# CakeML: bootstrapping a verified compiler

**Ramana Kumar**

COMP4161, 6 October 2016

www.data61.csiro.au

## Question

What is this function, `foo`, more often called?

$$\texttt{foo}\ f\ [\,] = [\,]$$
$$\texttt{foo}\ f\ (h \# t) = f\ h \# \texttt{foo}\ f\ t$$

## Question

What is this function, `foo`, more often called?

$$\texttt{foo}\ f\ [\,] = [\,]$$
$$\texttt{foo}\ f\ (h \# t) = f\ h \# \texttt{foo}\ f\ t$$

## Answer

$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (h \# t) = f\ h \# \texttt{map}\ f\ t$$

## Question

What about this one?

$$\texttt{bar}\ [\,] = 0$$
$$\texttt{bar}\ (h \# t) = \texttt{Suc}\ (\texttt{bar}\ t)$$

## Question

What about this one?

$$\texttt{bar}\ [\,] = 0$$
$$\texttt{bar}\ (h \# t) = \texttt{Suc}\ (\texttt{bar}\ t)$$

## Answer

$$\texttt{length}\ [\,] = 0$$
$$\texttt{length}\ (h \# t) = \texttt{Suc}\ (\texttt{length}\ t)$$

## Question

What about this one?

$\texttt{bar}\ [\,] = 0$

$\texttt{bar}\ (h \# t) = \texttt{Suc}\ (\texttt{bar}\ t)$

## Answer

$\texttt{length}\ [\,] = 0$

$\texttt{length}\ (h \# t) = \texttt{Suc}\ (\texttt{length}\ t)$

## Note

$7 = \texttt{Suc}\ (\texttt{Suc}\ (\texttt{Suc}\ (\texttt{Suc}\ (\texttt{Suc}\ (\texttt{Suc}\ (\texttt{Suc}\ 0))))))$

# Spot the differences

## Example 1

$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (h \# t) = f\ h \# \texttt{map}\ f\ t$$

## Example 2

$$\vdash\ (\forall f.\ \texttt{map}\ f\ [\,] = [\,]) \wedge$$
$$\quad \forall f\ h\ t.\ \texttt{map}\ f\ (h \# t) = f\ h \# \texttt{map}\ f\ t$$

# Spot the differences

### Example 1

$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (h \# t) = f\ h \# \texttt{map}\ f\ t$$

### Example 2

$$\vdash\ (\forall f.\ \texttt{map}\ f\ [\,] = [\,]) \land$$
$$\quad \forall f\ h\ t.\ \texttt{map}\ f\ (h \# t) = f\ h \# \texttt{map}\ f\ t$$

### Answer

Example 1 is a pair of equations.

# Spot the differences

## Example 1

$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (h \# t) = f\ h \# \texttt{map}\ f\ t$$

## Example 2

$$\vdash\ (\forall f.\ \texttt{map}\ f\ [\,] = [\,]) \land$$
$$\forall f\ h\ t.\ \texttt{map}\ f\ (h \# t) = f\ h \# \texttt{map}\ f\ t$$

## Answer

Example 1 is a pair of equations.
Example 2 is a theorem: it has a turnstile, a conjunction, and explicit universal quantification.

# Spot the differences

### Example 1

$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (h \# t) = f\ h \# \texttt{map}\ f\ t$$

### Example 2

$$\vdash\ (\forall f.\ \texttt{map}\ f\ [\,] = [\,]) \wedge$$
$$\forall f\ h\ t.\ \texttt{map}\ f\ (h \# t) = f\ h \# \texttt{map}\ f\ t$$

### Answer

Example 1 is a pair of equations.
Example 2 is a theorem: it has a turnstile, a conjunction, and explicit universal quantification.
(But they mean the same thing.)

# What you learned last month

### Question

Can you prove this?

$\forall\, l\ f.\ \texttt{length}\ (\texttt{map}\ f\ \ l) = \texttt{length}\ l$

# What you learned last month

**Question**

Can you prove this?

$\forall\, l\ f.\ \texttt{length}\ (\texttt{map}\ f\ l) = \texttt{length}\ l$

**Answer**

Yes! By induction on the list *l*, simplifying with the definitions of `map` and `length`.

# What you learned last month

### Question

Can you prove this?

$\forall\, l\, f.\ \texttt{length}\ (\texttt{map}\ f\ l) = \texttt{length}\ l$

### Answer

Yes! By induction on the list *l*, simplifying with the definitions of `map` and `length`.

But we are interested in even simpler theorems...

# Simple theorems

Question

Can you prove this?

`map length` [[ ]; [[ ]; [ ]]; [[ ]]] = [0; 2; 1]

# Simple theorems

## Question

Can you prove this?

`map length` [[ ]; [[ ]; [ ]]; [[ ]]] = [0; 2; 1]

Or this?

`length` (`map Suc` [1; 2; 0]) = 3

# Simple theorems

## Question

Can you prove this?

`map length` [[ ]; [[ ]; [ ]]; [[ ]]] = [0; 2; 1]

Or this?

`length` (`map Suc` [1; 2; 0]) = 3

## Answer

Simplification...

# Simple theorems

### Question

Can you prove this?

`map length` [[ ]; [[ ]; [ ]]; [[ ]]] = [0; 2; 1]

Or this?

`length` (`map Suc` [1; 2; 0]) = 3

### Answer

Simplification...

In fact, you only need the left-hand side of the equation in order to produce the theorem.

# Evaluation problems

Definition

An *evaluation problem* is a term that does not contain any variables (only known constants and concrete data).

# Evaluation problems

Definition

An *evaluation problem* is a term that does not contain any variables (only known constants and concrete data).
A solution is a theorem $\vdash tm = tm'$, where $tm'$ cannot be simplified further.

# Example

Consider the constant `while`, which satisfies the following equation.

$\vdash$ `while` $P\ g\ x = \text{if}\ P\ x\ \text{then}$ `while` $P\ g\ (g\ x)\ \text{else}\ x$

# Example

Consider the constant `while`, which satisfies the following equation.

$\vdash$ `while` $P\ g\ x = \text{if}\ P\ x\ \text{then}$ `while` $P\ g\ (g\ x)\ \text{else}\ x$

## An evaluation problem

What is the solution for this input term?

`while` $(\lambda x.\ x = 0)\ (\lambda x.\ x)\ 1$

# Example

Consider the constant `while`, which satisfies the following equation.

$$\vdash \texttt{while}\ P\ g\ x = \text{if}\ P\ x\ \text{then}\ \texttt{while}\ P\ g\ (g\ x)\ \text{else}\ x$$

## An evaluation problem

What is the solution for this input term?

$\texttt{while}\ (\lambda x.\ x = 0)\ (\lambda x.\ x)\ 1$

## Answer

$$\vdash \texttt{while}\ (\lambda x.\ x = 0)\ (\lambda x.\ x)\ 1 = 1$$

# Example

## Another evaluation problem

What about this input term?

`while` $(\lambda x.\ x = 0)\ (\lambda x.\ x)\ 0$

# Example

### Another evaluation problem

What about this input term?

`while` $(\lambda x.\ x = 0)\ (\lambda x.\ x)\ 0$

### Answer

...

# Example

## Another evaluation problem

What about this input term?

$\texttt{while}\ (\lambda x.\ x = 0)\ (\lambda x.\ x)\ 0$

## Answer

...

Simplification loops. There is no solution.

# Example

## Another evaluation problem

What about this input term?

`while` $(\lambda x.\ x = 0)\ (\lambda x.\ x)\ 0$

## Answer

...

Simplification loops. There is no solution.

## Note

But I thought HOL was a logic of total functions?

# Example

## Another evaluation problem

What about this input term?

`while` $(\lambda x.\ x = 0)\ (\lambda x.\ x)\ 0$

## Answer

...

Simplification loops. There is no solution.

## Note

But I thought HOL was a logic of total functions?

It is. `while` is total. We just cannot prove anything interesting about its value on the arguments above.

# Evaluation automation

How does simplification work?

# Evaluation automation

## How does simplification work?

Roughly, given a set of rewriting theorems,

1. Find a subterm that matches the left-hand side of one of the rewrite theorems.

# Evaluation automation

## How does simplification work?

Roughly, given a set of rewriting theorems,

1. Find a subterm that matches the left-hand side of one of the rewrite theorems.
2. Apply primitive proof steps to replace that subterm with the rewrite theorem's right-hand side.

# Evaluation automation

## How does simplification work?

Roughly, given a set of rewriting theorems,

1. Find a subterm that matches the left-hand side of one of the rewrite theorems.
2. Apply primitive proof steps to replace that subterm with the rewrite theorem's right-hand side.
3. Repeat until no subterms can be found.

# Evaluation automation

## How does simplification work?

Roughly, given a set of rewriting theorems,

1. Find a subterm that matches the left-hand side of one of the rewrite theorems.
2. Apply primitive proof steps to replace that subterm with the rewrite theorem's right-hand side.
3. Repeat until no subterms can be found.

Clearly this procedure can sometimes loop forever.

# Proof tools steer the kernel

Kernel as an API for theorems

# Proof tools steer the kernel

## Kernel as an API for theorems

- Theorem prover kernel provides primitive methods for constructing theorems.

# Proof tools steer the kernel

## Kernel as an API for theorems

- Theorem prover kernel provides primitive methods for constructing theorems.
- Tools (like the simplifier) call these methods.

# Proof tools steer the kernel

## Kernel as an API for theorems

- Theorem prover kernel provides primitive methods for constructing theorems.
- Tools (like the simplifier) call these methods.
- Therefore, tools do not need to be trusted: only kernel-sanctioned theorems can be produced.

# Proof tools steer the kernel

## Kernel as an API for theorems

- Theorem prover kernel provides primitive methods for constructing theorems.
- Tools (like the simplifier) call these methods.
- Therefore, tools do not need to be trusted: only kernel-sanctioned theorems can be produced.

Isabelle and HOL4 support this view (“LCF-style”).

# Evaluation within the logic

Call-by-value proof automation

# Evaluation within the logic

## Call-by-value proof automation

- High-performance simplification:
  - Choose a good *evaluation strategy*.
  - Use techniques from functional programming.

# Evaluation within the logic

## Call-by-value proof automation

- High-performance simplification:
  - Choose a good *evaluation strategy*.
  - Use techniques from functional programming.
- HOL4 includes such automation (called `EVAL`).
  It can be extended with user-defined automation.

# Evaluation within the logic

## Call-by-value proof automation

- High-performance simplification:
  - Choose a good *evaluation strategy*.
  - Use techniques from functional programming.
- HOL4 includes such automation (called `EVAL`).
  It can be extended with user-defined automation.
- Performance is *fundamentally limited*.
  - At best, simplification is akin to interpreting a program.
  - And, every step ultimately goes through the kernel.

# Evaluation outside the logic

## Trusted code generation

- Isabelle also offers another method:

# Evaluation outside the logic

## Trusted code generation

- Isabelle also offers another method:
  - Print the input term in a functional programming language.
  - Compile and run the program.
  - Read back the result.

# Evaluation outside the logic

## Trusted code generation

- Isabelle also offers another method:
  - Print the input term in a functional programming language.
  - Compile and run the program.
  - Read back the result.
- Faster than `EVAL`, because the program is compiled and optimised before it is run.

# Evaluation outside the logic

## Trusted code generation

- Isabelle also offers another method:
  - Print the input term in a functional programming language.
  - Compile and run the program.
  - Read back the result.
- Faster than `EVAL`, because the program is compiled and optimised before it is run.
- But, this does not produce a proof.
  - The result theorem needs to be asserted as an axiom.
  - Much care is required to ensure this axiom is plausible.

# Evaluation outside the logic

## Trusted code generation

- Isabelle also offers another method:
  - Print the input term in a functional programming language.
  - Compile and run the program.
  - Read back the result.
- Faster than `EVAL`, because the program is compiled and optimised before it is run.
- But, this does not produce a proof.
  - The result theorem needs to be asserted as an axiom.
  - Much care is required to ensure this axiom is plausible.

We will return to this later.

# Counting steps

Question

Can you count the number of reductions (applications of a single rewrite rule) taken in solving an evaluation problem?

# Counting steps

## Question

Can you count the number of reductions (applications of a single rewrite rule) taken in solving an evaluation problem?

## Answer

Yes: augment the simplifier so it counts how many rewrites it applies, and returns the count alongside the theorem.

# Counting steps

### Question

Can you count the number of reductions (applications of a single rewrite rule) taken in solving an evaluation problem?

### Answer

Yes: augment the simplifier so it counts how many rewrites it applies, and returns the count alongside the theorem.

### Example

Simplify and count: $\texttt{while}\ (\lambda\, x.\ x < 2)\ \texttt{Suc}\ 0$.

# Counting steps

### Question

Can you count the number of reductions (applications of a single rewrite rule) taken in solving an evaluation problem?

### Answer

Yes: augment the simplifier so it counts how many rewrites it applies, and returns the count alongside the theorem.

### Example

Simplify and count: `while` $(\lambda\, x.\ x < 2)$ `Suc` 0.
returns: $(\vdash$ `while` $(\lambda\, x.\ x < 2)$ `Suc` 0 = 2, 2 rewrites)

# Counting steps

### Question

Can you count the number of reductions (applications of a single rewrite rule) taken in solving an evaluation problem?

### Answer

Yes: augment the simplifier so it counts how many rewrites it applies, and returns the count alongside the theorem.

### Example

Simplify and count: $\texttt{while}\ (\lambda x.\ x < 2)\ \texttt{Suc}\ 0$.
returns: $(\vdash \texttt{while}\ (\lambda x.\ x < 2)\ \texttt{Suc}\ 0 = 2$, 2 rewrites)
(Actually: 216 primitive inference steps.)

# Counting steps inside the logic

Question

How about *reasoning about* the number of steps?

# Counting steps inside the logic

### Question

How about *reasoning about* the number of steps?

### Problem

The simplifier is outside the logic, just using the kernel API.

# Counting steps inside the logic

Question

How about *reasoning about* the number of steps?

Problem

The simplifier is outside the logic, just using the kernel API.
Inside the logic, the number of steps is completely invisible.

# Counting steps inside the logic

### Question

How about *reasoning about* the number of steps?

### Problem

The simplifier is outside the logic, just using the kernel API.
Inside the logic, the number of steps is completely invisible.

### Totally different approach

Formalise simplification within the logic.

# Counting steps inside the logic

### Question

How about *reasoning about* the number of steps?

### Problem

The simplifier is outside the logic, just using the kernel API.
Inside the logic, the number of steps is completely invisible.

### Totally different approach

Formalise simplification within the logic.
Use a deep embedding.

# Deep embeddings

### Question

What might this datatype be used for?

```
lit =
    IntLit int
  | Char char
  | StrLit string
  | Word8 byte
  | Word64 word64
```

# Deep embeddings

## Question

What might this datatype be used for?

```
lit =
    IntLit int
  | Char char
  | StrLit string
  | Word8 byte
  | Word64 word64
```

## Answer

```
exp =
    Lit lit
  | Var (string id)
  | Con (string id option) (exp list)
  | Fun string exp
  | App op (exp list)
```

# Functional semantics

## Some meanings

$\texttt{evaluate}\ st\ env\ [\texttt{Lit}\ l] = (st, \texttt{Rval}\ [\texttt{Litv}\ l])$

# Functional semantics

## Some meanings

evaluate *st* *env* [Lit *l*] = (*st*, Rval [Litv *l*])
evaluate *st* *env* [Fun *x* *e*] = (*st*, Rval [Closure *env* *x* *e*])

# Functional semantics

## Some meanings

```
evaluate st env [Lit l] = (st,Rval [Litv l])
evaluate st env [Fun x e] = (st,Rval [Closure env x e])
evaluate st env [Var n] =
  case lookup_var_id n env of
```

# Functional semantics

## Some meanings

```
evaluate st env [Lit l] = (st,Rval [Litv l])
evaluate st env [Fun x e] = (st,Rval [Closure env x e])
evaluate st env [Var n] =
 case lookup_var_id n env of
  None  ⇒  (st,Rerr (Rabort Rtype_error))
 | Some v  ⇒  (st,Rval [v])
```

# Functional semantics

## Some meanings

```
evaluate st env [Lit l] = (st,Rval [Litv l])
evaluate st env [Fun x e] = (st,Rval [Closure env x e])
evaluate st env [Var n] =
 case lookup_var_id n env of
   None  ⇒  (st,Rerr (Rabort Rtype_error))
 | Some v  ⇒  (st,Rval [v])
```

## Pulling apart closures

```
do_call [Closure env n e; v2] =
 Some (env with v := (n,v2) # env.v,e)
```

# Functional semantics

## Some meanings

evaluate $st$ $env$ [Lit $l$] = ($st$,Rval [Litv $l$])
evaluate $st$ $env$ [Fun $x$ $e$] = ($st$,Rval [Closure $env$ $x$ $e$])
evaluate $st$ $env$ [Var $n$] =
  case lookup_var_id $n$ $env$ of
    None ⇒ ($st$,Rerr (Rabort Rtype_error))
  | Some $v$ ⇒ ($st$,Rval [$v$])

## Pulling apart closures

do_call [Closure $env$ $n$ $e$; $v_2$] =
  Some ($env$ *with* v := ($n$,$v_2$) # $env$.v,$e$)
do_call [Litv $l$; $v_2$] = None
...

# Functional semantics has a clock

## Function applications tick

`evaluate` *st env* [`Call` $e_1$ $e_2$] =
case `evaluate` *st env* [$e_1$; $e_2$] of

# Functional semantics has a clock

## Function applications tick

```
evaluate st env [Call e1 e2] =
  case evaluate st env [e1; e2] of
    (st',Rval vs) ⇒
      (case do_call (reverse vs) of
```

# Functional semantics has a clock

## Function applications tick

```
evaluate st env [Call e1 e2] =
 case evaluate st env [e1; e2] of
  (st',Rval vs) ⇒
   (case do_call (reverse vs) of
     None ⇒ (st',Rerr (Rabort Rtype_error))
    | Some (env',e) ⇒
      if st'.clock = 0 then
        (st',Rerr (Rabort Rtimeout_error))
      else
        evaluate (st' with clock := st'.clock − 1) env' [e])
 | (st',Rerr _) ⇒ (st',Rerr _)
```

# Functional semantics has a clock

## Function applications tick

```
evaluate st env [Call e1 e2] =
 case evaluate st env [e1; e2] of
  (st',Rval vs) ⇒
   (case do_call (reverse vs) of
     None ⇒ (st',Rerr (Rabort Rtype_error))
    | Some (env',e) ⇒
      if st'.clock = 0 then
       (st',Rerr (Rabort Rtimeout_error))
      else
       evaluate (st' with clock := st'.clock − 1) env' [e])
 | (st',Rerr _) ⇒ (st',Rerr _)
```

The clock lets us prove termination for `evaluate`.

# CakeML

## Language features

- functions: higher-order, polymorphic, mutually recursive

# CakeML

## Language features

- functions: higher-order, polymorphic, mutually recursive

```
fn x => if x then "hi" else "bye";
let
  fun f 0 = true | f n = g (n-1)
  fun g n = n = 1 orelse f (n-1)
in f end
```

# CakeML

## Language features

- functions: higher-order, polymorphic, mutually recursive

```
fn x => if x then "hi" else "bye";
let
  fun f 0 = true | f n = g (n-1)
  fun g n = n = 1 orelse f (n-1)
in f end
```

- datatypes: recursive, pattern-matching

# CakeML

## Language features

- functions: higher-order, polymorphic, mutually recursive

```
fn x => if x then "hi" else "bye";
let
  fun f 0 = true | f n = g (n-1)
  fun g n = n = 1 orelse f (n-1)
in f end
```

- datatypes: recursive, pattern-matching
- state (references), exceptions, modules

# CakeML

## Language features

- functions: higher-order, polymorphic, mutually recursive

```
fn x => if x then "hi" else "bye";
let
  fun f 0 = true | f n = g (n-1)
  fun g n = n = 1 orelse f (n-1)
in f end
```

- datatypes: recursive, pattern-matching
- state (references), exceptions, modules

# CakeML

## Language features

- functions: higher-order, polymorphic, mutually recursive

```
fn x => if x then "hi" else "bye";
let
  fun f 0 = true | f n = g (n-1)
  fun g n = n = 1 orelse f (n-1)
in f end
```

- datatypes: recursive, pattern-matching
- state (references), exceptions, modules

A real programming language.

# CakeML

## Language features

- functions: higher-order, polymorphic, mutually recursive

```
fn x => if x then "hi" else "bye";
let
  fun f 0 = true | f n = g (n-1)
  fun g n = n = 1 orelse f (n-1)
in f end
```

- datatypes: recursive, pattern-matching
- state (references), exceptions, modules

A real programming language.
But many similarities to HOL.

# Interlude: ML

What is ML?

# Interlude: ML

## What is ML?

- A family of programming languages, including Standard ML and OCaml (and CakeML), developed by Milner and others in the 70s.

# Interlude: ML

## What is ML?

- A family of programming languages, including Standard ML and OCaml (and CakeML), developed by Milner and others in the 70s.
- Many theorem provers are written in ML, including Isabelle and HOL4.

# Interlude: ML

## What is ML?

- A family of programming languages, including Standard ML and OCaml (and CakeML), developed by Milner and others in the 70s.
- Many theorem provers are written in ML, including Isabelle and HOL4.
- Stands for "meta-language", because its original use was implementing LCF theorem prover, which has an "object language", namely, the logic.

# Interlude: ML

## What is ML?

- A family of programming languages, including Standard ML and OCaml (and CakeML), developed by Milner and others in the 70s.
- Many theorem provers are written in ML, including Isabelle and HOL4.
- Stands for "meta-language", because its original use was implementing LCF theorem prover, which has an "object language", namely, the logic.

Nowadays a general programming language, and used in industry.

# Interlude: ML

## What is ML?

- A family of programming languages, including Standard ML and OCaml (and CakeML), developed by Milner and others in the 70s.
- Many theorem provers are written in ML, including Isabelle and HOL4.
- Stands for "meta-language", because its original use was implementing LCF theorem prover, which has an "object language", namely, the logic.

Nowadays a general programming language, and used in industry.

## Characteristics

Functional, strict, impure, type safe, modular.

# Deep map

Remember this?

$\texttt{map}\ f\ [\,] = [\,]$

$\texttt{map}\ f\ (h \# t) = f\ h \# \texttt{map}\ f\ t$

# Deep map

## Remember this?

map $f$ [] = []
map $f$ $(h \# t)$ = $f$ $h \#$ map $f$ $t$

## Compare

```
Dletrec
 [("map","v3",
   Fun "v4"
    (Mat (VarS "v4")
      [(PconS "nil" [],ConS "nil" []);
       (PconS "::" [Pvar "v2"; Pvar "v1"],
        ConS "::"
         [Call (VarS "v3") (VarS "v2");
          Call (Call (VarS "map") (VarS "v3")) (VarS "v1")])]))]
```

# Deep map, pretty-printed

Easier to read in concrete syntax

```
fun map v3 v4 =
  case v4
  of [] => []
  |  v2::v1 => (v3 v2::(map v3 v1));
```

# Deep map, pretty-printed

Easier to read in concrete syntax

```
fun map v3 v4 =
  case v4
  of [] => []
  |  v2::v1 => (v3 v2::(map v3 v1));
```

Let us name this deeply-embedded declaration `map_dec`.

# Proofs about deep embeddings

## Another declaration

```
val it = map (fn x => (x + 1)) [1,2,0];
```

# Proofs about deep embeddings

## Another declaration

```
val it = map (fn x => (x + 1)) [1,2,0];
```

Call this `map_suc_dec`.

# Proofs about deep embeddings

### Another declaration

```
val it = map (fn x => (x + 1)) [1,2,0];
```

Call this `map_suc_dec`.

### Clock bound

As promised, we can now reason about the number of steps.

# Proofs about deep embeddings

## Another declaration

```
val it = map (fn x => (x + 1)) [1,2,0];
```

Call this `map_suc_dec`.

## Clock bound

As promised, we can now reason about the number of steps.

$\vdash$ `evaluate_decs` $st$ $env$ [`map_dec`; `map_suc_dec`] =
  ($st'$,_,`Rval` $res$) $\Rightarrow$
  $st$.clock $\geq 10$

# Proofs about deep embeddings

## Another declaration

```
val it = map (fn x => (x + 1)) [1,2,0];
```

Call this `map_suc_dec`.

## Clock bound

As promised, we can now reason about the number of steps.

$\vdash$ `evaluate_decs` *st env* [`map_dec`; `map_suc_dec`] =
  ($st'$,_,`Rval` *res*) $\Rightarrow$
  $st$.clock $\geq 10$

How hard was this to prove?

# Proofs about deep embeddings

## Another declaration

```
val it = map (fn x => (x + 1)) [1,2,0];
```

Call this `map_suc_dec`.

## Clock bound

As promised, we can now reason about the number of steps.

$\vdash$ `evaluate_decs` $st$ $env$ [`map_dec`; `map_suc_dec`] =
  ($st'$,_,`Rval` $res$) $\Rightarrow$
  $st$.clock $\geq 10$

How hard was this to prove?

Using `EVAL` the proof is short, but takes many seconds to run.

# More general proofs

Deep embeddings let us reason about the semantics in general.

# More general proofs

Deep embeddings let us reason about the semantics in general.

Type safety

# More general proofs

Deep embeddings let us reason about the semantics in general.

## Type safety

- We can define a type system over deeply-embedded syntax.

# More general proofs

Deep embeddings let us reason about the semantics in general.

## Type safety

- We can define a type system over deeply-embedded syntax.
- We can prove that well-typed programs never crash

# More general proofs

Deep embeddings let us reason about the semantics in general.

## Type safety

- We can define a type system over deeply-embedded syntax.
- We can prove that well-typed programs never crash (they only diverge or terminate with a value or un-handled exception).

# More general proofs

Deep embeddings let us reason about the semantics in general.

## Type safety

- We can define a type system over deeply-embedded syntax.
- We can prove that well-typed programs never crash (they only diverge or terminate with a value or un-handled exception).

## Alternative semantics

# More general proofs

Deep embeddings let us reason about the semantics in general.

## Type safety

- We can define a type system over deeply-embedded syntax.
- We can prove that well-typed programs never crash (they only diverge or terminate with a value or un-handled exception).

## Alternative semantics

- You may have seen relational big-step semantics, as well as small-step operational semantics.

# More general proofs

Deep embeddings let us reason about the semantics in general.

## Type safety

- We can define a type system over deeply-embedded syntax.
- We can prove that well-typed programs never crash (they only diverge or terminate with a value or un-handled exception).

## Alternative semantics

- You may have seen relational big-step semantics, as well as small-step operational semantics.
- We can prove equivalences between different versions of the semantics, and obtain a solid understanding of our language.

# Deep proofs are hard

Remember this?

$\vdash \forall l\ f.\ \texttt{length}\ (\texttt{map}\ f\ l) = \texttt{length}\ l$

# Deep proofs are hard

Remember this?

$\vdash\ \forall l\ f.\ \texttt{length}\ (\texttt{map}\ f\ l) = \texttt{length}\ l$

How do we prove it about the deep embedding?

# Deep proofs are hard

Remember this?

$\vdash\ \forall l\ f.\ \texttt{length}\ (\texttt{map}\ f\ l) = \texttt{length}\ l$

How do we prove it about the deep embedding?

Induct, simp?

# Deep proofs are hard

## Remember this?

$\vdash \ \forall l\ f.\ \texttt{length}\ (\texttt{map}\ f\ \ l) = \texttt{length}\ l$

How do we prove it about the deep embedding?

## Induct, simp?

Nope: the deep embedding gets in the way.

# Deep proofs are hard

Remember this?

$\vdash\ \forall l\ f.\ \texttt{length}\ (\texttt{map}\ f\ l) = \texttt{length}\ l$

How do we prove it about the deep embedding?

Induct, simp?

Nope: the deep embedding gets in the way.

It is possible, but much more cumbersome.

# Deep proofs are hard

## Remember this?

$\vdash \forall l\ f.\ \texttt{length}\ (\texttt{map}\ f\ l) = \texttt{length}\ l$

How do we prove it about the deep embedding?

## Induct, simp?

Nope: the deep embedding gets in the way.
It is possible, but much more cumbersome.
But can we get it automatically from the shallow proof?

# Deep proofs are hard

## Remember this?

$\vdash\ \forall l\ f.\ \texttt{length}\ (\texttt{map}\ f\ \ l) = \texttt{length}\ l$

How do we prove it about the deep embedding?

## Induct, simp?

Nope: the deep embedding gets in the way.
It is possible, but much more cumbersome.
But can we get it automatically from the shallow proof?

(You may have seen a similar thing before, e.g., Autocorres.)

# Connecting shallow to deep

Question

What is the deep counterpart of this term?

`Suc (Suc (Suc 0))`

# Connecting shallow to deep

### Question

What is the deep counterpart of this term?

`Suc (Suc (Suc 0))`

### Answer

`Litv (IntLit (toInt (Suc (Suc (Suc 0)))))`

# Connecting shallow to deep

**Question**

What is the deep counterpart of this term?

`Suc (Suc (Suc 0))`

**Answer**

`Litv (IntLit (toInt (Suc (Suc (Suc 0)))))`

(of type `v`, rather than `nat`)

# Connecting shallow to deep

Question

How about the unit value?

()

# Connecting shallow to deep

### Question

How about the unit value?

`()`

### Answer

`Conv None []`

# Connecting shallow to deep

### Question

How about the unit value?

`()`

### Answer

`Conv None []`

### Refinement invariants

We can characterise these relationships:

# Connecting shallow to deep

## Question

How about the unit value?

`()`

## Answer

`Conv None []`

## Refinement invariants

We can characterise these relationships:

$$\texttt{INT}\ i\ v \iff v = \texttt{Litv}\ (\texttt{IntLit}\ i)$$

# Connecting shallow to deep

## Question

How about the unit value?

`()`

## Answer

`Conv None []`

## Refinement invariants

We can characterise these relationships:

$$\texttt{INT}\ i\ v \iff v = \texttt{Litv}\ (\texttt{IntLit}\ i)$$

$$\texttt{NAT}\ n\ v \iff \texttt{INT}\ (\texttt{toInt}\ n)\ v$$

# Connecting shallow to deep

### Question

How about the unit value?

()

### Answer

Conv None [ ]

### Refinement invariants

We can characterise these relationships:

$$\texttt{INT}\ i\ v \iff v = \texttt{Litv}\ (\texttt{IntLit}\ i)$$

$$\texttt{NAT}\ n\ v \iff \texttt{INT}\ (\texttt{toInt}\ n)\ v$$

$$\texttt{UNIT}\ u\ v \iff v = \texttt{Conv None}\ [\,]$$

# Shallow to deep datatypes

## Question

What is the deep counterpart of this term?

[0; 2; 1]

# Shallow to deep datatypes

## Question

What is the deep counterpart of this term?

[0; 2; 1]

## Answer

```
ConvS "list" "::"
  [Litv (IntLit 0);
   ConvS "list" "::"
     [Litv (IntLit 1);
      ConvS "list" "::" [Litv (IntLit 2); ConvS "list" "nil" []]]]
```

# Shallow to deep datatypes

## Question

What is the deep counterpart of this term?
`[0; 2; 1]`

## Answer

```
ConvS "list" "::"
  [Litv (IntLit 0);
   ConvS "list" "::"
     [Litv (IntLit 1);
      ConvS "list" "::" [Litv (IntLit 2); ConvS "list" "nil" []]]]
```

## Refinement invariant

`LIST` *A ls v* means *v* relates to *ls*, if *A* relates the elements.

# Shallow to deep datatypes

## Question

What is the deep counterpart of this term?

[0; 2; 1]

## Answer

```
ConvS "list" "::"
  [Litv (IntLit 0);
   ConvS "list" "::"
     [Litv (IntLit 1);
      ConvS "list" "::" [Litv (IntLit 2); ConvS "list" "nil" []]]]
```

## Refinement invariant

LIST $A$ $ls$ $v$ means $v$ relates to $ls$, if $A$ relates the elements.

LIST $A$ [] $v$ $\iff$ $v$ = ConvS "list" "nil" []

# Shallow to deep datatypes

## Question

What is the deep counterpart of this term?

[0; 2; 1]

## Answer

```
ConvS "list" "::"
  [Litv (IntLit 0);
   ConvS "list" "::"
     [Litv (IntLit 1);
      ConvS "list" "::" [Litv (IntLit 2); ConvS "list" "nil" []]]]
```

## Refinement invariant

LIST $A$ $ls$ $v$ means $v$ relates to $ls$, if $A$ relates the elements.

$\texttt{LIST}\ A\ [\,]\ v \iff v = \texttt{ConvS "list" "nil"}\ [\,]$

$\texttt{LIST}\ A\ (h \# t)\ v \iff$

$\exists\, v_1\ v_2.\ v = \texttt{ConvS "list" "::"}\ [v_1;\ v_2] \land A\ h\ v_1 \land \texttt{LIST}\ A\ t\ v_2$

# Connecting shallow to deep

Question

What is the deep counterpart of this term?

$\lambda x.\ x + x$

# Connecting shallow to deep

**Question**

What is the deep counterpart of this term?

$\lambda x.\ x + x$

**Answer**

`Closure` *env* `"x" (App (Opn Plus) [VarS "x"; VarS "x"])`

# Connecting shallow to deep

## Question

What is the deep counterpart of this term?

$\lambda x.\ x + x$

## Answer

`Closure` *env* `"x" (App (Opn Plus) [VarS "x"; VarS "x"])`

There are many answers, for many *env*s.

# Connecting shallow to deep

## Question

What is the deep counterpart of this term?

$\lambda x.\ x + x$

## Answer

`Closure` *env* `"x" (App (Opn Plus) [VarS "x"; VarS "x"])`

There are many answers, for many *env*s.

(Not to mention the many equivalent expressions.)

# Connecting shallow to deep

### Question

What is the deep counterpart of this term?

$\lambda x.\ x + x$

### Answer

`Closure` *env* `"x" (App (Opn Plus) [VarS "x"; VarS "x"])`

There are many answers, for many *env*s.

(Not to mention the many equivalent expressions.)

### Refinement invariant

How can we characterise this relationship?

# Shallow to deep functions

Refinement invariant

$(\texttt{NAT} \rightarrow \texttt{NAT})\ f\ v$ means:

# Shallow to deep functions

Refinement invariant

$(\texttt{NAT} \to \texttt{NAT})\ f\ v$ means:

$v$ is a closure implementing the function $f$

# Shallow to deep functions

## Refinement invariant

(NAT $\rightarrow$ NAT) $f$ $v$ means:
$v$ is a closure implementing the function $f$
(which should be of type `nat` $\rightarrow$ `nat`, in this case)

# Shallow to deep functions

### Refinement invariant

$(\texttt{NAT} \to \texttt{NAT})\ f\ v$ means:
$v$ is a closure implementing the function $f$
(which should be of type $\texttt{nat} \to \texttt{nat}$, in this case)

### Definition

$(A \to B)\ f\ v \iff$

# Shallow to deep functions

## Refinement invariant

$(\texttt{NAT} \to \texttt{NAT})\ f\ v$ means:

$v$ is a closure implementing the function $f$

(which should be of type $\texttt{nat} \to \texttt{nat}$, in this case)

## Definition

$$(A \to B)\ f\ v \iff$$
$$\forall x\ v_1.$$
$$A\ x\ v_1 \Rightarrow$$

# Shallow to deep functions

## Refinement invariant

$(\mathtt{NAT} \rightarrow \mathtt{NAT})\ f\ v$ means:

$v$ is a closure implementing the function $f$

(which should be of type $\mathtt{nat} \rightarrow \mathtt{nat}$, in this case)

## Definition

$$
\begin{array}{l}
(A \rightarrow B)\ f\ v \iff \\
\quad \forall\, x\ v_1 . \\
\quad\quad A\ x\ v_1 \Rightarrow \\
\quad\quad\quad \exists\, v_2\ env\ exp\ k . \\
\quad\quad\quad\quad (\mathtt{do\_call}\ [v;\ v_1] = \mathtt{Some}\ (env, exp) \wedge \\
\quad\quad\quad\quad\ \mathtt{evaluate}\ (\mathtt{st}_0\ \textit{with}\ \mathrm{clock} := k)\ env\ [exp] = \\
\quad\quad\quad\quad\quad (\mathtt{st}_0, \mathtt{Rval}\ [v_2])) \wedge B\ (f\ x)\ v_2
\end{array}
$$

# Shallow to deep map

Question

What is the deep counterpart of this term?

`map`

# Shallow to deep map

## Question

What is the deep counterpart of this term?

`map`

## Answer

Any closure, $\texttt{map}_\mathsf{v}$, satisfying this refinement invariant:

$$((A \to B) \to \texttt{LIST}\ A \to \texttt{LIST}\ B)\ \texttt{map}\ \texttt{map}_\mathsf{v}$$

# Shallow to deep map

Question

What is the deep counterpart of this term?

`map`

Answer

Any closure, $\mathtt{map}_\mathsf{v}$, satisfying this refinement invariant:

$((A \rightarrow B) \rightarrow \mathtt{LIST}\ A \rightarrow \mathtt{LIST}\ B)\ \mathtt{map}\ \mathtt{map}_\mathsf{v}$

Is that enough?

# Shallow to deep map

## Question

What is the deep counterpart of this term?

`map`

## Answer

Any closure, $\mathtt{map}_\mathsf{v}$, satisfying this refinement invariant:

$((A \to B) \to \mathtt{LIST}\ A \to \mathtt{LIST}\ B)\ \mathtt{map}\ \mathtt{map}_\mathsf{v}$

## Is that enough?

Yes, only closures that behave like `map` satisfy this invariant.

# Shallow to deep expressions

Question

What is the deep counterpart of this term?

$(\lambda x.\ x + x)\ 3$

# Shallow to deep expressions

## Question

What is the deep counterpart of this term?

$(\lambda x.\ x + x)\ 3$

## Trick question

That term does not correspond to a value (it can be simplified).

# Shallow to deep expressions

### Question

What is the deep counterpart of this term?
$(\lambda x.\ x + x)\ 3$

### Trick question

That term does not correspond to a value (it can be simplified).

### Answer

The deep counterpart is an expression, not a value:

# Shallow to deep expressions

### Question

What is the deep counterpart of this term?

$(\lambda x.\ x + x)\ 3$

### Trick question

That term does not correspond to a value (it can be simplified).

### Answer

The deep counterpart is an expression, not a value:

```
Call (Fun "x" (App (Opn Plus) [VarS "x"; VarS "x"]))
  (Lit (IntLit 3))
```

# Shallow to deep expressions

## Correctness

What constitutes correspondence between shallow and deep?

# Shallow to deep expressions

## Correctness

What constitutes correspondence between shallow and deep?

Why is this

$(\lambda x.\ x + x)\ 3$

# Shallow to deep expressions

## Correctness

What constitutes correspondence between shallow and deep?

Why is this

$(\lambda x.\ x + x)\ 3$

refined by this

# Shallow to deep expressions

## Correctness

What constitutes correspondence between shallow and deep?

Why is this

$(\lambda x.\ x + x)\ 3$

refined by this

```
Call (Fun "x" (App (Opn Plus) [VarS "x"; VarS "x"]))
  (Lit (IntLit 3))
```

?

# Shallow to deep expressions

## Correctness

What constitutes correspondence between shallow and deep?

Why is this

$(\lambda x.\ x + x)\ 3$

refined by this

```
Call (Fun "x" (App (Opn Plus) [VarS "x"; VarS "x"]))
  (Lit (IntLit 3))
```

?

## Answer

The semantics justifies the connection.

# Shallow to deep expressions

## Correctness

What constitutes correspondence between shallow and deep?

Why is this

$(\lambda x.\ x + x)\ 3$

refined by this

```
Call (Fun "x" (App (Opn Plus) [VarS "x"; VarS "x"]))
  (Lit (IntLit 3))
```

?

## Answer

The semantics justifies the connection.

$$\vdash\ \exists k\ res.\ \texttt{evaluate}\ (\texttt{st}_0\ \textit{with}\ \text{clock} := k)\ env\ [\texttt{Call (Fun "x" (App (Opn Plus) [VarS "x"; VarS "x"])) (Lit (IntLit 3))}] = (\texttt{st}_0,\texttt{Rval}\ [res]) \wedge \texttt{NAT}\ ((\lambda x.\ x + x)\ 3)\ res$$

# Certificate theorems

### Definition

A *certificate theorem* for deep embedding *exp* and refinement invariant *A* states:

# Certificate theorems

**Definition**

A *certificate theorem* for deep embedding *exp* and refinement invariant $A$ states:

$\exists\, k\ res.$
$\texttt{evaluate}\ (\texttt{st}_0\ \textit{with}\ \text{clock} := k)\ env\ [exp] = (\texttt{st}_0, \texttt{Rval}\ [res]) \land A\ res$

# Certificate theorems

### Definition

A *certificate theorem* for deep embedding *exp* and refinement invariant $A$ states:

$\exists\, k\ res.$

$\texttt{evaluate}\ (\texttt{st}_0\ \textit{with}\ \text{clock} := k)\ env\ [exp] = (\texttt{st}_0,\texttt{Rval}\ [res]) \wedge$
$A\ res$

We abbreviate this by `Cert` *env exp A*.

# Certificate theorems

### Definition

A *certificate theorem* for deep embedding *exp* and refinement invariant *A* states:

$\exists\, k\ res.$
$\quad \texttt{evaluate}\ (\texttt{st}_0\ \textit{with}\ \texttt{clock} := k)\ env\ [exp] = (\texttt{st}_0,\texttt{Rval}\ [res]) \wedge$
$\quad A\ res$

We abbreviate this by `Cert` *env exp A*.

### Example

$\vdash\ \texttt{Cert}\ env\ (\texttt{ConS}\ \texttt{"::"}\ [\texttt{Con None}\ [\,];\ \texttt{ConS}\ \texttt{"nil"}\ [\,]])$
$\quad (\text{LIST UNIT}\ [()])$

# Certificate theorem for map

Question

What is the deep counterpart of `map`, considered as an expression?

# Certificate theorem for map

**Question**

What is the deep counterpart of `map`, considered as an expression?

**Answer**

Just a variable: `VarS "map"`.

# Certificate theorem for map

### Question

What is the deep counterpart of `map`, considered as an expression?

### Answer

Just a variable: `VarS "map"`.
But it is only correct in the right environment:

# Certificate theorem for map

Question

What is the deep counterpart of `map`, considered as an expression?

Answer

Just a variable: `VarS "map"`.

But it is only correct in the right environment:

$\vdash$ `lookup_var "map"` *env* = Some $\mathtt{map}_\mathsf{v}$ $\Rightarrow$

Cert *env* (`VarS "map"`) $(((a \to b) \to \mathtt{LIST}\ a \to \mathtt{LIST}\ b)\ \mathtt{map})$

# Certificate theorem for map

**Question**

What is the deep counterpart of `map`, considered as an expression?

**Answer**

Just a variable: `VarS "map"`.

But it is only correct in the right environment:

$$\vdash \texttt{lookup\_var "map"}\ \mathit{env} = \texttt{Some}\ \texttt{map}_{\mathsf{v}} \Rightarrow$$
$$\texttt{Cert}\ \mathit{env}\ (\texttt{VarS "map"})\ (((a \to b) \to \texttt{LIST}\ a \to \texttt{LIST}\ b)\ \texttt{map})$$

Now, how can we use this certificate theorem?

# Deep results for shallow proofs

Remember this?

$\vdash$ `length` (`map` $f$ $l$) = `length` $l$

# Deep results for shallow proofs

Remember this?

$\vdash \texttt{length}\ (\texttt{map}\ f\ l) = \texttt{length}\ l$

The deep version

# Deep results for shallow proofs

## Remember this?

$\vdash$ `length` (`map` $f$ $l$) = `length` $l$

## The deep version

$\vdash$ `lookup_var "map"` $env$ = `Some` $\texttt{map}_v$ $\wedge$
`lookup_var "length"` $env$ = `Some` $\texttt{length}_v$ $\Rightarrow$

# Deep results for shallow proofs

## Remember this?

$\vdash$ `length` (`map` $f$ $l$) = `length` $l$

## The deep version

$\vdash$ `lookup_var "map"` $env$ = `Some` $\texttt{map}_v$ $\wedge$
`lookup_var "length"` $env$ = `Some` $\texttt{length}_v$ $\Rightarrow$
`lookup_var "l"` $env$ = `Some` $l_v$ $\wedge$ LIST $a$ $l$ $l_v$ $\Rightarrow$

# Deep results for shallow proofs

## Remember this?

$\vdash$ `length` (`map` $f$ $l$) = `length` $l$

## The deep version

$\vdash$ `lookup_var "map"` *env* = `Some` $\texttt{map}_\mathsf{v}$ $\wedge$
`lookup_var "length"` *env* = `Some` $\texttt{length}_\mathsf{v}$ $\Rightarrow$
`lookup_var "l"` *env* = `Some` $l_\mathsf{v}$ $\wedge$ `LIST` $a$ $l$ $l_\mathsf{v}$ $\Rightarrow$
`lookup_var "f"` *env* = `Some` $f_\mathsf{v}$ $\wedge$ $(a \rightarrow b)$ $f$ $f_\mathsf{v}$ $\Rightarrow$

# Deep results for shallow proofs

## Remember this?

$\vdash$ `length` (`map` $f$ $l$) = `length` $l$

## The deep version

$\vdash$ `lookup_var "map"` $env$ = `Some` $\texttt{map}_\mathsf{v}$ $\wedge$
`lookup_var "length"` $env$ = `Some` $\texttt{length}_\mathsf{v}$ $\Rightarrow$
`lookup_var "l"` $env$ = `Some` $l_\mathsf{v}$ $\wedge$ `LIST` $a$ $l$ $l_\mathsf{v}$ $\Rightarrow$
`lookup_var "f"` $env$ = `Some` $f_\mathsf{v}$ $\wedge$ $(a \rightarrow b)$ $f$ $f_\mathsf{v}$ $\Rightarrow$
`Cert` $env$ (`Call` (`VarS "length"`) (`VarS "l"`))
(`NAT` (`length` $l$)) $\wedge$
`Cert` $env$
(`Call` (`VarS "length"`)
(`Call` (`Call` (`VarS "map"`) (`VarS "f"`)) (`VarS "l"`)))
(`NAT` (`length` (`map` $f$ $l$)))

# Deep results for shallow proofs

## Remember this?

$\vdash$ `length` (`map` $f$ $l$) = `length` $l$

## The deep version

$\vdash$ `lookup_var "map"` $env$ = `Some` $\texttt{map}_v$ $\wedge$
  `lookup_var "length"` $env$ = `Some` $\texttt{length}_v$ $\Rightarrow$
   `lookup_var "l"` $env$ = `Some` $l_v$ $\wedge$ `LIST` $a$ $l$ $l_v$ $\Rightarrow$
    `lookup_var "f"` $env$ = `Some` $f_v$ $\wedge$ ($a \rightarrow b$) $f$ $f_v$ $\Rightarrow$
     `Cert` $env$ (`Call` (`VarS "length"`) (`VarS "l"`))
      (`NAT` (`length` $l$)) $\wedge$
     `Cert` $env$
      (`Call` (`VarS "length"`)
        (`Call` (`Call` (`VarS "map"`) (`VarS "f"`)) (`VarS "l"`)))
      (`NAT` (`length` (`map` $f$ $l$)))

Follows directly from the certificate theorems for `map` and `length`.

# Certificate theorems compose

Derived rules

$\vdash$ `Cert` *env* (`Lit` (`IntLit` (`toInt` *n*))) (`NAT` *n*)

# Certificate theorems compose

## Derived rules

$\vdash \texttt{Cert}\ \mathit{env}\ (\texttt{Lit}\ (\texttt{IntLit}\ (\texttt{toInt}\ n)))\ (\texttt{NAT}\ n)$

$\vdash \texttt{Cert}\ \mathit{env}\ e_1\ ((A \to B)\ f) \Rightarrow$
$\quad \texttt{Cert}\ \mathit{env}\ e_2\ (A\ x) \Rightarrow \texttt{Cert}\ \mathit{env}\ (\texttt{Call}\ e_1\ e_2)\ (B\ (f\ x))$

# Certificate theorems compose

## Derived rules

$\vdash$ `Cert` *env* (`Lit` (`IntLit` (`toInt` *n*))) (`NAT` *n*)

$\vdash$ `Cert` *env* $e_1$ $((A \to B)\ f) \Rightarrow$
  `Cert` *env* $e_2$ $(A\ x) \Rightarrow$ `Cert` *env* (`Call` $e_1$ $e_2$) $(B\ (f\ x))$

$\vdash$ `Cert` *env* $e_1$ (`BOOL` $b_1$) $\Rightarrow$
  `Cert` *env* $e_2$ (`BOOL` $b_2$) $\Rightarrow$
   `Cert` *env*
    (`If` $e_1$ $e_2$
      (`App` (`Opb Leq`) [`Lit` (`IntLit` 0); `Lit` (`IntLit` 0)]))
    (`BOOL` $(b_1 \Rightarrow b_2)$)

# Certificate theorems compose

## Derived rules

$\vdash$ Cert *env* (Lit (IntLit (toInt $n$))) (NAT $n$)

$\vdash$ Cert *env* $e_1$ (($A \to B$) $f$) $\Rightarrow$
  Cert *env* $e_2$ ($A$ $x$) $\Rightarrow$ Cert *env* (Call $e_1$ $e_2$) ($B$ ($f$ $x$))

$\vdash$ Cert *env* $e_1$ (BOOL $b_1$) $\Rightarrow$
  Cert *env* $e_2$ (BOOL $b_2$) $\Rightarrow$
   Cert *env*
    (If $e_1$ $e_2$
      (App (Opb Leq) [Lit (IntLit 0); Lit (IntLit 0)]))
    (BOOL ($b_1 \Rightarrow b_2$))

$\vdash$ $A$ $x$ $v$ $\Rightarrow$
  lookup_var $n$ *env* = Some $v$ $\Rightarrow$ Cert *env* (VarS $n$) ($A$ $x$)

# Certificate theorems compose

## Derived rules

$\vdash$ `Cert` *env* (`Lit` (`IntLit` (`toInt` *n*))) (`NAT` *n*)

$\vdash$ `Cert` *env* $e_1$ $((A \to B)\ f) \Rightarrow$
`Cert` *env* $e_2$ $(A\ x) \Rightarrow$ `Cert` *env* (`Call` $e_1$ $e_2$) $(B\ (f\ x))$

$\vdash$ `Cert` *env* $e_1$ (`BOOL` $b_1$) $\Rightarrow$
`Cert` *env* $e_2$ (`BOOL` $b_2$) $\Rightarrow$
`Cert` *env*
(`If` $e_1$ $e_2$
(`App` (`Opb Leq`) [`Lit` (`IntLit` 0); `Lit` (`IntLit` 0)]))
(`BOOL` $(b_1 \Rightarrow b_2)$)

$\vdash$ $A\ x\ v \Rightarrow$
`lookup_var` *n* *env* = `Some` *v* $\Rightarrow$ `Cert` *env* (`VarS` *n*) $(A\ x)$

By composing certificates, we can generate a certified deep embedding by traversing a shallow term.

# Proof-producing code generation

## That is the idea

From shallow embeddings we can *automatically* generate *certified* deep embeddings.

# Proof-producing code generation

## That is the idea

From shallow embeddings we can *automatically* generate *certified* deep embeddings.

## CakeML code generation features

- Automatic certified code generation.

# Proof-producing code generation

## That is the idea

From shallow embeddings we can *automatically* generate *certified* deep embeddings.

## CakeML code generation features

- Automatic certified code generation.
- Supports recursive functions and datatypes.

# Proof-producing code generation

## That is the idea

From shallow embeddings we can *automatically* generate *certified* deep embeddings.

## CakeML code generation features

- Automatic certified code generation.
- Supports recursive functions and datatypes.
- Can generate modular code (ML structures).

# Proof-producing code generation

## That is the idea

From shallow embeddings we can *automatically* generate *certified* deep embeddings.

## CakeML code generation features

- Automatic certified code generation.
- Supports recursive functions and datatypes.
- Can generate modular code (ML structures).
- Can generate stateful code

# Proof-producing code generation

## That is the idea

From shallow embeddings we can *automatically* generate *certified* deep embeddings.

## CakeML code generation features

- Automatic certified code generation.
- Supports recursive functions and datatypes.
- Can generate modular code (ML structures).
- Can generate stateful code (room for improvement).

# Proof-producing code generation

## That is the idea

From shallow embeddings we can *automatically* generate *certified* deep embeddings.

## CakeML code generation features

- Automatic certified code generation.
- Supports recursive functions and datatypes.
- Can generate modular code (ML structures).
- Can generate stateful code (room for improvement).

“Certified implementations from verified algorithms”

# What we have seen so far

## Evaluation problems

Fast simplification within the logic using `EVAL`.

# What we have seen so far

## Evaluation problems

Fast simplification within the logic using `EVAL`.
"Evaluate" HOL terms as if with a functional-program interpreter.

# What we have seen so far

## Evaluation problems

Fast simplification within the logic using `EVAL`.
"Evaluate" HOL terms as if with a functional-program interpreter.

## Certified deep embeddings

# What we have seen so far

## Evaluation problems

Fast simplification within the logic using `EVAL`.
"Evaluate" HOL terms as if with a functional-program interpreter.

## Certified deep embeddings

Automatic generation of a *real* functional program from a HOL term, *plus*...

# What we have seen so far

## Evaluation problems

Fast simplification within the logic using `EVAL`.
"Evaluate" HOL terms as if with a functional-program interpreter.

## Certified deep embeddings

Automatic generation of a *real* functional program from a HOL term, *plus*... a certificate theorem stating that the CakeML code correctly implements the HOL term.

# What we have seen so far

## Evaluation problems

Fast simplification within the logic using `EVAL`.
"Evaluate" HOL terms as if with a functional-program interpreter.

## Certified deep embeddings

Automatic generation of a *real* functional program from a HOL term, *plus*... a certificate theorem stating that the CakeML code correctly implements the HOL term.

## Next up

Verified compilation

# What else can we do with syntax?

## Functions on syntax

Within the logic, we have defined semantic functions.

- evaluate, of type
  $\alpha\,\texttt{s} \rightarrow \texttt{senv} \rightarrow \texttt{exp}\,\texttt{list} \rightarrow \alpha\,\texttt{s} \times (\texttt{v}\,\texttt{list},\ \texttt{v})\,\texttt{result}$

# What else can we do with syntax?

## Functions on syntax

Within the logic, we have defined semantic functions.

- evaluate, of type
  $\alpha\ \texttt{s} \rightarrow \texttt{senv} \rightarrow \texttt{exp}\ \texttt{list} \rightarrow \alpha\ \texttt{s} \times (\texttt{v}\ \texttt{list},\ \texttt{v})\ \texttt{result}$
- welltyped, of type
  $\texttt{tenv} \rightarrow \texttt{exp} \rightarrow \texttt{bool}$

# What else can we do with syntax?

## Functions on syntax

Within the logic, we have defined semantic functions.

- `evaluate`, of type
  $\alpha$ `s` $\rightarrow$ `senv` $\rightarrow$ `exp list` $\rightarrow$ $\alpha$ `s` $\times$ (`v list`, `v`) `result`
- `welltyped`, of type
  `tenv` $\rightarrow$ `exp` $\rightarrow$ `bool`

## Another function

How about transforming the syntax? e.g.,

# What else can we do with syntax?

## Functions on syntax

Within the logic, we have defined semantic functions.

- `evaluate`, of type
  $\alpha\ \texttt{s} \rightarrow \texttt{senv} \rightarrow \texttt{exp list} \rightarrow \alpha\ \texttt{s} \times (\texttt{v list},\ \texttt{v})\ \texttt{result}$
- `welltyped`, of type
  $\texttt{tenv} \rightarrow \texttt{exp} \rightarrow \texttt{bool}$

## Another function

How about transforming the syntax? e.g.,

- `compile_exp`, of type
  $\texttt{cs} \rightarrow \texttt{exp} \rightarrow \texttt{cs} \times \texttt{byte list}$

# What else can we do with syntax?

## Functions on syntax

Within the logic, we have defined semantic functions.

- `evaluate`, of type
  $\alpha\,\texttt{s} \to \texttt{senv} \to \texttt{exp list} \to \alpha\,\texttt{s} \times (\texttt{v list},\ \texttt{v})\,\texttt{result}$
- `welltyped`, of type
  $\texttt{tenv} \to \texttt{exp} \to \texttt{bool}$

## Another function

How about transforming the syntax? e.g.,

- `compile_exp`, of type
  $\texttt{cs} \to \texttt{exp} \to \texttt{cs} \times \texttt{byte list}$

(You saw something like this in Assignment 2)

# Anatomy of a compiler

## Compiler definition

Would something like this work

$\texttt{compile\_exp}\ cs\ (\texttt{Lit}\ (\texttt{IntLit}\ 2)) = (cs, [184w;\ 2w;\ 0w;\ 0w;\ 0w])$

?

# Anatomy of a compiler

## Compiler definition

Would something like this work

`compile_exp` $cs$ (`Lit` (`IntLit` 2)) = ($cs$, [184$w$; 2$w$; 0$w$; 0$w$; 0$w$])

?

## Does this scale?

# Anatomy of a compiler

## Compiler definition

Would something like this work

`compile_exp` *cs* (`Lit` (`IntLit` 2)) = (*cs*, [184*w*; 2*w*; 0*w*; 0*w*; 0*w*])

?

## Does this scale?

No.

What do you do for `compile_exp` *cs* (`Fun` *x* *exp*), for example?

# Anatomy of a compiler

## Compiler definition

Would something like this work

`compile_exp` *cs* (`Lit` (`IntLit` 2)) = (*cs*, [184*w*; 2*w*; 0*w*; 0*w*; 0*w*])

?

## Does this scale?

No.

What do you do for `compile_exp` *cs* (`Fun` *x* *exp*), for example?

Compilation is rather more involved than the semantics.

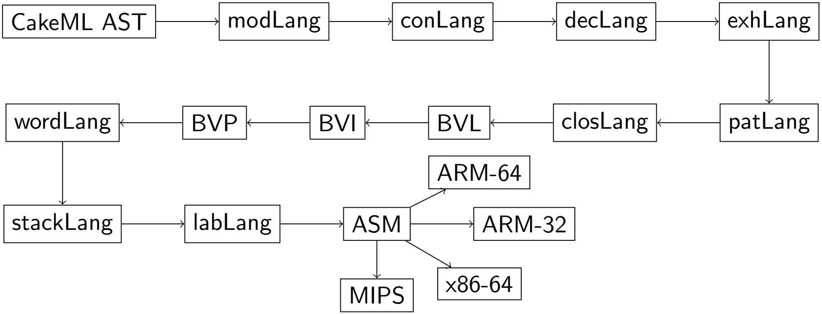
# Intermediate languages

Many phases

# Intermediate languages

## Many phases

CakeML compiler backend:



# Compiling pattern matching

## A small peek

exhLang ⟶ patLang compiles `case` to nested `if`.

# Compiling pattern matching

## A small peek

exhLang ⟶ patLang compiles `case` to nested `if`.

## Example

```
case (C0 1) of C1 => raise C2 | (C0 x) => x
```

# Compiling pattern matching

A small peek

exhLang ⟶ patLang compiles `case` to nested `if`.

Example

```
case (C0 1) of C1 => raise C2 | (C0 x) => x
```

compiles to

```
let C0 1 in if v0 = C1 then raise C2 else el 0 v0
```

# Compiling pattern matching

A small peek

exhLang ⟶ patLang compiles `case` to nested `if`.

Example

```
case (C0 1) of C1 => raise C2 | (C0 x) => x
```

compiles to

```
let C0 1 in if v0 = C1 then raise C2 else el 0 v0
```

Or, in the deep embedding

# Compiling pattern matching

## A small peek

exhLang → patLang compiles case to nested `if`.

## Example

```
case (C0 1) of C1 => raise C2 | (C0 x) => x
```
compiles to
```
let C0 1 in if v0 = C1 then raise C2 else el 0 v0
```

## Or, in the deep embedding

```
⊢ compile []
    (Mat (Con 0 [Lit (IntLit 1)])
      [(Pcon 1 [],Raise (Con 2 []));
       (Pcon 0 [Pvar "x"],Var "x")]) =
    Let (Con 0 [Lit (IntLit 1)])
     (If (App (Op Eq) [Vardb 0; Con 1 []])
       (Raise (Con 2 [])) (App (El 0) [Vardb 0]))
```

# Compiling patterns correctly

**Question**

What do we need to prove about `compile`?

# Compiling patterns correctly

### Question

What do we need to prove about `compile`?

### Answer

That it preserves semantics:

# Compiling patterns correctly

## Question

What do we need to prove about `compile`?

## Answer

That it preserves semantics: the semantics of the compiled program is the same as the semantics of the source program.

# Compiling patterns correctly

### Question

What do we need to prove about `compile`?

### Answer

That it preserves semantics: the semantics of the compiled program is the same as the semantics of the source program.

### In more detail

$\vdash\ \texttt{evaluate}_{\text{exh}}\ \mathit{env}_{\text{exh}}\ s_{\text{exh}}\ [\mathit{exp}_{\text{exh}}] = ({s'}_{\text{exh}}, r_{\text{exh}}) \Rightarrow$

# Compiling patterns correctly

## Question

What do we need to prove about `compile`?

## Answer

That it preserves semantics: the semantics of the compiled program is the same as the semantics of the source program.

## In more detail

$$\vdash\ \texttt{evaluate}_{\mathrm{exh}}\ \mathit{env}_{\mathrm{exh}}\ s_{\mathrm{exh}}\ [\mathit{exp}_{\mathrm{exh}}] = (s'_{\mathrm{exh}}, r_{\mathrm{exh}}) \Rightarrow$$
$$r_{\mathrm{exh}} \neq \texttt{Rerr}\ (\texttt{Rabort}\ \texttt{Rtype\_error}) \Rightarrow$$

# Compiling patterns correctly

## Question

What do we need to prove about `compile`?

## Answer

That it preserves semantics: the semantics of the compiled program is the same as the semantics of the source program.

## In more detail

$$\vdash \ \texttt{evaluate}_{\mathsf{exh}}\ \mathit{env}_{\mathsf{exh}}\ s_{\mathsf{exh}}\ [\mathit{exp}_{\mathsf{exh}}] = (s'_{\mathsf{exh}}, r_{\mathsf{exh}}) \Rightarrow$$
$$r_{\mathsf{exh}} \neq \texttt{Rerr}\ (\texttt{Rabort Rtype\_error}) \Rightarrow$$
$$\texttt{sem\_rel}\ (\mathit{env}_{\mathsf{exh}}, s_{\mathsf{exh}})\ (\mathit{env}_{\mathsf{pat}}, s_{\mathsf{pat}}) \Rightarrow$$

# Compiling patterns correctly

## Question

What do we need to prove about `compile`?

## Answer

That it preserves semantics: the semantics of the compiled program is the same as the semantics of the source program.

## In more detail

$$
\begin{array}{l}
\vdash \ \texttt{evaluate}_{\mathrm{exh}}\ env_{\mathrm{exh}}\ s_{\mathrm{exh}}\ [exp_{\mathrm{exh}}] = (s'_{\mathrm{exh}}, r_{\mathrm{exh}}) \Rightarrow \\
\quad r_{\mathrm{exh}} \neq \texttt{Rerr (Rabort Rtype\_error)} \Rightarrow \\
\quad\ \texttt{sem\_rel}\ (env_{\mathrm{exh}}, s_{\mathrm{exh}})\ (env_{\mathrm{pat}}, s_{\mathrm{pat}}) \Rightarrow \\
\quad\quad \exists\, s'_{\mathrm{pat}}\ r_{\mathrm{pat}}. \\
\quad\quad\quad \texttt{evaluate}_{\mathrm{pat}}\ env_{\mathrm{pat}}\ s_{\mathrm{pat}}\ [\texttt{compile}\ (\texttt{bvs}\ env_{\mathrm{exh}})\ exp_{\mathrm{exh}}] = \\
\quad\quad\quad\ (s'_{\mathrm{pat}}, r_{\mathrm{pat}}) \wedge \texttt{s\_rel}\ s'_{\mathrm{exh}}\ s'_{\mathrm{pat}} \wedge \texttt{res\_rel}\ r_{\mathrm{exh}}\ r_{\mathrm{pat}}
\end{array}
$$

# Verified compilation

## Compiler correctness theorem shape

- If the source program $e$ evaluates in $s_1$ to result $r_1$,

# Verified compilation

## Compiler correctness theorem shape

- If the source program $e$ evaluates in $s_1$ to result $r_1$,
- and if $s_1$ is related to $s_2$,

# Verified compilation

## Compiler correctness theorem shape

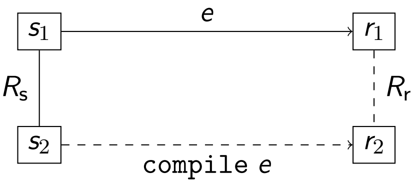
- If the source program $e$ evaluates in $s_1$ to result $r_1$,
- and if $s_1$ is related to $s_2$,
- then `compile` $e$ evaluates in $s_2$ to result $r_2$, and $r_1$ is related to $r_2$.

# Verified compilation

## Compiler correctness theorem shape

- If the source program $e$ evaluates in $s_1$ to result $r_1$,
- and if $s_1$ is related to $s_2$,
- then `compile` $e$ evaluates in $s_2$ to result $r_2$, and $r_1$ is related to $r_2$.

## In a picture

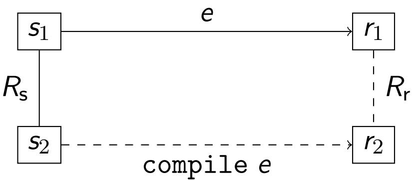


# Verified compilation

## Compiler correctness theorem shape

- If the source program $e$ evaluates in $s_1$ to result $r_1$,
- and if $s_1$ is related to $s_2$,
- then `compile` $e$ evaluates in $s_2$ to result $r_2$, and $r_1$ is related to $r_2$.

## In a picture



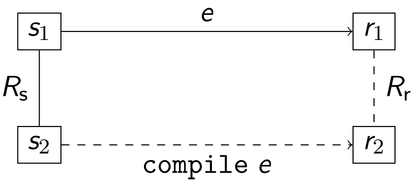

Proof idea:

# Verified compilation

## Compiler correctness theorem shape

- If the source program $e$ evaluates in $s_1$ to result $r_1$,
- and if $s_1$ is related to $s_2$,
- then `compile` $e$ evaluates in $s_2$ to result $r_2$, and $r_1$ is related to $r_2$.

## In a picture



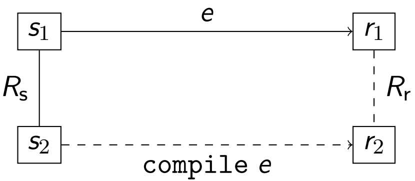

Proof idea: induction on source semantics.

# Verified compilation

## Compiler correctness theorem shape

- If the source program $e$ evaluates in $s_1$ to result $r_1$,
- and if $s_1$ is related to $s_2$,
- then `compile` $e$ evaluates in $s_2$ to result $r_2$, and $r_1$ is related to $r_2$.

## In a picture




Proof idea: induction on source semantics. A natural fit.

# Verified compilation

Is that enough?

# Verified compilation

## Is that enough?

- As a compiler user, we do not want to have to assume the source program evaluates to a result.

# Verified compilation

## Is that enough?

- As a compiler user, we do not want to have to assume the source program evaluates to a result.
- Rather we want to know that *whatever the compiled program does*, that behaviour is permitted by the source semantics.

# Verified compilation

## Is that enough?

- As a compiler user, we do not want to have to assume the source program evaluates to a result.
- Rather we want to know that *whatever the compiled program does*, that behaviour is permitted by the source semantics.

## But which programs might not evaluate?

# Verified compilation

## Is that enough?

- As a compiler user, we do not want to have to assume the source program evaluates to a result.
- Rather we want to know that *whatever the compiled program does*, that behaviour is permitted by the source semantics.

## But which programs might not evaluate?

- Programs that crash...

# Verified compilation

## Is that enough?

- As a compiler user, we do not want to have to assume the source program evaluates to a result.
- Rather we want to know that *whatever the compiled program does*, that behaviour is permitted by the source semantics.

## But which programs might not evaluate?

- Programs that crash...
- Programs that diverge (loop forever)...

# Verified compilation

## Is that enough?

- As a compiler user, we do not want to have to assume the source program evaluates to a result.
- Rather we want to know that *whatever the compiled program does*, that behaviour is permitted by the source semantics.

## But which programs might not evaluate?

- Programs that crash...
- Programs that diverge (loop forever)...
- It depends on what style of semantics you use.

# Functional big-step

## Big-step for compiler verification

- Big-step semantics are defined inductively over the syntax.
- Just like the compiler, so there is a natural proof structure.

# Functional big-step

## Big-step for compiler verification

- Big-step semantics are defined inductively over the syntax.
- Just like the compiler, so there is a natural proof structure.
- Functional big-step is naturally total: crashes are explicit.

# Functional big-step

## Big-step for compiler verification

- Big-step semantics are defined inductively over the syntax.
- Just like the compiler, so there is a natural proof structure.
- Functional big-step is naturally total: crashes are explicit.

## The clock enables divergence preservation

- If we prove the compiler preserves timeouts,

# Functional big-step

## Big-step for compiler verification

- Big-step semantics are defined inductively over the syntax.
- Just like the compiler, so there is a natural proof structure.
- Functional big-step is naturally total: crashes are explicit.

## The clock enables divergence preservation

- If we prove the compiler preserves timeouts,
- then the compiled code diverges if and only if the source code diverges.

# Divergence preservation

## Definition (or theorem)

*exp diverges* iff:

$$\forall k.\ \exists s'.\ \texttt{evaluate}\ (s\ \textit{with}\ \mathrm{clock}\ := k)\ \mathit{env}\ [\mathit{exp}] = (s', \texttt{Rerr}\ (\texttt{Rabort}\ \mathit{Rtimeout}))$$

# Divergence preservation

## Definition (or theorem)

*exp diverges* iff:

$\forall k.$

$\exists s'.$

`evaluate` $(s$ *with* clock $:= k)$ *env* $[exp]$ =
$(s'$, `Rerr` (`Rabort` *Rtimeout*))

## Consequence

If the compiled code

- times out whenever the source code times out, and,

# Divergence preservation

## Definition (or theorem)

*exp diverges* iff:

$$\forall k.\ \exists s'.\ \texttt{evaluate}\ (s\ \textit{with}\ \text{clock}\ := k)\ \textit{env}\ [\textit{exp}] = (s', \texttt{Rerr}\ (\texttt{Rabort}\ \textit{Rtimeout}))$$

## Consequence

If the compiled code

- times out whenever the source code times out, and,
- converges whenever the source code converges,

# Divergence preservation

## Definition (or theorem)

*exp diverges* iff:

$\forall k.$

$\quad\exists s'.$

$\quad\quad$ `evaluate` $(s$ *with* clock $:= k)$ *env* $[exp] =$

$\quad\quad\quad(s',$`Rerr` $($`Rabort` *Rtimeout*$))$

## Consequence

If the compiled code

- times out whenever the source code times out, and,
- converges whenever the source code converges,

then it diverges iff the source code diverges.

# Divergence preservation

## Definition (or theorem)

*exp diverges* iff:

$\forall\, k.$

$\exists\, s'.$

$\texttt{evaluate}\ (s\ \textit{with}\ \mathrm{clock}\ :=\ k)\ \textit{env}\ [\textit{exp}] =$

$(s', \texttt{Rerr}\ (\texttt{Rabort}\ \textit{Rtimeout}))$

## Consequence

If the compiled code

- times out whenever the source code times out, and,
- converges whenever the source code converges,

then it diverges iff the source code diverges.
(As long as both source and target semantics are *deterministic*.)

# Verified compilation for CakeML

So far we briefly saw one small phase of the compiler from abstract syntax.

# Verified compilation for CakeML

So far we briefly saw one small phase of the compiler from abstract syntax. This is typically called the *backend*.

# Verified compilation for CakeML

So far we briefly saw one small phase of the compiler from abstract syntax. This is typically called the *backend*.

## Other pieces of a compiler

- Lexing and parsing: from concrete syntax to abstract syntax.

# Verified compilation for CakeML

So far we briefly saw one small phase of the compiler from abstract syntax. This is typically called the *backend*.

## Other pieces of a compiler

- Lexing and parsing: from concrete syntax to abstract syntax.
- Type inference: reject ill-typed programs.

# Verified compilation for CakeML

So far we briefly saw one small phase of the compiler from abstract syntax. This is typically called the *backend*.

## Other pieces of a compiler

- Lexing and parsing: from concrete syntax to abstract syntax.
- Type inference: reject ill-typed programs.

The top-level compiler for CakeML has the following type:
`cs` → `string` → `compiler_result`

# Verified compilation for CakeML

So far we briefly saw one small phase of the compiler from abstract syntax. This is typically called the *backend*.

## Other pieces of a compiler

- Lexing and parsing: from concrete syntax to abstract syntax.
- Type inference: reject ill-typed programs.

The top-level compiler for CakeML has the following type:

```
cs → string → compiler_result
where compiler_result =
        ParseError
      | TypeError
      | Success cs (byte list)
```

# Verified compilation for CakeML

## Source semantics

- Specified grammar for concrete syntax, and how it maps to abstract syntax.

# Verified compilation for CakeML

## Source semantics

- Specified grammar for concrete syntax, and how it maps to abstract syntax.
- Specified type system for whole programs.

# Verified compilation for CakeML

## Source semantics

- Specified grammar for concrete syntax, and how it maps to abstract syntax.
- Specified type system for whole programs.
- If a program parses and type checks, then its semantics is one of:

# Verified compilation for CakeML

## Source semantics

- Specified grammar for concrete syntax, and how it maps to abstract syntax.
- Specified type system for whole programs.
- If a program parses and type checks, then its semantics is one of:
  - Terminate with a value or exception, or,

# Verified compilation for CakeML

## Source semantics

- Specified grammar for concrete syntax, and how it maps to abstract syntax.
- Specified type system for whole programs.
- If a program parses and type checks, then its semantics is one of:
  - Terminate with a value or exception, or,
  - Diverge.

# Verified compilation for CakeML

## Source semantics

- Specified grammar for concrete syntax, and how it maps to abstract syntax.
- Specified type system for whole programs.
- If a program parses and type checks, then its semantics is one of:
  - Terminate with a value or exception, or,
  - Diverge.
- The latest version of CakeML adds a trace of I/O events to each of these options.

# Verified compilation for CakeML

## Target semantics

- Instruction semantics for each target machine (x86-64, ARM-32, etc.).

# Verified compilation for CakeML

## Target semantics

- Instruction semantics for each target machine (x86-64, ARM-32, etc.).
- Specifies a machine state (memory, registers, etc.), and a “next state” relation for each instruction.

# Verified compilation for CakeML

## Target semantics

- Instruction semantics for each target machine (x86-64, ARM-32, etc.).
- Specifies a machine state (memory, registers, etc.), and a "next state" relation for each instruction.
- Validated (in some cases) by evaluation of the model compared with execution of real hardware.

# Verified compilation for CakeML

## Correctness theorem

- If the semantics state *st* and compiler state *cs* are related correctly, then consider the result of `compile` *cs prog*:

# Verified compilation for CakeML

## Correctness theorem

- If the semantics state *st* and compiler state *cs* are related correctly, then consider the result of `compile` *cs prog*:
- If `ParseError`, then *prog* does not satisfy the grammar.

# Verified compilation for CakeML

## Correctness theorem

- If the semantics state *st* and compiler state *cs* are related correctly, then consider the result of `compile` *cs prog*:
- If `ParseError`, then *prog* does not satisfy the grammar.
- If `TypeError`, then *prog* is not well typed in *st*.

# Verified compilation for CakeML

## Correctness theorem

- If the semantics state *st* and compiler state *cs* are related correctly, then consider the result of `compile` *cs prog*:
- If `ParseError`, then *prog* does not satisfy the grammar.
- If `TypeError`, then *prog* is not well typed in *st*.
- Otherwise, consider all machine states *ms* in which the compiled code is loaded correctly.

# Verified compilation for CakeML

## Correctness theorem

- If the semantics state *st* and compiler state *cs* are related correctly, then consider the result of `compile` *cs* *prog*:
- If `ParseError`, then *prog* does not satisfy the grammar.
- If `TypeError`, then *prog* is not well typed in *st*.
- Otherwise, consider all machine states *ms* in which the compiled code is loaded correctly. The semantics of running (repeatedly stepping) *ms* includes:

# Verified compilation for CakeML

## Correctness theorem

- If the semantics state *st* and compiler state *cs* are related correctly, then consider the result of `compile` *cs prog*:
- If `ParseError`, then *prog* does not satisfy the grammar.
- If `TypeError`, then *prog* is not well typed in *st*.
- Otherwise, consider all machine states *ms* in which the compiled code is loaded correctly. The semantics of running (repeatedly stepping) *ms* includes:
  - Divergence if *prog* can diverge in *st*.

# Verified compilation for CakeML

## Correctness theorem

- If the semantics state *st* and compiler state *cs* are related correctly, then consider the result of `compile` *cs prog*:
- If `ParseError`, then *prog* does not satisfy the grammar.
- If `TypeError`, then *prog* is not well typed in *st*.
- Otherwise, consider all machine states *ms* in which the compiled code is loaded correctly. The semantics of running (repeatedly stepping) *ms* includes:
  - Divergence if *prog* can diverge in *st*.
  - Termination if *prog* can terminate in *st*.

# Verified compilation for CakeML

## Correctness theorem

- If the semantics state *st* and compiler state *cs* are related correctly, then consider the result of `compile` *cs prog*:
- If `ParseError`, then *prog* does not satisfy the grammar.
- If `TypeError`, then *prog* is not well typed in *st*.
- Otherwise, consider all machine states *ms* in which the compiled code is loaded correctly. The semantics of running (repeatedly stepping) *ms* includes:
  - Divergence if *prog* can diverge in *st*.
  - Termination if *prog* can terminate in *st*.
  - Termination with a resource error (e.g., out of memory).

# Verified compilation for CakeML

## Correctness theorem

- If the semantics state *st* and compiler state *cs* are related correctly, then consider the result of `compile` *cs prog*:
- If `ParseError`, then *prog* does not satisfy the grammar.
- If `TypeError`, then *prog* is not well typed in *st*.
- Otherwise, consider all machine states *ms* in which the compiled code is loaded correctly. The semantics of running (repeatedly stepping) *ms* includes:
  - Divergence if *prog* can diverge in *st*.
  - Termination if *prog* can terminate in *st*.
  - Termination with a resource error (e.g., out of memory).
- Furthermore the I/O events match up with the semantics.

# Verified compilation for CakeML

## Correctness theorem

- If the semantics state *st* and compiler state *cs* are related correctly, then consider the result of `compile` *cs prog*:
- If `ParseError`, then *prog* does not satisfy the grammar.
- If `TypeError`, then *prog* is not well typed in *st*.
- Otherwise, consider all machine states *ms* in which the compiled code is loaded correctly. The semantics of running (repeatedly stepping) *ms* includes:
  - Divergence if *prog* can diverge in *st*.
  - Termination if *prog* can terminate in *st*.
  - Termination with a resource error (e.g., out of memory).
- Furthermore the I/O events match up with the semantics.

Shorthand: "`compile` *cs prog* implements *prog*"

# What we have seen so far

## Evaluation problems

Fast simplification within the logic using `EVAL`.
"Evaluate" HOL terms as if with a functional-program interpreter.

## Certified deep embeddings

Automatic generation of a *real* functional program from a HOL term, *plus*... a certificate theorem stating that the CakeML code correctly implements the HOL term.

## Verified compilation

# What we have seen so far

## Evaluation problems

Fast simplification within the logic using `EVAL`.
"Evaluate" HOL terms as if with a functional-program interpreter.

## Certified deep embeddings

Automatic generation of a *real* functional program from a HOL term, *plus*... a certificate theorem stating that the CakeML code correctly implements the HOL term.

## Verified compilation

An algorithm turning CakeML code into machine code, *plus*...

# What we have seen so far

## Evaluation problems

Fast simplification within the logic using `EVAL`.
"Evaluate" HOL terms as if with a functional-program interpreter.

## Certified deep embeddings

Automatic generation of a *real* functional program from a HOL term, *plus*... a certificate theorem stating that the CakeML code correctly implements the HOL term.

## Verified compilation

An algorithm turning CakeML code into machine code, *plus*... a theorem stating that the semantics of the compiled code is permitted by the semantics of the source code.

# Compiler as shallow embedding

### Question

How can we run the verified compiler?

# Compiler as shallow embedding

### Question

How can we run the verified compiler?

### Problem?

Recall, the compiler is a function in HOL...

# Compiler as shallow embedding

### Question

How can we run the verified compiler?

### Problem?

Recall, the compiler is a function in HOL...

### Answer

Running the compiler is an *evaluation problem*.

# Compiler as shallow embedding

Question

How can we run the verified compiler?

Problem?

Recall, the compiler is a function in HOL...

Answer

Running the compiler is an *evaluation problem*.
We can use `EVAL` to run the compiler in the logic.

# Evaluating the compiler

## Remember this?

`map_suc_dec`, pretty-printed:

```
val it = map (fn x => (x + 1)) [1,2,0];
```

# Evaluating the compiler

## Remember this?

`map_suc_dec`, pretty-printed:

```
val it = map (fn x => (x + 1)) [1,2,0];
```

## Example

Input term:

`compile_ast` $\texttt{cs}_0$ `[Tdec map_dec; Tdec map_suc_dec]`.

# Evaluating the compiler

### Remember this?

`map_suc_dec`, pretty-printed:

```
val it = map (fn x => (x + 1)) [1,2,0];
```

### Example

Input term:

`compile_ast` $cs_0$ `[Tdec map_dec; Tdec map_suc_dec]`.

Produces:

$\vdash$ `compile_ast` $cs_0$ `[Tdec map_dec; Tdec map_suc_dec]` =
`Success` $cs_1$ `map_suc_code`

for some $cs_1$ and `map_suc_code`.

# Compiler as deep embedding

**Question**

But how can we run the compiler quicker?

# Compiler as deep embedding

### Question

But how can we run the compiler quicker?

### Recall

The compiler is a function in HOL...

# Compiler as deep embedding

## Question

But how can we run the compiler quicker?

## Recall

The compiler is a function in HOL...

## Possible answer

Can we use proof-producing code generation?

# Compiler as deep embedding

### Question

But how can we run the compiler quicker?

### Recall

The compiler is a function in HOL...

### Possible answer

Can we use proof-producing code generation?

Yes, to produce CakeML code implementing the compiler.

# Compiler as deep embedding

Generating code implementing the compiler

What is the deep counterpart of `compile`?

# Compiler as deep embedding

## Generating code implementing the compiler

What is the deep counterpart of `compile`?

Some declaration

`compile_dec = Dletrec [("compile","cs",Fun "prog" ...)]`

satisfying...

# Compiler as deep embedding

## Generating code implementing the compiler

What is the deep counterpart of `compile`?

Some declaration

`compile_dec = Dletrec [("compile","cs",Fun "prog" ...)]`

satisfying...

## Certificate theorem

$\vdash$ `EnvContains` *env* `compiler_decs` $\Rightarrow$
  `Cert` *env* (`VarS "compile"`) ((`CS` $\rightarrow$ `STRING` $\rightarrow$ `CR`) `compile`)

# Compiler as deep embedding

## Generating code implementing the compiler

What is the deep counterpart of compile?

Some declaration

$\texttt{compile\_dec} = \texttt{Dletrec}\ [(\texttt{"compile"},\texttt{"cs"},\texttt{Fun}\ \texttt{"prog"}\ \ldots)]$

satisfying...

## Certificate theorem

$\vdash \texttt{EnvContains}\ \mathit{env}\ \texttt{compiler\_decs} \Rightarrow$
$\quad \texttt{Cert}\ \mathit{env}\ (\texttt{VarS}\ \texttt{"compile"})\ ((\texttt{CS} \to \texttt{STRING} \to \texttt{CR})\ \texttt{compile})$

compiler_decs includes all the preliminary declarations required to define the compiler.

# Compiler as deep embedding

## Generating code implementing the compiler

What is the deep counterpart of `compile`?

Some declaration

`compile_dec = Dletrec [("compile","cs",Fun "prog" ...)]`

satisfying...

## Certificate theorem

$\vdash$ `EnvContains` *env* `compiler_decs` $\Rightarrow$

`Cert` *env* `(VarS "compile")` `((CS` $\rightarrow$ `STRING` $\rightarrow$ `CR) compile)`

`compiler_decs` includes all the preliminary declarations required to define the compiler.

## But how do we run it?

# Compiler as deep embedding

## Generating code implementing the compiler

What is the deep counterpart of `compile`?

Some declaration

`compile_dec = Dletrec [("compile","cs",Fun "prog" ...)]`

satisfying...

## Certificate theorem

$\vdash$ `EnvContains` *env* `compiler_decs` $\Rightarrow$

`Cert` *env* `(VarS "compile")` `((CS` $\rightarrow$ `STRING` $\rightarrow$ `CR) compile)`

`compiler_decs` includes all the preliminary declarations required to define the compiler.

## But how do we run it?

Now we have the compiler as CakeML code...

# The perhaps obvious next step

## To run CakeML code, first compile it

- Evaluation problem:
  `compile_ast` $cs_0$ [`Struct "CakeML" compiler_decs`]

# The perhaps obvious next step

To run CakeML code, first compile it

- Evaluation problem:
  `compile_ast` $cs_0$ [`Struct "CakeML" compiler_decs`]
- Use `EVAL` to solve it.

# The perhaps obvious next step

## To run CakeML code, first compile it

- Evaluation problem:
  `compile_ast` $\mathtt{cs}_0$ `[Struct "CakeML" compiler_decs]`
- Use `EVAL` to solve it. (Takes several hours.)

# The perhaps obvious next step

## To run CakeML code, first compile it

- Evaluation problem:
  `compile_ast` $\texttt{cs}_0$ `[Struct "CakeML" compiler_decs]`
- Use `EVAL` to solve it. (Takes several hours.)
- Produces a *bootstrapping theorem*:

# The perhaps obvious next step

## To run CakeML code, first compile it

- Evaluation problem:
  `compile_ast` $\texttt{cs}_0$ `[Struct "CakeML" compiler_decs]`
- Use `EVAL` to solve it. (Takes several hours.)
- Produces a *bootstrapping theorem*:

⊢ `compile_ast` $\texttt{cs}_0$ `[Struct "CakeML" compiler_decs]` =
  `Success` $\texttt{cs}_2$ `compiler_code`

# The perhaps obvious next step

## To run CakeML code, first compile it

- Evaluation problem:
  `compile_ast` $cs_0$ `[Struct "CakeML" compiler_decs]`
- Use `EVAL` to solve it. (Takes several hours.)
- Produces a *bootstrapping theorem*:

⊢ `compile_ast` $cs_0$ `[Struct "CakeML" compiler_decs]` =
  `Success` $cs_2$ `compiler_code`

## To run machine code, print and execute

- At this point we must step out of the logic.

# The perhaps obvious next step

## To run CakeML code, first compile it

- Evaluation problem:

  `compile_ast` $\texttt{cs}_0$ `[Struct "CakeML" compiler_decs]`
- Use `EVAL` to solve it. (Takes several hours.)
- Produces a *bootstrapping theorem*:

  ⊢ `compile_ast` $\texttt{cs}_0$ `[Struct "CakeML" compiler_decs] =`

  `Success` $\texttt{cs}_2$ `compiler_code`

## To run machine code, print and execute

- At this point we must step out of the logic.
- We assume our machine model (and loader etc.) is correct.

# Bootstrapping

## What we have

1. Correctness theorem:

# Bootstrapping

## What we have

1. Correctness theorem:
   $\vdash \forall \mathit{cs}\ \mathit{prog}\,.\ \texttt{compile\_ast}\ \mathit{cs}\ \mathit{prog}$ implements $\mathit{prog}$

# Bootstrapping

## What we have

1. Correctness theorem:
   $\vdash \forall\, cs\ prog\,.\ \texttt{compile\_ast}\ cs\ prog$ implements $prog$
2. Certificate theorem:

# Bootstrapping

## What we have

1. Correctness theorem:
   $\vdash \forall\, \mathit{cs}\ \mathit{prog}\,.\ \texttt{compile\_ast}\ \mathit{cs}\ \mathit{prog}\ \text{implements}\ \mathit{prog}$
2. Certificate theorem:
   $\vdash\ \texttt{EnvContains}\ \mathit{env}\ \texttt{compiler\_decs} \Rightarrow$
   $\quad \texttt{Cert}\ \mathit{env}\ (\texttt{VarS}\ \texttt{"compile"})\ ((\texttt{CS} \rightarrow \texttt{STRING} \rightarrow \texttt{CR})\ \texttt{compile})$

# Bootstrapping

## What we have

1. Correctness theorem:
   $\vdash \forall\, cs\ prog\,.\ \texttt{compile\_ast}\ cs\ prog\ \text{implements}\ prog$
2. Certificate theorem:
   $\vdash\ \texttt{EnvContains}\ env\ \texttt{compiler\_decs} \Rightarrow$
   $\quad \texttt{Cert}\ env\ (\texttt{VarS "compile"})\ ((\texttt{CS} \to \texttt{STRING} \to \texttt{CR})\ \texttt{compile})$
3. Bootstrapping theorem:

# Bootstrapping

## What we have

1. Correctness theorem:
   $\vdash \forall\, cs\ prog\,.\ \texttt{compile\_ast}\ cs\ prog\ \text{implements}\ prog$
2. Certificate theorem:
   $\vdash\ \texttt{EnvContains}\ env\ \texttt{compiler\_decs} \Rightarrow$
   $\quad \texttt{Cert}\ env\ (\texttt{VarS "compile"})\ ((\texttt{CS} \rightarrow \texttt{STRING} \rightarrow \texttt{CR})\ \texttt{compile})$
3. Bootstrapping theorem:
   $\vdash\ \texttt{compile\_ast}\ \texttt{cs}_0\ [\texttt{Struct "CakeML" compiler\_decs}] =$
   $\quad \texttt{Success}\ \texttt{cs}_2\ \texttt{compiler\_code}$

# Bootstrapping

## What we have

1. Correctness theorem:
   $\vdash \forall\, cs\ prog\,.$ `compile_ast` $cs\ prog$ implements $prog$
2. Certificate theorem:
   $\vdash$ `EnvContains` $env$ `compiler_decs` $\Rightarrow$
   `Cert` $env$ (`VarS "compile"`) ((`CS` $\rightarrow$ `STRING` $\rightarrow$ `CR`) `compile`)
3. Bootstrapping theorem:
   $\vdash$ `compile_ast` $\mathtt{cs}_0$ [`Struct "CakeML" compiler_decs`] =
   `Success` $\mathtt{cs}_2$ `compiler_code`

## Put them together

- 1 and 3: $\vdash$ `compiler_code` implements `compiler_decs`.

# Bootstrapping

## What we have

1. Correctness theorem:
   $\vdash \forall\, cs\ prog\,.\ \texttt{compile\_ast}\ cs\ prog\ \text{implements}\ prog$
2. Certificate theorem:
   $\vdash\ \texttt{EnvContains}\ env\ \texttt{compiler\_decs} \Rightarrow$
   $\quad \texttt{Cert}\ env\ (\texttt{VarS "compile"})\ ((\texttt{CS} \rightarrow \texttt{STRING} \rightarrow \texttt{CR})\ \texttt{compile})$
3. Bootstrapping theorem:
   $\vdash\ \texttt{compile\_ast}\ \texttt{cs}_0\ [\texttt{Struct "CakeML" compiler\_decs}] =$
   $\quad \texttt{Success}\ \texttt{cs}_2\ \texttt{compiler\_code}$

## Put them together

- 1 and 3: $\vdash$ compiler_code implements compiler_decs.
- plus 2: $\vdash$ compiler_code implements compile.

# Compiler verification

Result

We have verified machine code implementing the compiler.

# Compiler verification

### Result

We have verified machine code implementing the compiler.

### Dimensions of compiler verification

- How far the compiler goes:

# Compiler verification

## Result

We have verified machine code implementing the compiler.

## Dimensions of compiler verification

- How far the compiler goes:
  string $\rightarrow$ AST $\rightarrow$ ILs $\rightarrow \cdots \rightarrow$ asm $\rightarrow$ bytes

# Compiler verification

## Result

We have verified machine code implementing the compiler.

## Dimensions of compiler verification

- How far the compiler goes:
  string $\rightarrow$ AST $\rightarrow$ ILs $\rightarrow \cdots \rightarrow$ asm $\rightarrow$ bytes
- Which level of the compiler is verified:

# Compiler verification

## Result

We have verified machine code implementing the compiler.

## Dimensions of compiler verification

- How far the compiler goes:
  string $\rightarrow$ AST $\rightarrow$ ILs $\rightarrow \cdots \rightarrow$ asm $\rightarrow$ bytes
- Which level of the compiler is verified:
  algorithm (shallow), high-level code (deep), machine code

# Compiler verification

## Result

We have verified machine code implementing the compiler.

## Dimensions of compiler verification

- How far the compiler goes:
  string $\rightarrow$ AST $\rightarrow$ ILs $\rightarrow \cdots \rightarrow$ asm $\rightarrow$ bytes
- Which level of the compiler is verified:
  algorithm (shallow), high-level code (deep), machine code
- CakeML covers the full spectrum of both dimensions.

# A loose end

## Alternative to simplification outside the logic

- Can we use the verified compiler to get *fast* evaluation without needing to assert axioms?

# A loose end

## Alternative to simplification outside the logic

- Can we use the verified compiler to get *fast* evaluation without needing to assert axioms?

## Yes, but not yet

# A loose end

## Alternative to simplification outside the logic

- Can we use the verified compiler to get *fast* evaluation without needing to assert axioms?

## Yes, but not yet

- Still need to run the machine code outside the logic, and lose the connection.

# A loose end

## Alternative to simplification outside the logic

- Can we use the verified compiler to get *fast* evaluation without needing to assert axioms?

## Yes, but not yet

- Still need to run the machine code outside the logic, and lose the connection.
- Work in progress: building a verified theorem prover that includes evaluation by compilation...

# What we have seen

## Evaluation problems

Fast simplification within the logic using `EVAL`.
"Evaluate" HOL terms as if with a functional-program interpreter.

## Certified deep embeddings

Automatic generation of a *real* functional program from a HOL term, *plus*... a certificate theorem stating that the CakeML code correctly implements the HOL term.

## Verified compilation

An algorithm turning CakeML code into machine code, *plus*... a theorem stating that the semantics of the compiled code is permitted by the semantics of the source code.

## Bootstrapping

# What we have seen

## Evaluation problems

Fast simplification within the logic using `EVAL`.
"Evaluate" HOL terms as if with a functional-program interpreter.

## Certified deep embeddings

Automatic generation of a *real* functional program from a HOL term, *plus*... a certificate theorem stating that the CakeML code correctly implements the HOL term.

## Verified compilation

An algorithm turning CakeML code into machine code, *plus*... a theorem stating that the semantics of the compiled code is permitted by the semantics of the source code.

## Bootstrapping

Combining the above to get a verified compiler in machine code.

# CakeML

## People involved

# CakeML

## People involved

Currently: Anthony Fox (Cambridge),

# CakeML

## People involved

Currently: Anthony Fox (Cambridge), Armaël Guéneau (ENS Lyon),

# CakeML

## People involved

Currently: Anthony Fox (Cambridge), Armaël Guéneau (ENS Lyon), Ramana Kumar (Data61),

# CakeML

## People involved

Currently: Anthony Fox (Cambridge), Armaël Guéneau (ENS Lyon), Ramana Kumar (Data61), Magnus Myreen (Chalmers),

# CakeML

## People involved

Currently: Anthony Fox (Cambridge), Armaël Guéneau (ENS Lyon), Ramana Kumar (Data61), Magnus Myreen (Chalmers), Michael Norrish (Data61),

# CakeML

## People involved

Currently: Anthony Fox (Cambridge), Armaël Guéneau (ENS Lyon), Ramana Kumar (Data61), Magnus Myreen (Chalmers), Michael Norrish (Data61), Scott Owens (Kent),

# CakeML

## People involved

Currently: Anthony Fox (Cambridge), Armaël Guéneau (ENS Lyon), Ramana Kumar (Data61), Magnus Myreen (Chalmers), Michael Norrish (Data61), Scott Owens (Kent), Yong Kiam Tan (CMU)

# CakeML

## People involved

Currently: Anthony Fox (Cambridge), Armaël Guéneau (ENS Lyon), Ramana Kumar (Data61), Magnus Myreen (Chalmers), Michael Norrish (Data61), Scott Owens (Kent), Yong Kiam Tan (CMU)

## Effort

Started in 2012. 3-6 people working on it. Builds on lots of previous work (mainly HOL4).

# CakeML

## People involved

Currently: Anthony Fox (Cambridge), Armaël Guéneau (ENS Lyon), Ramana Kumar (Data61), Magnus Myreen (Chalmers), Michael Norrish (Data61), Scott Owens (Kent), Yong Kiam Tan (CMU)

## Effort

Started in 2012. 3-6 people working on it. Builds on lots of previous work (mainly HOL4).

## CakeML is free software

# CakeML

## People involved

Currently: Anthony Fox (Cambridge), Armaël Guéneau (ENS Lyon), Ramana Kumar (Data61), Magnus Myreen (Chalmers), Michael Norrish (Data61), Scott Owens (Kent), Yong Kiam Tan (CMU)

## Effort

Started in 2012. 3-6 people working on it. Builds on lots of previous work (mainly HOL4).

## CakeML is free software

You can be involved!
`https://cakeml.org`