
Combine adjacent memory allocations
Imperative language
Remove data abstraction
Simplify program
Machine-like types
WordLang:
Select target instructions

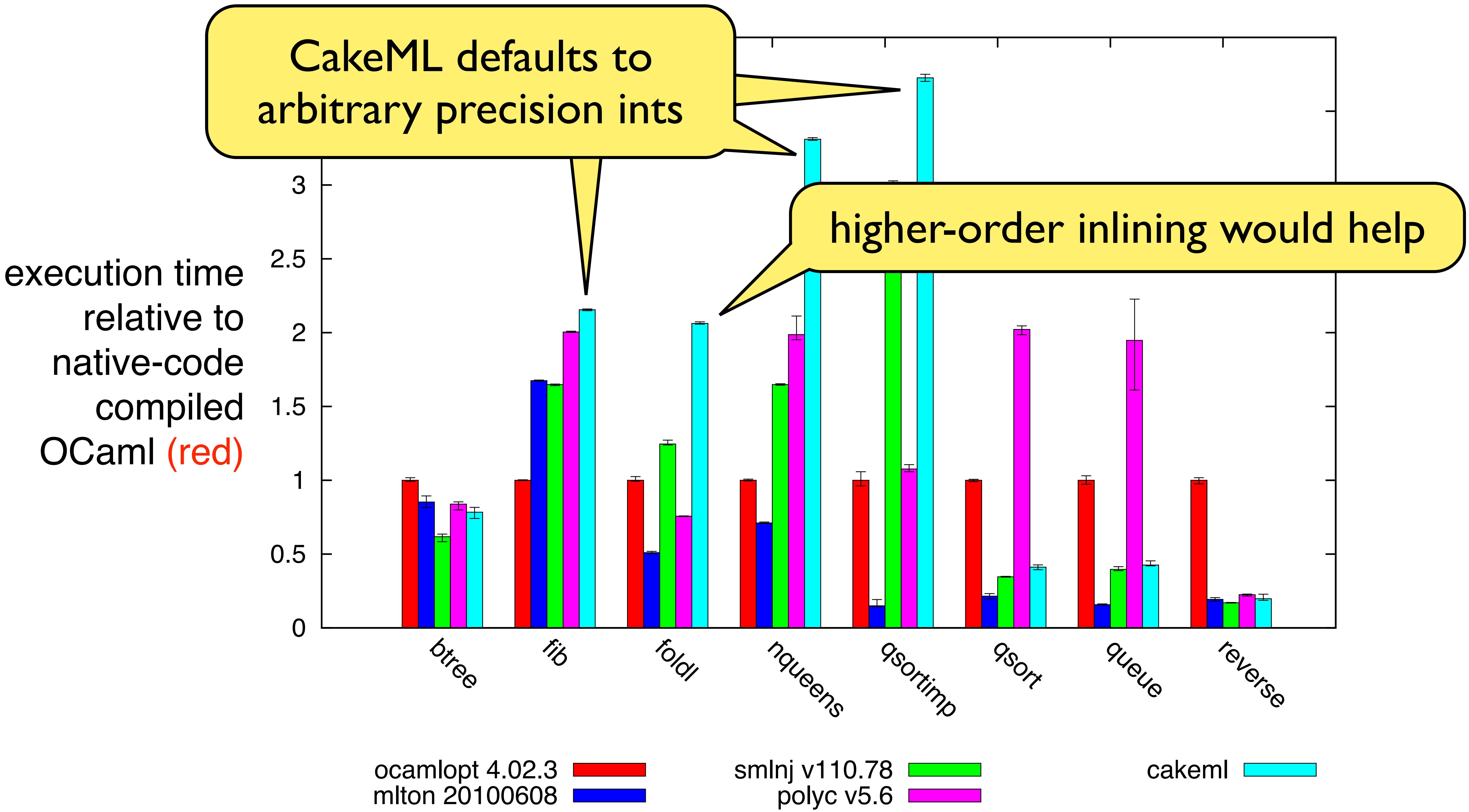


Machine-like types

Imperative compiler
with an FP twist:
garbage collector,
live-var annotations,
fast exception
mechanisms (for ML)

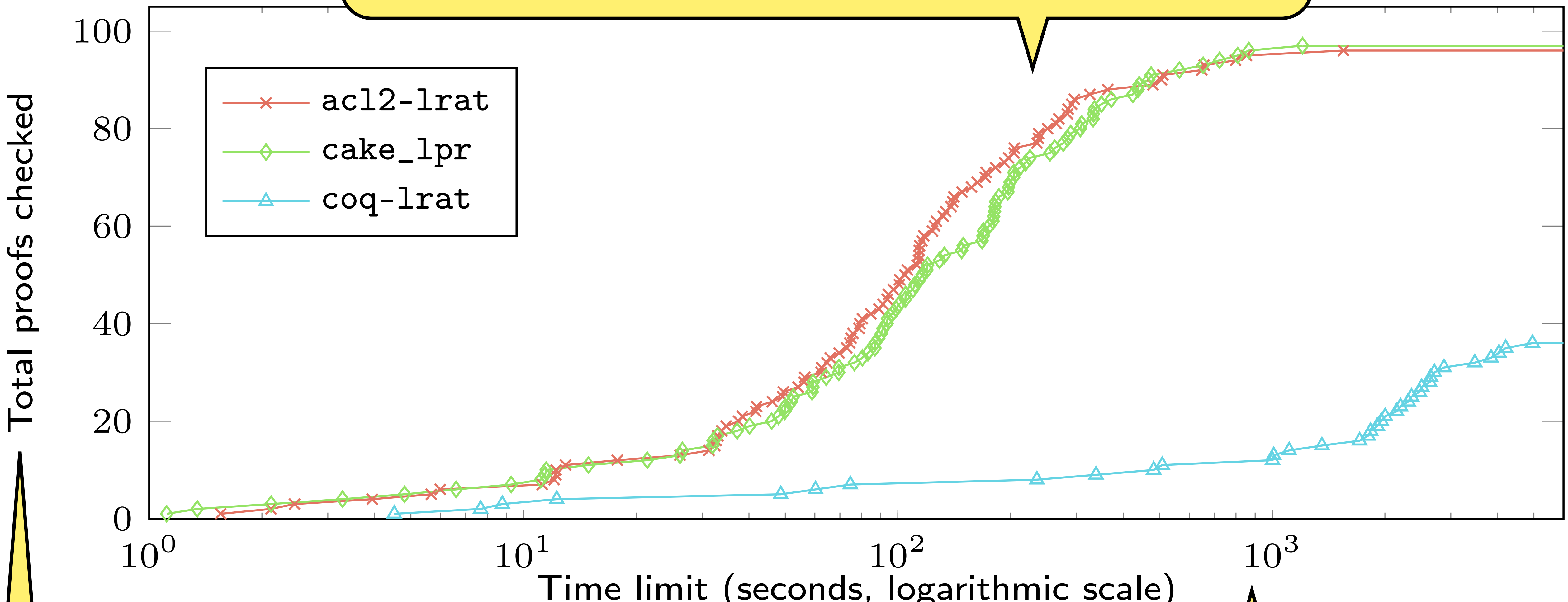
Targets 5 architectures

Performance numbers



Verified checking of LRAT

Verified cake_lpr implementation is near highly optimised ACL2-generated checker



Percent of SAT Race 2019 proofs checked …

… within (per instance) time limit.

Tan et al. TACAS'21

Demo of cake_lpr checker

```
myreen@oven2:~/demo/sat-proofs$
```

Other developments

How to verify manually written CakeML code?

Background:

Arthur Charguéraud developed CFML for reasoning about OCaml code in Coq

A verified programming logic for CakeML:

Armaël Guéneau adapted CFML for reasoning about CakeML code (including state, exceptions and I/O)

Son Ho implemented significant proof automation for Armaël's program logic for CakeML

Infinite runs and liveness

Infinite runs:

Compiler proofs talk about infinite runs.

⟶ Program logic should be able to!

⟶ CakeML CF adapted to reasoning about infinite runs and liveness properties.

Johannes Åman Pohjola

Problem with liveness

Problem: Compiler correctness allows any CakeML program to exit early with an out-of-memory (OOM) error.

Thus: non-terminating CakeML programs might terminate once compiled…

Shape of compiler correctness theorem:

$$\text{machine\_sem}\ \mathit{ffi}\ (\text{compile}\ c\ \mathit{prog}) \subseteq$$
$$\text{extend\_with\_resource\_limit}\ (\text{source\_sem}\ \mathit{ffi}\ \mathit{prog})$$

source behaviours extended with early OOM termination

Proving absence of OOM

Solution: A verified ***space cost semantics*** for CakeML that allows liveness properties to transfer to machine code.

Shape of new alternative theorem:

if space cost semantics says *prog* is safe …

$$\text{is\_safe\_for\_space}\ \mathit{ffi}\ c\ \mathit{prog}\ \ldots \Rightarrow$$
$$\text{machine\_sem}\ \mathit{ffi}\ (\text{compile}\ c\ \mathit{prog}) = \text{source\_sem}\ \mathit{ffi}\ \mathit{prog}$$

… then semantics is preserved exactly

… and liveness properties proved at source carry to machine code!

This talk

Part 1: The core of the CakeML project

research questions, main ideas,

verified compilation, end-to-end correctness

Part 2: Extensions and collaborations

hardware, HOL light, other compilers,

proof checkers, collaborations

This talk

Part 1: The core of the CakeML project

research questions, main ideas,

verified compilation, end-to-end correctness

Part 2: Extensions and collaborations

hardware, HOL light, other compilers,

proof checkers, collaborations

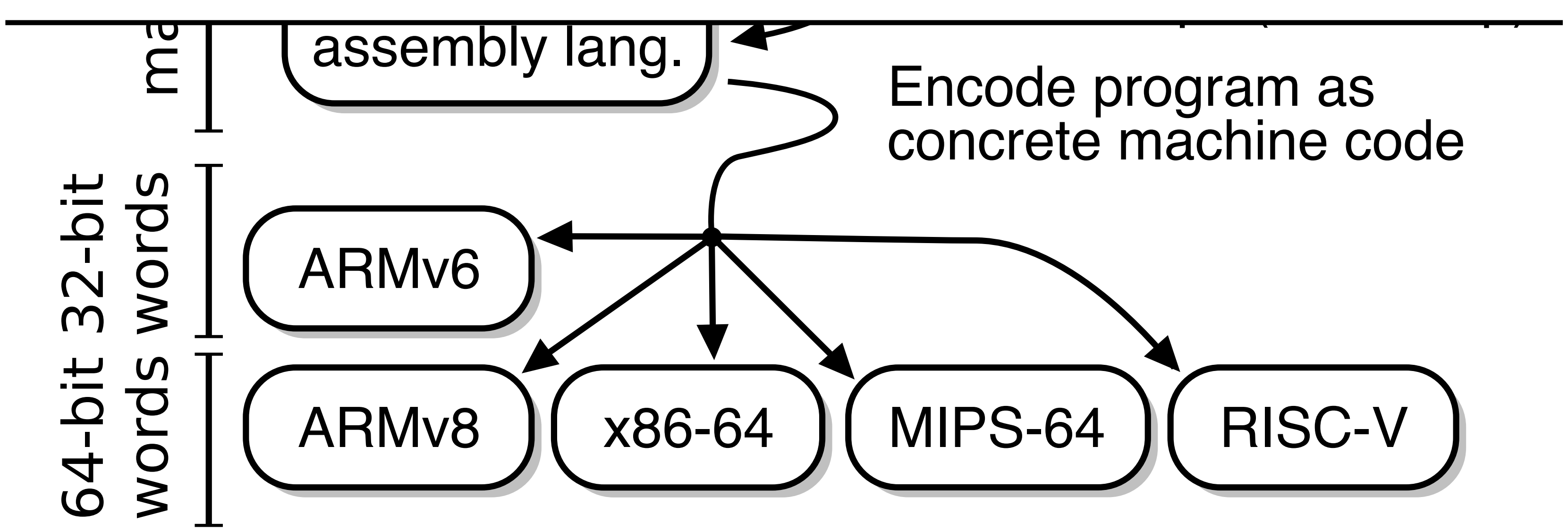
Verified stack

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
CakeML+Silver [PLDI'19]
Erbsen et al. [PLDI'21]

CakeML

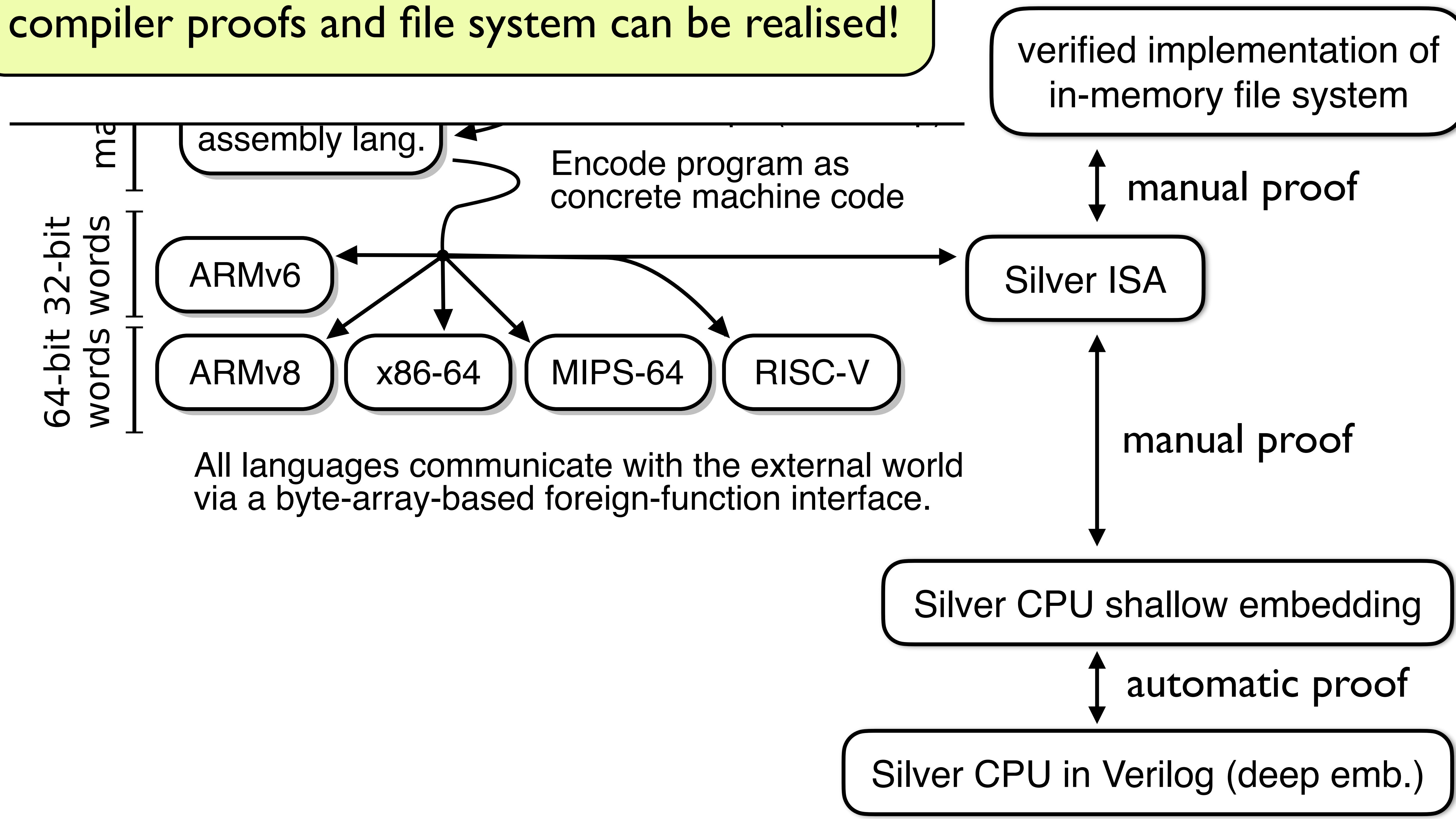
compiler work produced end-to-end verification that ended at the software-hardware interface (ISA).



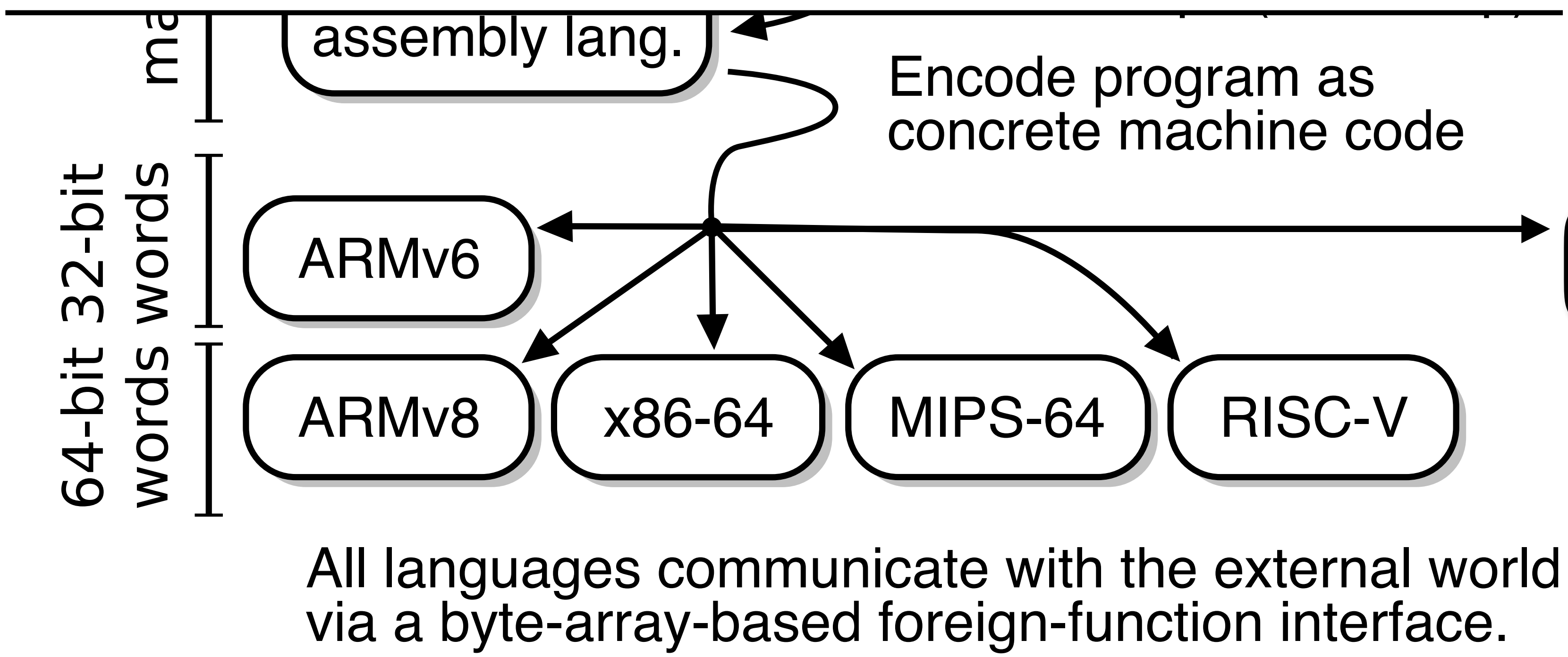
All languages communicate with the external world via a byte-array-based foreign-function interface.

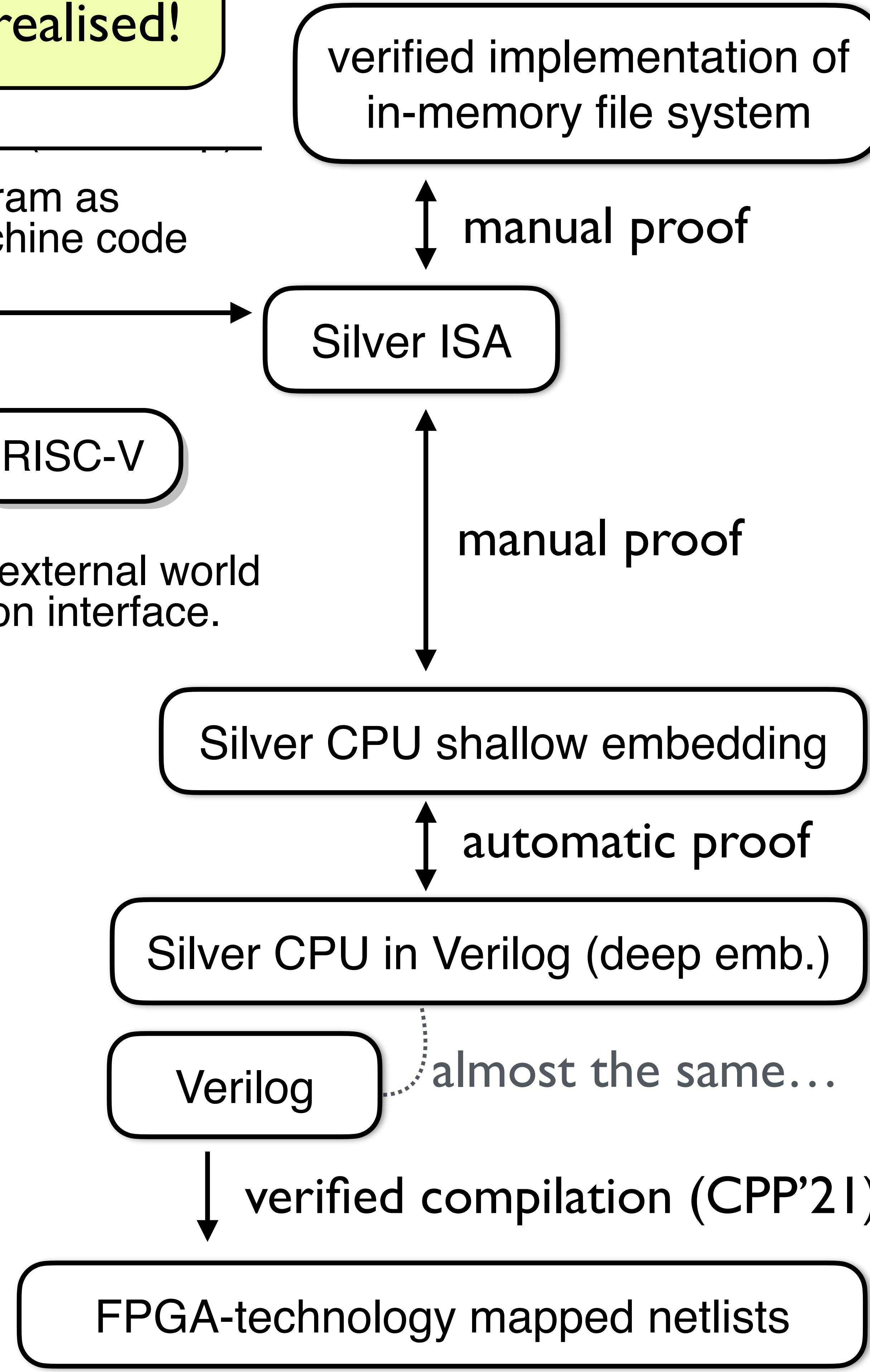
Extending into hardware

Valuable: shows that assumptions made in compiler proofs and file system can be realised!



compiler proofs and file system can be realised!

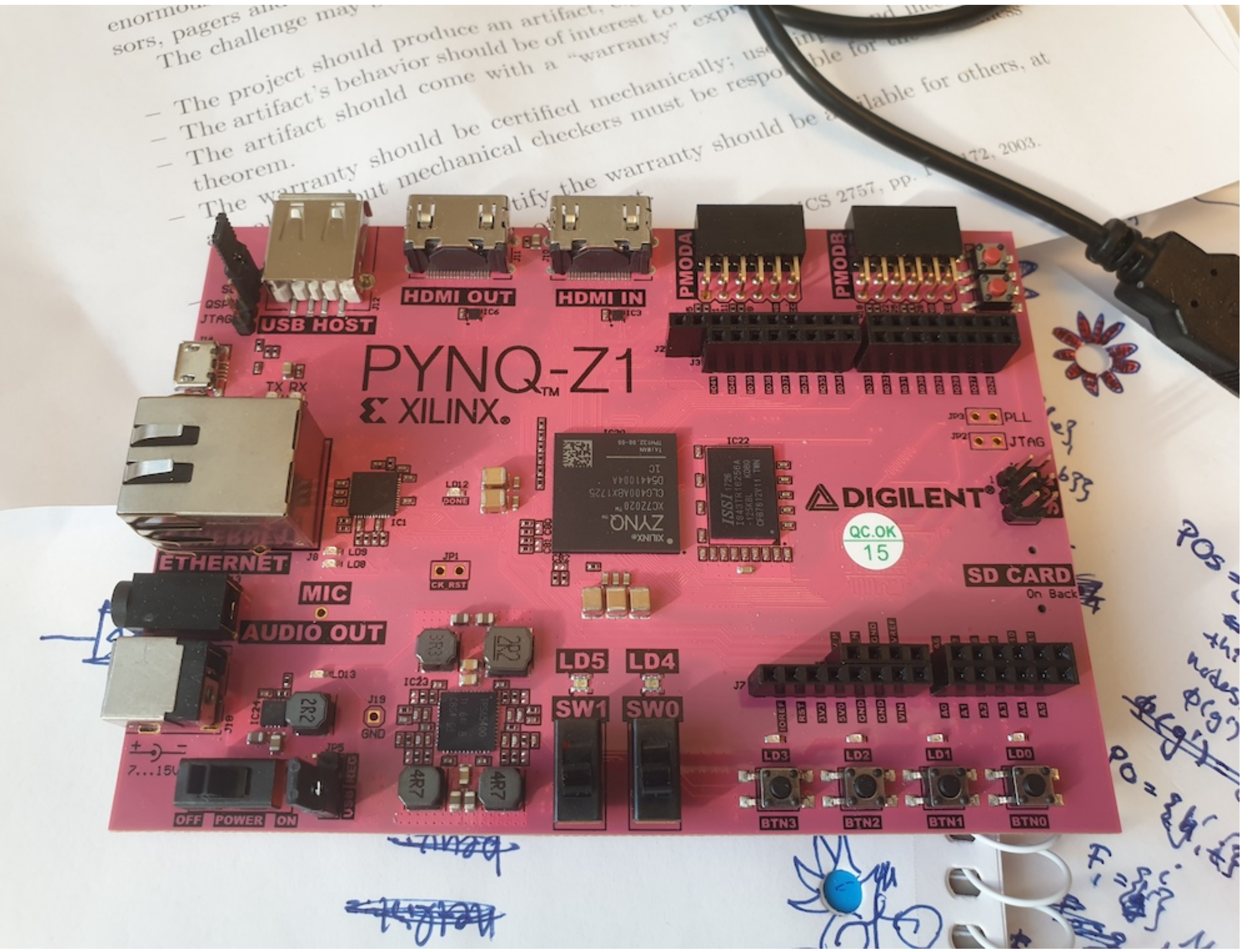




The hardware work is all by:

Andreas Lööw

How real is it?



The verified CPU ***can run non-trivial programs***, including the entire CakeML compiler.

This talk

Part 1: The core of the CakeML project

research questions, main ideas,

verified compilation, end-to-end correctness

Part 2: Extensions and collaborations

hardware, HOL light, other compilers,

proof checkers, collaborations

HOL light

Is an ITP by John Harrison for higher-order logic (HOL):

- Shares the lead in Freek Wiedijk's 100 Theorems challenge with Isabelle/HOL (86/100)
- Major formalizations of various branches of mathematics: (multivariate-) real analysis, complex analysis, …
- Proof of *the Kepler conjecture* (Flyspeck project)

Candle

proved to be sound

Candle is a verified clone of HOL light.

it really looks and feels like HOL light

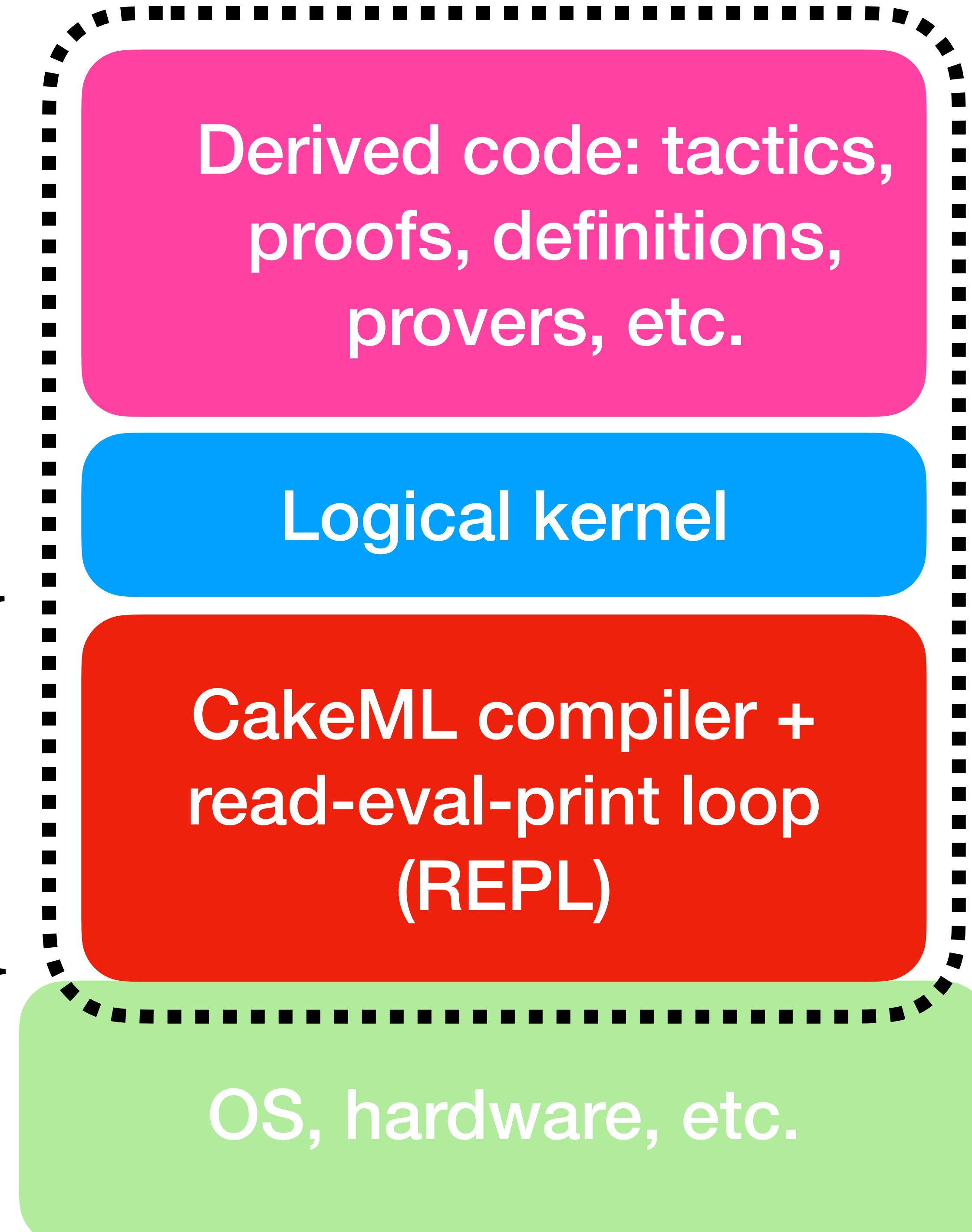
I can show a demo if we have time…

Candle

Candle has an LCF-style design (logical kernel):

we proved an end-to-end soundness theorem:

… machine code running prover will only output theorems that are true according to semantics of higher-order logic.



Candle: A Verified Implementation of HOL Light

Oskar Abrahamsson ✉
Chalmers University of Technology, Gothenburg, Sweden

Magnus O. Myreen ✉
Chalmers University of Technology, Gothenburg, Sweden

Ramana Kumar ✉
London, UK

Thomas Sewell ✉
University of Cambridge, UK

ITP 2022

builds on these results

Cakes that Bake Cakes: Dynamic Computation in CakeML

PLDI 2023

Thomas Sewell

A major effort to insert the CakeML compiler into the CakeML runtime (enables dynamic compilation)

This talk

Part 1: The core of the CakeML project

research questions, main ideas,

verified compilation, end-to-end correctness

Part 2: Extensions and collaborations

hardware, HOL light other compilers,

proof checkers, collaborations

Haskell-like language

PureCake: A Verified Compiler for a Lazy Functional Language

PLDI 2023

Some code PureCake can compile:

```
numbers :: [Integer]
numbers =
  let num n = n : num (n + 1)
  in num 0

factA :: Integer -> Integer -> Integer
factA a n =
  if n < 2 then a
  else factA (a * n) (n - 1)

factorials :: [Integer]
factorials = map (factA 1) numbers
```

```
app :: (a -> IO b) -> [a] -> IO ()
app f l = case l of
            []  -> return ()
            h:t -> do f h ; app f t

main :: IO ()
main = do
  arg1 <- read_arg1
  -- fromString == 0 on malformed input
  let i = fromString arg1
      facts = take i factorials
  app (\i  -> print $ toString i) facts
```

Language	Compiler implementation	Comments on verification
PureCake source		
	lexing, parsing, desugaring	can reject input; this pass is not verified
PureLang: ce from Fig. 2 pure call-by-name (subst. semantics)	split **letrecs** and simplify	proved to preserve ≅ (Sec 3.4)
	run type inferencer	rejects ill-typed programs, proved sound
	demands analysis annotates with seqs	proved to preserve ≈ (Sec 4.4)
	run type inferencer (again)	rejects ill-typed programs, proved sound
front end / *back end*		
	translate into call-by-value; introduce **delay** & **force**; avoid **delay** (**force** (**var** _))	proof was broken down to *five relations*; implementation stays within their composition
ThunkLang: pure call-by-value (subst. semantics)	lift lambdas out of **lets** and **letrecs**, and simplify some **force** expressions	implementation stays within the *transitive closure* of a syntactic relation that preserves semantics
	reformulate to make step into StateLang simpler	proof uses the composition of *three relations*: 1. expands the IO monad into stateful operations 2. replaces Delay, Box, Force with stateful operations 3. tidies up the result
EnvLang: pure call-by-value (env. semantics)	compile **delay**, **box**, **force** and IO monads to stateful ops	
StateLang: impure call-by-value (env. semantics)	push _ · **unit** inwards	implementation stays within the *transitive closure* of a syntactic relation that preserves semantics
	make every lambda bind a variable to align with CakeML	
CakeML source	translate into CakeML and attach helper functions	

CakeML as target language

… and another compiler:

PLOS 2023

Pancake

Verified Systems Programming Made Sweeter

Johannes Åman Pohjola
j.amanpohjola@unsw.edu.au
UNSW Sydney
Australia

Hira Taqdees Syeda*
syedahir@amazon.com
Chalmers University of Technology
Gothenburg, Sweden

Miki Tanaka
miki.tanaka@unsw.edu.au
UNSW Sydney
Australia

Krishnan Winter
k.winter@student.unsw.edu.au
UNSW Sydney
Australia

Tsun Wang Sau
t.sau@student.unsw.edu.au
UNSW Sydney
Australia

Benjamin Nott
b.nott@student.unsw.edu.au
UNSW Sydney
Australia

Tiana Tsang Ung
t.tsangung@student.unsw.edu.au
UNSW Sydney
Australia

Craig McLaughlin
c.mclaughlin@unsw.edu.au
UNSW Sydney
Australia

Remy Seassau†
remy.seassau@cs.ox.ac.uk
UNSW Sydney
Australia

Magnus O. Myreen
myreen@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Michael Norrish
michael.norrish@anu.edu.au
Australian National University
Canberra, Australia

Gernot Heiser
gernot@unsw.edu.au
UNSW Sydney
Australia

This talk

Part 1: The core of the CakeML project

research questions, main ideas,

verified compilation, end-to-end correctness

Part 2: Extensions and collaborations

hardware, HOL light, other compilers,

proof checkers, collaborations

Applications

Some tools generate proof traces / files / logs.

e.g. SAT solvers generate DRAT proofs of UNSAT

Verified checkers = Good applications for CakeML tools!

Great opportunities for collaborations!

cake_lpr demo

Marijn Heule — checker for DRAT / LPR proofs

Jakob Nordström et al. — pseudo boolean checker / VeriPB

Ambros Gleixner — verifier for integer programming results

Eva Darulova — floating-point error bounds

Questions?

Summary

The CakeML project has developed:

- a formal semantics for an SML/OCaml-style language
- a bootstrapped verified compiler
- scalable proof-producing code generation
- separation logic for non-terminating code (liveness)
- a verified space cost semantics (proves absence of OOM)
- efficient verified applications (e.g. UNSAT proof checker)

Current work: using CakeML to implement other languages

Let's work together! Get in touch myreen@chalmers.se

Size of the effort

465 204 lines of definition & tactic proofs

23 918 lines of code for proof automation

1 630 lines of Makefiles and Holmakefiles

21 545 git commits (https://code.cakeml.org/)

Candle demo

```
#
```

Demo of CakeML compiler

```
myreen@oven2:~/demo/latest-version$
```

Demo by Andreas Lööw