Improving the verified compiler

Ramana Kumar

Magnus Myreen

Michael Norrish

Scott Owens
People involved

We are ‘verification people’.

We come with questions! .. and hope for discussion.

verified compilation from CakeML to bytecode

verified parsing (syntax is compatible with SML)

verified x86 implementations

proof-producing code generation from HOL

operational semantics

Magnus Myreen (Chalmers)

Michael Norrish (NICTA, ANU)

Ramana Kumar (Uni. Cambridge)

Scott Owens (Uni. Kent)

Magnus Myreen (Chalmers)

Michael Norrish (NICTA, ANU)

Ramana Kumar (Uni. Cambridge)
Why ML for verification?

ML is a nice clean high-level language.

The semantics is clear and easy to reason about… informally, at least.

Can we make formal reasoning about ML easy too?
Purpose of CakeML

1) carve out a subset of (mostly) StandardML that is easy to reason about formally

2) build a formally verified implementation that can be used: efficient, with good libraries, etc.

3) demonstrate how verified code can be produced *easily* using CakeML as a tool
First application

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 compiler for CakeML

**Part 2:** in-progress improvements, discussion
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 CakeML prints only those results permitted by its semantics. The theorem handles both incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping.

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 aspects of a compiler along the full scope of both of these dimensions: one, the compilation algorithm for converting a program from a source string to a list of numbers representing machine code, and two, the execution of that algorithm as implemented in machine code.

Part 1: verified compiler for CakeML
Dimensions of Compiler Verification

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

how far compiler goes

Our verification covers the full spectrum of both dimensions.

- compiler algorithm
- implementation in ML
- implementation in machine code
- interactive call in read-eval-print loop runtime

the thing that is verified
The CakeML language is designed to be both easy to program in and easy to reason about formally.

- ✓ higher-order functions
- ✓ mutual recursion and polymorphism
- ✓ datatypes and (nested) pattern matching
- ✓ references and (user-defined) exceptions
- ✓ modules, signatures, abstract types

CakeML, the language = Standard ML without I/O or functors

i.e. with almost everything else:
- ✓ arrays, vectors
- ✓ byte arrays, bytes (limited support)
- ✓ strings (limited support), no chars
- ✓ type abbreviations

Update! New since POPL’14:
- ✓ arrays, vectors
- ✓ byte arrays, bytes (limited support)
- ✓ strings (limited support), no chars
- ✓ type abbreviations

Reality: CakeML, the language = Standard ML without I/O or functors

Not always simple.
Contributions of POPL’14 paper

**Artefacts**
- Specifications
- Verified Algorithms

**Proof techniques**
- Divergence Preservation
- Bootstrapping

- **Proof development** where everything fits together.

**main new technique**: use verified compiler to produce verified implementation.
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.
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 x86-64 machine code

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

Part 1: verified compiler for CakeML

Part 2: in-progress improvements, discussion
Status as of POPL’14

Compiler phases:

- string → tokens → AST → IL → bytecode → x86

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

Almost no optimisations…

Because: it made the invariants supporting the compiler proof uniform. doesn't mean verification of a better compiler is too difficult, just that it takes time, so we want to have a good design first
On-going improvements

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.

IL-N → ASM

Anthony Fox joins project and helps with final phases

ARM
x86-64
MIPS-64
asm.js
Compiler improvements since POPL’14

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

- string → tokens → AST → IL-1
- module compilation
- declaration compilation
- pattern-match compilation (nested ifs)
- special treatment of known calls
- ... → IL-2
- ... → register allocation
- inlining, allocation opt.
- special treatment of known calls
- How and where should closures be compiled?
- Should we do lambda-lifting?
- ARM
- x86-64
Some Questions

ML experts, where do optimisations yield most rewards?

What optimisations will be most valuable?

Verifying the correctness of optimisations is expensive…

Verification of optimisations in language with lambda requires complex state-relation for the correctness proof.

Low-level optimisations requires interfacing with the invariants of the copying GC…

Anthony Fox joins project and helps with final phases
1) carve out a subset of (mostly) StandardML that is easy to reason about formally

2) build a formally verified implementation that can be used: efficient, with good libraries, etc.

3) demonstrate how verified code can be produced easily using CakeML as a tool

what should these be?

should we follow SML?

suggestions for cool large(ish) examples?
Towards interesting examples

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
A foreign-function interface (FFI) will be added to CakeML.

**Aim:** use CakeML to easily develop verified clients for seL4, NICTA’s formally verified OS

**FFI:**
- would have a simple byte-array interface
- formally modelled as an oracle in the semantics
- can be instated to fit with seL4’s fast IPC mechanism
  (IPC = inter-process communication)

**Research visit planned for Dec-Jan 2014/15**
Our Questions

ML experts, where do you see verification being valuable for ML?

Interesting examples?

Where should we try to optimise most?
Summary

**Contributions so far:**
First(?) **bootstrapping** 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**?