CakeML

From functions to machine code

… *with proof all the way!*

Speaker: Magnus Myreen

DeepSpec workshop — PLDI 2018

The CakeML project

Main goals:

An *ITP as a programming environment* with proofs.

Provide a *code extraction mechanism* that relates *via proof* the extracted (machine) code with the original HOL functions.

By-product:

A *realistic verified ML compiler* built as a base for research and teaching.

Example

program that computes frequencies of words

```
hello! said the sailor
the captain and the
navigator replied hello!
```

```
and: 1
captain: 1
hello!: 2
navigator: 1
replied: 1
said: 1
sailor: 1
the: 3
```

Specification

There is some list of words (*ws*) that matches the words in the file …

… the list is sorted …

valid_wordfreq_output $file\_contents\ output \iff$

$\exists\, ws.$

set ws = set (words_of $file\_contents$) $\wedge$ sorted $(\lambda\, x\ y.\ x < y)\ ws\ \wedge$

$output$ = concat (map $(\lambda\, w.$ format_output $(w$,frequency $file\_contents\ w))\ ws)$

… and the output has a line per word indicating the frequency of the word.

Pure implementation

Simple functional program in the logic of an ITP:

compute_wordfreq_output $input\_lines =$
map format_output (to_list (foldl insert_line empty_tree $input\_lines$))
insert_line t $line$ $=$ foldl insert_word t (words_of $line$)

idea: builds an ordered binary tree and then flattens it

An easy-to-prove correctness theorem:

$\vdash$ valid_wordfreq_output $file\_contents$
(concat (compute_wordfreq_output (lines_of $file\_contents$)))

Monads for I/O part

```
wordfreq () =
do
  args ← commandline (arguments ());
  filename ← hd args;
  lines ← stdio (inputLinesFrom filename);
  stdio (print_list (compute_wordfreq_output lines))
od otherwise
do
  name ← commandline (name ());
  stdio (print_err (“usage: ” ^ name ^ “ filename\n”))
od
```

monadic function for I/O in “Haskell-style”

Verification easy thanks to straightforward shallow embedding.

Correctness of compiler output

most compiler verification projects promise this, but few deliver fully

The CakeML toolchain allows ***transportation***
of correctness properties to the compiler output.

CakeML delivers:

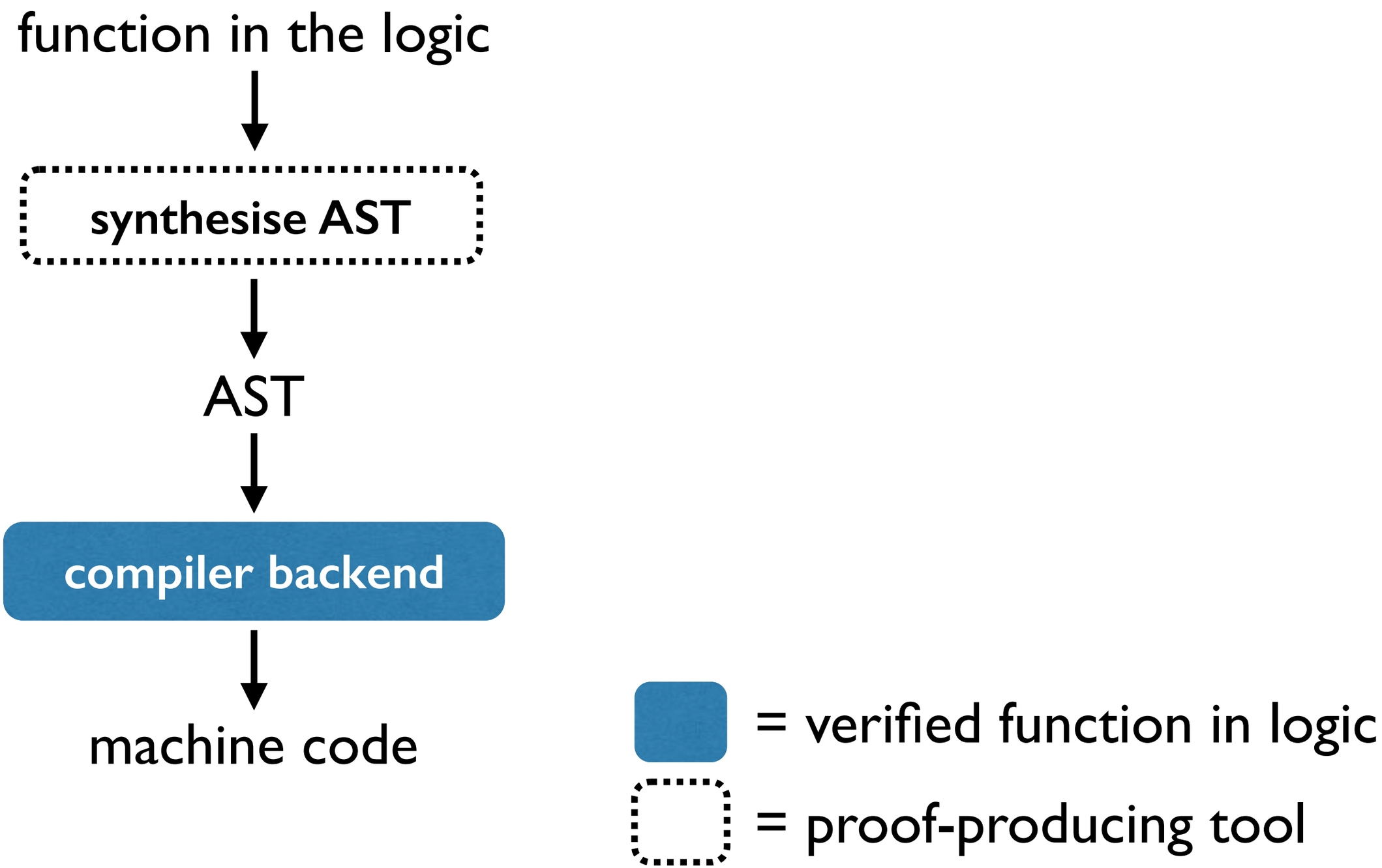
theorem proved inside an ITP

$$
\begin{array}{l}
\vdash \mathsf{wfCL}\ [pname;\ fname] \wedge \mathsf{wfFS}\ fs \wedge \mathsf{hasFreeFD}\ fs \wedge \\
\quad \mathsf{get\_file\_contents}\ fs\ fname = \mathsf{Some}\ file\_contents \wedge \\
\quad \mathsf{x64\_installed}\ \mathsf{compiler\_output}\ (\mathsf{basis\_ffi}\ [pname;\ fname]\ fs)\ mc\ ms \Rightarrow \\
\quad \exists\, io\_events\ ascii\_output. \\
\qquad \mathsf{machine\_sem}\ mc\ (\mathsf{basis\_ffi}\ [pname;\ fname]\ fs)\ ms \subseteq \\
\qquad \mathsf{extend\_with\_resource\_limit}\ \{\ \mathsf{Terminate\ Success}\ io\_events\ \}\ \wedge \\
\qquad \mathsf{extract\_fs}\ fs\ io\_events = \mathsf{Some}\ (\mathsf{add\_stdout}\ fs\ ascii\_output)\ \wedge \\
\qquad \mathsf{valid\_wordfreq\_output}\ file\_contents\ ascii\_output
\end{array}
$$

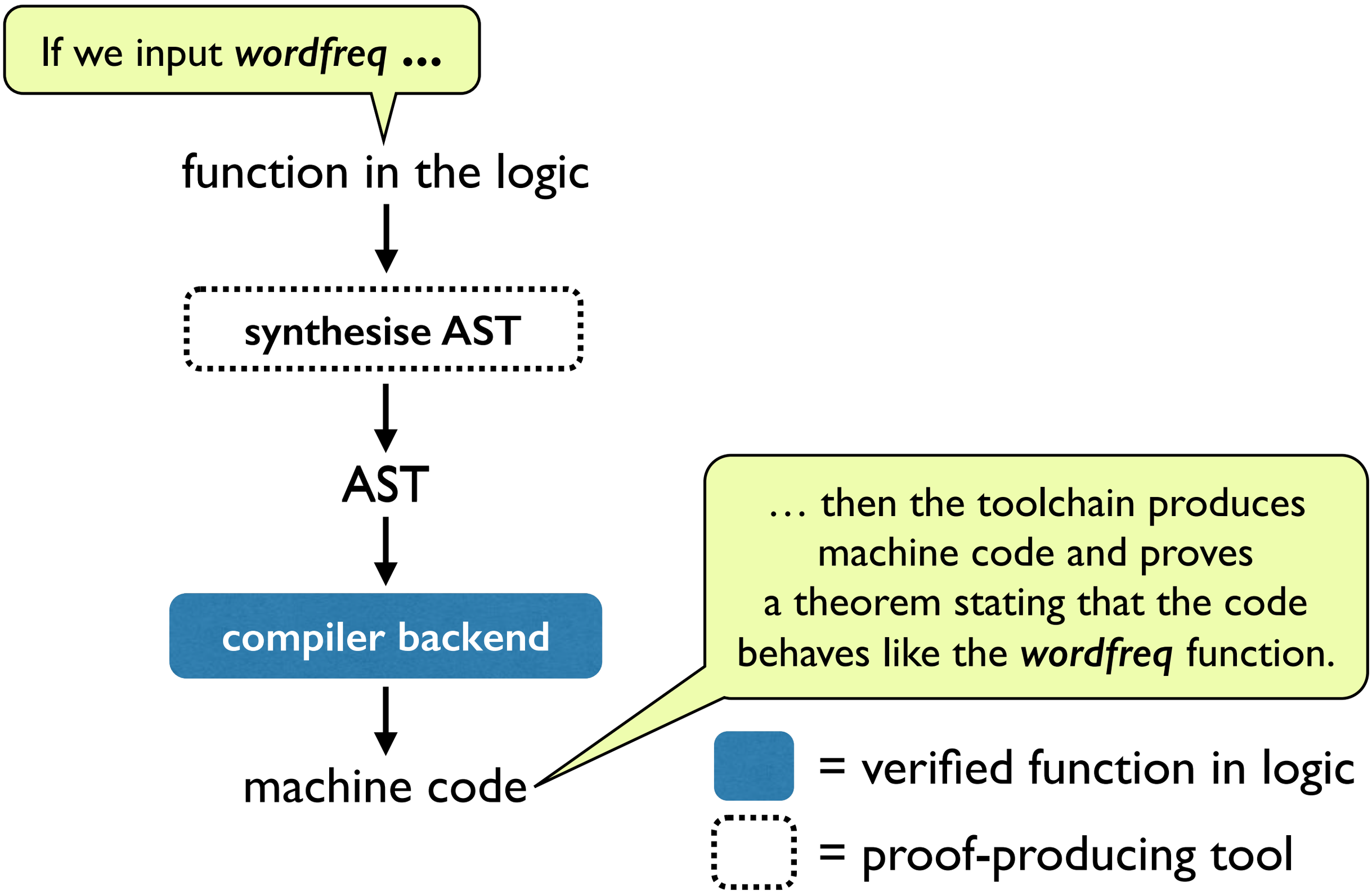
Toolchain

for proof-producing code extraction

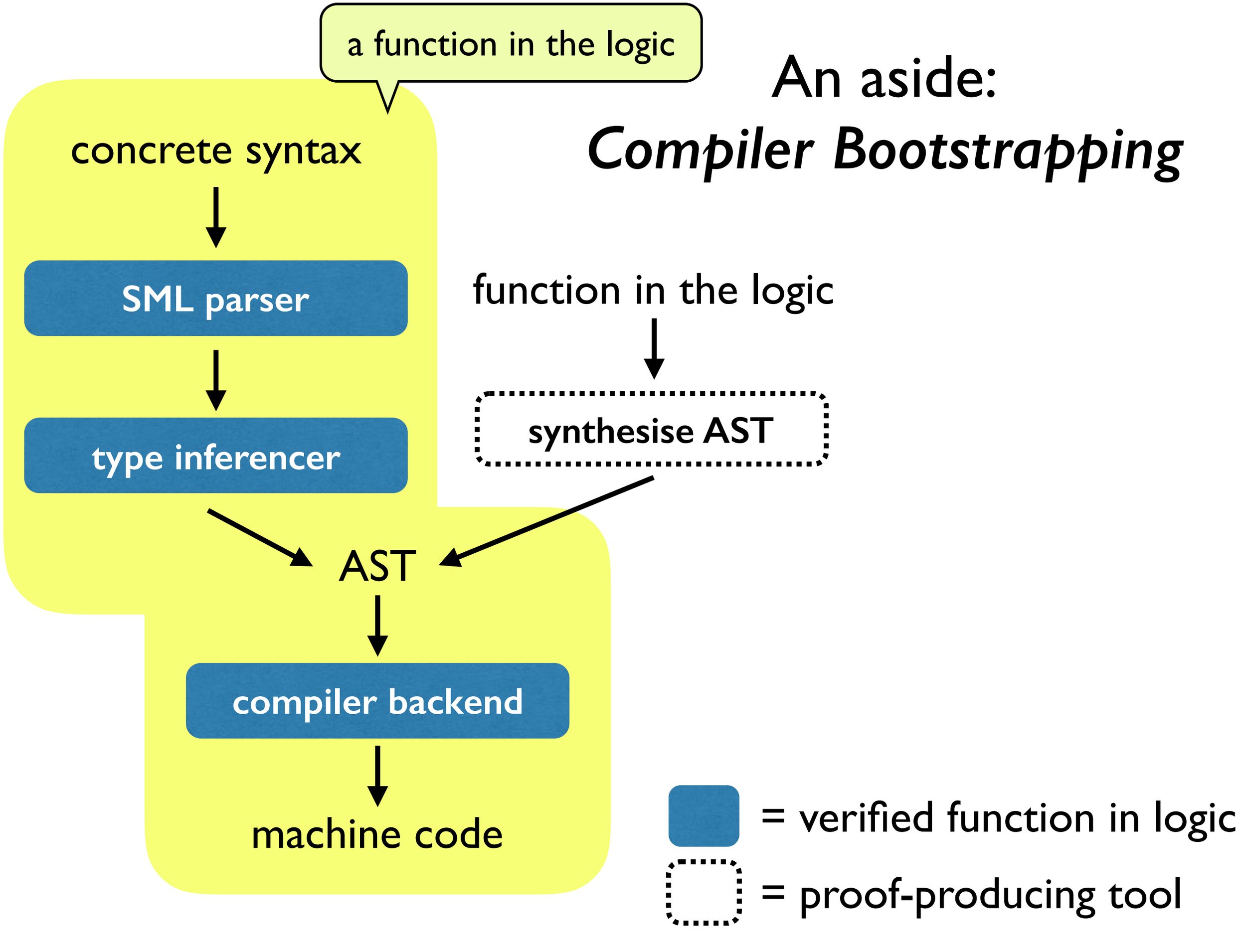
Components of toolchain



Components of toolchain



An aside:
Compiler Bootstrapping



An aside:
Compiler Bootstrapping

If we input
the CakeML compiler …

function in the logic

↓

synthesise AST

AST

↓

compiler backend

↓

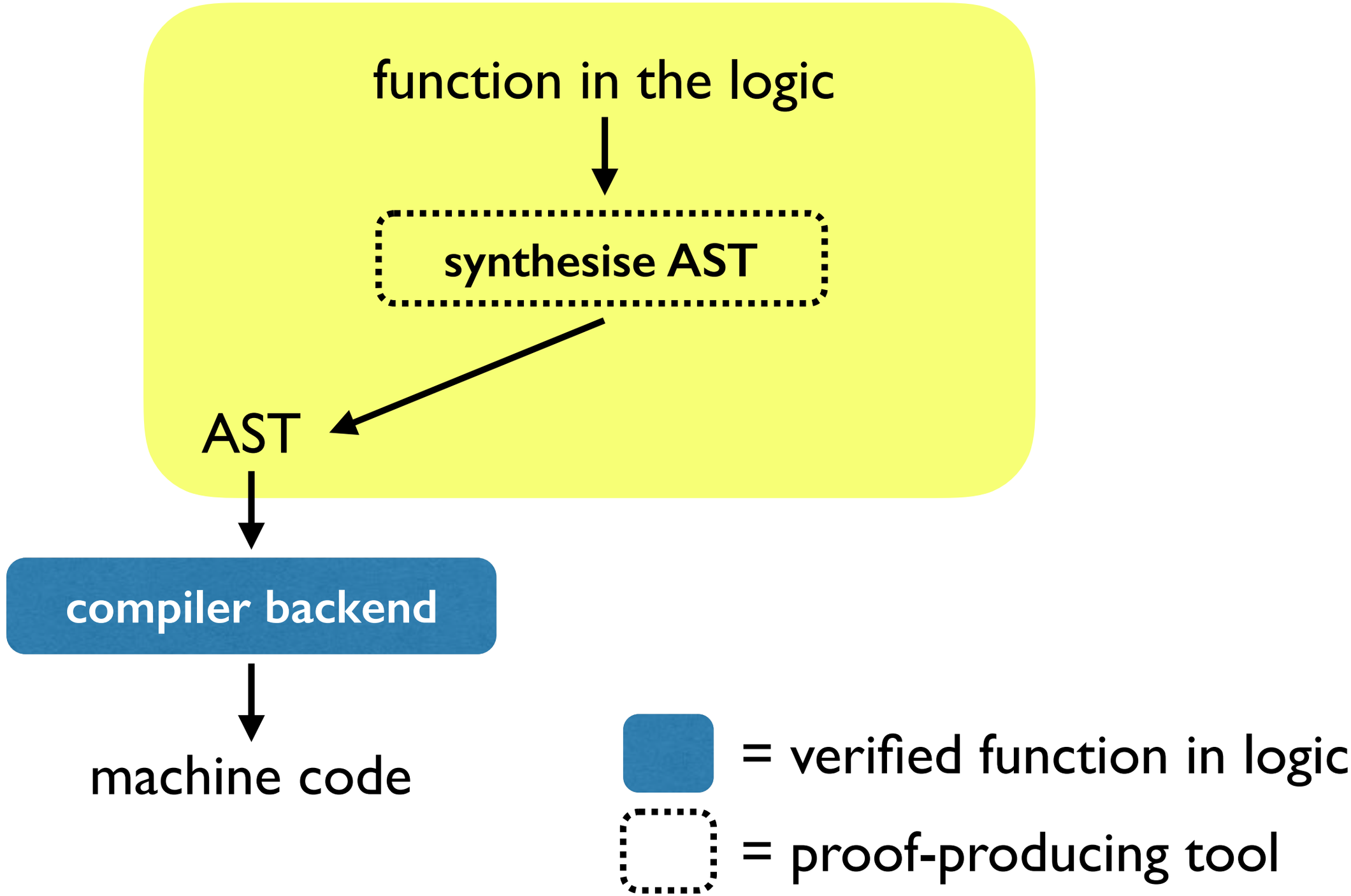
machine code

… then gen. code behaves like
the CakeML compiler function.

= verified function in logic

= proof-producing tool

Proof-producing synthesis



Proof-producing synthesis

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 …

$$\text{Eval}\ env\ \lfloor 1 \rfloor\ (\text{int}\ 1)$$

… integer 1 in the logic.

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

Automation

Each stage automation proves:

$$assumptions \Rightarrow \mathsf{Eval}\ env\ code\ (ref\_inv\ t)$$

Examples:

$$\vdash \mathsf{T} \Rightarrow \mathsf{Eval}\ env\ \lfloor \mathtt{1} \rfloor\ (\mathsf{int}\ 1)$$

$$\vdash \mathsf{Eval}\ env\ \lfloor \mathtt{x} \rfloor\ (\mathsf{int}\ x) \Rightarrow \mathsf{Eval}\ env\ \lfloor \mathtt{x} \rfloor\ (\mathsf{int}\ x)$$

Automation

Examples:

$$\vdash \mathsf{T} \Rightarrow \mathsf{Eval}\ env\ \lfloor \mathtt{1} \rfloor\ (\mathsf{int}\ 1)$$

$$\vdash \mathsf{Eval}\ env\ \lfloor \mathtt{x} \rfloor\ (\mathsf{int}\ x) \Rightarrow \mathsf{Eval}\ env\ \lfloor \mathtt{x} \rfloor\ (\mathsf{int}\ x)$$

Lemmas ...

$$\begin{array}{l} \vdash \mathsf{Eval}\ env\ x_1\ (\mathsf{int}\ n_1) \Rightarrow \\ \quad \mathsf{Eval}\ env\ x_2\ (\mathsf{int}\ n_2) \Rightarrow \\ \quad \mathsf{Eval}\ env\ \lfloor x_1\ \mathsf{+}\ x_2 \rfloor\ (\mathsf{int}\ (n_1 + n_2)) \end{array}$$

... are used to prove compound terms:

$$\vdash \mathsf{Eval}\ env\ \lfloor \mathtt{x} \rfloor\ (\mathsf{int}\ x) \Rightarrow \mathsf{Eval}\ env\ \lfloor \mathtt{x\ +\ 1} \rfloor\ (\mathsf{int}\ (x + 1))$$

Closures are similar

Lemma for lambda:

$$\vdash (\forall\, v\ x.\ a\ x\ v \Rightarrow \mathsf{Eval}\ (env\ [n \mapsto v])\ body\ (b\ (f\ x))) \Rightarrow$$
$$\mathsf{Eval}\ env\ \lfloor \texttt{fn n => } body \rfloor\ ((a \longrightarrow b)\ f)$$

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

Allows us to prove:

$$\vdash \mathsf{T} \Rightarrow \mathsf{Eval}\ env\ \lfloor \texttt{fn x => x + 1} \rfloor\ ((\mathsf{int} \longrightarrow \mathsf{int})\ (\lambda\, x.\ x + 1))$$

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

$$\vdash \mathsf{Eval}\ env\ \lfloor \texttt{x} \rfloor\ (\mathsf{int}\ x) \Rightarrow \mathsf{Eval}\ env\ \lfloor \texttt{x + 1} \rfloor\ (\mathsf{int}\ (x + 1))$$

Mini demo

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

Recent addition:

Support for monadic functions for I/O and stateful code

see IJCAR'18 preprint at cakeml.org

(Significantly improved speed of register allocator.)

Synthesis from monadic funs

For pure we use Eval:

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

For monadic functions we use EvalM:

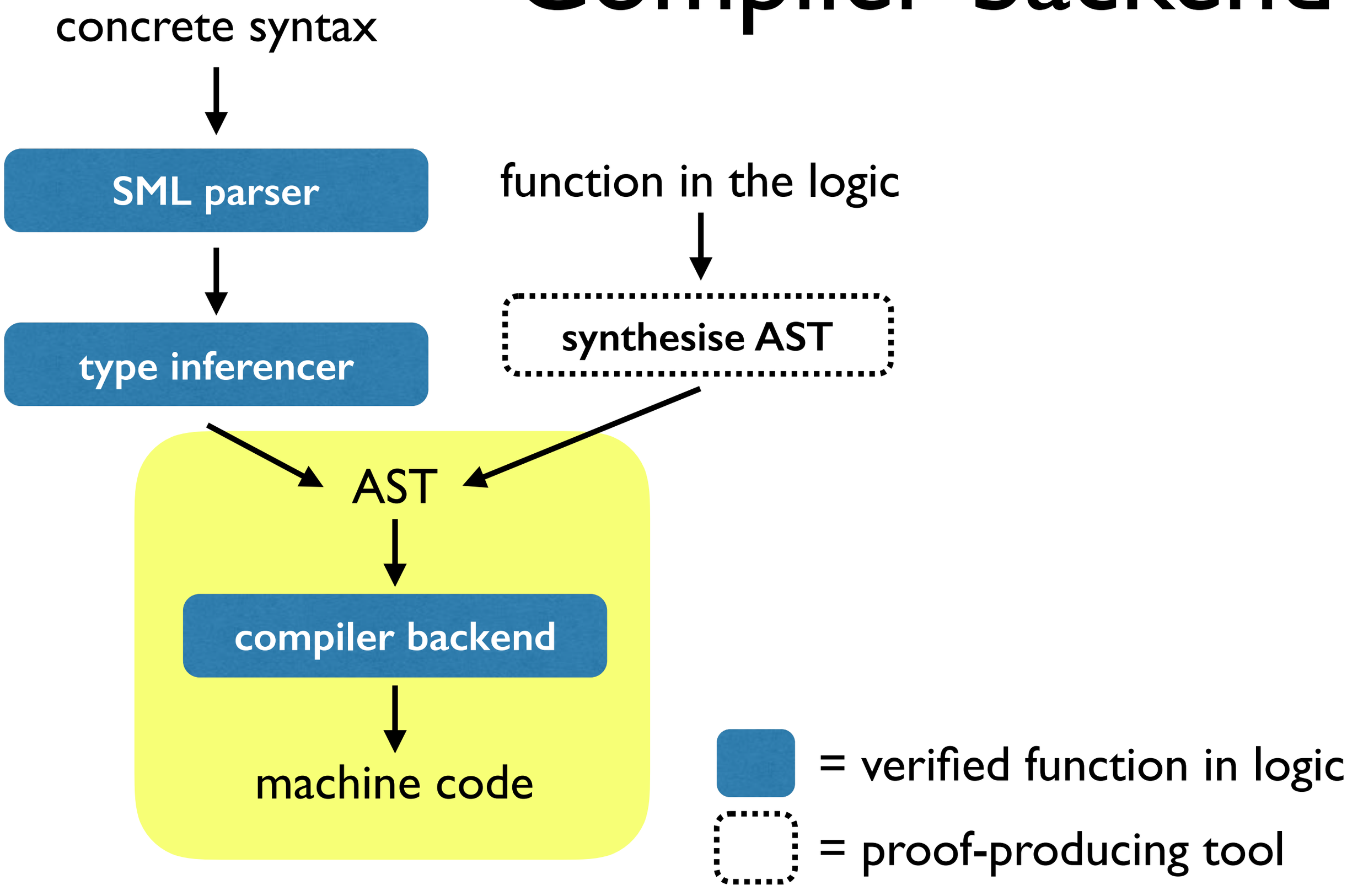
$$\begin{array}{l}\text{EvalM}\ ro\ env\ st\ exp\ P\ H\ \Longleftrightarrow \\ \forall\, s. \\ \quad \text{state\_rel}\ H\ st\ s\ \Rightarrow \\ \quad \exists\, s_2\ res\ st_2\ ck. \\ \quad\quad (\text{evaluate}\ (s\ with\ \text{clock}\ :=\ ck)\ env\ [exp] = (s_2, res)) \wedge \\ \quad\quad P\ st\ (st_2, res) \wedge \text{state\_rel\_frame}\ ro\ H\ (st, s)\ (st_2, s_2)\end{array}$$

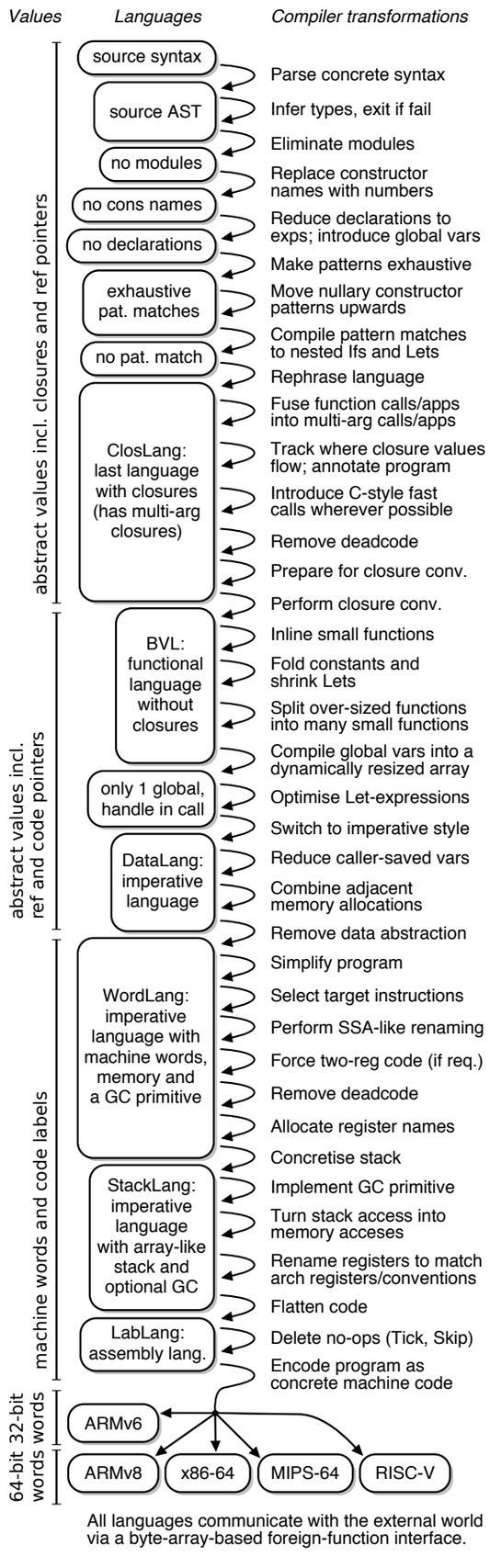
H is heap assertion relating monad state to CakeML state

where: $\text{state\_rel}\ (h, p)\ st\ s\ \Longleftrightarrow\ (h\ st * \mathsf{T})\ (\text{st2heap}\ p\ s)$

separating conjunction

Compiler backend





Compiler backend:

12 intermediate languages (ILs)

and many within-IL optimisations

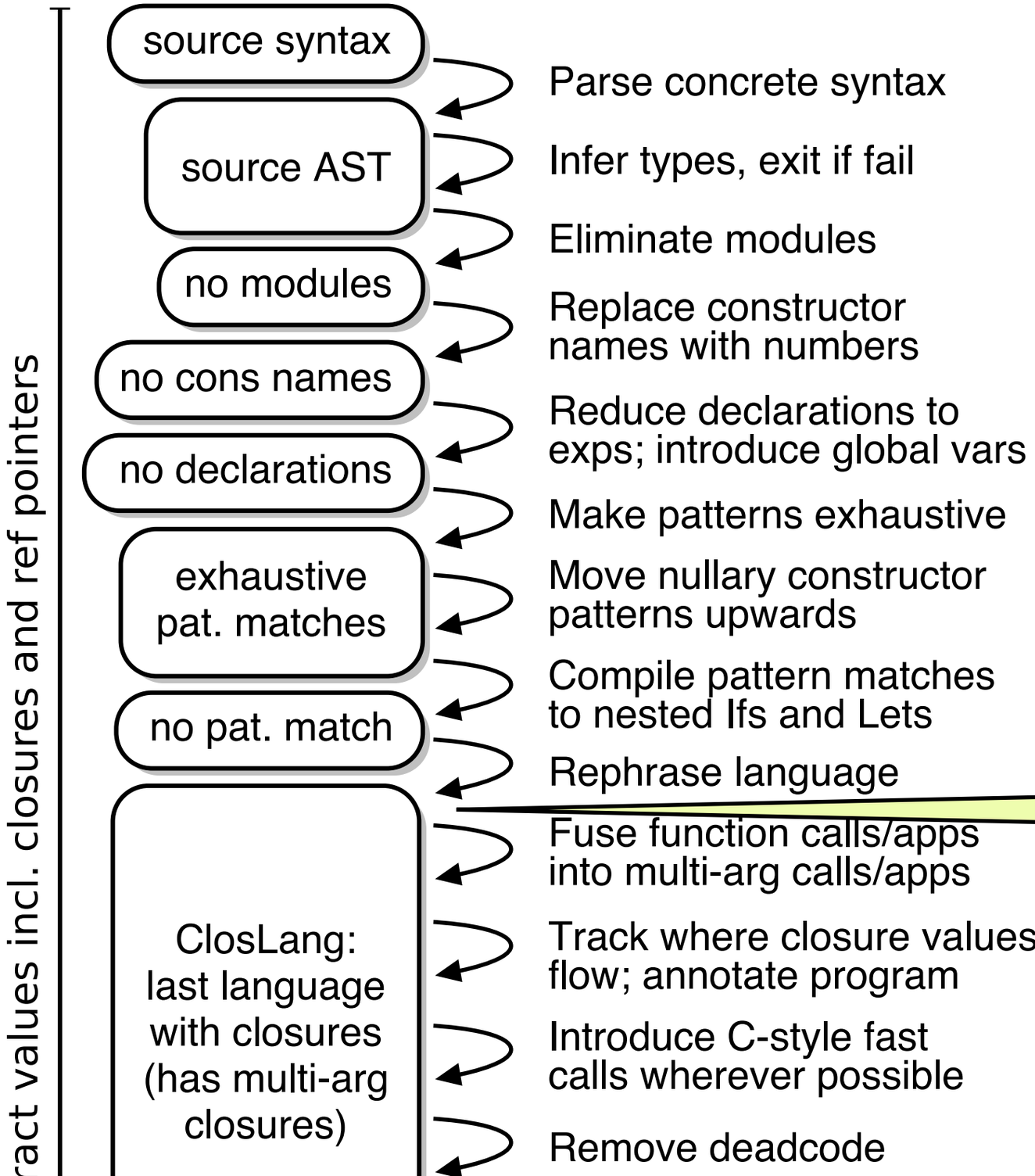
each IL at the right level of abstraction

for the benefit of proofs and compiler implementation

Next slide zooms in

Values used by the semantics

Values *Languages* *Compiler transformations*



Both proved sound and complete.

Parser and type inferencer as before

Early phases reduce the number of language features

Language with multi-argument closures

abstract values incl. closures

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

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.

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

Compile global vars into a dynamically resized array

Simple first-order functional language

abstract values incl. ref and code pointers

only 1 global, handle in call

Optimise Let-expressions

Switch to imperative style

DataLang: imperative language

Reduce caller-saved vars

Combine adjacent memory allocations

Remove data abstraction

Imperative language

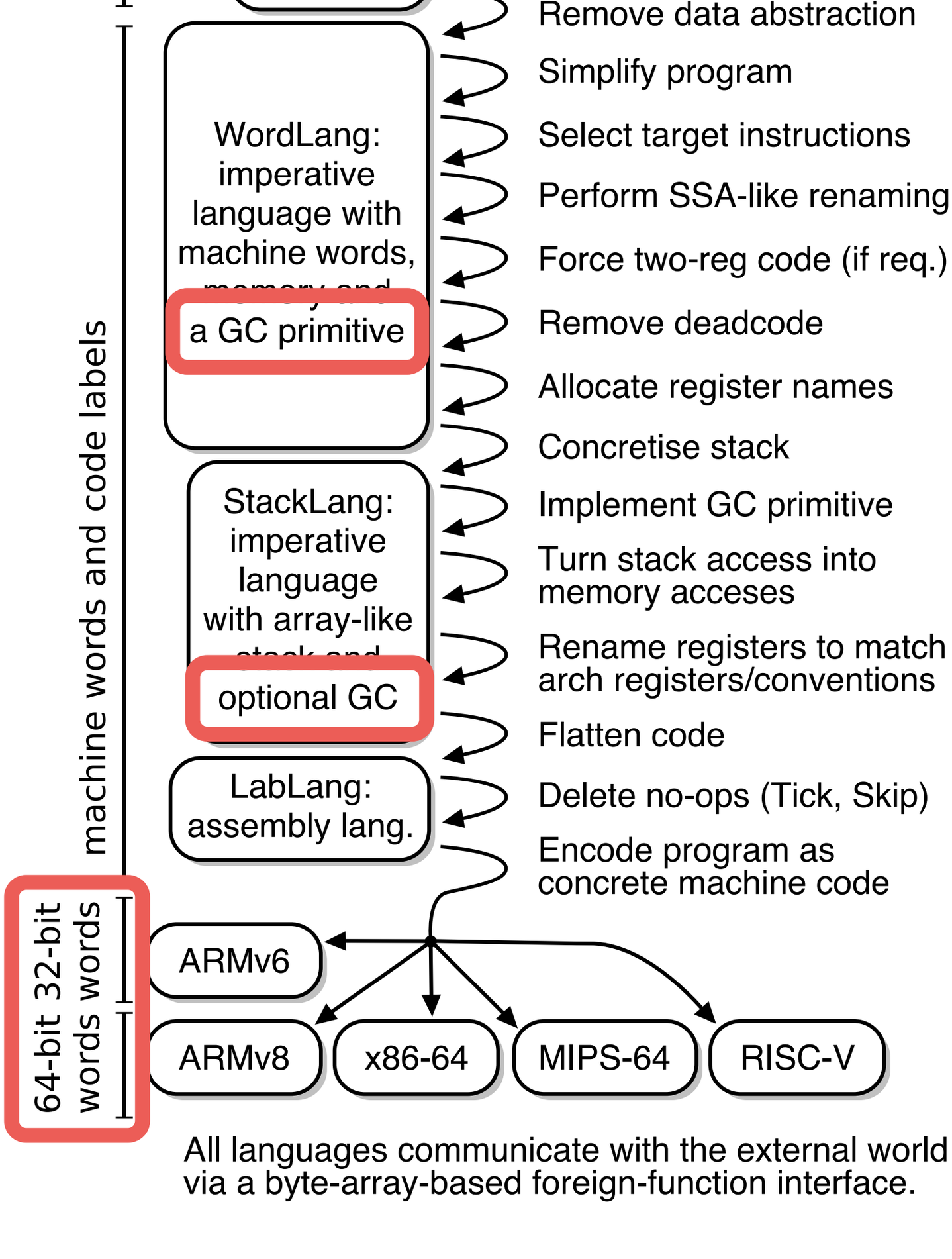
WordLang: imperative

Simplify program

Select target instructions

Machine-like types

Perform SSA-like renaming

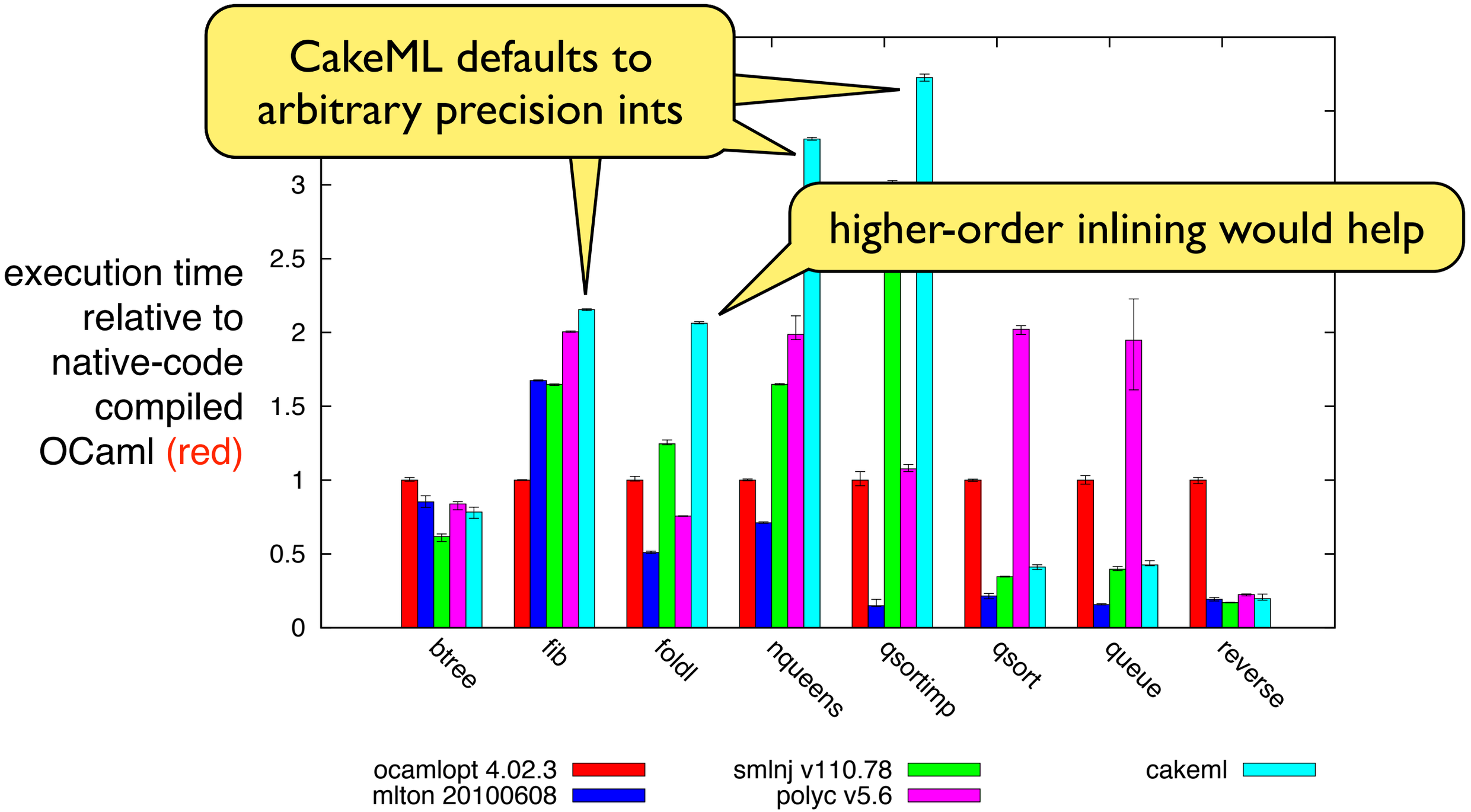


Machine-like types

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

Targets 5 architectures

(Old) Performance numbers



Last slide

Questions?

Summary:

CakeML provides a ***code extraction mechanism***
that relates ***via proof*** the extracted (machine) code
with the original functions from the logic of an ITP.

includes support for I/O and stateful code (monads)

In the pipeline:

- A major new language feature (and applications)
- Support for proving correctness of infinite executions

Get involved! → CakeML is an open project: cakeml.org

Extra: Compiler correctness

$$
\begin{array}{l}
\vdash \mathsf{config\_ok}\ cc\ mc \Rightarrow \\
\quad \mathsf{case\ compile}\ cc\ prelude\ input\ \mathsf{of} \\
\quad\ \mathsf{Success}\ (bytes,ffi\_limit)\ \Rightarrow \\
\quad\quad \exists\, behaviours. \\
\quad\quad\ \mathsf{cakeml\_semantics}\ ffi\ prelude\ input = \\
\quad\quad\ \mathsf{Execute}\ behaviours \wedge \\
\quad\quad \forall\, ms. \\
\quad\quad\ \mathsf{code\_installed}\ (bytes,cc,ffi,ffi\_limit,mc,ms) \Rightarrow \\
\quad\quad\quad \mathsf{machine\_sem}\ mc\ ffi\ ms \subseteq \\
\quad\quad\quad \mathsf{extend\_with\_resource\_limit}\ behaviours \\
\quad | \ \mathsf{Failure\ ParseError}\ \Rightarrow \\
\quad\quad \mathsf{cakeml\_semantics}\ ffi\ prelude\ input = \mathsf{CannotParse} \\
\quad | \ \mathsf{Failure\ TypeError}\ \Rightarrow \\
\quad\quad \mathsf{cakeml\_semantics}\ ffi\ prelude\ input = \mathsf{IllTyped} \\
\quad | \ \mathsf{Failure\ CompileError}\ \Rightarrow\ \mathsf{true}
\end{array}
$$