A New Verified Compiler Backend for CakeML

Main contributors to date: Anthony Fox, Ramana Kumar, Magnus Myreen, Michael Norrish, Scott Owens, Yong Kiam Tan
What?

1. A *programming language* in the style of Standard ML and OCaml.
2. An *ecosystem* of proofs and verification tools
3. A *verified, end-to-end development*
Verified compilation...
State of the art

CompCert

Compiles C source code to assembly

Good performance numbers

Ecosystem: **Verified Software Toolchain - Princeton University**

Leroy et al. Source: http://compcert.inria.fr/
Verified compilation...  
...for functional languages?

Answer: Many, but all are ‘toy’.

Attempt: CakeML first ‘realistic’ verified ML compiler (plus ecosystem).
The CakeML language

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
✓ arbitrary-precision integers
✓ modules, signatures, abstract types
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
✓ arbitrary-precision integers
✓ modules, signatures, abstract types

Update! New since POPL’14:
✓ foreign-function interface
✓ mutable arrays, byte arrays, bytes
✓ vectors strings, chars
✓ type abbreviations
Ecosystem

Proof-producing synthesis
- HOL functions → CakeML AST → CakeML AST → machine code

Verified compiler backend
- CakeML AST

Verified parsing
- ASCII → CakeML AST

Verified type inference
- CakeML AST → typeable yes/no

Soon: Proof-producing verification-condition generation
- CakeML AST → Characteristic Formula i.e. a ‘verification condition’

Also: x86 implementation with read-eval-print-loop
This talk: **Compiler verification**

**user expectations**

**gap**

observational behaviour of source code

proved connection

modelled behaviour of generated machine code

**gap**

real behaviour of hardware

**Verified compiler backend**

CakeML AST → machine code

The entire development is in the HOL4 theorem prover.
The CakeML compiler

Version 1 & 2

Verified compiler backend

CakeML AST \rightarrow machine code
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 a breadth of topics including lexing, parsing, type checking, incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping.

Our contributions are twofold. The first is simply in building a system that is end-to-end verified, demonstrating that each piece of such a verification effort can in practice be composed and verified, without any over-simplifying assumptions. The second is developing novel approaches to some of the more challenging aspects of the verification effort: we apply the verified compiler to itself to produce a verified machine-code implementation of the compiler. Additionally, our compiler proof handles diverging input programs with a lightweight approach based on logical timeout exceptions. The entire development was carried out in the HOL4 theorem prover.

Categories and Subject Descriptors

D.2.4 [Software/Program Verification]: Correctness proofs, Formal and Verifying and Reasoning about Programs—Mechanical verification, Specification techniques, Invariants

Keywords

machine code verification; read-eval-print loop; verified parsing; verified type checking; verified garbage collection.

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 verified compilers for general-purpose languages has addressed all aspects of a compiler along two 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.

Our purpose in this paper is to explain how we have verified a compiler along the full scope of both of these dimensions for a practical, general-purpose programming language. Our language is called CakeML, and it is a strongly typed, impure, strict functional language. By verified, we mean...
Dimensions of Compiler Verification

how far compiler goes

First verification to cover the full spectrum of both dimensions.

the thing that is verified
Intuition for Bootstrapping

**Proof-producing synthesis**

- HOL functions
- CakeML AST
- CakeML AST
- machine code

**Verified compiler backend**

**Verified parsing**

- ASCII
- CakeML AST
- CakeML AST
- typeable yes/no

**Verified type inference**

- CakeML AST
- CakeML AST
- CakeML AST
- machine code
Intuition for Bootstrapping

verified x86 implementation of parsing, type inference, and compilation
Version 1 as in 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 possible…
Poor design.
Version 2

Goals:

*Design compatible with optimisations.*

*Acceptable performance.*

*Strategy:* take inspiration from OCaml compiler (for some parts).
Result:

12 intermediate languages (ILs) and many within-IL optimisations

each IL at the right level of abstraction

for the benefit of proofs and compiler implementation
Values used by the semantics

Values
- abstract values incl. closures and ref pointers

Languages
- source syntax
- source AST
- no modules
- no cons. names
- no declarations
- full pat. match
- no pat. match
- last language with closures (has multi-arg. closures)
- func. lang. without closures

Compiler transformations
- Parse concrete syntax
- Infer types, exit if fail
- Eliminate modules
- Replace constructor names with numbers
- Reduce declarations to exps; introduce global vars
- Make patterns exhaustive
- Compile pattern matches to nested Ifs and Lets
- Rephrase language
- Fuse function calls/apps into multi-arg calls/apps
- Eliminate dead code
- Prepare for closure conv.
- Perform closure conv.
- Fold constants
- Shrink Lets
- Compile global vars into a dynamically resized array

Parser and type inferencer as before

Early phases reduce the number of language features

Language with multi-argument closures

Abstract imperative language

Languages
- imperative language with array-like stack and optional GC
- imperative language with machine words, memory and a GC primitive

Compiler transformations
- Select target instructions
- Perform SSA-like renaming
- Force two-reg code (if req.)
- Allocate register names
- Concretise stack
- Implement GC primitive
- Turn stack access into memory accesses
- Rename register to match arch registers/conventions
- Flatten code
- labelled assembly lang.
- Delete no-ops (Tick, Skip)
- Encode program as concrete machine code

Values used by the semantics
- abstract values incl. code pointers and refs
- machine words and code labels

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

- Simple first-order functional language
- Imperative language
- Machine-like types

Compiler transformations

- Abstract values incl. closures
- Abstract values incl. code pointers and refs
- Code labels

Values

- Abstract values incl. closures and ref pointers
- Abstract values incl. code pointers and refs
- Machine words and code labels

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

- 32-bit words
- 64-bit words
- ARMv6
- ARMv8
- x86-64
- MIPS-64
- RISC-V

- Combine adjacent memory allocations
- Reduce caller-saved vars
- Select target instructions
- Perform SSA-like renaming
- Force two-reg code (if req.)
- Allocate register names
- Concretise stack
- Implement GC primitive
- Turn stack access into memory accesses
- Rename register to match arch registers/conventions
- Flatten code
- Labelled assembly lang.
- Delete no-ops (Tick, Skip)
- Encode program as concrete machine code
- ARMv6
- ARMv8
- x86-64
- MIPS-64
- RISC-V

Compile global vars into a dynamically resized array

- Fusing function calls/apps into multi-arg calls/apps
- Eliminate dead code
- Prepare for closure conv.
- Perform closure conv.
- Fold constants
- Shrink Lets
- Compile global vars into a dynamically resized array
- Switch to imperative style
- Reduce caller-saved vars
- Combine adjacent memory allocations
- Remove data abstraction
- Select target instructions
- Perform SSA-like renaming
- Force two-reg code (if req.)
- Allocate register names
- Concretise stack
- Implement GC primitive
- Turn stack access into memory accesses
- Rename register to match arch registers/conventions
- Flatten code
- Labelled assembly lang.
- Delete no-ops (Tick, Skip)
- Encode program as concrete machine code
- ARMv6
- ARMv8
- x86-64
- MIPS-64
- RISC-V
All languages communicate with the external world via a byte-array-based foreign-function interface.
Some details

Closure representation:

**Closures are values with a code pointer:**
Block closure_tag
([CodePtr ptr; Number arg_count] @ free_var_vals)

**For mutually recursive closures:**
Block closure_tag
[CodePtr ptr; Number arg_count; RefPtr ref_ptr]

expected number of arguments
(multi-argument closures)
More details

Configurable data representation

Example pointer value:

0...00110011101 00 01 010 1

- address value
- padding
- length
- tag
- marker

These can be left out

Speeds up pattern matching, if present
Even more details

Stack contains information about live vars for the GC

Details of one stack frame:
Semantics & Proofs
Semantics

Each intermediate language has a formal semantics.

We define these using a *functional big-step style* (ESOP’16) where the semantics is an evaluation function in logic.

*Extract of abstract first-order lang:*

evaluate ([Var \( n \)], \( env, s \)) =
  if \( n < \text{len} \ env \) then (Rval \([\text{nth} \ n \ env], s\))
  else (Rerr (Rabort Rtype_error), s)

evaluate ([If \( x_1 \ x_2 \ x_3 \)], \( env, s \)) =
  case evaluate ([\( x_1 \)], \( env, s \)) of
    (Rval \( vs, s_1 \)) \Rightarrow
      if \( \text{Boolv true} = \text{hd} \ vs \) then evaluate ([\( x_2 \)], \( env, s_1 \))
      else if \( \text{Boolv false} = \text{hd} \ vs \) then evaluate ([\( x_3 \)], \( env, s_1 \))
      else (Rerr (Rabort Rtype_error), s_1)
  | (Rerr \( e, s_1 \)) \Rightarrow (Rerr \( e, s_1 \))

*Top-level observable FFI* semantics defined using evaluate.
Proof

Proof styles:

*Standard induction on evaluation function*

- proofs in direction of compilation
- no co-induction needed for divergence pres. (ESOP’16)

*Untyped logical relation (ind. on compile function)*

Each part of the compiler preserves obs. semantics:

\[ \text{compile } \text{config prog} = \text{new_prog} \wedge \text{syntactic_condition prog} \wedge \text{Fail } \notin \text{semantics } \text{ffi prog} \implies \text{semantics } \text{ffi new_prog} \subseteq \text{extend_with_resource_limit} \text{ (semantics } \text{ffi prog}) \]

---

Type-safe source implies this

due to out-of-memory error
Difficult cases

GC and register allocator interaction

Solution: we use a semantics that allows reordering of stack variables.
Size, Effort, Speed

**Compiler Size:** 6,000 lines of function definitions
(excludes target-specific instruction encoders & config)

**Proof Size:** 100,000 lines of HOL proof script

**Effort:** 6 people, 2 years, but not full time

**Speed:** next slide…

(Numbers up-to-date as of April 2016)
Simple Benchmarks

- **fib**
  - CakeML v1: Faster
  - CakeML v2: Slower

- **qsort**
  - CakeML v1: Slower
  - CakeML v2: Faster

- **queue**
  - CakeML v1: Slower
  - CakeML v2: Faster

- **btree**
  - CakeML v1: Faster
  - CakeML v2: Slower
Simple Benchmarks

Contributing factor: CakeML has arbitrary precision arithmetic
Simple Benchmarks

Why?

Version 1 *can compile big programs* (in-logic)

Version 2 *in-logic* evaluation is *too slow* for large examples

we are working to improve this

why not outside?
Immediate future work

fun map f [] = []
  | map f (x::xs) = f x :: map f xs;

val list_add1 = map (fn n => n + 1);

Any app of a known function needs to be optimised to a fast procedure call.

Inlining should produce a copy of map specialised for fn n => n+1
This talk: New compiler’s design compatible with optimisations

Big-picture: Ecosystem around a clean formalised ML language

Why? End-to-end verification, and end-to-end verified applications

Questions?

Magnus Myreen   Yong Kiam Tan   Ramana Kumar   Anthony Fox   Scott Owens   Michael Norrish