CakeML: bootstrapping a verified compiler

Ramana Kumar

COMP4161, 20 October 2015

DATA 61 CSIRO

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?

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

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

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

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?

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

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]

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).
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` $P\ g\ (g\ x)\ \text{else}\ x$

An evaluation problem

What is the solution for this input term?

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

Answer

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

Example

Another evaluation problem

What about this input term?

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

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?

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

- 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

- 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:
  - 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?

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?

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
```

Answer

```
exp =
    Lit lit
  | Var (string id)
  | Con (string id option) (exp list)
  | Fun string exp
  | App op (exp list)
  | If exp exp exp
  | Raise exp
  | Handle exp ((pat × exp) list)
  | ...
```

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 := $(n, v_2) \# env.\mathrm{v}, e$)
do_call [Litv *l*; v_2] = None
...

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

DATA 61 CSIRO

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?

- 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?

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

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

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 =
  (fn 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::[]);
```

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 ≥ 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.

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?

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

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))`

Answer

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

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

Connecting shallow to deep

Question

How about the unit value?

`()`

Answer

`Conv None []`

Refinement invariants

We can characterise these relationships:

$$\mathtt{INT}\ i\ v \iff v = \mathtt{Litv}\ (\mathtt{IntLit}\ i)$$
$$\mathtt{NAT}\ n\ v \iff \mathtt{INT}\ (\mathtt{toInt}\ n)\ v$$
$$\mathtt{UNIT}\ u\ v \iff v = \mathtt{Conv}\ \mathtt{None}\ [\,]$$

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" []

LIST A $(h \# t)$ $v \iff$

$\exists v_1\ v_2.\ v$ = ConvS "list" "::" $[v_1;\ v_2] \wedge A\ h\ v_1 \wedge$ LIST $A\ t\ v_2$

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

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

v is a closure implementing the function f

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

Definition

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

Shallow to deep map

Question

What is the deep counterpart of this term?

`map`

Answer

Some closure, satisfying this refinement invariant:

$((A \rightarrow B) \rightarrow \texttt{LIST}\ A \rightarrow \texttt{LIST}\ B)\ \texttt{map}$

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

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?

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.

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

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*.

Example

$\vdash$ `Cert` *env* (`ConS "::" [Con None []; ConS "nil" []]`)
(LIST UNIT [()])

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:

$$\vdash \texttt{EnvContains}\ [\texttt{map_dec}]\ \mathit{env} \Rightarrow$$
$$\quad \texttt{Cert}\ \mathit{env}\ (\texttt{VarS "map"})\ (((a \to b) \to \texttt{LIST}\ a \to \texttt{LIST}\ b)\ \texttt{map})$$

Now, how can we use this certificate theorem?

Deep results for shallow proofs

Remember this?

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

The deep version

$\vdash$ `EnvContains [map_dec; length_dec]` env $\Rightarrow$
  `LIST` a l v_1 $\wedge$ `lookup_var "l"` env = `Some` v_1 $\Rightarrow$
   ($a \to b$) f v_2 $\wedge$ `lookup_var "f"` env = `Some` v_2 $\Rightarrow$
    `Cert` env
     `(Call (VarS "length")`
      `(Call (Call (VarS "map") (VarS "f")) (VarS "l")))`
     `(NAT (length (map` f l`)))` $\wedge$
    `Cert` env `(Call (VarS "length") (VarS "l"))`
     `(NAT (length` 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)

$\vdash$ `Cert` env e_1 (($A \rightarrow B$) 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` ($b_1 \Rightarrow b_2$))

$\vdash$ `lookup_var` n env = `Some` v $\Rightarrow$
  A x 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.

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

Another function

How about transforming the syntax? e.g.,

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

(You saw something like this in Assignment 2)

Anatomy of a compiler

Compiler definition

Would something like this work

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

?

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

CakeML compiler backend:

CakeML AST → modLang → conLang → decLang → exhLang → patLang → closLang → BVL → BVI → BVP → wordLang → stackLang → labLang → ASM

ASM → ARM-64
ASM → ARM-32
ASM → x86-64
ASM → MIPS

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`?

Answer

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

In more detail

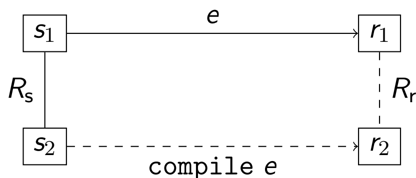
$$\vdash \mathtt{evaluate}_{\mathrm{exh}}\ ck\ env_{\mathrm{exh}}\ s_{\mathrm{exh}}\ exp_{\mathrm{exh}}\ (s'_{\mathrm{exh}}, r_{\mathrm{exh}}) \Rightarrow$$
$$r_{\mathrm{exh}} \neq \mathtt{Rerr}\ (\mathtt{Rabort}\ \mathtt{Rtype_error}) \Rightarrow$$
$$\mathtt{sem_rel}\ (env_{\mathrm{exh}}, s_{\mathrm{exh}})\ (env_{\mathrm{pat}}, s_{\mathrm{pat}}) \Rightarrow$$
$$\exists s'_{\mathrm{pat}}\ r_{\mathrm{pat}}.$$
$$\mathtt{evaluate}_{\mathrm{pat}}\ ck\ env_{\mathrm{pat}}\ s_{\mathrm{pat}}\ (\mathtt{compile}\ (\mathrm{bvs}\ env_{\mathrm{exh}})\ exp_{\mathrm{exh}})$$
$$(s'_{\mathrm{pat}}, r_{\mathrm{pat}}) \wedge \mathtt{s_rel}\ s'_{\mathrm{exh}}\ s'_{\mathrm{pat}} \wedge \mathtt{res_rel}\ r_{\mathrm{exh}}\ r_{\mathrm{pat}}$$

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?

- 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 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:
$\forall k.$
$\exists s'.$
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,
- 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. 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.
- 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 (under development) 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.).
- 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*:
- 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

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?

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::[]);
```

Example

Input term:

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

Produces:

⊢ compile_ast $\mathtt{cs}_0$ [Tdec map_dec; Tdec map_suc_dec] =
 Success $\mathtt{cs}_1$ map_suc_code

for some $\mathtt{cs}_1$ and map_suc_code.

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`?

Some declaration

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

satisfying...

Certificate theorem

⊢ `EnvContains compiler_decs` *env* ⇒
    `Cert` *env* `(VarS "compile")` `((CS` → `STRING` → `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` $\texttt{cs}_0$ `[Struct "CakeML" compiler_decs]`
- Use `EVAL` to solve it. (Takes about half an hour.)
- 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:
   $\vdash \forall\, cs\ prog\,.$ `compile_ast` *cs prog* `implements` *prog*
2. Certificate theorem:
   $\vdash$ `EnvContains compiler_decs` *env* $\Rightarrow$
   `Cert` *env* (`VarS "compile"`) ((`CS` $\rightarrow$ `STRING` $\rightarrow$ `CR`) `compile`)
3. Bootstrapping theorem:
   $\vdash$ `compile_ast` $\texttt{cs}_0$ [`Struct "CakeML" compiler_decs`] =
   `Success` $\texttt{cs}_2$ `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.

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?

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

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

CakeML

DATA 61 CSIRO

People involved

Currently: Anthony Fox (Cambridge), Ramana Kumar (Data61), Magnus Myreen (Chalmers), Michael Norrish (Data61), Scott Owens (Kent), Yong Kiam Tan (A*STAR)

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`