The CakeML Project

A functional programming language
A verified compiler
Verified applications
Theorem proving technology
Formal Verification: Two Extremes

<table>
<thead>
<tr>
<th>Full functional correctness</th>
<th>Bug finding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rich security properties</td>
<td>Simple security properties</td>
</tr>
<tr>
<td>Interactive</td>
<td>Automatic</td>
</tr>
<tr>
<td>1,000s of lines</td>
<td>1,000,000 of lines</td>
</tr>
<tr>
<td>Not necessarily mainstream</td>
<td>C, Java &amp; ASM</td>
</tr>
</tbody>
</table>
The CakeML Language

Design: “The CakeML language is designed to be both easy to program in and easy to reason about formally”

Reality: CakeML, the language ≅ Standard ML without functors

i.e. with almost everything else:
✓ higher-order functions
✓ mutual recursion and polymorphism
✓ datatypes and (nested) pattern matching
✓ references and (user-defined) exceptions
✓ modules, signatures, abstract types
✓ polymorphic/byte arrays/vectors, FFI calls
? right-to-left evaluation, prefers curried style
The CakeML Language

was originally

Design: “The CakeML language is designed to be both easy to program in and easy to reason about formally”

Reality: CakeML, the language $\cong$ Standard ML without functors

i.e. with almost everything else:
✓ higher-order functions
✓ mutual recursion and polymorphism
✓ datatypes and (nested) pattern matching
✓ references and (user-defined) exceptions
✓ modules, signatures, abstract types
✓ polymorphic/byte arrays/vectors, FFI calls
? right-to-left evaluation, prefers curried style
The CakeML Language

was originally

Design: “The CakeML language is designed to be both easy to program in and easy to reason about formally”

It is still clean, but not always simple.

Reality: CakeML, the language ≈ Standard ML without functors

i.e. with almost everything else:

✓ higher-order functions
✓ mutual recursion and polymorphism
✓ datatypes and (nested) pattern matching
✓ references and (user-defined) exceptions
✓ modules, signatures, abstract types
✓ polymorphic/byte arrays/vectors, FFI calls
?
right-to-left evaluation, prefers curried style
The CakeML Language

was originally

Design: “The CakeML language is designed to program in and easy to reason about formally.

It is still clean, but not always simple.

Reality: CakeML, the language \(\cong\) Standard ML without functors

i.e. with almost everything else:

✓ higher-order functions
✓ mutual recursion and polymorphism
✓ datatypes and (nested) pattern matching
✓ references and (user-defined) exceptions
✓ modules, signatures, abstract types
✓ polymorphic/byte arrays/vectors, FFI calls

? right-to-left evaluation, prefers curried style

Significant effort wrt the semantics
The CakeML Compiler

 Lex/parsing/infer types

 Compile and optimise (12 ILs)

 Select instructions, resolve labels

 https://cakeml.org
The CakeML Compiler

Lex/parsing/infer types

Compile and optimise (12 ILs)

Select instructions, resolve labels

string -> concrete syntax

untyped AST

generic assembly

All implemented in HOL
(a pure, total functional PL)

bytes

x86/ARM/MIPS/RISC-V

All languages communicate with the external world via a byte-array-based foreign-function interface.

https://cakeml.org
Performance numbers *before* bignums were added (Nov 2016)

Which colour is what ML implementation?
Performance numbers *before* bignums were added (Nov 2016)

CakeML is sometimes faster than OCaml
Performance numbers *before* bignums were added (Nov 2016)

CakeML is sometimes faster than OCaml
Performance numbers after bignums were added (Feb 2017)
Dimensions of Compiler Verification
Dimensions of Compiler Verification

source code
abstract syntax
intermediate language
bytecode
machine code

how far compiler goes
Dimensions of Compiler Verification

- source code
- abstract syntax
- intermediate language
- bytecode
- machine code

how far compiler goes

compiler algorithm
implementation in ML
implementation in machine code
interface with the underlying system

the thing that is verified
Dimensions of Compiler Verification

- source code
- abstract syntax
- intermediate language
- bytecode
- machine code

how far compiler goes

Our verification will cover the full spectrum of both dimensions.

compiler algorithm
implementation in ML
implementation in machine code
interface with the underlying system

the thing that is verified
Ecosystem

Proof-producing synthesis

HOL functions → CakeML AST

Verified compiler backend

CakeML AST → CakeML AST → machine code

Verified parsing

ASCII → CakeML AST

Verified type inference

CakeML AST → typeable yes/no

Proof-producing verification-condition generation

CakeML AST → Characteristic Formula

i.e. a ‘verification condition’

Also: x86 implementation with read-eval-print-loop
Unix-style utilities

Stateful

Input stream

Processing

Output stream

Stateful

Pure or stateful
Applications

Unix-style utilities

• cat
• sort
• grep
• diff+patch
• bootstrapped compiler

Standard library for CakeML:

• module: char I/O stdin/stdout
• module: reading of files
• module: reading command-line arguments
• standard modules: lists, vectors, arrays, strings, characters, etc.
Programming in HOL

\[(\text{length } []) = 0\] ∧
\[(\text{length } (h::t) = 1 + \text{length } t)\]

\[\forall x \ y. \text{length } (x++y) = \text{length } x + \text{length } y\]

*Induct_on `x` THEN SRW_TAC [] []*

*EVAL (`length [1;2;3]`) = (\[\forall x \ y. \text{length } (x++y) = \text{length } x + \text{length } y\]) = \[\forall x \ y. \text{length } (x++y) = \text{length } x + \text{length } y\] = 3*

*Secure evaluation*

*Or Isabelle/HOL or Coq or …*

*Theorem*

*Proof*
Programming in HOL

\((\text{compile conf std\_in} = \ldots)\)

\(\forall \text{conf p. good\_init init init'} \text{ conf} \Rightarrow \text{sem init p} = \text{sem\_x86 init'} \text{ (compile conf p)}\)

\[[13 \text{ IL semantics, > 100,000 lop}]\]

\(\text{EVAL (`\text{compile \ldots \text{"val x = \ldots"\"})} = \) \\
(\text{compile \ldots \text{"val x = \ldots\"} = 0x48,0x39 \ldots} \)

\(-\text{Secure evaluation}\)}}
Programming in HOL

\((\text{compile\ conf\ std\_in} = \ldots)\)

\(\vdash \forall \text{conf\ p.\ good\_init\ init\ init'}\ \text{conf} \Rightarrow\sem\ init\ p = \sem_{x86}\ init'\ (\text{compile\ conf\ p})\)

\([13\ IL\ semantics, > 100,000\ lop]\)

\(\text{EVAL \{`\text{compile ... “val x = …“}’\}} = (\vdash\text{compile ... “val x = …“} = 0x48,0x39 \ldots)\)

~15,000 loc

Theorem

Proof

Slow: 15 minutes to 2 days

Secure evaluation
fun length [] = 0
  | length (h::t) = 1 + length t

⊢ length (x++y) ↓ v ⇒
  ∃v_1 v_2. length x ↓ v_1 ∧ length y ↓ v_2 ∧ v = v_1 + v_2

Myreen & Owens, ICFP ’12
CFML: Characteristic Formulae for ML
Arthur Charguéraud

“CFML can be used to verify Caml programs using the Coq proof assistant.”

Arthur’s PhD topic

We want CF for CakeML
Arthur’s student Armaël Guéneau → Chalmers visit
What is CF?

Verification conditions for ML programs.

For standard Hoare logic:

It suffices to show: \( P \Rightarrow wp(c, Q) \)

To prove: \( \{ P \} c \{ Q \} \)

For Caml programs:

It suffices to show: \( cf \ e \ H \ Q \)

To prove: \( \{ H \} e \{ Q \} \)

\( cf \) is a function similar to \( wp \).

It produces a verification condition (higher-order sep. logic).
Weaknesses of Arthur’s CFML for Caml

CFML: The cf function is defined in OCaml (i.e. outside of Coq).

CFML: Soundness proved mostly outside of Coq (pen and paper).
   CFML: Soundness proved w.r.t. idealise semantics of OCaml.

CFML: does not support I/O or exceptions.
Aims with CakeML CF

CFML: The cf function is defined in OCaml (i.e. outside of Coq)

CFML: Soundness proved mostly outside of Coq (pen and paper).
CFML: Soundness proved w.r.t. idealise semantics of OCaml.

CFML: does not support I/O or exceptions.
Aims with CakeML CF

CFML: The cf function is defined in OCaml (i.e. outside of Coq)

CakeML CF: defines cf as a function in the logic

CFML: Soundness proved mostly outside of Coq (pen and paper).
CFML: Soundness proved w.r.t. idealise semantics of OCaml.

CakeML CF: soundness proved in the logic w.r.t. CakeML semantics

CFML: does not support I/O or exceptions.

CakeML CF: supports all CakeML language features
(incl. I/O and exceptions)

Weakness of CakeML CF: clunkier values (deep embedding), tactics etc.
Soundness thm

Moreover, as detailed in Section 3, we could extend the approach to handle new language features that are not supported by CFML.

The soundness theorem, which justifies proving properties about a characteristic formula to give equivalent properties about the program itself, is stated as follows. If the characteristic formula for the deeply embedded expression $e$ (and environment $env$) holds for some shallowly embedded pre-condition $H$ and shallowly embedded post-condition $Q$, i.e., $\text{cf}(e, env, H, Q)$, then, starting from a state satisfying $H$, $e$ is guaranteed to successfully evaluate in CakeML's functional big-step semantics [20], and reach a new state $st_0$ and value $v$ satisfying $Q$.

This mechanised proof eliminates the last bits of paper proof that need to be trusted in CFML. Section 2 details the main steps leading to the proof.

We extend the CF framework introduced in CFML to handle two new language features: exceptions, and I/O through CakeML's foreign-function interface (FFI). These extensions are proved sound with respect to the CakeML semantics, and neatly make our framework able to handle all features of the CakeML programming language.

The extension which adds support for I/O is implemented by carefully modifying the state to set function, shown in the soundness theorem above. We modified the state to set function so that it makes visible the state of the FFI in the pre- and post-conditions. There were numerous tricky details to get right in the definition of state to set because the design goal was to make I/O reasoning local in the style of separation logic. Our support for I/O is local in that the proof for a piece of code which only uses, say, the print-to-stdout FFI ports does not impose any assumptions on the behaviour, state, or even existence of other FFI ports, e.g., ports for reading-from-stdin. In the spirit of separation logic, our framework allows combining different assertions about the FFI using CF's equivalent to the separation logic frame rule. Section 3 provides details on how we modified state to set to make the FFI available in CF proofs.

Support for exceptions is implemented by making the post-conditions differentiate whether the result is a normal return with a value or a value raised as an exception. The new framework is able to reason about exception handling.

(CakeML's module system is also supported in our CF framework, but supporting modules did not require extending the original ideas of CFML.)

\[
\begin{align*}
\vdash \text{cf } e \text{ env } H & \quad Q \quad \Rightarrow \\
\forall st. \\
H (\text{state}\_\text{to}\_\text{set } st) & \Rightarrow \\
\exists st' \ h_f \ h_g \ v \ ck. \\
\text{evaluate } (st \text{ with clock } := ck) \text{ env } [e] & = (st', \text{Rval } [v]) \land \\
\text{split } (\text{state}\_\text{to}\_\text{set } st') & (h_f, h_g) \land Q \ v \ h_f
\end{align*}
\]

\text{(Version before support for exceptions was added.)}
I/O semantics in CakeML (FFI)

The CakeML state carries an oracle (with a type variable):

\[
\theta \text{ ffi_state } = \\
\langle | \text{ oracle}: (\text{string } \to \theta \to \text{ byte list } \to \theta \text{ oracle_result}) ; \\
\text{ ffi_state} : \theta ; \\
\text{ final_event} : (\text{final_event option}) ; \\
\text{ io_events} : (\text{io_event list}) | > \\
\text{ final_event } = \text{ Final_event string (byte list) ffi_outcome} \\
\text{ ffi_outcome } = \text{ FFI_diverged } | \text{ FFI_failed} \\
\text{ io_event } = \text{ IO_event string ((byte } \times \text{ byte) list)} \\
\theta \text{ oracle_result } = \text{ Oracle_return } \theta (\text{ byte list}) | \text{ Oracle_diverge } | \text{ Oracle_fail}
\]
I/O semantics in CakeML (FFI)

The CakeML state carries an oracle (with a type variable):

\[
\theta \text{ ffi_state} = \\
\langle \text{oracle} : (\text{string} \to \theta \to \text{byte list} \to \theta \text{ oracle_result}) ; \\
\text{ffi_state} : \theta ; \\
\text{final_event} : (\text{final_event} \text{ option}) ; \\
\text{io_events} : (\text{io_event list}) \rangle \\
\text{final_event} = \text{Final_event} \text{ string (byte list) ffi_outcome} \\
\text{ffi_outcome} = \text{FFI_diverged} | \text{FFI_failed} \\
\text{io_event} = \text{IO_event} \text{ string ((byte \times byte) list)} \\
\theta \text{ oracle_result} = \text{Oracle_return} \theta (\text{byte list}) | \text{Oracle_diverge} | \text{Oracle_fail}
\]
I/O continued

Reminder about the soundness theorem:

\[ \vdash \text{cf } e \text{ env } H \text{ Q } \Rightarrow \
\forall \text{st.} \\
H (\text{state_to_set } \text{st}) \Rightarrow \\
\exists \text{st'} \text{ hf hg v ck.} \\
\text{evaluate (st with clock } := \text{ ck) env [e] } = (\text{st'}, \text{Rval } [v]) \land \\
\text{split (state_to_set } \text{st'}) (\text{hf}, \text{hg}) \land \text{Q v hf} \]

Make state_to_set include a partitioned image of the FFI state so that we can write:

\[ (\text{IO } s_1 \text{ u1 } [n] \ast \text{IO } s_2 \text{ u2 } ns \ast \ldots) (\text{state_to_set } pp \text{ st}) \]

where:

\[ \text{IO } st \text{ u } ns = (\lambda s. \exists ts. s = \{ \text{FFI_part } st \text{ u } ns \text{ ts } \}) \]
Spec for part of cat

\[\vdash \text{FILENAME } fnm \ fnv \land \text{numOpenFDs } fs \ < 255 \Rightarrow\]
\[\{\text{CATFS } fs \ast \text{STDOUT } out\}\]
\[\text{cat1_v} \cdot [fnv]\]
\[\{\text{POST}\]
\[(\lambda u.\]
\[\exists \ content.\]
\[\langle \text{UNIT } () u \rangle \ast \langle \text{alist_lookup } fs.files fnm = \text{Some } content \rangle \ast \]
\[\text{CATFS } fs \ast \text{STDOUT } (out \ @ content))\]
\[(\lambda e.\]
\[\langle \text{BadFileName_exn } e \rangle \ast \langle \neg \text{inFS_fname } fnm fs \rangle \ast \text{CATFS } fs \ast \]
\[\text{STDOUT } out)\}\]
Bootstrapping

function in logic (compiler)

- parsing
- type inference
- compilation
Bootstrapping

- parsing
- type inference
- compilation

function in logic (compiler)

CakeML program (compiler-ML)

⊢ compiler-ML implements compiler

Proof-producing synthesis (ICFP'12)
Bootstrapping

function in logic (compiler)

CakeML program (compiler-ML)

⊢ compiler-ML implements compiler

⊢ compiler (compiler-ML) = compiler-x86

Proof-producing synthesis (ICFP’12)

by evaluation in the logic
Bootstrapping

function in logic (compiler)

Proof-producing synthesis (ICFP’12)

CakeML program (compiler-ML)

⊢ compiler-ML implements compiler

by evaluation in the logic

⊢ compiler (compiler-ML) = compiler-x86

by compiler correctness

⊢ ∀c. (compiler c) implements c
Bootstrapping

function in logic (compiler)

CakeML program (compiler-ML)

⊢ compiler-ML implements compiler

⊢ compiler (compiler-ML) = compiler-x86

⊢ ∀ c. (compiler c) implements c

Theorem: ⊢ compiler-x86 implements compiler

Proof-producing synthesis (ICFP’12)

parsing

type inference

compilation

by evaluation in the logic

by compiler correctness
fun main u =
  let
    val cl = Commandline.arguments ()
  in
    case compiler_x64 cl (read_all []) of
      (c, e) => (print_app_list c; print_err e)
  end

`cl ≠ [] ∧ EVERY validArg cl ∧ LENGTH (FLAT cl) + LENGTH cl ≤ 256 ⇒
app (p:'ffi ffi_proj) ^ (fetch_v "main" st)
  [Conv NONE []]
  (STDOUT out * STDERR err * (STDIN inp F * COMMANDLINE cl))
  (POSTv uv. &UNIT_TYPE () uv *
    STDOUT (out ++ (FLAT (MAP explode
      (append (FST(compiler_x64 (TL(MAP implode cl))
        inp))))))) *
  STDERR (err ++ explode (SND(compiler_x64 (TL(MAP implode cl)) inp)) *
    (STDIN "" T * COMMANDLINE cl))`,

24
fun main u = 
  let
    val cl = Commandline.arguments ()
in
  case compiler_x64 cl (read_all []) of
    (c, e) => (print_app_list c; p),
end

`cl ≠ [] ∧ EVERY validArg cl ∧ LENGTH (FLAT cl) + LENGTH cl ≤ 256 ⇒
app (p:'ffi ffi_proj) ^(fetch_v "main" st) [Conv NONE []]
(STDOUT out * STDERR err * (STDIN inp F * COMMANDLINE cl))
(POSTv uv. &UNIT_TYPE () uv *
  STDOUT (out ++ (FLAT (MAP explode
    (append (FST(compiler_x64 (TL(MAP implode cl))
      inp)))))) *
  STDERR (err ++ explode (SND(compiler_x64 (TL(MAP implode cl)) inp))) *
  (STDIN "" T * COMMANDLINE cl))`,

Good command line
To a compiler application

```plaintext
fun main u = 
  let
    val cl = Commandline.arguments ()
  in
    case compiler_x64 cl (read_all []) of
      (c, e) => (print_app_list c; print_err e)
  end

Unit arg.
```
fun main u = 
  let 
    val cl = Commandline.arguments () 
  in 
    case compiler_x64 cl (read_all []) of 
      (c, e) => (print_app_list c; print_err e) 
  end 

`cl ≠ [] ∧ EVERY validArg cl ∧ LENGTH (FLAT cl) + LENGTH app (p:'ffi ffi_proj) ^(fetch_v "main" st) 
            [Conv NONE []]
  (STDOUT out * STDERR err * (STDIN inp F * COMMANDLINE cl)) 
  (POSTv uv. &UNIT_TYPE () uv * 
    STDOUT (out ++ (FLAT (MAP explode 
                        (append (FST(compiler_x64 (TL(MAP implode cl)) 
                                   inp)))))))) * 
  STDERR (err ++ explode (SND(compiler_x64 (TL(MAP implode cl)) inp))) * 
           (STDIN "" T * COMMANDLINE cl))`,

Precond.
fun main u =
  let
    val cl = Commandline.arguments ()
  in
    case compiler_x64 cl (read_all []) of
      (c, e) => (print_app_list c; print_err e)
  end

`cl ≠ [] ∧ EVERY validArg cl ∧ |FLAT (TL(MAP implode cl)) | ≤ 256 ⇒
  app (p:'ffi ffi_proj) ^{(fetch_v "main" st)}
  [Conv NONE []]
  (STDOUT out * STDERR err * (STDIN inp F * COMMANDLINE cl))
  (POSTv uv. &UNIT_TYPE () uv *
    STDOUT (out ++ (FLAT (MAP explode
      (append (FST(compiler_x64 (TL(MAP implode cl))
        inp)))))) *
    STDERR (err ++ explode (SND(compiler_x64 (TL(MAP implode cl)) inp))) *
    (STDIN "" T * COMMANDLINE cl))`,

Unit res.
fun main u = 
  let 
    val cl = Commandline.arguments () 
  in 
    case compiler_x64 cl (read_all []) of 
      (c, e) => (print_app_list c; print_err e) 
  end 

`cl ≠ [] ∧ EVERY validArg cl ∧ LENGTH (FLAT cl) + LENGTH cl ≤ 256 ⇒ 
app (p:'ffi ffi_proj) ^(fetch_v "main" [] [Conv NONE []]) 
(STDOUT out * STDERR err * (STDIN inp F * COMMANDLINE cl)) 
(POSTv uv. &UNIT_TYPE () uv * 
  STDOUT (out ++ (FLAT (MAP explode 
    (append (FST(compiler_x64 (TL(MAP implode cl)) 
      inp))))) *) 
  STDERR (err ++ explode (SND(compiler_x64 (TL(MAP implode cl)) inp))))) * 
  (STDIN "" T * COMMANDLINE cl))`,

Output
To a compiler application

fun main u = 
  let
    val cl = CommandLine.arguments ()
  in
    case compiler_x64 cl (read_all []) of
      (c, e) => (print_app_list c; print_err e)
  end

`cl ≠ [] ∧ EVERY validArg cl ∧ LENGTH (FLAT cl) + LENGTH cl ≤ 256 ⇒
  app (p:'ffi ffi_proj) ^(fetch_v "main" st)
    [Conv NONE []]
  (STDOUT out * STDERR err * (STDOUT (out ++ (FLAT (MAP explode
    (append (FST(compiler_x64 (TL(MAP implode cl))
      inp)))))) * 
  STDERR (err ++ explode (SND(compiler_x64 (TL(MAP implode cl))
    inp)))) * 
  (STDIN "" T * COMMANDLINE cl))`,

Error msgs.
To a compiler application

fun main u =
  let
    val cl = Commandline.arguments ()
in
    case compiler_x64 cl (read_all []) of
      (c, e) => (print_app_list c; print_err e)
  end

`cl ≠ [] ∧ EVERY validArg cl ∧ LENGTH (FLAT cl) + LENGTH cl ≤ 256 ⇒
app (p:'ffi ffi_proj) ^(fetch_v "main" st)
  [Conv NONE []]
  (STDOUT out * STDERR err * (STDIN inp F * COMMANDLINE cl))
  (POSTv uv. &UNIT_TYPE () uv *
   STDOUT (out ++ (FLAT (MAP explode
     (append (FST(compiler_x64 (TL(MAP implode cl))
       inp)))))) *
   STDERR (err ++ explode (SND(compiler_x64 (TL(MAP implode cl)) inp))) *
   (STDIN "" T * COMMANDLINE cl)),

24
Compiler for an ML-like programming language
Mechanically verified in HOL-4
A tool to support the construction of verified systems