Verification of an ML compiler

Lecture 4: Compiler bootstrapping and new directions

Marktoberdorf Summer School MOD 2017

Magnus O. Myreen, Chalmers University of Technology

Bootstrapping

Some people say:

a compiler is not "real" until it is self-hosting

meaning: it can compile itself

"bootstrap"

Bootstrapping

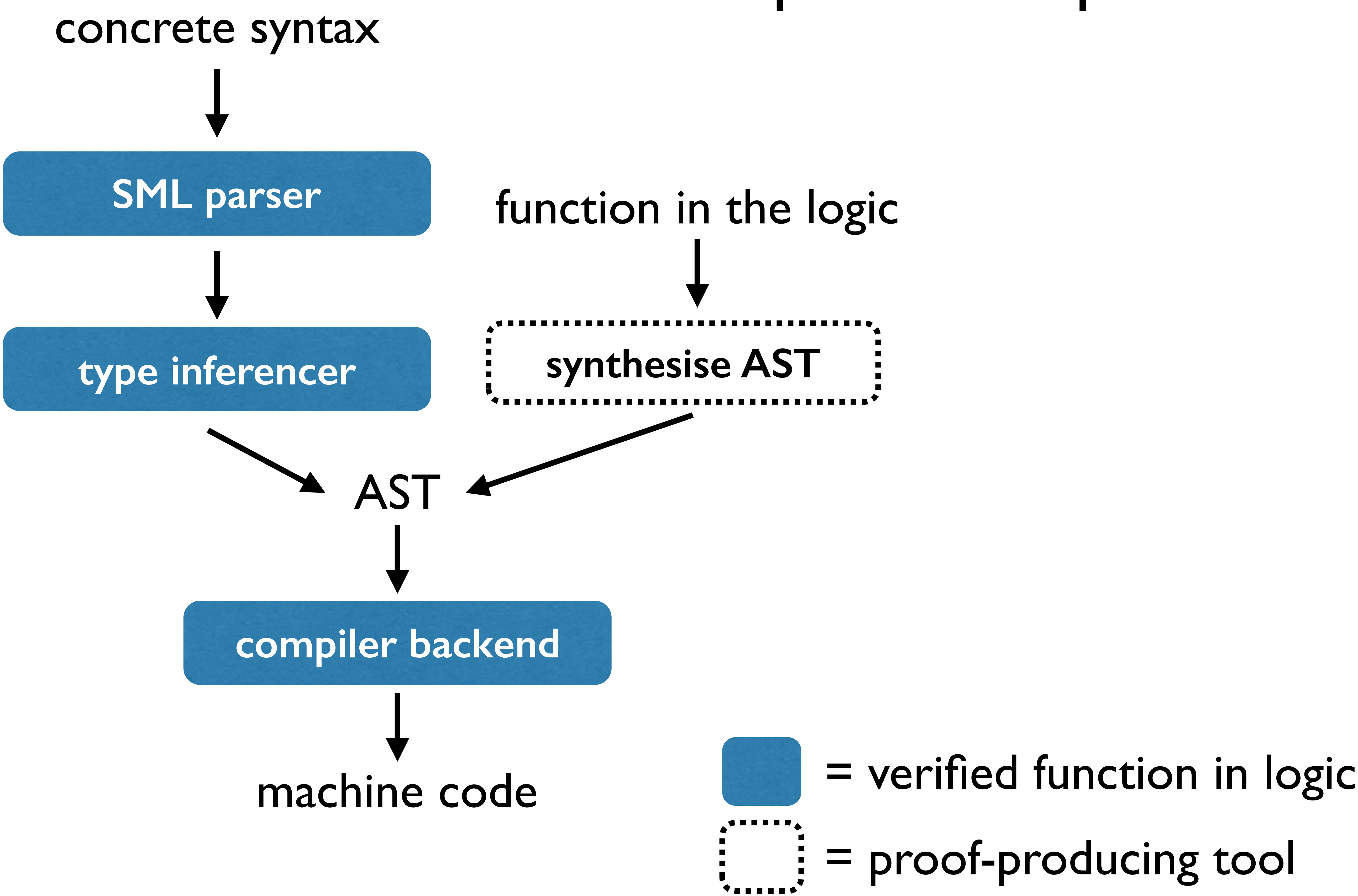
Some people say:

a compiler is not "real" until it is self-hosting

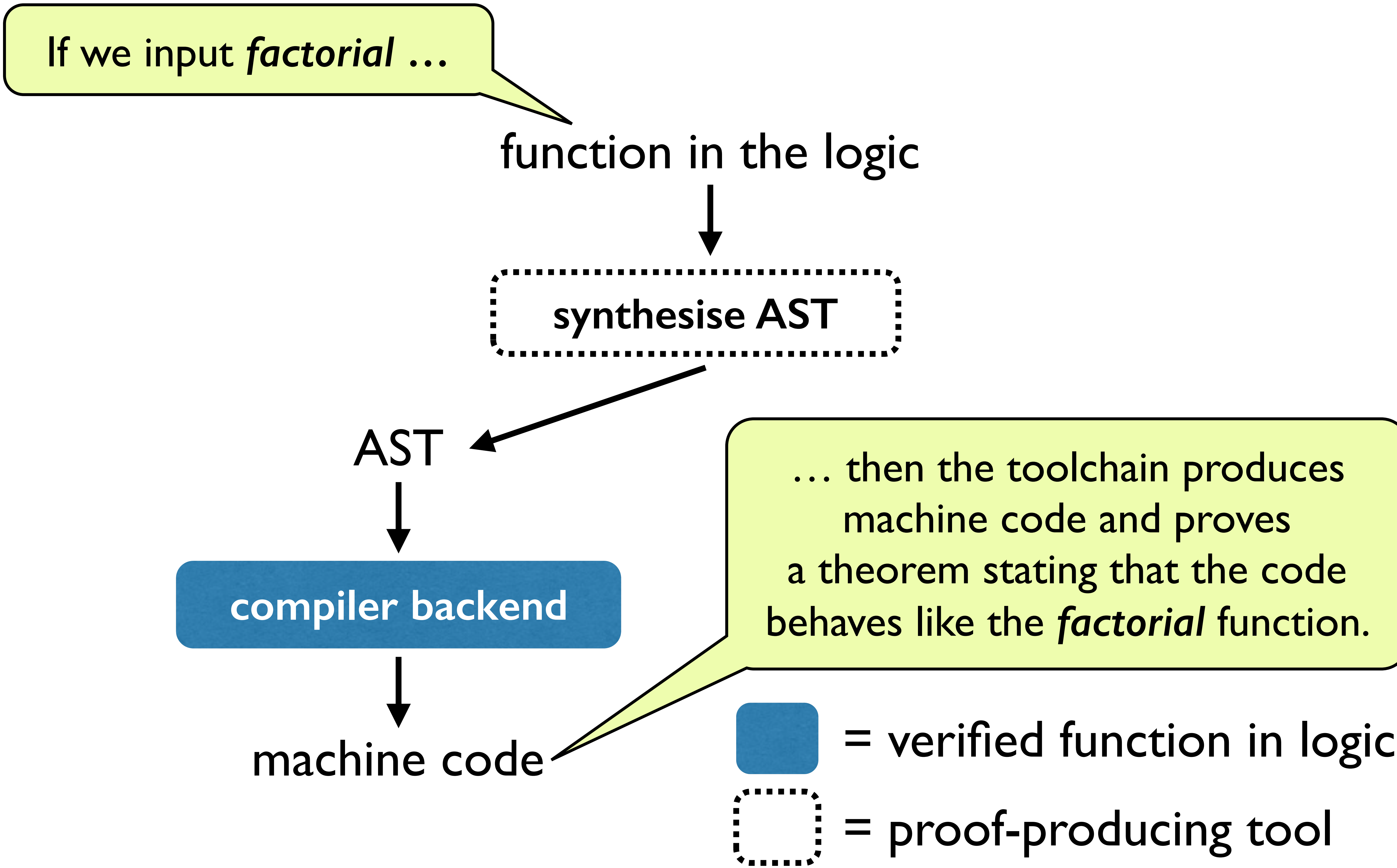
This lecture:

1. how the CakeML compiler was bootstrapped
2. where CakeML is going next (version 3)

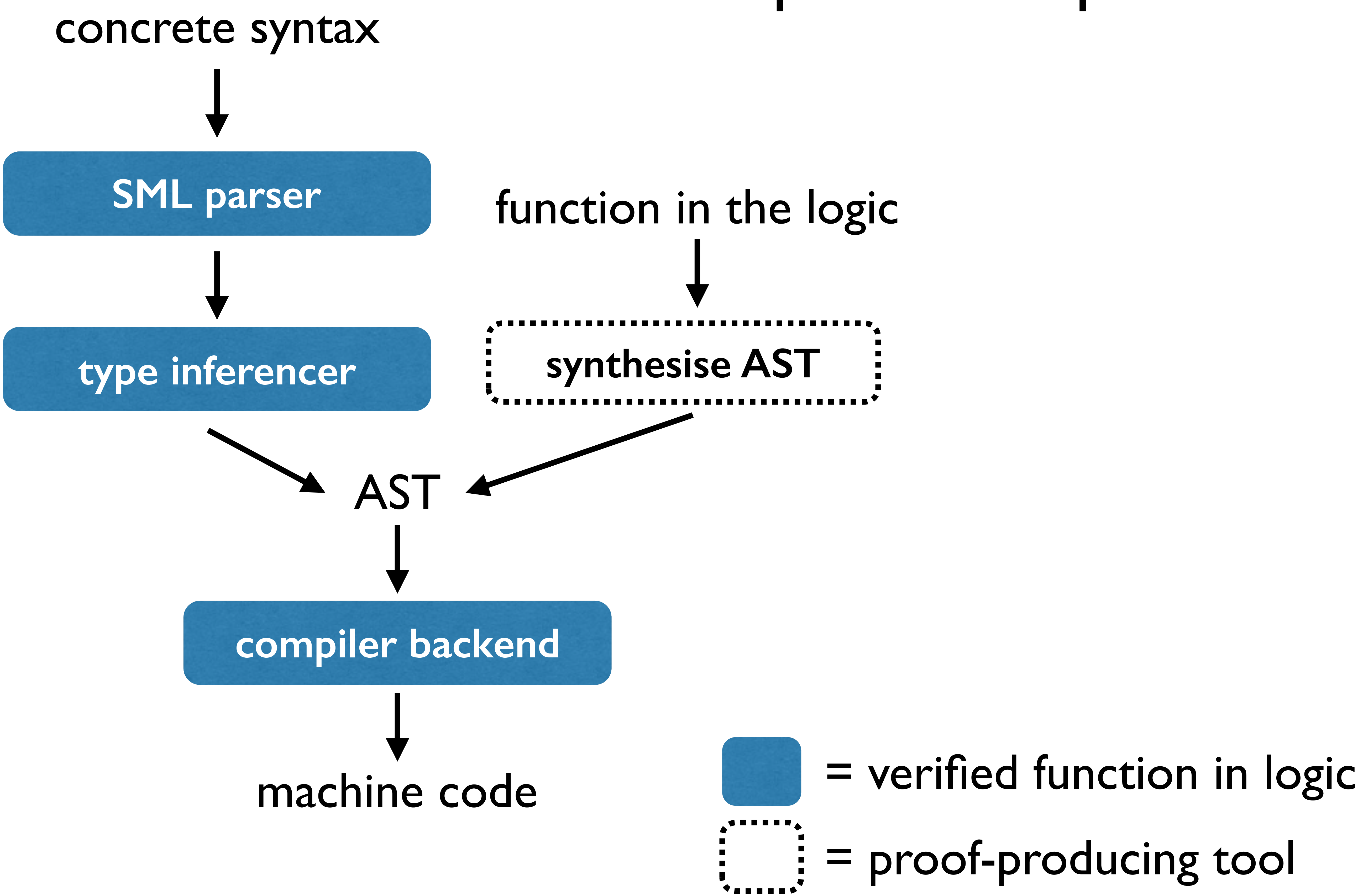
Required components

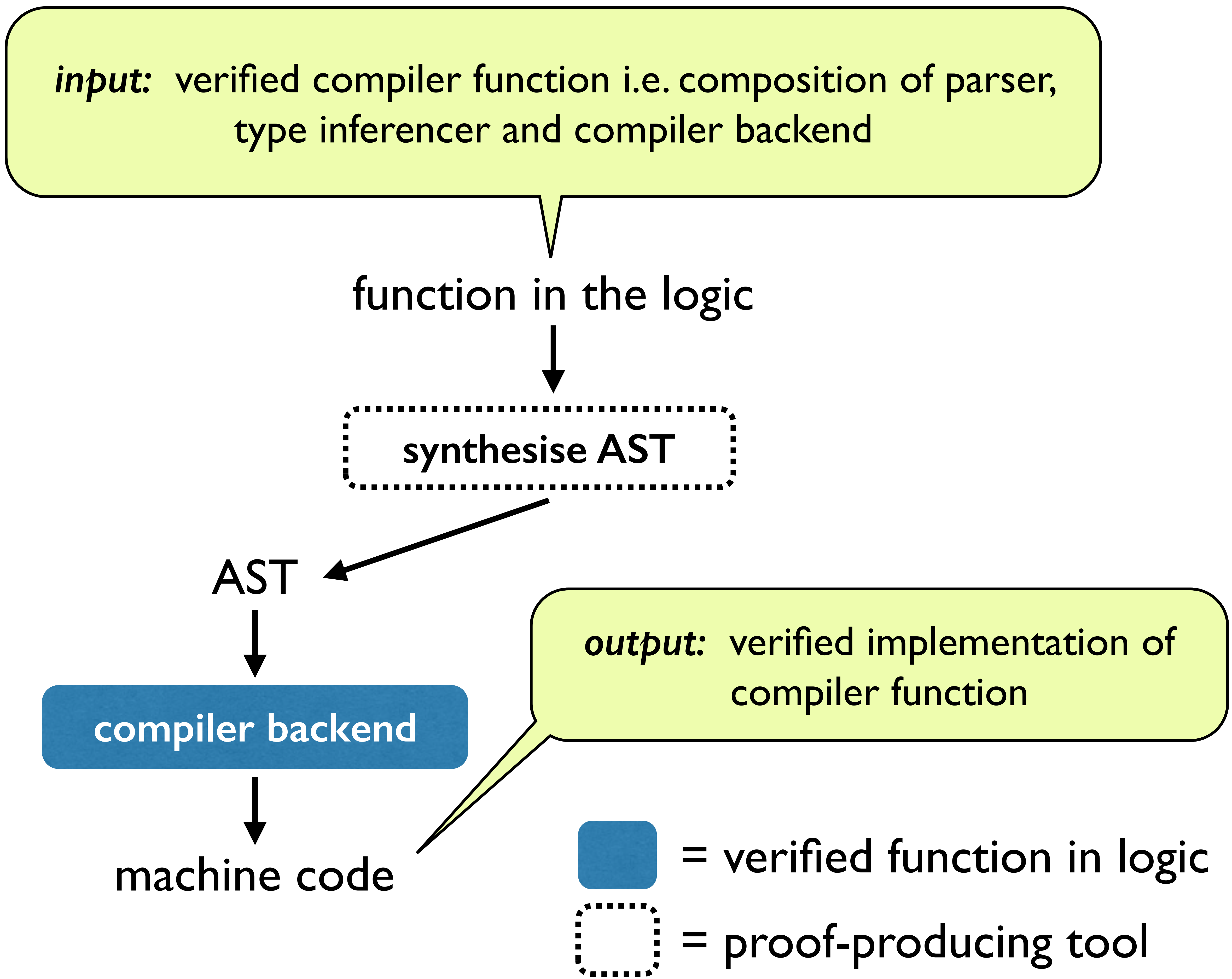


Proof producing code generation



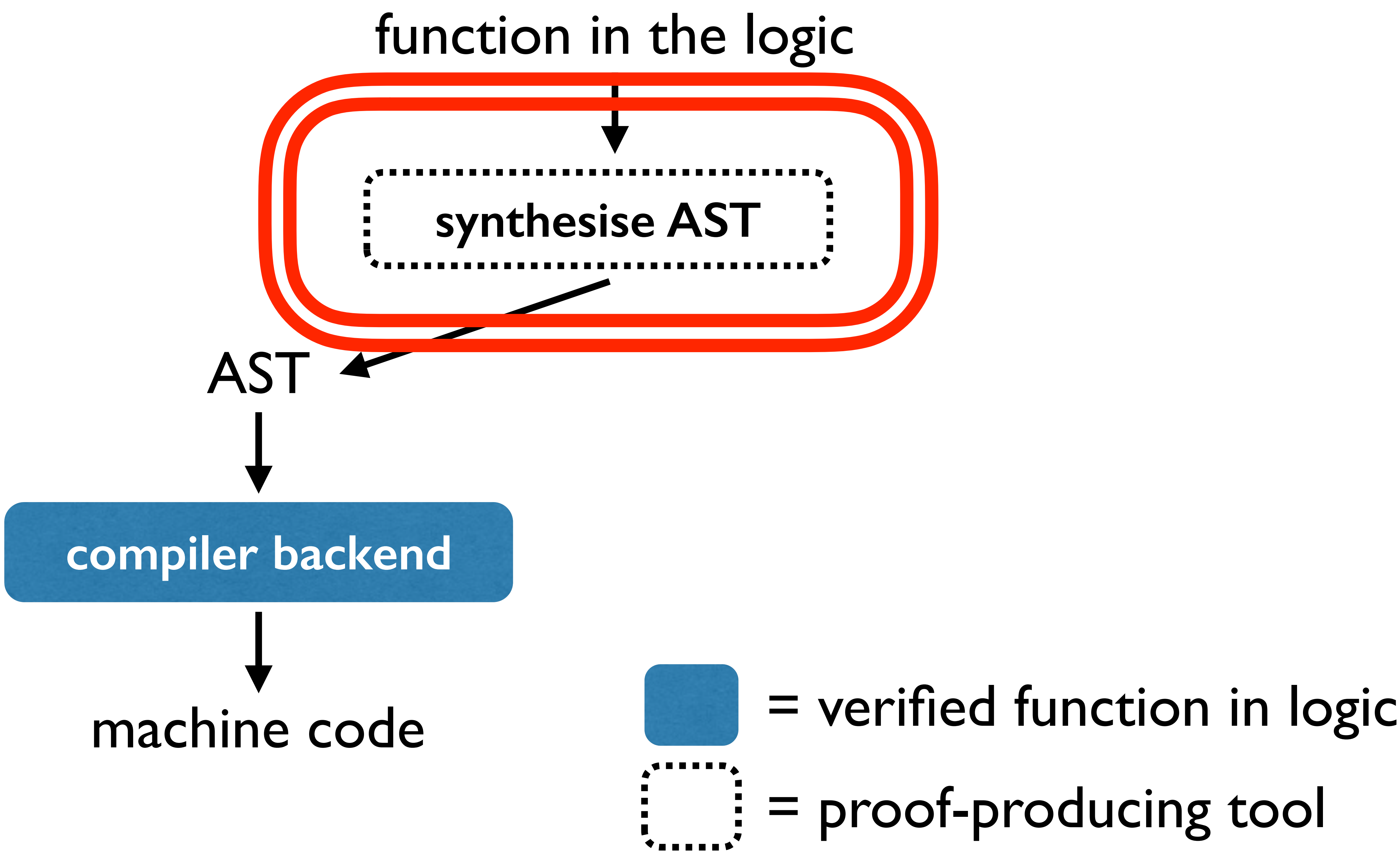
Required components





input: verified compiler function i.e. composition of parser, type inferencer and compiler backend
function in the logic
synthesise AST
AST
compiler backend
machine code
output: verified implementation of compiler function
= verified function in logic
= proof-producing tool

The Missing Piece



“The CakeML Translator”

function in the logic

↓

synthesise AST

↓

AST

Performs a bottom-up translation of HOL functions into AST.

Proves: the generated AST behaves like the HOL function

expressed as a ***typed logical relation***

Definitions

The translator states its theorems using a relation called Eval.

$$\begin{array}{l} \mathsf{Eval}\ \mathit{env}\ \mathit{exp}\ \mathit{post} = \\ \quad \exists \mathit{val}.\ \mathsf{evaluate}\ \mathit{env}\ \mathit{exp}\ \varepsilon = (\mathsf{Rval}\ \mathit{val}, \varepsilon)\ \wedge\ \mathit{post}\ \mathit{val} \end{array}$$

We can express relationships between HOL and CakeML:

$$\mathsf{Eval}\ \mathit{env}\ \lfloor 5 \rfloor\ (\mathsf{int}\ 5)$$

HOL integer

CakeML AST

if we define int as follows:

$$\mathsf{int}\ i \quad = \quad \lambda v.\ (v = \mathsf{Lit}\ i) \qquad \text{where } i \text{ is an integer}$$

relates HOL *i* with CakeML value *v*, if *v* is an integer

Bottom-up construction

Each stage derives a theorem of the form:

$$\textit{assumptions} \implies \mathsf{Eval}\ \textit{env}\ \textit{exp}\ (\textit{inv}\ t)$$

Examples:

$$\begin{aligned}
\mathsf{Eval}\ \textit{env}\ \lfloor \mathtt{n} \rfloor\ (\mathsf{int}\ n) \quad &\implies \quad \mathsf{Eval}\ \textit{env}\ \lfloor \mathtt{n} \rfloor\ (\mathsf{int}\ n) \\
\mathsf{true} \quad &\implies \quad \mathsf{Eval}\ \textit{env}\ \lfloor 5 \rfloor\ (\mathsf{int}\ 5)
\end{aligned}$$

Bottom-up construction

Examples:

$$\text{Eval } env \ \lfloor \mathtt{n} \rfloor \ (\text{int } n) \quad \Longrightarrow \quad \text{Eval } env \ \lfloor \mathtt{n} \rfloor \ (\text{int } n)$$

$$\text{true} \quad \Longrightarrow \quad \text{Eval } env \ \lfloor \mathtt{5} \rfloor \ (\text{int } 5)$$

Lemmas are used to translate compound terms:

$$\begin{array}{l} \forall e_1 \ e_2 \ i \ j. \\ \quad \text{Eval } env \ \lfloor e_1 \rfloor \ (\text{int } i) \ \wedge \\ \quad \text{Eval } env \ \lfloor e_2 \rfloor \ (\text{int } j) \Longrightarrow \\ \quad \text{Eval } env \ \lfloor e_1 \ \mathtt{+} \ e_2 \rfloor \ (\text{int } (i+j)) \end{array}$$

Example:

$$\text{Eval } env \ \lfloor \mathtt{n} \rfloor \ (\text{int } n) \Longrightarrow \text{Eval } env \ \lfloor \mathtt{n+5} \rfloor \ (\text{int } (n+5))$$

Functions

A function *f* from int to int:

this arrow combines relations

$$\text{Eval}\ env\ \lfloor \texttt{f} \rfloor\ ((\text{int} \to \text{int})\ f)$$

The arrow is defined as follows:

$$(a \to b)\ f$$
$$=$$
$$\lambda cl.\ \forall x\ v.\ a\ x\ v \Longrightarrow$$
$$\exists u.\ \text{evaluate\_closure}\ cl\ v\ u \land b\ (f\ x)\ u$$

for any inputs x and v, related by a …

… when closure cl is applied to v …

… some u is returned such that b relates it with the result of *f x*

Function Application

$$\text{Eval } \mathit{env}\ \lfloor \mathtt{f} \rfloor\ ((a \to b)\ f) \land$$
$$\text{Eval } \mathit{env}\ \lfloor \mathtt{x} \rfloor\ (a\ x) \Longrightarrow$$
$$\text{Eval } \mathit{env}\ \lfloor \mathtt{f\ \ x} \rfloor\ (b\ (f\ x))$$

Example

We can now derive:

$$\text{Eval } \mathit{env}\ \lfloor \texttt{f} \rfloor\ ((\mathsf{int} \to \mathsf{int})\ f) \implies \text{Eval } \mathit{env}\ \lfloor \texttt{f 5} \rfloor\ (\mathsf{int}\ (f\ 5))$$

assumption about variable *f*

conclusion about *f 5*

Lambda-abstraction

instantiates the env

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

smaller env

Example:

$$\mathsf{Eval}\ env\ \lfloor \texttt{fn n => n+5} \rfloor\ ((\mathsf{int} \to \mathsf{int})\ (\lambda n.\ n + 5))$$

derived from theorem with assumption about n

Type variables

$$\text{Eval } env \lfloor \texttt{fn f => fn x => f (f x)} \rfloor$$
$$(((a \to a) \to a \to a)\ (\lambda f\ x.\ f\ (f\ x)))$$

variable with type α → v → bool

has type α

Crucially: *a* can be instantiated once a more concrete type is used

e.g. with the **int** relation which has type variable with type int → v → bool

User-defined constant

Suppose twice *is defined to be* $\lambda f\ x.\ f\ (f\ x)$

One derives:

$$\text{Eval}\ env\ \lfloor\texttt{fn f => fn x => f (f x)}\rfloor$$
$$(((a \to a) \to a \to a)\ (\lambda f\ x.\ f\ (f\ x)))$$

… and then the following:

assumes `twice` is bound in ML *env*

$$\text{DeclAssum}\ \lfloor\texttt{val twice = fn f => fn x => f (f x);}\rfloor\ env \Longrightarrow$$
$$\text{Eval}\ env\ \lfloor\texttt{twice}\rfloor\ (((a \to a) \to a \to a)\ \text{twice})$$

CakeML name `twice` is related to HOL const twice

Algorithm

Function translation is easy in non-recursive case:

Step 1: bottom-up traversal following body of HOL definition

Step 2: replace body with HOL name (rewriting) and "store" CakeML code in env

assumes `twice` is bound in ML *env*

DeclAssum $\lfloor$`val twice = fn f => fn x => f (f x);`$\rfloor$ $env \Longrightarrow$
Eval env $\lfloor$`twice`$\rfloor$ $(((a \to a) \to a \to a)$ twice$)$

CakeML name `twice` is related to HOL name twice

Recursive functions?

$$\text{gcd}\ m\ n\ =\ \text{if}\ 0 < n\ \text{then gcd}\ n\ (m\ \text{mod}\ n)\ \text{else}\ m$$

a translation must make an assumption about the CakeML-HOL relation for gcd

… but that's what we are proving!

Solution

an abbreviation

A restrictive assumption:

restricts the forall inside the arrow

$$\color{red}{\mathrm{A}\ m\ n} = \mathrm{Eval}\ \mathit{env}\ \lfloor \texttt{gcd} \rfloor\ ((\mathrm{eq\ nat}\ m \rightarrow \mathrm{eq\ nat}\ n \rightarrow \mathrm{nat})\ \mathrm{gcd})$$

where $\quad \mathrm{eq}\ a\ x \;=\; \lambda y\ v.\ (x = y) \wedge a\ y\ v$

Bottom-up translation of the rec. call produces:

$$\begin{array}{l}
\mathrm{Eval}\ \mathit{env}\ \lfloor \texttt{m} \rfloor\ (\mathrm{nat}\ m) \wedge \\
\mathrm{Eval}\ \mathit{env}\ \lfloor \texttt{n} \rfloor\ (\mathrm{nat}\ n)\ \wedge\ n \neq 0 \implies \\
\mathrm{Eval}\ \mathit{env}\ \lfloor \texttt{m mod n} \rfloor\ (\mathrm{nat}\ (m\ \mathrm{mod}\ n))
\end{array}$$

Solution

an abbreviation

A restrictive assumption:

restricts the forall inside the arrow

$$\textsf{A}\ m\ n \ = \ \textsf{Eval}\ env\ \lfloor \texttt{gcd} \rfloor\ ((\textsf{eq nat}\ m \to \textsf{eq nat}\ n \to \textsf{nat})\ \textsf{gcd})$$

where $\quad \textsf{eq}\ a\ x \ = \ \lambda y\ v.\ (x = y) \wedge a\ y\ v$

At the top-level:

$$\textsf{DedclAssum}\ \lfloor \texttt{fun gcd m = fn n => ...} \rfloor\ env \Longrightarrow$$
$$\forall m\ n.\ (0 < n \Longrightarrow \textsf{A}\ n\ (m\ \textsf{mod}\ n)) \Longrightarrow \textsf{A}\ m\ n$$

This looks familiar...

Solution

The termination proof for gcd produces an ***induction theorem*** of the form:

$$\forall P.\;\; (\forall m\, n.\, (0 < n \implies P\, n\, (m \;\mathsf{mod}\; n)) \implies P\, m\, n) \implies (\forall m\, n.\, P\, m\, n)$$

The induction theorem is used to remove these assumptions and finish the translation.

At the top-level:

$$\mathsf{DedclAssum}\; \lfloor \texttt{fun gcd m = fn n => ...} \rfloor\; \mathit{env} \implies$$
$$\forall m\, n.\, (0 < n \implies \mathsf{A}\, n\, (m \;\mathsf{mod}\; n)) \implies \mathsf{A}\, m\, n$$

Result

The termination proof for gcd produces an ***induction theorem*** of the form:

$$\forall P.\ \ (\forall m\, n.\ (0 < n \implies P\, n\, (m \bmod n)) \implies P\, m\, n) \implies (\forall m\, n.\ P\, m\, n)$$

Final result:

$$\begin{array}{l} \mathsf{DeclAssum}\ \lfloor \texttt{fun gcd m = fn n => ...} \rfloor\ \mathit{env} \implies \\ \mathsf{Eval}\ \mathit{env}\ \lfloor \texttt{gcd} \rfloor\ ((\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat})\ \mathsf{gcd}) \end{array}$$

similar result to non-recursive case

Can all HOL functions be translated to ML?

Can all HOL functions be translated to ML?

No

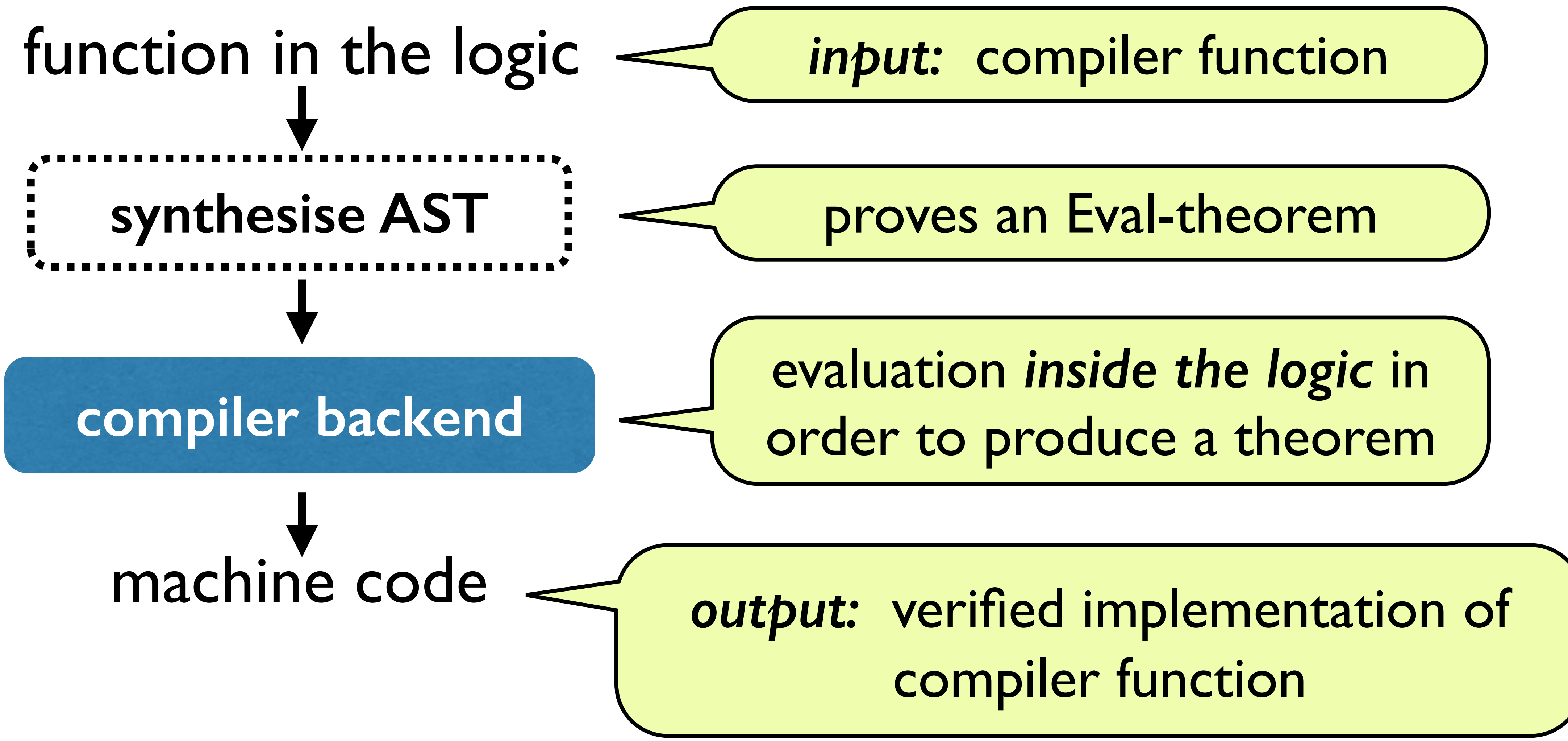
HOL has more powerful semantics for = than ML

HOL's equality can compare functions, ML's cannot

HOL allows underspecification (e.g. missing cases) and Hilbert's choice.

leads to side conditions in the translator theorems brief HOL4 demo (if time allows)

Idea



function in the logic
synthesise AST
compiler backend
machine code
input: compiler function
proves an Eval-theorem
evaluation *inside the logic* in order to produce a theorem
output: verified implementation of compiler function

Idea

function in the logic

↓

synthesise AST

↓

compiler backend

input: compiler function

proves an Eval-theorem

evaluation inside the logic in order to produce a theorem

TIME OUT (> 24 hours)

register allocator has bad complexity:
at least $\mathbf{O(n^3)}$ where **n** is number of variables

Translation validation

in the context evaluation by rewriting in the logic

Register allocator is too slow for in-logic evaluation

Solution:

1. evaluate compiler *to just before* register allocation

$\vdash$ **compile** *config* [source_prog] =
imperative_to_target (**reg_alloc** *config* [graph] [IL-prog])

don't expand definition

2. extract clash graph [graph]; *find colouring outside of logic*

3. *instantiate config* to include solution to colouring problem

4. make **reg_alloc** function *checks if valid colouring exists* inside *config*, if so use the colouring

Translation validation

in the context evaluation by rewriting in the logic

strange looking config, but theorem still fits compiler correctness theorem

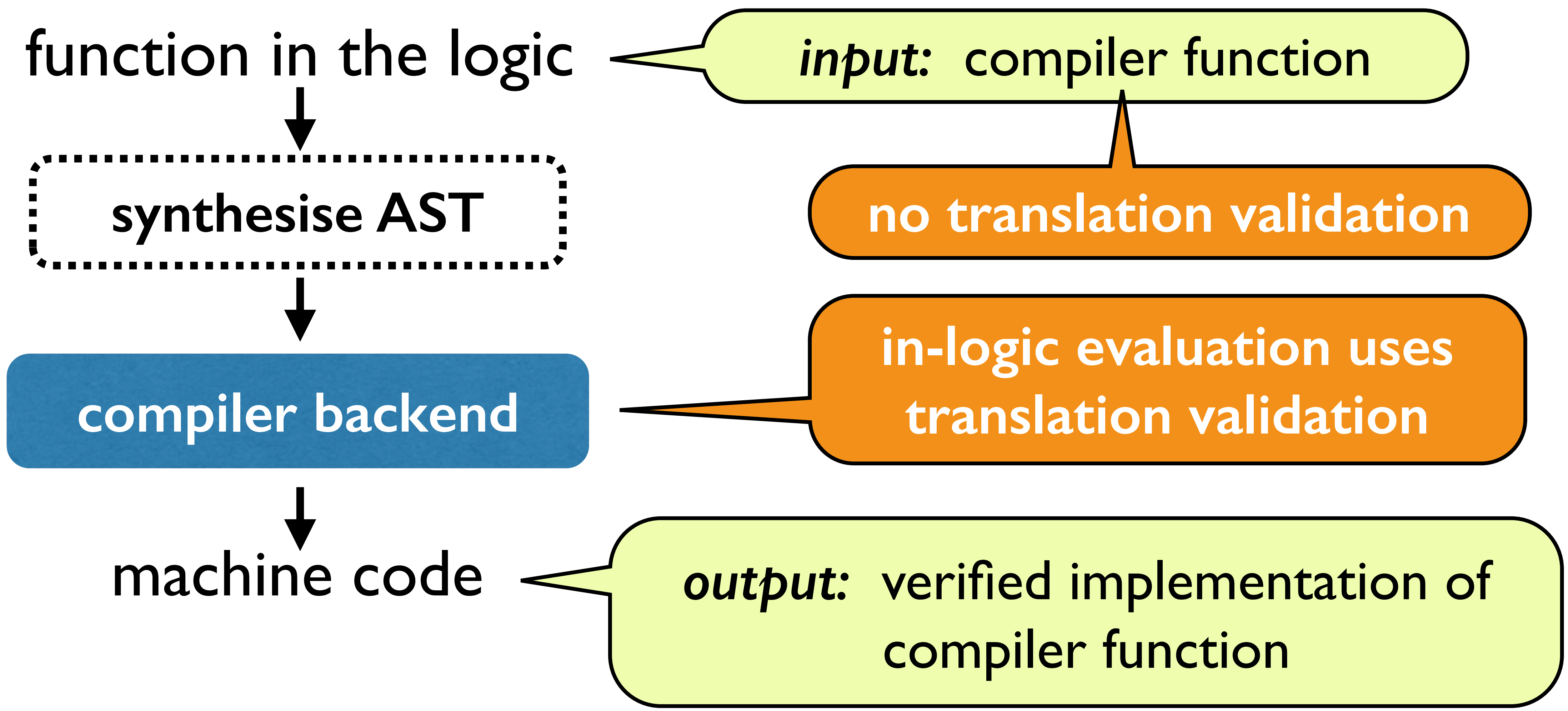
Resulting theorem:

$\vdash$ **compile** (*config with colourings …*) [source_prog] =

```
[0x48,0x39,0xF3,0x0F,0x83,0x0B,0x00,0x00,0x00,0xBF,0x07,0x00,
 0x00,0x00,0xE9,0xDD,0xFF,0xFF,0xFF,0x90,0x48,0x39,0xDA,0x0F,
 0x83,0x0B,0x00,0x00,0x00,0xBF,0x08,0x00,0x00,0x00,0xE9,0xC9,
 0xFF,0xFF,0xFF,0x90,0x48,0x89,0xD8,0x48,0x29,0xF0,0x49,0xB8,
 0xF8,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x00,0x49,0x39,0xC0,0x0F,
 0x83,0x0B,0x00,0x00,0x00,0xBF,0x03,0x00,0x00,0x00,0xE9,0xA5,
 0xFF,0xFF,0xFF,0x90,0x48,0x89,0xD8,0x41,0xB8,0xB8,0x1A,0x00,
 0x00,0x4C,0x01,0xC0,0x41,0xB8,0xF8,0x07,0x00,0x00,0x4C,0x39,
 0xC0,0x0F,0x83,0x0B,0x00,0x00,0x00,0xBF,0x04,0x00,0x00,0x00,
 0xE9,0x7F,0xFF,0xFF,0xFF,0x90,0x48,0x89,0xD0,0x48,0x29,0xD8,
```

What we learnt

Verified compilers can be bootstrapped.



Current research: adding an efficient *Eval* primitive to the CakeML language and its implementation

Extra slides about current research

Current research: adding an efficient ***Eval*** primitive to the CakeML language and its implementation

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

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

Version 2 does not.

ad hoc implementation and proof

For version 3, 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 reflection 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:

A list of declarations …

```
eval : environment; -> ast -> environment;
```

… 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;

environment n;

val _ = eval n (parse “val _ = (res := 1+2)”);

print_int (!res);
```

Declares an environment (incl `res`)

… which is used by **eval**

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.

Semantics of Eval

```
evaluate env state (Eval n x) =
  case evaluate_list env state [n,x] of
  | (Rval [Environment tenv env', decs],s) =>
      (case has_type tenv decs of
        | None => (Rerr (Raise NoType),s)
        | Some tenv' =>
            (case evaluate_decs env' s' decs of
             | (Rval env'',s'') => (Rval (Environment tenv' env''),s''))
             | res => res))
  | (Rval _,s') => (Rerr Error,s')
  | res => res
```

evaluate arguments

type check

evaluate given declarations

if successful, return new environment

(functional big-step clock tick omitted above)

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.

The compiler

To keep formulas free of clutter, let's assume:

$$\text{compile} \;:\; \text{ast} \to \text{byte list}$$

$$\text{compile} \;=\; \text{pass}_{n-1} \circ \text{pass}_{n-2} \circ \ldots \circ \text{pass}_1 \circ \text{pass}_0$$

This is not entirely true: the compiler has config and state.

First compiler pass

Typewriter font is compiler-generated AST

```
pass0 (Eval n x) =
  let
    val (n,x) = (pass0 n, pass0 x)
  in
    case infer_types n x of
      None => raise NoType
    | Some n’ => (InstallAndRun (compile x); n’)
  end
```

calls normal source code in prelude

New primitive in every IL

… that transports machine code downwards.

Semantics of InstallAndRun

For IL k:

evaluate *env state* (InstallAndRun *x*) =
  case evaluate *env state x* **of**
  | (Rval *v*,*s*) =>
    let (*env'*,*exp*,*s'*) = next_guess *s* **in**
      if $v \neq (\text{pass}_{n-1} \circ \ldots \circ \text{pass}_k)\ exp$
      then Rerr Error
      else evaluate *env' s' exp*
  | *res* => *res*

evaluate argument

uses oracle to guess the expression in IL k

if the guess is wrong then exec gets stuck

otherwise, evaluate the expression

The state contains an oracle: an infinite sequence expressions.
The next_guess function pops an element from the sequence.

Sketch of theorem for pass$_0$

source semantics

evaluate *env s ast* = (*res*,*s'*) $\wedge$ *res* $\neq$ Rerr Error $\wedge$
state_rel *s t* $\wedge$ env_rel *env env'* $\Rightarrow$

semantics of first IL

$\exists$*guesses*.
evaluate *env'* (set_guesses *t guesses*) (pass$_0$ *ast*) = (*res'*,*t'*) $\wedge$
res_rel *res res'* $\wedge$ state_rel *s' t'*

there is some sequence of guesses that works

Complicated:
This proof needs to use our proof of soundness and completeness for type inferencer.

Subsequent passes

(Mostly) just propagate InstallAndRun:

pass_k (InstallAndRun x) = InstallAndRun (pass_k x)

Late stage: write input bytes to memory and then runs InstallAndRun.

Simplifies InstallAndRun

At the bottom, InstallAndRun becomes clear-icache-and-jump.

InstallAndRun is almost a no-op

Theorem for other passes

semantics of IL k

evaluate *env s exp* = (*res*,*s'*) ∧ *res* ≠ Rerr Error ∧
state_rel *s t* ∧ env_rel *env env'* ⇒

semantics of IL k+1

evaluate *env' t* (pass$_k$ *exp*) = (*res'*,*t'*) ∧
res_rel *res res'* ∧ state_rel *s' t'*

Here state_rel relates the guesses:

state_rel *s t* =
… ∧ ∀*n*. *s*.guesses *n* = pass$_k$ (*s*.guesses *n*)

Good news: ought to be an easy modification.
Bad news: every compiler pass needs to be updated.

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.