Verification of an ML compiler

Lecture 4:
Compiler bootstrapping and new directions

Marktoberdorf Summer School MOD 2017

Magnus O. Myreen, Chalmers University of Technology
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

- concrete syntax
  - SML parser
    - type inferencer
      - synthesise AST
        - function in the logic
          - verified function in logic
          - proof-producing tool

- machine code

- compiler backend

Proof producing code generation

If we input \texttt{factorial} ... 

function in the logic

\[ \text{synthesise AST} \]

\[ \text{AST} \]

\[ \text{compiler backend} \]

\[ \text{machine code} \]

... then the toolchain produces machine code and proves a theorem stating that the code behaves like the \texttt{factorial} function.

\[ \text{= verified function in logic} \]

\[ \text{= proof-producing tool} \]
Required components

concrete syntax

SML parser

function in the logic

type inferencer

synthesise AST

AST

compiler backend

machine code

= verified function in logic

= proof-producing tool
**input:** verified compiler function i.e. composition of parser, type inferencer and compiler backend

**function in the logic**

**synthesise AST**

**output:** verified implementation of compiler function

= verified function in logic

= proof-producing tool
The Missing Piece

function in the logic

synthesise AST

AST

compiler backend

machine code

= verified function in logic

= proof-producing tool
“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 \texttt{Eval}.

\[
\text{Eval } env \ exp \ post = \exists \textit{val}. \ 	ext{evaluate } env \ exp \ \varepsilon = (Rval \ val, \varepsilon) \land post \ val
\]

We can express relationships between HOL and CakeML:

\[
\text{Eval } env \ [5] \ (\text{int } 5)
\]

if we define \texttt{int} as follows:

\[
\texttt{int } i = \lambda v. (v = \texttt{Lit } i) \quad \text{where } i \text{ is an integer}
\]

relates HOL \(i\) with CakeML value \(v\), if \(v\) is an integer
Bottom-up construction

Each stage derives a theorem of the form:

\[ \text{assumptions} \implies \text{Eval } \text{env } \text{exp} \ (\text{inv } t) \]

Examples:

- \( \text{Eval } \text{env} \ [n] \ (\text{int } n) \implies \text{Eval } \text{env} \ [n] \ (\text{int } n) \)
- \( \text{true} \implies \text{Eval } \text{env} \ [5] \ (\text{int } 5) \)
Bottom-up construction

Examples:

\[
\text{Eval } env \ [n] (\text{int } n) \implies \text{Eval } env \ [n] (\text{int } n)
\]
\[
\text{true} \implies \text{Eval } env \ [5] (\text{int } 5)
\]

Lemmas are used to translate compound terms:

\[
\forall e_1 \ e_2 \ i \ j. \\
\text{Eval } env \ [e_1] (\text{int } i) \land \\
\text{Eval } env \ [e_2] (\text{int } j) \implies \\
\text{Eval } env \ [e_1 + e_2] (\text{int } (i + j))
\]

Example:

\[
\text{Eval } env \ [n] (\text{int } n) \implies \text{Eval } env \ [n+5] (\text{int } (n + 5))
\]
A function $f$ from $\text{int}$ to $\text{int}$:

$$\text{Eval } \text{env } [f] ((\text{int} \rightarrow \text{int}) \ f)$$

The arrow is defined as follows:

$$(a \rightarrow b) \ f$$

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

... some $u$ is returned such that $b$ relates it with the result of $f \ x$
Function Application

Eval \textit{env} \left[f\right] ((a \rightarrow b) f) \land

Eval \textit{env} \left[x\right] (a x) \Longrightarrow

Eval \textit{env} \left[f \ x\right] (b (f x))
Example

We can now derive:

\[
\text{Eval } env \ [f] \ ((\text{int } \rightarrow \text{int}) \ f) \implies \text{Eval } env \ [f \ 5] \ (\text{int} \ (f \ 5))
\]

assumption about variable \( f \)  

conclusion about \( f \ 5 \)
Lambda-abstraction

$$(\forall x \forall v. \; a \; x \; v \implies \text{Eval} \; (env[n \mapsto v]) \; [\text{body}] \; (b \; (f \; x))) \implies$$

$\text{Eval} \; env \; [fn \; n \mapsto \text{body}] \; ((a \mapsto b) \; f)$$

Example:

$\text{Eval} \; env \; [fn \; n \mapsto n+5] \; ((\text{int} \mapsto \text{int}) \; (\lambda n. \; n+5))$

derived from theorem with assumption about $n$
Type variables

\[ \text{Eval } env \left[ \text{fn } f \Rightarrow \text{fn } x \Rightarrow f (f x) \right] \]
\[ (((a \rightarrow a) \rightarrow a \rightarrow a) (\lambda f x. f (f x))) \]

variable with type \( \alpha \rightarrow \nu \rightarrow \text{bool} \)

has type \( \alpha \)

\textbf{Crucially:} \( \alpha \) can be instantiated once a more concrete type is used

e.g. with the \texttt{int} relation which has type variable with type \texttt{int} \( \rightarrow \nu \rightarrow \text{bool} \)
**User-defined constant**

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

One derives:

\[
\text{Eval } env \ [fn \ f \Rightarrow \ fn \ x \Rightarrow f \ (f \ x)] \\
(((a \rightarrow a) \rightarrow a \rightarrow a) \ (\lambda f \ x. \ f \ (f \ x)))
\]

... and then the following:

\[
\text{DeclAssum } [\text{val } \text{twice} = fn \ f \Rightarrow fn \ x \Rightarrow f \ (f \ x);] \ env \ \rightarrow \\
\text{Eval } env \ [\text{twice}] \ (((a \rightarrow a) \rightarrow a \rightarrow a) \ \text{twice})
\]

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

 DeclAssum `val twice = fn f => fn x => f (f x);` `env`
 Eval `env` twice `(((a → a) → a → a) twice)`

CakeML name `twice` is related to HOL name `twice`

assumes `twice` is bound in ML env
Recursive functions?

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

A restrictive assumption:

\[ A \ m \ n = \text{Eval env } [gcd] ((\text{eq nat } m \rightarrow \text{eq nat } n \rightarrow \text{nat}) \ gcd) \]

where \[ \text{eq } a \ x = \lambda y. (x = y) \land a \ y \ v \]

Bottom-up translation of the rec. call produces:

\[
\begin{align*}
\text{Eval env } [m] (\text{nat } m) \land \\
\text{Eval env } [n] (\text{nat } n) \land n \neq 0 \implies \\
\text{Eval env } [m \mod n] (\text{nat } (m \mod n))
\end{align*}
\]
Solution

A restrictive assumption:

\[ A \, m \, n = \text{Eval}\, \text{env}\, [\text{gcd}]\, ((\text{eq\, nat\,} m \rightarrow \text{eq\, nat}\, n \rightarrow \text{nat})\, \text{gcd}) \]

where \[ \text{eq\, a\, x} = \lambda y\, v.\, (x = y) \land a\, y\, v \]

At the top-level:

\[ \text{DedclAssum}\, [\text{fun}\, \text{gcd}\, m = \text{fn}\, n\, \Rightarrow\, \ldots]\, \text{env}\, \Rightarrow\, \forall m\, n.\, (0 < n \Rightarrow A\, n\, (m\, \text{mod}\, n)) \Rightarrow A\, m\, n \]

This looks familiar…
Solution

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

\[ \forall P. \ (\forall m\ n. \ (0 < n \Rightarrow P\ n\ (m \mod n)) \Rightarrow P\ m\ n) \Rightarrow (\forall m\ n. \ P\ m\ n) \]

At the top-level:

\[ \text{DedclAssum} \ [\text{fun} \ \text{gcd} \ m = \text{fn} \ n \Rightarrow \ldots] \ \text{env} \ \Rightarrow \ \forall m\ n. \ (0 < n \Rightarrow A\ n\ (m \mod n)) \Rightarrow A\ m\ n \]
Result

The termination proof for gcd produces an \textit{induction theorem} of the form:

\[
\forall P. \ (\forall m \ n. \ (0 < n \implies P \ n \ (m \mod n)) \implies P \ m \ n) \implies (\forall m \ n. \ P \ m \ n)
\]

\textbf{Final result:}

\[
\text{DedclAssum} \ [\text{fun} \ gcd \ m = \text{fn} \ n \Rightarrow \ldots] \ env \ \rightarrow \\
\text{Eval} \ env \ [gcd] \ ((\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \ gcd)
\]

\begin{itemize}
\item similar result to non-recursive case
\end{itemize}
Can all HOL functions be translated to ML?
Can all HOL functions be translated to ML?

No

HOL has more powerful semantics for equality than ML.

- HOL’s equality can compare functions, ML’s cannot.
- HOL allows underspecification (e.g. missing cases) and Hilbert’s choice.

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

- **input:** compiler function
- **synthesise AST**
  - proves an Eval-theorem
- **compiler backend**
  - evaluation *inside the logic* in order to produce a theorem
- **machine code**
  - **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 $O(n^3)$ 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

\[ \text{compile config [source_prog]} = \text{imperative_to_target (reg_alloc config [graph] [IL-prog])} \]

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

Resulting theorem:

\[ \vdash \text{compile} \ (\text{config with colourings } \ldots) \ [\text{source\_prog}] = \]

\[ [0x48, 0x39, 0xF3, 0x0F, 0x83, 0x0B, 0x00, 0x00, 0x00, 0xBF, 0x07, 0x00, 0x00, 0x00, 0xE9, 0xDD, 0xFF, 0xFF, 0xFF, 0x90, 0x48, 0x39, 0x90, 0x0x08, 0x0x00, 0x0x00, 0x0x00, 0x0x00, 0x0xF3, 0x0x0F, 0x83, 0x0B, 0x00, 0x00, 0x00, 0xBF, 0x08, 0x00, 0x00, 0x00, 0xE9, 0xC9, 0x8F, 0xFF, 0xFF, 0x0x90, 0x48, 0x39, 0x0xF3, 0x0x0F, 0x83, 0x0B, 0x00, 0x00, 0x00, 0xBF, 0x08, 0x00, 0x00, 0x00, 0xE9, 0xC9, 0x8F, 0x8F, 0xFF, 0xFF, 0xF0, 0x48, 0x89, 0xD8, 0x48, 0x29, 0xF0, 0x49, 0x8B, 0x49, 0x8F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF0, 0x48, 0x39, 0x90, 0x0x08, 0x0x00, 0x0x00, 0x0x00, 0x0x00, 0x0x0F, 0x0x03, 0x0x00, 0x0x00, 0x0x00, 0x0x00, 0x0xE9, 0xA5, 0x8F, 0x8F, 0xFF, 0xFF, 0xFF, 0x0x90, 0x48, 0x39, 0x90, 0x0x08, 0x0x00, 0x0x00, 0x0x00, 0x0x00, 0x0x0F, 0x0x03, 0x0x00, 0x0x00, 0x0x00, 0x0x00, 0x0xE9, 0xA5, 0x8F, 0x8F, 0xFF, 0xFF, 0xFF, 0x0x90, 0x48, 0x39, 0x90, 0x0x08, 0x0x00, 0x0x00, 0x0x00, 0x0x00, 0x0x0F, 0x0x03, 0x0x00, 0x0x00, 0x0x00, 0x0x00, 0x0xE9, 0xA5, 0x8F, 0x8F, 0xFF, 0xFF, 0xFF, 0x0x90, 0x48, 0x39, 0x90, 0x0x08, 0x0x00, 0x0x00, 0x0x00, 0x0x00, 0x0x0F, 0x0x03, 0x0x00, 0x0x00, 0x0x00, 0x0x00, 0x0xE9, 0xA5] \]
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:

```ml
fun loop n = 
  case read () of
    NONE => () 
  | SOME input => loop (eval n (parse_wrap_print_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:

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

loop basis_environment;;
```

Type of eval primitive:

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

A list of declarations ...

... is evaluated in a given environment.

Returns the input environment extended with the new decls.
We propose that references are used:

```plaintext
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

```plaintext
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.
evaluate \( \text{env state} \) (Eval \( n \ x \)) =

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

*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:

\[
\text{compile} : \text{ast} \rightarrow \text{byte list} \\
\text{compile} = \text{pass}_{n-1} \circ \text{pass}_{n-2} \circ \ldots \circ \text{pass}_1 \circ \text{pass}_0
\]

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

\[
\text{pass}_0 (\text{Eval } n \ x) = \\
\text{let} \\
\quad \text{val } (n,x) = (\text{pass}_0 n, \text{pass}_0 x) \\
\text{in} \\
\quad \text{case infer_types } n \ x \text{ of} \\
\quad \quad \text{None } \Rightarrow \text{raise NoType} \\
\quad \quad \mid \text{Some } n' \Rightarrow (\text{InstallAndRun } (\text{compile } x); n') \\
\text{end}
\]

Typewriter font is compiler-generated AST

calls normal source code in prelude

New primitive in every IL

... that transports machine code downwards.
Semantics of InstallAndRun

For IL k:

evaluate \( \text{env state} \) \((\text{InstallAndRun } x)\) =

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

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

\[ \text{source semantics} \]

\[ \text{evaluate } env\ s\ ast = (res,s') \land res \neq \text{Rerr Error} \land \]
\[ \text{state}_{-}\text{rel } s\ t \land \text{env}_{-}\text{rel } env\ env' \Rightarrow \]
\[ \exists guesses. \]
\[ \text{evaluate } env' (\text{set_guesses } t \ guesses) (\text{pass}_0 \ ast) = (res',t') \land \]
\[ \text{res}_{-}\text{rel } res\ res' \land \text{state}_{-}\text{rel } s't' \]

\[ \text{semantics of first IL} \]

\[ \text{there is some sequence of guesses that works} \]

\[ \text{Complicated:} \]
\[ \text{This proof needs to use our proof of soundness and completeness for type inferencer.} \]
Subsequent passes

(Mostly) just propagate InstallAndRun:

$$\text{pass}_k (\text{InstallAndRun } x) = \text{InstallAndRun} (\text{pass}_k x)$$

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

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

- Simplifies InstallAndRun
- InstallAndRun is almost a no-op
Theorem for other passes

evaluate \( env \ s \ exp = (res, s') \land res \neq \text{Rerr Error} \land \)
state\_rel \( s \ t \land \text{env\_rel env env'} \Rightarrow \)

evaluate \( env' \ t \ (\text{pass}_k \ exp) = (res', t') \land \)
res\_rel \( res \ res' \land \text{state\_rel s'} \ t' \)

Here state\_rel relates the guesses:

\[
\text{state\_rel s t} = \\
\ldots \land \forall n. \ s.\text{guesses} n = \text{pass}_k (s.\text{guesses} n)
\]

**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:

```ml
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:

```ml
fun loop n =
  case read () of
    NONE => ()
  | SOME input =>
    loop (eval n (parse_wrap_print_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.