Towards verified theorem provers

Part 1: proving the soundness of Milawa

QED+20, Vienna

Magnus O. Myreen — Computer Laboratory, University of Cambridge, UK
Jared Davis — Centaur Technology, Inc., Austin TX, USA

(also mentions work with Rob Arthan, Ramana Kumar, Michael Norrish, Scott Owens)
Verified theorem provers

Part 1: proving the soundness of Milawa

Part 2: (by Ramana) towards a verified impl. of HOL light
Part I: proving the soundness of Milawa

A self-verifying theorem prover
Jared Davis — PhD work

A verified runtime for a verified theorem prover
Magnus Myreen, Jared Davis — ITP’11
Proving Milawa sound

- semantics of Milawa’s logic
- inference rules of Milawa’s logic
- Milawa theorem prover (kernel approx. 1700 lines of Milawa Lisp)
- Lisp semantics
- Lisp implementation (x86) (approx. 7000 64-bit x86 instructions)
- semantics of x86-64 machine

Milawa’s soundness [ITP’14]

verification of a Lisp implementation [ITP’11]
A very short introduction

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

---

**The Milawa theorem prover**

- custom tools
- FOL provers
- decision procedures
- simplifier
- rewriter
- SAT/SMT

---

- ‘auto’ tactics
- rewriting
- case splitting
- derived rules

---

work by Jared Davis
Brief Milawa demo
Proving Milawa sound

- semantics of Milawa’s logic
- inference rules of Milawa’s logic
- Milawa theorem prover (kernel approx. 1700 lines of Milawa Lisp)
- Lisp semantics
- Lisp implementation (x86) (approx. 7000 64-bit x86 instructions)
- semantics of x86-64 machine

Jitawa verified Lisp

Milawa’s soundness [ITP’14]

verification of a Lisp implementation [ITP’11]
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 from ITP’11

A—C combine to a top-level theorem that relates the logic’s semantics with the execution of the x86 machine code.
This talk

A. formalise Milawa’s logic

B. prove that Milawa's kernel is faithful to the logic

C. connect the verified Lisp implementation
Syntax

sexp ::= Val num | Sym string | Dot sexp sexp  

prim ::= If | Equal | Not | Symbolp | Symbol_less  
       | Natp | Add | Sub | Less | Cons | Cons  
       | Car | Cdr | Rank | Ord_less | Ordp  

func ::= PrimitiveFun prim  
      | Fun string  

term ::= Const sexp  
       | Var string  
       | App func (term list)  
       | LamApp (string list) term (term list)  

formula ::= ¬formula  
           | formula \ formulal  
           | term = term
Context

Syntax, semantics and inference rules depend on a context.

A context is a finite partial map from \textit{string} to \textit{string list} \times \textit{func\_body} \times (\textit{sexp list} \rightarrow \textit{sexp})

\[
\begin{align*}
\text{func\_body} & ::= \text{Body term} & \text{concrete term (e.g. recursive function)} \\
& | \text{Witness term string} & \text{property, element name} \\
& | \text{None} & \text{no function body given}
\end{align*}
\]
Semantics

\[(\models_{\pi} p) = \text{formula\_ok}_{\pi} p \land \forall i. \text{eval\_formula} i \ \pi \ p\]

**syntax makes sense**

**truth value**

\[
\begin{align*}
\text{eval\_formula} i \ \pi \ (\neg p) & = \neg(\text{eval\_formula} i \ \pi \ p) \\
\text{eval\_formula} i \ \pi \ (p \lor q) & = \text{eval\_formula} i \ \pi \ p \lor \text{eval\_formula} i \ \pi \ q \\
\text{eval\_formula} i \ \pi \ (x = y) & = (\text{eval\_term} i \ \pi \ x \ = \ \text{eval\_term} i \ \pi \ y) \\
\text{eval\_term} i \ \pi \ (\text{Const} \ c) & = c \\
\text{eval\_term} i \ \pi \ (\text{Var} \ v) & = i(v) \\
\text{eval\_term} i \ \pi \ (\text{App} \ f \ xs) & = \text{eval\_app} (f, \text{map} (\text{eval\_term} i \ \pi) \ xs, \pi) \\
\text{eval\_term} i \ \pi \ (\text{LambdaApp} \ vs \ x \ xs) & = \text{let} \ ys = \text{map} (\text{eval\_term} i \ \pi) \ xs \ \text{in} \\
& \quad \text{eval\_term} [vs \mapsto ys] \ \pi \ x \\
\text{eval\_app} (\text{PrimitiveFun} \ p, \text{args}, \pi) & = \text{eval\_primitive} p \ \text{args} \\
\text{eval\_app} (\text{Fun} \ \text{name}, \text{args}, \pi) & = \text{let} (\_\_interp) = \pi(\text{name}) \ \text{in} \\
& \quad \text{interp}(\text{args}) \\
\text{eval\_primitive} \ \text{Add} \ [\text{Val} \ 2, \text{Val} \ 3] & = \text{Val} \ 5 \\
\text{eval\_primitive} \ \text{Add} \ [\text{Val} \ 2, \text{Sym} \ "a"] & = \text{Val} \ 2 \\
\text{eval\_primitive} \ \text{Cons} \ [\text{Val} \ 2, \text{Sym} \ "a"] & = \text{Dot} (\text{Val} \ 2) (\text{Sym} \ "a")
\end{align*}
\]
Well-formedness of context

Semantics only makes sense for well-formed contexts.

For every entry,

\[ \pi(name) = (formals, \text{Body} \ body, \text{interp}) \]

it must be that:

- the formals are all distinct
- the body is well-formed w.r.t. the context
- the interpretation satisfies the defining equation:

\[ \forall i. \ \text{interp}(\text{map } i \ \text{formals}) = \text{eval}_\text{term } i \ \pi \ body \]

Similarly for the other function types.
(a few of the) Inference rules

\[ \vdash_\pi a \lor (b \lor c) \]
\[ \vdash_\pi (a \lor b) \lor c \quad \text{associativity} \]
\[ a \in \text{milawa\_axioms} \]
\[ \vdash_\pi a \quad \text{(basic axiom)} \]

function definition in context

\[ \pi(name) = (\text{formals}, \text{Body} body, \text{interp}) \]
\[ \vdash_\pi \text{App} (\text{Fun} name) (\text{map} \text{Var} \text{formals}) = body \]

body of function

defining equation

facts about Lisp primitives
Soundness of logic

Soundness of inference rules:

\[ \forall \pi\ p. \ context\_ok\ \pi\ \land\ (\vdash\ \pi\ p) \implies (\models\ \pi\ p) \]

- induction rule most interesting, Kaufmann&Slind [TPHOLs’07]

Soundness of definition mechanism:

\[ \forall \pi\ name\ formals\ body. \]

context\_ok\ \(\pi\ \land\ \text{definition\_ok}\ (name,\ formals,\ body,\ \pi) \implies\]
context\_ok\ (\(\pi[name \mapsto (formals,\ body,\ \text{new\_interp}\ \pi\ name\ formals\ body])\))

- req. proving that termination conditions imply that a semantic interpretation exists as a function in HOL
This talk

A. formalise Milawa’s logic

B. prove that Milawa's kernel is faithful to the logic

C. connect the verified Lisp implementation
This talk

A. formalise Milawa’s logic

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
Proving Milawa faithful to its logic

Verification must be w.r.t. semantics of Lisp [ITP’11].

Semantics of Lisp’s read-eval-print loop:

1. parse ASCII characters into s-expressions
2. translate s-expressions into program AST
3. evaluate program AST
4. print results, goto 1.

Need to verify program down to concrete source code.
Steps towards an easier verification

- run the Lisp parser (in the logic) on Milawa’s kernel

Each top-level function definition in ASCII

```lisp
(defun lookup-safe (a x)
  (if (consp x)
      (if (equal a (car (car x)))
        (if (consp (car x))
          (car x)
          (cons (car (car x)) (cdr (car x))))
      (lookup-safe a (cdr x)))
  nil))
```

becomes a program AST

```
App Define [Const (Sym "LOOKUP-SAFE"), Const (...), Const (...)]
```
Steps towards an easier verification

When

\[
\text{App Define } [\text{Const (Sym "LOOKUP-SAFE" )}, \text{Const (...)}, \text{Const (...)}]
\]

is evaluated, the op. sem. adds a definition to its context:

- function name: "LOOKUP-SAFE"
- parameter list: "A", "X"
- function body: If (App (PrimitiveFun Consp) [Var "X"])
  \(\) (If (App (PrimitiveFun Equal) [...] )
  \(\) (If (App (PrimitiveFun Consp) [...] (...) ()))
  \(\) (App (Fun "LOOKUP-SAFE") [...]))
  \(\) (Const (Sym "NIL"))

We could do verification over this deep embedding.

...but a shallow embedding is easier to work with.
Steps towards an easier verification

function name: "LOOKUP–SAFE"
parameter list: "A", "X"
function body: If (App (PrimitiveFun Consp) [Var "X"])
   (If (App (PrimitiveFun Equal) [...]))
   (If (App (PrimitiveFun Consp) [...] (...) (...))
      (App (Fun "LOOKUP–SAFE") [...])))
   (Const (Sym "NIL"))

We translate deep embedding into convenient shallow emb. [ITP’12]

lookup_safe a x = if consp x then
   if a = car (car x) then
      if consp (car x) then
         car x
      else cons (car (car x)) (cdr (car x))
   else lookup_safe a (cdr x)
else Sym "NIL"
Steps towards an easier verification

We translate deep embedding into convenient shallow emb.

\[ \text{lookup\_safe } a \; x = \begin{cases} \text{if consp } x \text{ then} & \begin{cases} \text{if } a = \text{car } (\text{car } x) \text{ then} & \begin{cases} \text{if consp } (\text{car } x) \text{ then} & \text{car } x \\ \text{else cons } (\text{car } (\text{car } x)) (\text{cdr } (\text{car } x)) & \end{cases} \\ \text{else lookup\_safe } a (\text{cdr } x) \\ \text{else Sym } "\text{NIL}" \end{cases} \end{cases} \]

and produce a certificate theorem relating the deep and shallow embeddings.

\[ \ldots \rightarrow (\text{Fun } "\text{LOOKUP\_SAFE}",[a,x],\text{state}) \xrightarrow{\text{ap}} (\text{lookup\_safe } a \; x,\text{state}) \]
Verification proof

prove that Milawa’s (reflective) kernel is faithful to logic

A routine verification exercise.

Points of interest:

Milawa’s initial proof checker was a large function

Top-level loop has complicated invariant, relates:

- program state
- current Lisp op.sem. state
- logical context

Bugs found? Yes, two very minor (no soundness bugs)
Verification proof

Theorem:

\[ \exists \text{ans \, k \, output \, ok}. \]
\[ \text{milawa\_main \, \texttt{cmds \ init\_state} = (\text{ans}, (k, output, ok))} \land \]
\[ ( \text{ok} \implies (\text{ans} = \text{Sym \ "SUCCESS"}) \land \]
\[ \text{let result} = \text{compute\_output \ \texttt{cmds}} \text{ in} \]
\[ \text{every\_line \ line\_ok \ result} \land \]
\[ \text{output} = \text{output\_string \ result} \]

where

\[ \text{line\_ok} (\pi, l) = (l = \text{"NIL"}) \lor \]
\[ (\exists n. (l = \text{"(PRINT \ (n \ \ldots \ )")}) \land \text{is\_number \ n} \lor \]
\[ (\exists \phi. (l = \text{"(PRINT \ (THEOREM \ \phi)")}) \land \text{context\_ok} \ \pi \land \models_{\pi} \ \phi) \]
This talk

A. formalise Milawa’s logic

B. prove that Milawa’s kernel is faithful to the logic

C. connect the verified Lisp implementation
Correctness of Jitawa Lisp [ITP’11]

Top-level correctness theorem:

\[
\begin{align*}
\{ \text{init\_state } & \text{input} \ast \text{pc } \text{pc} \ast \langle \text{terminates\_for \ input} \rangle \} \\
\text{pc} &: \text{code\_for\_entire\_jitawa\_implementation} \\
\{ \text{error\_message} \lor \exists \text{output}. & \langle []', \text{input} \rangle \xrightarrow{\text{exec}} \langle \text{output}', \text{true} \rangle \ast \text{final\_state } \text{output} \}
\end{align*}
\]
Correctness of Jitawa Lisp [ITP’11]

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

Each execution is allowed to fail with an error message.

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

If there is no error message, then the result is described by the high-level op. semantics.
There must be enough memory and input is Milawa’s kernel followed by call to main for some input.

\( \forall \text{input } pc. \)

\[
\{ \text{init\_state (milawa\_implementation ++ "}(\text{milawa\_main 'input}))) * pc pc \}
\]

\( pc : \text{code\_for\_entire\_jitawa\_implementation} \)

\[
\{ \text{error\_message } \lor (\text{let result = compute\_output (parse input) in} \\
\langle \text{every\_line line\_ok result} \rangle \ast \langle \text{final\_state (output\_string result ++ "SUCCESS")} \rangle) \}
\]

Machine code terminates either with error message, or …

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

The x86 for the compile function was produced as follows:

- verified compiler as function in logic
- compiled to machine-code
- verified x86

Very cumbersome....

...should have compiled the verified compiler using itself!
Bootstrapping the compiler

Instead: we bootstrap the verified compile function, we evaluate the compiler on a deep embedding of itself within the logic:

```
EVAL "``compile COMPILE``"
```

derives a theorem:

\[
\text{compile COMPILE} = \text{compiler-as-machine-code}
\]

The first(?) bootstrapping of a formally verified compiler.
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

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 the dimensions: one, the compilation
Summary

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 from ITP’11

Questions?