A New Verified Compiler Backend for CakeML

Yong Kiam Tan*  Magnus O. Myreen  Ramana Kumar
Anthony Fox  Scott Owens  Michael Norrish
CakeML

A Verified Implementation of ML

- Has: references, modules, signatures, datatypes, exceptions, ...
- Doesn’t have: functors, module nesting, let-polymorphism
Functional big-step semantics

Versions of formalized semantics in HOL4:
- Relational small-step and big-step semantics
- Functional big-step semantics (ESOP’16)
Functional big-step semantics

Versions of formalized semantics in HOL4:
- Relational small-step and big-step semantics
- Functional big-step semantics (ESOP’16)

\[
\text{eval } \text{st } \text{env} \ (\text{Let } n \ e \ e') = \\
\text{case eval } \text{st } \text{env} \ e \ of \\
(\text{st'}, \text{Rval } v) \Rightarrow \\
\text{eval st'} \text{env}[n\rightarrow v] \ e' \\
\mid \text{res} \Rightarrow \text{res}
\]

\[
\text{eval } \text{st } \text{env} \ (\text{Var } n) = \\
\text{case lookup_var } n \text{ env of} \\
\text{SOME } v \Rightarrow (\text{st}, \text{Rval } v) \\
\mid \text{NONE} \Rightarrow (\text{st}, \text{Rerr})
\]
End-to-end verified compilation from CakeML to x86-64
End-to-end verified compilation from CakeML to x86-64

Compiler bootstrap via proof producing translation
v1.0 drawbacks

- Too low-level for FP optimizations, awkward for target-specific optimizations
v1.0 drawbacks

- Too low-level for FP optimizations, awkward for target-specific optimizations
- Only one target (x86-64) available
CakeML compiler v2.0

- New backend designed for verified optimizations
- Configurable compiler + multiple targets
- Basic support for Foreign Function Interface (FFI)
- 12 ILs for optimizations at different abstraction levels
Abstract values with closures

- 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.
Value abstraction levels

Abstract values with closures

- 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.

Abstract values w/o closures

- BVL: func. lang. without closures
- only 1 global
- DataLang: imperative language
- Perform closure conv.
- Fold constants
- Shrink Lets
- Compile global vars into a dynamically sized array
- Switch to imperative style
- Reduce caller-saved vars
- Combine adjacent memory allocations
Value abstraction levels

Abstract values with closures
- 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.

Abstract values w/o closures
- BVL: func. lang. without closures
  - only 1 global
- DataLang: imperative language
- Perform closure conv.
  - Fold constants
  - Shrink Lets
  - Compile global vars into a dynamically sized array
  - Switch to imperative style
  - Reduce caller-saved vars
  - Combine adjacent memory allocations

Concrete 32/64-bit words
- WordLang: imperative language with machine words, memory and a GC primitive
  - Concretise stack
  - Implement GC primitive
  - Turn stack access into memory accesses
  - Rename registers to match arch registers/conventions
  - Flatten code
  - Delete no-ops (Tick, Skip)
  - Encode program as concrete machine code

Languages
- StackLang: imperative language with array-like stack and optional GC
- LabLang: assembly lang.
- ARMv6
- ARMv8
- x86-64
- MIPS-64
- RISC-V
Compiler frontend

- Frontend largely similar to v1.0

- Source syntax
- 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

- Source AST

- No modules

- No cons. names

- No declarations

- Full pat. match

- No pat. match

- ClosLang:
- Last language with closures
- Has multi-arg. closures

- Eliminate modules
- Infer types, exit if fail

- 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

- BVL: Func. lang. without closures
- Fold constants
- Shrink Lets

- Only 1 global
- Compile global vars into a dynamically resized array

- DataLang: Imperative language
- Switch to imperative style

- WordLang: Imperative language with machine words, memory and a GC primitive
- Remove data abstraction
- Select target instructions
- Perform SSA-like renaming
- Force two-reg code (if req.)
- Allocate register names
- Concretise stack

- StackLang: Imperative language with array-like stack and optional GC
- Implement GC primitive
- Turn stack access into memory accesses
- Rename registers to match arch registers/conventions
- Flatten code

- LabLang: Assembly lang.
- Delete no-ops (Tick, Skip)
- Encode program as concrete machine code

- ARMv6
- 32-bit words
- ARMv8
- 64-bit words
- X86-64
- MIPS-64
- RISC-V

Languages

Compiler transformations

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

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

Reduce caller-saved vars
Compiler frontend

- 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

- ClosLang: last language with closures (has multi-arg. closures)

- Eliminate modules
- Infer types, exit if fail

- 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

- Frontend largely similar to v1.0
- + Foreign Function Interface (FFI)

Languages

- Compiler transformations

Values

- abstract values incl. closures and ref pointers
- machine words and code labels

Languages

- ARMv6
- x86-64
- MIPS-64
- RISC-V

Languages

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

Reduce caller-saved vars
Compiler frontend

- Frontend largely similar to v1.0
- + Foreign Function Interface (FFI)
- New IL: ClosLang, a simple functional language
• Functional-style optimizations
e.g. multi-arg. functions
\[
\text{fn } x \Rightarrow \text{fn } y \Rightarrow x + y
\]
\[
\sim\sim\sim
\text{fn } <x,y> \Rightarrow x + y
\]
### Compiler backend (1)

<table>
<thead>
<tr>
<th>ClosLang: last language with closures (has multi-arg. closures)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rephrase language</td>
</tr>
<tr>
<td>Fuse function calls/apps into multi-arg calls/apps</td>
</tr>
<tr>
<td>Eliminate dead code</td>
</tr>
<tr>
<td>Prepare for closure conv.</td>
</tr>
<tr>
<td>Perform closure conv.</td>
</tr>
<tr>
<td>Fold constants</td>
</tr>
<tr>
<td>Shrink Lets</td>
</tr>
<tr>
<td>Compile global vars into a dynamically resized array</td>
</tr>
<tr>
<td>Switch to imperative style</td>
</tr>
<tr>
<td>Reduce caller-saved vars</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BVL: func. lang. without closures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only 1 global</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DataLang:</th>
</tr>
</thead>
</table>

#### Functional-style optimizations

- e.g. multi-arg. functions
  - `fn x => fn y => x + y`
  - `fn <x,y> => x + y`

#### Closure conversion

- 0: `<y,c> => c.0 + y`
- 1: `<x> => cl(0,1,x)`

- 0: `<x,y> => x + y`
Compiler backend (1)

- Functional-style optimizations
e.g. multi-arg. functions
  \[\text{fn } x \Rightarrow \text{fn } y \Rightarrow x + y\]
  \[\approx\]
  \[\text{fn } <x,y> \Rightarrow x + y\]

- Closure conversion
  0: \(<y,c> \Rightarrow c.0 + y\)
  1: \(<x> \Rightarrow cl(0,1,x)\)

- Switch to imperative semantics
CakeML backend (2)

- Values in terms of 32-/64-bit words
- Configurable data representation

---

DataLang: imperative language
- 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

WordLang: imperative language with machine words, memory and a GC primitive

---

Languages
- Compiler transformations
- Values
  - abstract values incl. closures and ref pointers
  - abstract values incl. code pointers and refs
  - machine words and code labels

32-bit
- ARMv6

64-bit
- x86-64
- MIPS-64
- RISC-V

---

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

Pointer format:

```
0...00110011101  00  01  010  1
```

- padding
- length
- tag
- marker

Idea: Don’t need all bits to address available memory. Encode some constructor tags using pointer bits. User selects number of bits used.

```
case x of
  INL i => e
  | INR j => e'
  ⇝
  temp := x_word && 15w;
  if (temp == 5w) {
    e
  } else {
    e'
  }
```
Pointer representation

Pointer format:

0...00110011101 00 01 010 1

↑ padding ↑ length ↑

address value  tag  marker

- Idea: Don’t need all bits to address available memory
- Encode some constructor tags using pointer bits
- User selects number of bits used

```c
case x of
    INL i => e
  | INR j => e'
  ~> temp := x_word && 15w;
  if (temp == 5w) { e }
  else { e' }
```
Values in terms of 32-/64-bit words

Configurable data representation

Introduce abstract GC specification

DataLang: imperative language
- 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

WordLang: imperative language with machine words, memory and a GC primitive
- Implement GC primitive
- Turn stack access into memory accesses
- Rename registers to match arch registers/conventions
- Flatten code

LabLang: assembly lang.
- Delete no-ops (Tick, Skip)
- Encode program as concrete machine code

Languages

Compiler transformations

Values
- abstract values incl. closures and ref pointers
- abstract values incl. code pointers and refs
- machine words and code labels

32-bit
- 64-bit

ARMv6
- x86-64
- MIPS-64
- RISC-V

Important target-specific optimizations
- Instruction selection
- (Simplified) SSA pass
- Dead code elimination
- Register allocation

All languages communicate with the external world via a byte-array-based foreign-function interface.
CakeML backend (2)

- Values in terms of 32-/64-bit words
- Configurable data representation
- Introduce abstract GC specification
- Important target-specific optimizations
  - Instruction selection
  - (Simplified) SSA pass
  - Dead code elimination
  - Register allocation
CakeML backend (3)

- Concretize stack representation
  
  \[
  \begin{align*}
  &r0 := 5; \\
  &r1 := r0 + s5 \\
  \Rightarrow \\
  &r0 := 5; \\
  &rt := stack\_load(5); \\
  &r1 := r0 + rt \\
  \Rightarrow \\
  &r0 := 5; \\
  &rt := mem\_load(sp+5*4w); \\
  &r1 := r0 + rt
  \end{align*}
  \]

Languages

- Values
  - abstract values incl. closures and ref pointers
  - abstract values incl. code pointers and refs
  - machine words and code labels

- 32-bit words
  - 64-bit words

- ARMv6
  - ARMv8
  - x86-64
  - MIPS-64
  - RISC-V

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

- Reduce caller-saved vars

- Why two separate steps?

- LabLang: assembly lang.
  - Delete no-ops (Tick, Skip)
  - Encode program as concrete machine code

- StackLang: imperative language
  - with array-like stack and optional GC
  - Concretise stack
  - Implement GC primitive
  - Turn stack access into memory accesses
  - Rename registers to match arch registers/conventions
  - Flatten code
  - Delete no-ops (Tick, Skip)
  - Encode program as concrete machine code

- Concretize stack representation
  
  \[
  \begin{align*}
  &r0 := 5; \\
  &r1 := r0 + s5 \\
  \Rightarrow \\
  &r0 := 5; \\
  &rt := stack\_load(5); \\
  &r1 := r0 + rt \\
  \Rightarrow \\
  &r0 := 5; \\
  &rt := mem\_load(sp+5*4w); \\
  &r1 := r0 + rt
  \end{align*}
  \]
Stack frames contain both caller saved and spilled variables.

Idea: provide GC with static liveness information.
CakeML backend (3)

- Concretize stack representation (+ implement GC)

StackLang: imperative language with array-like stack and optional GC
  - Concretise stack
  - Implement GC primitive

LabLang: assembly lang.
  - Delete no-ops (Tick, Skip)
  - Encode program as concrete machine code

ARMv6
ARMv8
x86-64
MIPS-64
RISC-V

Languages
Compiler transformations
Values

- abstract values incl. closures and ref pointers
- abstract values incl. code pointers and refs
- machine words and code labels

32-bit
64-bit

ARMv6
ARMv8
x86-64
MIPS-64
RISC-V

32-bit
64-bit

Languages communicate with the external world via a byte-array-based foreign-function interface.
Reduce caller-saved vars
CakeML backend (3)

- Concretize stack representation (+ implement GC)
- Flatten code, compile to assembly
StackLang: imperative language with array-like stack and optional GC
LabLang: assembly lang.
ARMv6
ARMv8
x86-64
MIPS-64
RISC-V

Concretise stack
Implement GC primitive
Turn stack access into memory accesses
Rename registers to match arch registers/conventions
Flatten code
Delete no-ops (Tick, Skip)
Encode program as concrete machine code

- Concretize stack representation (+ implement GC)
- Flatten code, compile to assembly
- Encode assembly to target ISA, e.g. x64, ARM, etc.

CakeML backend (3)
Simple benchmarks

Benchmark times relative to ocamlc

- btree: 3.5x
- fib: 12x
- foldl: 8.0x
- qsort: 3.5x
- queue: 2.9x
- reverse: 1.6x
Compiler verification

Observational semantics preservation

\[
\vdash \text{compile } \text{config prog} = \text{prog}' \land \\
\text{syntactic\_condition prog} \land \\
\text{Fail} \notin \text{semantics}_A \text{ ffi prog} \Rightarrow \\
\text{semantics}_B \text{ ffi prog}' = \text{semantics}_A \text{ ffi prog}
\]

If the semantics of a program, \( \text{prog} \), is not a failure, then the semantics of the compiled program, \( \text{prog}' \) is equivalent to \( \text{prog}' \)'s semantics.

Get overall compiler correctness theorem by composing these theorems.
Compiler verification

Observational semantics preservation (caveat)

\[ \vdash \text{compile config prog} = \text{prog}' \land \]
\[ \text{syntactic\_condition prog} \land \]
\[ \text{Fail} \not\in \text{semantics}_A \text{ ffi prog} \Rightarrow \]
\[ \text{semantics}_B \text{ ffi prog}' \subseteq \text{add\_resource\_limit (semantics}_A \text{ ffi prog}) \]

If the semantics of a program, prog, is not a failure, then the semantics of the compiled program, prog' is equivalent to prog’s semantics (up to the point where it stops with an out-of-memory error).

Get overall compiler correctness theorem by composing these theorems
Compiler verification techniques / tricks

Compiler correctness theorem

\[ \vdash \text{compile config } prog = prog' \land \]
\[ \text{eval}_A prog = (rstate, res) \land \]
\[ \text{state} \_ \text{rel} \text{ state } state' \land res \neq \text{Error} \Rightarrow \]
\[ \exists rstate' res'. \]
\[ \text{eval}_B prog' state' = (rstate', res') \land \]
\[ \text{state} \_ \text{rel} rstate rstate' \land \text{res} \_ \text{rel} res res' \]

- Usual style: induct “forward” over compilation function with an appropriate state relation
- Other styles: logical relations, “backward” compiler proofs
Proof size: 100 K lines of HOL4, 6 developers, 2 years.

- Division of work across ILs
- Only prove *soundness* of each pass
- Fine-grained control over semantics of each IL
Register allocation via graph coloring

Idea: most heuristics involve picking an order to color vertices greedily.

Verification: prove that the coloring algorithm never picks clashing colors.

\[ u \xrightarrow{} x \xrightarrow{} y \xrightarrow{} v \]

\[ u \xrightarrow{} x \xrightarrow{} y \xrightarrow{} v \]
Register allocation via graph coloring

Idea: most heuristics involve picking an order to color vertices greedily

Verification: prove that the coloring algorithm never picks clashing colors
Control over IL semantics

- Data abstraction removal has complex invariants
- Simplification: specify GC abstractly, implement later
- Actual interaction still complex (see paper for details)
Ongoing & future work

Ongoing:
- Charguéraud’s characteristic formulae (ICFP’10,’11)
- New compiler optimizations, e.g. peephole optimizations
- v2.0 compiler bootstrap
- Generational garbage collection (Adam)
Ongoing & future work

Ongoing:
- Charguéraud’s characteristic formulae (ICFP’10,’11)
- New compiler optimizations, e.g. peephole optimizations
- v2.0 compiler bootstrap
- Generational garbage collection (Adam)

Planned:
- Verified reflection (+ a REPL)
- Verified theorem prover
- Your idea here! (or see https://cakeml.org/projects.html)
We are at ICFP! Feel free to approach us for more details.

Rough guide:
- **Green** (Ramana, Scott, Michael)
- **Red** (Magnus)
- **Blue** (Yong Kiam)
- **Magenta** (Magnus, Anthony)
CakeML compiler bootstrap

1. Proof-producing translation

CakeML Compiler as HOL functions

CakeML Compiler as CakeML functions

2. Evaluate in logic

Bootstrapped CakeML Compiler
Termination preservation

- Compilation theorem:
  \[ \text{eval} \ (\text{prog}, \text{st with clock:=c}) = (\text{res}, \text{st'}) \implies \exists k. \text{eval} \ (\text{cprog}, \text{st with clock:=c+k}) = (\text{res}, \text{st'}) \]

- Simpler case: If prog does not time out at clock c, cprog does not time out at c+k.

- Harder case: If prog diverges (i.e. times out on all clocks)
  Suppose for contradiction that cprog does not time out for some clock c:
  \[ \text{eval} \ (\text{cprog}, \text{st with clock:=c}) = (\text{res}, \text{st'}) \lor \text{res} <> \text{Timeout} \]
Termination preservation

- Clock determinism lemma:
  
  \[
  \text{eval (prog, st with clock:=c)} = (\text{res, st'}) \land \\
  \text{res} \neq \text{Timeout} \implies \\
  \text{eval (prog, st with clock:=c+k)} = (\text{res, ...})
  \]

- We have:
  
  \[
  \text{eval (prog, st with clock:=c)} = (\text{Timeout, st'})
  \]

- Using compilation theorem:
  
  \[
  \text{eval (cprog, st with clock:=c+k)} = (\text{Timeout, st'})
  \]

- Contradiction by clock determinism on cprog at clock c.
Top-level correctness and FFI assumptions

Top-level correctness theorem

Any binary produced by a successful evaluation of the compiler function will either behave exactly according to the observable behaviour of the source semantics, or behave the same as the source up to some point at which it terminates with an out-of-memory error.

Assumptions:

- Compiler configuration is well-formed
- Generated program runs in a environment where the external world only modifies memory outside CakeML's memory region
- The behaviour of the FFI in the machine semantics matches the behaviour of the abstract FFI parameter provided to the source semantics