Certification of high-level and low-level programs, IHP, Paris, 2014

CakeML
A verified implementation of ML

Ramana Kumar
Magnus Myreen
Michael Norrish
Scott Owens
Background

From my PhD (2009):

**Verified Lisp interpreter**
in ARM, x86 and PowerPC machine code

Collaboration with Jared Davis (2011):

**Verified Lisp read-eval-print loop**
in 64-bit x86 machine code, with dynamic compilation

.plus verification of an ACL2-like theorem prover)

Can we do the same for ML?

**A verified implementation of ML**

.plus verification of a HOL-like theorem prover?)

Other HOL4 hackers also have relevant interests…
People involved

- Ramana Kumar (Uni. Cambridge)
  - operational semantics
  - verified compilation from CakeML to bytecode
  - verified type inferencer
  - verified parsing (syntax is compatible with SML)
  - verified x86 implementations
  - proof-producing code generation from HOL

- Michael Norrish (NICTA, ANU)

- Scott Owens (Uni. Kent)

- Magnus Myreen (Uni. Cambridge)
Overall aim

to make proof assistants into trustworthy and practical program development platforms

Trustworthy code extraction:

functions in HOL (shallow embedding) → proof-producing translation [ICFP’12, JFP’14] → CakeML program (deep embedding) → verified compilation of CakeML [POPL’14] → x86-64 machine code (deep embedding)
This talk

Part 1: verified implementation of CakeML

Part 2: current status, HOL light, future
POPL'14

CakeML: A Verified Implementation of ML

Ramana Kumar * 1  Magnus O. Myreen † 1  Michael Norrish 2  Scott Owens 3

1 Computer Laboratory, University of Cambridge, UK
2 Canberra Research Lab, NICTA, Australia ‡
3 School of Computing, University of Kent, UK

Abstract
We have developed and mechanically verified an ML system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop (REPL) in x86-64 machine code. Our correctness theorem ensures that this REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and we bootstrap the compiler to get a machine-code implementation of the compiler. Another consequence of bootstrapping is that we can include the compiler on equal footing with other CakeML programs. We then apply the compiler to itself, avoiding a tedious manual refinement proof relating the compilation algorithm for converting a program from a source string to a list of numbers representing machine code, and we synthesise a CakeML implementation of the compiler inside the logic; we bootstrap it. This leads to a list of contributions that are carefully designed to be simultaneously suitable for proving meta-theoretic language properties and for supporting a verified implementation. (Section 3) An extension of a proof-producing synthesis pathway [22] originally from logic to ML, now to machine code (via verified compilation). (Sections 4–6, 10)

1. Introduction
The last decade has seen a strong interest in verified compilation: and there have been significant, high-profile results, many based on the CompCert compiler for C [1, 14, 16, 29]. This interest is easy to justify: in the context of program verification, an unverified compiler forms a large and complex part of the trusted computing base. However, to our knowledge, none of the existing work on compilers for general-purpose languages has addressed all three dimensions: one, the compilation algorithm for converting a program from a source string to a list of high-level instructions; two, the compilation back-end that turns those instructions into a list of numbers representing machine code; and three, the correctness theorem that connects the two.
Dimensions of Compiler Verification

Our verification covers the full spectrum of both dimensions.
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 I/O or 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
Contributions of POPL’14 paper

**Artefacts**
- Specifications
- Verified Algorithms

**Proof techniques**
- Divergence Preservation
- Bootstrapping

**Proof development**
- light-weight approach to divergence preservation with big-step op. sem.
- main new technique: use verified compiler to produce verified implementation

where everything fits together.
Approach

Proof by refinement:

**Step 1:** specification of CakeML language
  - big-step and small-step operational semantics

**Step 2:** functional implementation in logic
  - read-eval-print-loop as verified function in logic

**Step 3:** production of verified x86-64 code
  - produced mostly by bootstrapping the compiler
Operational semantics

Big-step semantics:

- big-step evaluation relation
- environment semantics (cf. substitution sem.)
- produces TypeError for badly typed evaluations (e.g. 1+nil)
- stuck = divergence

Equivalent small-step semantics:

- used for type soundness proof and definition of divergence

Read-eval-print-loop semantics.

Semantics written in Lem, see Mulligan et al. [ICFP’14]
Functional implementation

Read-eval-print loop defined as rec. function in the logic:
**Specification:**

Context-free grammar (CFG) for significant subset of SML
Executable lexer.

**Implementation:**

Parsing-Expression-Grammar (PEG) Parser

- inductive evaluation relation
- executable interpreter for PEGs

**Correctness:**

Soundness and completeness

- induction on length of token list/parse tree and non-terminal rank
**Specification:**

Declarative type system.

**Implementation:**

Based on Milner’s Algorithm W

Purely functional (uses state-exception monad)

**Correctness:**

Proved sound w.r.t. declarative type system

Re-use of previous work on verified unification
Purpose:

Translates (typechecked) CakeML into CakeML Bytecode.

Implementation:

Translation via an intermediate language (IL).

- de Bruijn indices
- big-step operational semantics

CakeML to IL: makes language more uniform
IL to IL: removes pattern-matching, lightweight opt.
IL to Bytecode: closure conversion, data refinement, tail-call opt.
Semantics of bytecode execution

Instructions:

\[
\begin{align*}
bc_{\text{inst}} & ::= \text{Stack } bc_{\text{stack\_op}} | \text{PushExc} | \text{PopExc} \\
& \quad | \text{Return} | \text{CallPtr} | \text{Call } l o c \\
& \quad | \text{PushPtr } l o c | \text{Jump } l o c | \text{JumpIff } l o c \\
& \quad | \text{Ref} | \text{Deref} | \text{Update} | \text{Print} | \text{PrintC char} \\
& \quad | \text{Label } n | \text{Tick} | \text{Stop} \\
bc_{\text{stack\_op}} & ::= \text{Pop} | \text{Pops } n | \text{Shift } n \ n | \text{PushInt } \text{int} \\
& \quad | \text{Cons } n \ n | \text{El } n | \text{TagEq } n | \text{IsBlock } n \\
& \quad | \text{Load } n | \text{Store } n | \text{LoadRev } n \\
& \quad | \text{Equal} | \text{Less} | \text{Add} | \text{Sub} | \text{Mult} | \text{Div} | \text{Mod} \\
l o c & ::= \text{Lab } n | \text{Addr } n
\end{align*}
\]

Small-step semantics; values and state:

\[
\begin{align*}
bc_{\text{value}} & ::= \text{Number int} | \text{RefPtr } n | \text{Block } n \ bc_{\text{value}}^* \\
& \quad | \text{CodePtr } n | \text{StackPtr } n \\
bc_{\text{state}} & ::= \{ \text{stack} : bc_{\text{value}}^*; \text{refs} : n \mapsto bc_{\text{value}}; \\
& \quad \text{code} : bc_{\text{inst}}^*; \text{pc} : n; \text{handler} : n; \\
& \quad \text{output} : \text{string}; \text{names} : n \mapsto \text{string}; \\
& \quad \text{clock} : n^? \}
\end{align*}
\]
Semantics of bytecode execution

Sample rules:

\[
\begin{align*}
\text{fetch}(bs) &= \text{Stack} \ (\text{Cons} \ t \ n) \quad bs.\text{stack} = vs @ xs \quad |vs| = n \\
bs &\rightarrow (\text{bump} \ bs)\{\text{stack} = \text{Block} \ t \ (\text{rev} \ vs) :: xs\}
\end{align*}
\]

\[
\begin{align*}
\text{fetch}(bs) &= \text{Return} \quad bs.\text{stack} = x :: \text{CodePtr} \ ptr :: xs \\
bs &\rightarrow bs\{\text{stack} = x :: xs; \ pc = ptr\}
\end{align*}
\]

\[
\begin{align*}
\text{fetch}(bs) &= \text{CallPtr} \quad bs.\text{stack} = x :: \text{CodePtr} \ ptr :: xs \\
bs &\rightarrow bs\{\text{stack} = x :: \text{CodePtr} \ (\text{bump} \ bs).pc :: xs; \ pc = ptr\}
\end{align*}
\]
**Correctness:**

Proved in the direction of compilation.

Shape of correctness theorem:

\[(\exp, \text{env}) \Downarrow_{\text{ev}} \val \implies \text{“the code for } \exp \text{ is installed in } \bs \text{ etc.”} \implies \exists \bs'. \bs \rightarrow^* \bs' \land \text{“} \bs' \text{ contains } \val \text{”}\]

Bytecode semantics step relation
What about divergence?

We want: generated code diverges if and only if source code diverges
Idea: add logical clock

Big-step semantics:
- has an optional clock component
- clock ‘ticks’ decrements every time a function is applied
- once clock hits zero, execution stops with a TimeOut

Why do this?
- because now big-step semantics describes both terminating and non-terminating evaluations

∀exp env clock. ∃res. (exp, env, Some clock) ↓ev res

either: Result or TimeOut

produced by the semantics
Divergence

Evaluation diverges if
\[ \forall \text{clock}. \ (exp, env, \text{Some clock}) \Downarrow_{\text{ev}} \text{TimeOut} \]

for all clock values \quad \text{TimeOut happens}

Compiler correctness proved in conventional forward direction:

\[(exp, env) \Downarrow_{\text{ev}} \text{val} \quad \Rightarrow \quad \text{“the code for } exp \text{ is installed in } bs \text{ etc.”} \quad \Rightarrow \quad \exists bs'. \ bs \rightarrow^* bs' \land \text{“} bs' \text{ contains } val \text{”} \]

Bytecode has clock \quad \ldots \text{ that stays in sync with CakeML clock}

Theorem: bytecode diverges if and only if CakeML eval diverges
Step 3: production of verified x86-64 code
Verified x86-64 Implementation

- lexer
- parsing
- type inference
- compilation
- bytecode execution
- garbage collector
- bignum library

Verified x86-64 code generated using **bootstrapping** of the verified compiler.

JIT: translates Bytecode to machine code; jumps to generated machine code.

**hand-crafted** verified machine code based on previous work.

Real executable also has 30-line unverified C wrapper.
Translation into x86-64

Extract of definition:

Each bytecode instruction maps to some x86

\[
x64 \, i \, \text{Pop} = [0x48, \, 0x58]
\]

\[
x64\_\text{code} \, i \, (x::xs) = x64 \, i \, x \, @ \, x64\_\text{code} \, (i + \ldots) \, xs
\]

Correctness:

Each Bytecode instruction is correctly executed by generated x86-64 code.

\[
bs \rightarrow bs' \implies \text{temporal } \{(base, x64\_\text{code} \, 0 \, bs\_\text{code})\}
\]

\[
(\text{now } (bc\_\text{heap} \, bs \, (base, \, aux)) \Rightarrow \text{later } (\text{now } (bc\_\text{heap} \, bs' \, (base, \, aux)) \lor \text{now } (\text{out\_of\_memory\_error} \, aux)))
\]

heap invariant / memory abstraction
Bootstrapping the verified compiler
Production of verified x86-64

CakeML program (COMPILE) such that:

\[ \vdash \text{COMPILE implements } \text{compile} \]

by proof-producing synthesis [ICFP’12]

CakeML program (COMPILE) such that:

\[ \vdash \text{COMPILE implements } \text{compile} \]

Proof by evaluation inside the logic:

\[ \vdash \text{compile-to-x64 COMPILE} = \text{x64-code} \]

Compiler correctness theorem:

\[ \vdash \forall \text{prog. compile-to-x64 prog implements prog} \]

Combination of theorems:

\[ \vdash \text{x64-code implements compile} \]
Proof sketch

Theorem 22

Theorem 21 and temporal logic.

Top-level theorem

Top-level specification:

\[ \text{REPL}_s \; l \; \text{inp} \; \text{res} \]

result output: list of strings that ends in either Terminate or Diverge

input string

Correctness theorem:

temporal entire_machine_code_implementation

(now \((\text{init} \; \text{inp} \; \text{aux}) \Rightarrow \))

later \(((\exists l \; \text{res}. \; \text{repl_returns} \; (\text{out} \; \text{res}) \; \text{aux} \land \) \((\text{REPL}_s \; l \; \text{inp} \; \text{res} \land \text{terminates} \; \text{res})\))

\(\lor\)

\((\exists l \; \text{res}. \; \text{repl_diverged} \; (\text{out} \; \text{res}) \; \text{aux} \land \) \((\text{REPL}_s \; l \; \text{inp} \; \text{res} \land \neg \text{terminates} \; \text{res})\))

\(\lor\)

now \((\text{out_of_memory_error} \; \text{aux})\))
Numbers

Performance:
Slow: interpreted OCaml is 1x faster (... future work!)

Effort:
~70k lines of proof script in HOL4
< 5 man-years, but builds on a lot of previous work

Size:
875,812 instructions of verified x86-64 machine code

- implementation generates more instructions at runtime
- large due to bootstrapping
This talk

**Part 1:** verified implementation of CakeML

**Part 2:** current status, HOL light, future
Current status

Current compiler:

string → tokens → AST → IL → bytecode → x86

huge step

Bytecode simplified proofs of read-eval-print loop, but made optimisation impossible.
Future plans

**Refactored compiler:** split into more conventional compiler phases

- **string** → **tokens** → **AST** → **IL-1**
  - module compilation
  - closure compilation
  - pattern-match compilation
  - removal of memory abstraction
  - register allocation
  - ... as separate phases.

**Anthony Fox joins project and helps with final phases**
Verified examples on CakeML

Verification infrastructure:

• have: synthesis tool that maps HOL into CakeML [ICFP’12]
• future: integration with Arthur Charguéraud’s characteristic formulae technology [ICFP’10, ICFP’11]

for developing cool verified examples.
Big example: verified HOL light

**ML** was originally developed to host theorem provers.

Aim: verified HOL theorem prover.

We have:
- syntax, semantics and soundness of HOL (stateful, stateless)
- verified implementation of the HOL light kernel in CakeML (produced through synthesis)

Still to do:
- soundness of kernel $\Rightarrow$ soundness of entire HOL light
- run HOL light standard library on top of CakeML

Freek Wiedijk is translating HOL light sources to CakeML
Summary

Contributions so far:

First(?) **bootstrap**ing of a formally **verified compiler**. **New** lightweight method for **divergence preservation**.

Current work:

Formally **verified** implementation of **HOL light**. Verified **I/O** (foreign-function interface). **seL4**. Compiler improvements (new ILs, opt, targets).

Long-term aim:

An **ecosystem** of tools and proofs around CakeML lang.

Questions? Suggestions?