A Verified Lisp Implementation for A Verified Theorem Prover

Scheme workshop 2016, Nara, Japan

Magnus O. Myreen — University of Cambridge, but now at Chalmers University of Technology
Jared Davis — Centaur Technology, Inc., but now at Apple
Result:

A Verified Lisp Implementation for A Verified Theorem Prover

Claim:

The most comprehensive proof-based evidence of a theorem prover's soundness to date.
2005:

I’m a PhD student working on verification of machine code (factorial, length of a linked list)

Theme: exploring how to make verification scale.

Result:

A Verified Lisp Implementation for A Verified Theorem Prover

???
The start:

I’m a PhD student working on verification of machine code (factorial, length of a linked list)
Context: interactive theorem proving

Aim: to prove deep functional properties of machine code.

Proofs are performed in HOL4 — a fully expansive theorem prover.

All proofs expand at runtime into primitive inferences in the HOL4 kernel.

The kernel implements the axioms and inference rules of higher-order logic.
Context: interactive theorem proving

photo idea: Larry Paulsson
Machine code

Machine code,

\[ \text{E1510002 B0422001 C0411002 01AFFFFFB} \]

is impossible to read, write or maintain manually.

However, for theorem-prover-based formal verification:

machine code is clean and tractable!

Reason:

- all types are concrete: \text{word32}, \text{word8}, \text{bool}.
- state consists of a few simple components: a few registers, a memory and some status bits.
- each instruction performs only small well-defined updates.
Challenges of Machine Code

Challenges:
- several large, detailed models
- unstructured code
- very low-level and limited resources
During my PhD, I developed the following infrastructure:

...each part will be explained in the next slides.
Hoare triples

Each model can be evaluated, e.g. ARM instruction `add r0,r0,r0` is described by theorem:

\[
\begin{align*}
\vdash (\text{ARM}_\text{READ}\_\text{MEM} \ ((31 > 2) \ (\text{ARM}_\text{READ}\_\text{REG} \ 15w \ \text{state})) \ \text{state} = 0xE0800000w) \land \neg \text{state.undefined} \Rightarrow \\
(\text{NEXT}\_\text{ARM}\_\text{MMU} \ cp \ \text{state} = \\
\text{ARM}_\text{WRITE}\_\text{REG} \ 15w \ (\text{ARM}_\text{READ}\_\text{REG} \ 15w \ \text{state} + 4w) \ \\
(\text{ARM}_\text{WRITE}\_\text{REG} \ 0w \\
(\text{ARM}_\text{READ}\_\text{REG} \ 0w \ \text{state} + \text{ARM}_\text{READ}\_\text{REG} \ 0w \ \text{state}) \ \text{state}))
\end{align*}
\]

As a total-correctness machine-code Hoare triple:

\[
\begin{align*}
\vdash \text{SPEC ARM\_MODEL} & \quad \text{Informal syntax for this talk:} \\
(aR \ 0w \ x \ast \ aPC \ p) & \{ \text{R0} \ x \ast \ PC \ p \} \\
\{(p,0xE0800000w)\} & \quad p : E0800000 \\
(aR \ 0w \ (x+x) \ast \ aPC \ (p+4w)) & \{ \text{R0} \ (x+x) \ast \ PC \ (p+4) \}
\end{align*}
\]
Definition of Hoare triple

\[ \{p\} \ c \ \{q\} \iff \forall s \ r. \ (p \ast r \ast \text{code } c) \ s \implies \exists n. \ (q \ast r \ast \text{code } c) \ (\text{run } n \ s) \]

Program logic can be used directly for verification.
But direct reasoning in this embedded logic is tiresome.
Decompiler

Decompiler automates Hoare triple reasoning.

**Example:** Given some ARM machine code,

0: E3A00000    mov r0, #0
4: E3510000    L: cmp r1, #0
8: 12800001    addne r0, r0, #1
12: 15911000    ldrne r1, [r1]
16: 1AFFFFFFFB    bne L

the decompiler automatically extracts a readable function:

\[ f(r_0, r_1, m) = \text{let } r_0 = 0 \text{ in } g(r_0, r_1, m) \]

\[ g(r_0, r_1, m) = \text{if } r_1 = 0 \text{ then } (r_0, r_1, m) \text{ else } \]

\[ \text{let } r_0 = r_0 + 1 \text{ in } \]

\[ \text{let } r_1 = m(r_1) \text{ in } \]

\[ g(r_0, r_1, m) \]
Decompile, correct?

Decompiler automatically proves a certificate theorem:

\[ f_{pre}(r_0, r_1, m) \Rightarrow \]
\[ \{ (R0, R1, M) \text{ is } (r_0, r_1, m) \ast PC \ p \ast S \} \]
\[ p : E3A00000 \ E3510000 \ 12800001 \ 15911000 \ 1AFFFFFFB \]
\[ \{ (R0, R1, M) \text{ is } f(r_0, r_1, m) \ast PC \ (p + 20) \ast S \} \]

which informally reads:

for any initially value \((r_0, r_1, m)\) in reg 0, reg 1 and memory, the code terminates with \(f(r_0, r_1, m)\) in reg 0, reg 1 and memory.
Decomposition verification example

To verify code: prove properties of function $f$,

\[
\forall x \, l \, a \, m. \ \text{list}(l, a, m) \Rightarrow f(x, a, m) = (\text{length}(l), 0, m)
\]
\[
\forall x \, l \, a \, m. \ \text{list}(l, a, m) \Rightarrow f_{\text{pre}}(x, a, m)
\]

since properties of $f$ carry over to machine code via the certificate.

**Proof reuse**: Given similar x86 and PowerPC code:

```
31C085F67405408B36EBF7
38A000002C14000B408200107E0A02E38A500014BFFFFFF0
```

which decompiles into $f'$ and $f''$, respectively. Manual proofs above can be reused if $f = f' = f''$. 
Decompile how to

How to decompile:

1. derive Hoare triple theorems using Cambridge ARM model
2. compose Hoare triples
3. extract function

(Loops result in recursive functions.)

\[
\begin{align*}
\{ & R0 \ i \ * \ R1 \ j \ * \ PC \ p \} \\
\text{p+0 :}& \{ \ R0 \ (i+j) \ * \ R1 \ j \ * \ PC \ (p+4) \} \\
\{ & R0 \ i \ * \ PC \ (p+4) \} \\
\text{p+4 :}& \{ \ R0 \ (i \gg 1) \ * \ PC \ (p+8) \} \\
\{ & LR \ lr \ * \ PC \ lr \} \\
\text{p+8 :}& \{ \ LR \ lr \ * \ PC \ lr \} \\
\{ & R0 \ i \ * \ R1 \ j \ * \ LR \ lr \ * \ PC \ lr \} \\
\text{p :} & e0810000 \ e1a000a0 \ e12ffff1e \\
\{ & R0 \ ((i+j)>>1) \ * \ R1 \ j \ * \ LR \ lr \ * \ PC \ lr \} \\
\text{avg (i,j) = (i+j)>>1}
\end{align*}
\]
Decompiler cont.

Implementation:

- ML program which fully automatically performs forward proof
- no heuristics and no dangling proof obligations
- loops result in tail-recursive functions

Case studies:

- verified copying garbage collector
- bignum library routines
Part 2:

I want more automation and abstraction!
Proof-producing compilation

Synthesis often more practical. Given function $f$,

$$f(r_1) = \text{if } r_1 < 10 \text{ then } r_1 \text{ else let } r_1 = r_1 - 10 \text{ in } f(r_1)$$

our compiler generates ARM machine code:

```
E351000A L: cmp r1,#10
2241100A subcs r1,r1,#10
2AFFFFFFFFC bcs L
```

and automatically proves a certificate HOL theorem:

$$\vdash \{ R1 \ r_1 \ast PC \ p \ast s \}$$

$$p : E351000A \ 2241100A \ 2AFFFFFFFFC$$

$$\{ R1 \ f(r_1) \ast PC \ (p+12) \ast s \}$$
Compilation, example cont.

One can prove properties of \( f \) since it lives inside HOL:

\[
\forall x. \ f(x) = x \mod 10
\]

Properties proved of \( f \) translate to properties of the machine code:

\[
\{ \text{R1 } r_1 \mod 10 \} \text{ PC } (p+12) \text{ * s}
\]

Additional feature: the compiler can use the above theorem to extend its input language with: \( \text{let } r_1 = r_1 \mod 10 \text{ in } \)
Implementation

To compile function $f$:

1. generate, without proof, code from input $f$;
2. decompile, with proof, a function $f'$ from generated code;
3. prove $f = f'$.

Features:

- code generation *completely separate* from proof
- supports many light-weight *optimisations* without any additional proof burden: instruction reordering, conditional execution, dead-code elimination, duplicate-tail elimination, ...
- allows for significant *user-defined extensions*
Infrastructure again

Idea: create LISP implementations via compilation.

HOL4 functions for LISP parse, eval, print

verified code for LISP primitives car, cdr, cons, etc.

machine-code Hoare triple

ARM, x86, PowerPC code and certificate theorems

compiler

decompiler

ARM, x86, PowerPC
Lisp formalised

LISP s-expressions defined as data-type SExp:

\[ \text{Num} : \mathbb{N} \rightarrow \text{SExp} \]
\[ \text{Sym} : \text{string} \rightarrow \text{SExp} \]
\[ \text{Dot} : \text{SExp} \rightarrow \text{SExp} \rightarrow \text{SExp} \]

LISP primitives were defined, e.g.

\[ \text{cons } x \ y = \text{Dot } x \ y \]
\[ \text{car} (\text{Dot } x \ y) = x \]
\[ \text{plus} (\text{Num } m) (\text{Num } n) = \text{Num} (m + n) \]

The semantics of LISP evaluation was taken to be Gordon’s formalisation of ‘LISP 1.5’-like evaluation
Extending the compiler

We define heap assertion ‘lisp (v₁, v₂, v₃, v₄, v₅, v₆, l)’ and prove implementations for primitive operations, e.g.

\[
\text{is}_\text{pair } v₁ \Rightarrow \\
\{ \text{lisp} (v₁, v₂, v₃, v₄, v₅, v₆, l) * \text{pc } p \} \\
p : \text{E5934000} \\
\{ \text{lisp} (v₁, \text{car } v₁, v₃, v₄, v₅, v₆, l) * \text{pc } (p + 4) \} \\
\text{size } v₁ + \text{size } v₂ + \text{size } v₃ + \text{size } v₄ + \text{size } v₅ + \text{size } v₆ < l \Rightarrow \\
\{ \text{lisp} (v₁, v₂, v₃, v₄, v₅, v₆, l) * \text{pc } p \} \\
p : \text{E50A3018 E50A4014 E50A5010 E50A600C ...} \\
\{ \text{lisp} (\text{cons } v₁ v₂, v₂, v₃, v₄, v₅, v₆, l) * \text{pc } (p + 332) \} \\
\]

with these the compiler understands:

let v₂ = \text{car } v₁ in ...
let v₁ = \text{cons } v₁ v₂ in ...
Reminder

How to decompile:
1. derive Hoare triple theorems using Cambridge ARM model
2. compose Hoare triples
3. extract function

(Loops result in recursive functions.)
Abstract. This paper reports on a case study, which we believe is the first to produce a formally verified end-to-end implementation of a functional programming language running on commercial processors. Interpreters for the core of McCarthy’s LISP 1.5 were implemented in ARM, x86 and PowerPC machine code, and proved to correctly parse, evaluate and print LISP s-expressions. The proof of evaluation required working on top of verified implementations of memory allocation and garbage collection. All proofs are mechanised in the HOL4 theorem prover.
Running the Lisp interpreter

Nintendo DS lite (ARM)  MacBook (x86)  old MacMini (PowerPC)

\( \text{(pascal-triangle } '((1)) '6) \)

returns:

\[
(\begin{array}{cccccccc}
1 & 6 & 15 & 20 & 15 & 6 & 1 \\
1 & 5 & 10 & 10 & 5 & 1 \\
1 & 4 & 6 & 4 & 1 \\
1 & 3 & 3 & 1 \\
1 & 2 & 1 \\
1 & 1 \\
1 \\
\end{array})
\]
Part 3:

A sudden need for a serious Lisp implementation.
Two projects meet

My theorem prover is written in Lisp. Can I try your verified Lisp?

Umm.. sure!

Does your Lisp support ..., ..., and ...?

No, but it could ...

Jared Davis

A self-verifying theorem prover

Magnus Myreen

Verified Lisp implementations

verified LISP on ARM, x86, PowerPC
Running Milawa

Milawa’s bootstrap proof:

- 4 gigabyte proof file: >500 million unique conses
- takes 16 hours to run on a state-of-the-art runtime (CCL)

verified LISP on ARM, x86, PowerPC

hopelessly “toy”
Running Milawa

Milawa’s bootstrap proof:

- 4 gigabyte proof file: >500 million unique conses
- takes 16 hours to run on a state-of-the-art runtime (CCL)

Result:
- a new verified Lisp which is able to host the Milawa thm prover

Jitawa: verified **LISP** with **JIT compiler**
A short introduction to Milawa

- Milawa is styled after theorem provers such as NQTHM and ACL2,
- has a small trusted logical kernel similar to LCF-style provers,
- ... but does not suffer the performance hit of LCF’s fully expansive approach.
Comparison with LCF approach

**LCF-style approach**
- all proofs pass through the core’s primitive inferences
- extensions steer the core

**the Milawa approach**
- all proofs must pass the core
- the **core proof checker** can be replaced at runtime
Requirements on runtime

Milawa uses a subset of Common Lisp which

is for most part **first-order pure functions** over
natural numbers, symbols and conses,

uses primitives: cons car cdr consp natp symbolp
    equal + - < symbol-< if

macros: or and list let let* cond
    first second third fourth fifth

and a simple form of lambda-applications.

(Lisp subset defined on later slide.)
Requirements on runtime

...but Milawa also

• uses destructive updates, hash tables
• prints status messages, timing data
• uses Common Lisp’s checkpoints
• forces function compilation
• makes dynamic function calls
• can produce runtime errors

(Lisp subset defined on later slide.)
Runtime must scale

Designed to scale:

• just-in-time compilation for speed
  ‣ functions compile to native code

• target 64-bit x86 for heap capacity
  ‣ space for $2^{31}$ (2 billion) cons cells (16 GB)

• efficient scannerless parsing + abbreviations
  ‣ must cope with 4 gigabyte input

• graceful exits in all circumstances
  ‣ allowed to run out of space, but must report it
Workflow

1. specified input language: syntax & semantics
2. verified necessary algorithms, e.g.
   • compilation from source to bytecode
   • parsing and printing of s-expressions
   • copying garbage collection
3. proved refinements from algorithms to x86 code
4. plugged together to form read-eval-print loop

~30,000 lines of HOL4 scripts
AST of input language

\[
\begin{align*}
term & ::= \text{Const } \text{sexp} \mid \text{Var } \text{string} \mid \text{App } \text{func } (\text{term } \text{list}) \mid \text{If } \text{term } \text{term } \text{term} \mid \text{LambdaApp } (\text{string } \text{list}) \text{ term } (\text{term } \text{list}) \mid \text{Or } (\text{term } \text{list}) \mid \text{And } (\text{term } \text{list}) \mid \text{List } (\text{term } \text{list}) \mid \text{Let } ((\text{string } \times \text{term}) \text{ list}) \text{ term} \mid \text{LetStar } ((\text{string } \times \text{term}) \text{ list}) \text{ term} \mid \text{Cond } ((\text{term } \times \text{term}) \text{ list}) \mid \text{First } \text{term} \mid \text{Second } \text{term} \mid \text{Third } \text{term} \mid \text{Fourth } \text{term} \mid \text{Fifth } \text{term} \\
\text{sexp} & ::= \text{Val } \text{num} \mid \text{Sym } \text{string} \mid \text{Dot } \text{sexp } \text{sexp} \\
\text{func} & ::= \text{Define} \mid \text{Print} \mid \text{Error} \mid \text{Funcall} \mid \text{PrimitiveFun } \text{primitive} \mid \text{Fun } \text{string} \\
\text{primitive} & ::= \text{Equal} \mid \text{Symbolp} \mid \text{SymbolLess} \mid \text{Consp} \mid \text{Cons} \mid \text{Car} \mid \text{Cdr} \mid \text{Natp} \mid \text{Add} \mid \text{Sub} \mid \text{Less}
\end{align*}
\]
compile: AST $\rightarrow$ bytecode list

bytecode $::=$ Pop pop one stack element
| PopN $num$ pop $n$ stack elements
| PushVal $num$ push a constant number
| PushSym $string$ push a constant symbol
| LookupConst $num$ push the $n$th constant from system state
| Load $num$ push the $n$th stack element
| Store $num$ overwrite the $n$th stack element
| DataOp $primitive$ add, subtract, car, cons, ...
| Jump $num$ jump to program point $n$
| JumpIfNil $num$ conditionally jump to $n$
| DynamicJump jump to location given by stack top
| Call $num$ static function call (faster)
| DynamicCall dynamic function call (slower)
| Return return to calling function
| Fail signal a runtime error
| Print print an object to stdout
| Compile compile a function definition
How do we get just-in-time compilation?

Treating code as data:

\[ \forall p \ c \ q. \ \{p\} \ c \ \{q\} = \{p \ast \text{code } c\} \emptyset \{q \ast \text{code } c\} \]

Definition of Hoare triple:

\[ \{p\} \ c \ \{q\} = \forall s \ r. \ (p \ast r \ast \text{code } c) \ s \Rightarrow \exists n. \ (q \ast r \ast \text{code } c) \ (\text{run } n \ s) \]
I/O and efficient parsing

Jitawa implements a read-eval-print loop:

Use of external C routines adds assumptions to proof:

- reading next string from stdin
- printing null-terminated string to stdout
Read-eval-print loop

• Result of reading lazily, writing eagerly
• Eval = compile then jump-to-compiled-code
• Specification: read-eval-print until end of input
Correctness theorem

There must be enough memory and I/O assumptions must hold.

This machine-code Hoare triple holds only for terminating executions.

\[
\{ \text{init\_state } \textit{io} \star \text{pc } \textit{p} \star \langle \text{terminates\_for } \textit{io} \rangle \} \\
\textit{p} : \text{code\_for\_entire\_jitawa\_implementation} \\
\{ \text{error\_message } \lor \ \exists \textit{io}' . \langle \langle [], \textit{io} \rangle \xrightarrow{\text{exec}} \textit{io}' \rangle \star \text{final\_state } \textit{io}' \} \\
\]

Each execution is allowed to fail with an error message.

If there is no error message, then the result is described by the high-level op. semantics.
$ cat verified_code.s

/* Machine code automatically extracted from a HOL4 theorem. */
/* The code consists of 7423 instructions (31840 bytes). */

.byte 0x48, 0xB, 0x5F, 0x18
.byte 0x4C, 0x8B, 0x7F, 0x10
.byte 0x48, 0x8B, 0x47, 0x20
.byte 0x48, 0x8B, 0x4F, 0x28
.byte 0x48, 0x8B, 0x57, 0x08
.byte 0x48, 0x37
.byte 0x4C, 0x8B, 0x47, 0x60
.byte 0x4C, 0x8B, 0x4F, 0x68
.byte 0x4C, 0x8B, 0x57, 0x58
.byte 0x48, 0x01, 0xC1
.byte 0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0x83, 0xC0, 0x04
.byte 0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
.byte 0x48, 0x83, 0xC0, 0x04
...
Running Milawa on Jitawa

Running Milawa’s 4-gigabyte bootstrap process:

- CCL: 16 hours
- SBCL: 22 hours
- Jitawa: 128 hours (8x slower than CCL)

Jitawa’s compiler performs almost no optimisations.

Parsing the 4 gigabyte input:

- CCL: 716 seconds (9x slower than Jitawa)
- Jitawa: 79 seconds
Part 4:
The end-to-end result
Proving Milawa sound

- semantics of Milawa’s logic
- inference rules of Milawa’s logic
- Milawa theorem prover (kernel approx. 2000 lines of Milawa Lisp)
- Lisp semantics
- Lisp implementation (x86) (approx. 7000 64-bit x86 instructions)
- semantics of x86-64 machine

Proving soundness of the source code

Verification of a Lisp implementation

Assumes x86 model, C wrapper, OS, hardware
Milawa theorem prover
(kernel approx. 2000 lines of Milawa Lisp)

Proving the top-level theorem

The top-level theorem:
relates the logic’s semantics
with the execution of the x86 machine code.

Steps:

A. formalise Milawa’s logic
   - syntax, semantics, inference, soundness

B. prove that Milawa's kernel is faithful to the logic
   - run the Lisp parser (in the logic) on Milawa’s kernel
   - translate (with proof) deep embedding into shallow
   - prove that Milawa’s (reflective) kernel is faithful to logic

C. connect the verified Lisp implementation
   - compose with the correctness thm for Lisp system
Theorem: Milawa is sound down to x86

There must be enough memory and input is Milawa’s kernel followed by call to main for some input.

∀input pc.
{ init_state (milawa_implementation ++ "(milawa-main 'input)") * pc pc }
  pc : code_for_entire_jitawa_implementation
{ error_message ∨ (let result = compute_output (parse input) in
  ⟨every_line line_ok result⟩ *
  final_state (output_string result ++ "SUCCESS")) }

Machine code terminates either with error message, or ...

... output lines that are all true w.r.t. the semantics of the logic.

line_ok (π, l) = (l = "NIL") ∨
  (∃n. (l = "(PRINT (n . . . ))") ∧ is_number n) ∨
  (∃φ. (l = "(PRINT (THEOREM φ))") ∧ context_ok π ∧ ⊨_π φ)
Final Part:
Learning from the mistakes. Doing it better.
A better compiler compiler?

The x86 for the compile function was produced as follows:

A bit cumbersome....

…should have compiled the verified compiler using itself!
Instead: we should bootstrap the verified compile function, i.e. evaluate the compiler on a deep embedding of itself within the logic:

\[
\text{EVAL `compile COMPILE`} \quad \text{derivates a theorem:}
\]

\[
\text{compile COMPILE} = \text{compiler-as-machine-code}
\]
CakeML: A Verified Implementation of ML

Ramana Kumar 1 Magnus 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 an implementation of the ML language called CakeML, which supports a substantial subset of Standard ML.

CakeML is implemented as an interactive read-eval-print loop inside the logic; we write, and can run, the compiler as a function in the logic, and we synthesise a CakeML implementation of the compiler finally from logic to ML, now to machine code (via verified synthesise CakeML from that using our previous technique [22], which puts the compiler on equal footing with other CakeML programs. We then apply the compiler to itself, avoids a tedious manual refinement proof relating the compilation algorithm to its implementation, as well as providing a moderately large example program. More specifically, • the compiler correctness theorem thereby applies to the machine-code implementation of the compiler.

The first bootstrapping of a formally verified compiler.

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; two, the execution of that algorithm as implemented in machine code; and three, the correctness of the machine code. We call this the first bootstrapping of a formally verified compiler.
A New Verified Compiler Backend for CakeML

Yong Kiam Tan
IHPC, A*STAR, Singapore
tanyongkiam@gmail.com

Magnus O. Myreen
Chalmers University of Technology, Sweden
myreen@chalmers.se

Ramana Kumar
Data61, CSIRO / UNSW, Australia
ramana.kumar@data61.csiro.au

Anthony Fox
University of Cambridge, UK
anthony.fox@cl.cam.ac.uk

Scott Owens
University of Kent, UK
s.a.owens@kent.ac.uk

Michael Norrish
Data61, CSIRO
michael.norrish@data61.csiro.au

Abstract

We have developed and mechanically verified a new compiler backend for CakeML. Our new compiler features a sequence of intermediate languages that allows it to incrementally compile away high-level features and enables verification at the right levels of semantic detail. In this way, it resembles mainstream (unverified) compilers for strict functional languages. The compiler supports 12 intermediate languages, 5 target architectures: x86-64, ARMv6, ARMv8, MIPS-64, and RISC-V. It includes register allocation via Iterated Register Coalescing. It has efficient, configurable data representations, exceptions that unwind the call stack, register allocation, and properly compiles the call stack into memory, including the semantics of ML-style exception mechanism. The new compiler has a fully featured source language, namely a functional programming language to date.

To make the verification tractable, the compiler’s design must fit together. We focus particularly on the interaction between the compiler’s design and the compiler verifier. This means that the compiler’s intermediate languages, including their semantics, need to be carefully constructed to support precise specification of tractable invariants. Of course, we also consider the compiler’s code. We have developed and verified a new compiler backend for CakeML. Our new compiler features a sequence of intermediate languages that allows it to incrementally compile away high-level features and enables verification at the right levels of semantic detail. In this way, it resembles mainstream (unverified) compilers for strict functional languages. The compiler supports 12 intermediate languages, 5 target architectures: x86-64, ARMv6, ARMv8, MIPS-64, and RISC-V. It includes register allocation via Iterated Register Coalescing. It has efficient, configurable data representations, exceptions that unwind the call stack, register allocation, and properly compiles the call stack into memory, including the semantics of ML-style exception mechanism. The new compiler has a fully featured source language, namely a functional programming language to date.

To make the verification tractable, the compiler’s design must fit together. We focus particularly on the interaction between the compiler’s design and the compiler verifier. This means that the compiler’s intermediate languages, including their semantics, need to be carefully constructed to support precise specification of tractable invariants. Of course, we also consider the compiler’s code.
Looking back…

2005:

I’m a PhD student working on verification of machine code (factorial, length of a linked list)

A Verified Lisp Implementation for A Verified Theorem Prover

???

???
I’m a PhD student working on verification of machine code (factorial, length of a linked list)

- basic reasoning about real machine code
- powerful automation
- verification of garbage collectors
- synthesis from (abstract) functional specs
- verified Lisp interpreters
- verified just-in-time compiler for Lisp

Result:

A Verified Lisp Implementation for A Verified Theorem Prover

Questions?

Thank you for inviting me!
Intuition for Bootstrapping

Proof-producing synthesis

HOL functions $\rightarrow$ CakeML AST $\rightarrow$ CakeML AST $\rightarrow$ machine code

Verified parsing

ASCII $\rightarrow$ CakeML AST $\rightarrow$ CakeML AST $\rightarrow$ typeable yes/no

Verified compiler backend

Verified type inference
Intuition for Bootstrapping

verified x86 implementation of parsing, type inference, and compilation