The reflective Milawa theorem prover is sound down to the machine code that runs it.
Soundness
Soundness

We design our provers to be sound.
Soundness

We design our provers to be sound.

We verify programs with them.
Soundness

We design our provers to be sound.

We verify programs with them.

Why not prove the theorem provers sound?
Soundness

We design our provers to be sound.

We verify programs with them.

Why not prove the theorem provers sound?

This talk: explains how soundness was proved for the Milawa theorem prover.
Previous work

A self-verifying theorem prover
Jared Davis — PhD work
Previous work

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

- 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 Milawa sound

- **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**

verification of a Lisp implementation
[ITP’11]
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

verification of a Lisp implementation [ITP’11]
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

This paper [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
Comparison with LCF approach

**LCF-style approach**

- all proofs pass through the core’s primitive inferences
- extensions steer the core
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

work by Jared Davis
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

work by Jared Davis
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
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
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

*work by Jared Davis*
Steps
Steps

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

\[
\begin{align*}
sexp & ::= \text{Val } num \mid \text{Sym } string \mid \text{Dot } sexp sexp & \text{S-expression} \\
prim & ::= \text{If} \mid \text{Equal} \mid \text{Not} \mid \text{Symbolp} \mid \text{Symbol_less} \\
& \quad \mid \text{Natp} \mid \text{Add} \mid \text{Sub} \mid \text{Less} \mid \text{Consp} \mid \text{Cons} \\
& \quad \mid \text{Car} \mid \text{Cdr} \mid \text{Rank} \mid \text{Ord_less} \mid \text{Ordp} \\
func & ::= \text{PrimitiveFun } prim & \text{primitive functions} \\
& \quad \mid \text{Fun } string & \text{user-defined} \\
term & ::= \text{Const } sexp & \text{constant S-expression} \\
& \quad \mid \text{Var } string & \text{variable} \\
& \quad \mid \text{App } func (term \ list) & \text{function application} \\
& \quad \mid \text{LamApp } (string \ list) \ term (term \ list) & \lambda \text{ formals body actuals} \\
formula & ::= \neg \text{formula} & \text{negation} \\
& \quad \mid \text{formula} \lor \text{formula} & \text{disjunction} \\
& \quad \mid \text{term} = \text{term} & \text{term equality}
\end{align*}
\]
Context

Syntax, semantics and inference rules depend on a context.
Context

Syntax, semantics and inference rules depend on a context.

A context is a finite partial map from string to string list × func_body × (sexp list → sexp)
Syntax, semantics and inference rules depend on a context.

A context is a finite partial map from string to string list × func_body × (sexp list → sexp)
Syntax, semantics and inference rules depend on a context.

A context is a finite partial map from string to string list $\times$ func_body $\times$ (sexp list $\rightarrow$ sexp).
Syntax, semantics and inference rules depend on a context.

A context is a finite partial map from string to string list × func_body × (sexp list → sexp)

```
func_body ::= Body term  concrete term (e.g. recursive function)
  | Witness term string   property, element name
  | None                  no function body given
```

parameters

syntax of body
Syntax, semantics and inference rules depend on a context.

A context is a finite partial map from string to string list × func_body × (sexp list → sexp)

Syntax of body

parameters

semantic interpretation

\[
\text{func}_\text{body} ::= \text{Body term} \\
| \text{Witness term string} \\
| \text{None}
\]

can represent:
- concrete term (e.g. recursive function)
- property, element name
- no function body given
Semantics
Semantics

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

$$ \models_{\pi} p = \text{formula\_ok}_{\pi} p \land \forall i. \text{eval\_formula} \ i \ \pi \ p $$

syntax makes sense
Semantics

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

- syntax makes sense
- truth value
Semantics

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

syntax makes sense

truth value

eval\textunderscore formula } i \pi (\lnot p) = \lnot(\text{eval\textunderscore formula } i \pi p)
eval\textunderscore formula } i \pi (p \lor q) = \text{eval\textunderscore formula } i \pi p \lor \text{eval\textunderscore formula } i \pi q

eval\textunderscore formula } i \pi (x = y) = (\text{eval\textunderscore term } i \pi x = \text{eval\textunderscore term } i \pi y)
Semantics

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

**syntax makes sense**

**truth value**

\[
\begin{align*}
\text{eval}_{\text{formula}} i \pi (\neg p) &= \neg(\text{eval}_{\text{formula}} i \pi p) \\
\text{eval}_{\text{formula}} i \pi (p \lor q) &= \text{eval}_{\text{formula}} i \pi p \lor \text{eval}_{\text{formula}} i \pi q \\
\text{eval}_{\text{formula}} i \pi (x = y) &= (\text{eval}_{\text{term}} i \pi x = \text{eval}_{\text{term}} i \pi y) \\
\text{eval}_{\text{term}} i \pi (\text{Const } c) &= c \\
\text{eval}_{\text{term}} i \pi (\text{Var } v) &= i(v) \\
\text{eval}_{\text{term}} i \pi (\text{App } f \; xs) &= \text{eval}_{\text{app}} (f, \text{map} (\text{eval}_{\text{term}} i \pi) \; xs, \pi) \\
\text{eval}_{\text{term}} i \pi (\text{LambdaApp } vs \; x \; xs) &= \text{let } ys = \text{map} (\text{eval}_{\text{term}} i \pi) \; xs \text{ in } \\
&\quad \text{eval}_{\text{term}} [vs \mapsto ys] \pi x
\end{align*}
\]
Semantics

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

**syntax makes sense**

**truth value**

\[
\begin{align*}
\text{eval\_formula}\ i\ \pi\ (\neg p) &= \neg(\text{eval\_formula}\ i\ \pi\ p) \\
\text{eval\_formula}\ i\ \pi\ (p \lor q) &= \text{eval\_formula}\ i\ \pi\ p \lor \text{eval\_formula}\ i\ \pi\ q \\
\text{eval\_formula}\ i\ \pi\ (x = y) &= (\text{eval\_term}\ i\ \pi\ x = \text{eval\_term}\ i\ \pi\ y) \\
\text{eval\_term}\ i\ \pi\ (\text{Const}\ c) &= c \\
\text{eval\_term}\ i\ \pi\ (\text{Var}\ v) &= i(v) \\
\text{eval\_term}\ i\ \pi\ (\text{App}\ f\ xs) &= \text{eval\_app}\ (f, \text{map}\ (\text{eval\_term}\ i\ \pi)\ xs, \pi) \\
\text{eval\_term}\ i\ \pi\ (\text{LambdaApp}\ vs\ x\ xs) &= \text{let}\ ys = \text{map}\ (\text{eval\_term}\ i\ \pi)\ xs\ \text{in}\ \text{eval\_term}\ [vs \mapsto ys]\ \pi\ x \\
\text{eval\_app}\ (\text{PrimitiveFun}\ p,\ args,\ \pi) &= \text{eval\_primitive}\ p\ args \\
\text{eval\_app}\ (\text{Fun}\ name,\ args,\ \pi) &= \text{let}\ (_,\ _,\ interp) = \pi(\text{name})\ \text{in}\ interp(args)
\end{align*}
\]
Semantics

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

**syntax makes sense** **truth value**

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

Semantics only makes sense for well-formed contexts.
Well-formedness of context

Semantics only makes sense for well-formed contexts.

For every entry,

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

it must be that:

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

\[ \forall i. \interp(\text{map } i \text{ formals}) = \text{eval_term } i \pi \text{ body} \]
Well-formedness of context

Semantics only makes sense for well-formed contexts.

For every entry,

$$\pi(name) = (formals, Body \ body, interp)$$

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:

$$\forall i. \ interp(map \ i \ formals) = \ eval\_term \ i \ \pi \ body$$

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

\[
\frac{\vdash_\pi a \lor (b \lor c)}{\vdash_\pi (a \lor b) \lor c} \quad \text{(associativity)}
\]

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

\[
\pi(name) = (formals, \text{Body body, interp}) \quad \vdash_\pi \text{App (Fun name) (map Var formals)} = \text{body}
\]
(a few of the) Inference rules

\[
\begin{align*}
\pi(a \lor (b \lor c)) & \vdash (a \lor b) \lor c \\
\pi(a) & \vdash \pi(a) \\
\pi(name) & = (formals, \text{Body body, interp}) \\
\pi(\text{App (Fun name) (map Var formals)}) & = \text{body}
\end{align*}
\]
(a few of the) Inference rules

\[
\begin{align*}
& \vdash \pi \ a \lor (b \lor c) \\
& \vdash \pi \ (a \lor b) \lor c \quad \text{(associativity)}
\end{align*}
\]

\[
\begin{align*}
& a \in \text{milawa\_axioms} \\
& \vdash \pi \ a
\end{align*}
\]

(basic axiom)

function definition in context

\[
\begin{align*}
& \pi(name) = (\text{formals}, \text{Body body, interp}) \\
& \vdash \pi \ \text{App (Fun name)} \ (\text{map Var formals}) = \text{body}
\end{align*}
\]

facts about Lisp primitives
(a few of the) Inference rules

\[ \vdash_{\pi} a \lor (b \lor c) \]
\[ \vdash_{\pi} (a \lor b) \lor c \] (associativity)

\[ a \in \text{milawa\_axioms} \]
\[ \vdash_{\pi} a \] (basic axiom)

function definition in context

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

facts about Lisp primitives
(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

body of function

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

defining equation

facts about Lisp primitives
Soundness of logic

Soundness of inference rules:

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

Soundness of inference rules:

\( \forall \pi \ p. \ \text{context}_\text{ok} \ \pi \wedge (\vdash_\pi \ p) \implies (\models_\pi \ p) \)

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

Soundness of inference rules:

$$\forall \pi \ p. \ \text{context\_ok } \pi \land (\vdash \pi \ p) \implies (\models \pi \ p)$$

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

Soundness of definition mechanism:

$$\forall \pi \ \text{name formals body}.$$

$$\text{context\_ok } \pi \land \text{definition\_ok (name, formals, body, } \pi) \implies$$

$$\text{context\_ok (} \pi[\text{name } \mapsto (\text{formals, body, new\_interp } \pi\ \text{name formals body})])$$
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:

\[ \forall \pi \ name \ formals \ body. \]
\[ context\_ok \ \pi \land definition\_ok \ (name, formals, body, \pi) \implies \]
\[ context\_ok \ (\pi[name \mapsto (formals, body, new\_interp \ \pi \ name \ formals \ body)]) \]

- 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

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].
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:
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.
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
Steps towards an easier verification

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

Each top-level function definition in ASCII

(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))
Steps towards an easier verification

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

Each top-level function definition in ASCII

```
(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

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

When

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

is evaluated, the op. sem. adds a definition to its context:
Steps towards an easier verification

When

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

When

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

When

App Define [Const (Sym "LOOKUP−SAFE"), Const (...), 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. [ITP’12]

\[
\text{lookup}\_\text{safe } a \ x = \text{ if consp } x \text{ then }
\begin{align*}
& \quad \text{ if } a = \text{ car (car } x) \text{ then } \\
& \qquad \text{ if consp (car } x) \text{ then } \\
& \quad \quad \text{ car } x \\
& \quad \text{ else cons (car (car } x)) \text{ (cdr (car } x)) \\
& \quad \text{ else lookup}\_\text{safe } a \text{ (cdr } x) \\
& \quad \text{ else Sym "NIL" }
\end{align*}
\]

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

\[
\ldots \implies (\text{Fun "LOOKUP-SAFE", } [a, x], \text{state}) \xrightarrow{\text{ap}} (\text{lookup}\_\text{safe } a \ x, \text{state})
\]
Steps towards an easier verification

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

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

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

\[
\ldots \quad \Rightarrow (\text{Fun "LOOKUP-SAFE", } [a, x, state] \xrightarrow{\text{ap}} (\text{lookup\_safe } a \ x, state)
\]

name in deep embedding
Steps towards an easier verification

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

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

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

\[\ldots \implies \text{(Fun "LOOKUP-SAFE", [a, x], state) } \xrightarrow{\text{ap}} \text{(lookup-safe } a \ x, \text{state)}\]

name in deep embedding  shallow embedding
Steps towards an easier verification

We translate deep embedding into convenient shallow emb.

\[
\text{lookup\_safe}\ a\ x\ =\ \begin{cases} 
\text{if consp}\ x\ \text{then} \\
\quad \begin{cases} 
\text{if } a = \text{car}\ (\text{car}\ x) \text{ then} \\
\quad \begin{cases} 
\text{if consp}\ (\text{car}\ x) \text{ then} \\
\quad \text{car}\ x \\
\quad \text{else cons}\ (\text{car}\ (\text{car}\ x))\ (\text{cdr}\ (\text{car}\ x))
\end{cases}
\end{cases}
\end{cases}
\]

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

\[
\ldots \implies (\text{Fun } "\text{LOOKUP-SAFE}" , [a, x], state) \xrightarrow{ap} (\text{lookup\_safe}\ a\ x, state)
\]
Verification proof

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

A routine verification exercise.
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
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?
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:

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

where

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

\[
\{ \text{init-state } \text{input} \ast \text{pc } \text{pc} \ast \langle \text{terminates_for input} \rangle \} \\
\text{pc} : \text{code_for_entire_jitawa_implementation} \\
\{ \text{error_message} \lor \exists \text{output. } \langle [] , \text{input} \rangle \xrightarrow{\text{exec}} (\text{output}, \text{true}) \rangle \ast \text{final_state } \text{output} \} 
\]
Correctness of Jitawa Lisp [ITP’11]

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

{ init_state input * pc pc * (terminates_for input) }

pc : code_for_entire_jitawa_implementation

{ error_message \lor \exists output. (([], input) \xrightarrow{\text{exec}} (output, true)) * final_state output }
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.

\[
\begin{align*}
\{ \text{init\_state } \text{input} & \ast \text{pc } \text{pc} \ast (\text{terminates\_for } \text{input}) \} \\
\text{pc} : \text{code\_for\_entire\_jitawa\_implementation} \\
\{ \text{error\_message} \lor \exists \text{output. } (\langle [], \text{input} \rangle \xrightarrow{\text{exec}} (\text{output}, \text{true}) \rangle \ast \text{final\_state } \text{output} \} 
\end{align*}
\]
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.

If there is no error message, then the result is described by the high-level op. semantics.
Correctness of Jitawa Lisp [ITP’11]

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

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

\[
\{ \text{init.state } input \star pc \; pc \star \langle \text{terminates for } input \rangle \} \\
\text{pc : code for entire Jitawa implementation} \\
\{ \text{error.message } \lor \exists output. \langle ([], input) \xrightarrow{\text{exec}} (output, \text{true}) \rangle \star \text{final.state } output \}
\]

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.
Correctness of Jitawa Lisp [ITP’11]

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.

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

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

\[
\{ \text{init	extunderscore state } \text{input} \ast \text{pc } \text{pc} \ast \langle \text{terminates	extunderscore for } \text{input} \rangle \} \\
\text{pc} : \text{code	extunderscore for	extunderscore entire	extunderscore jitawa	extunderscore implementation} \\
\{ \text{error	extunderscore message} \lor \exists \text{output}. \langle \langle [], \text{input} \rangle \xrightarrow{\text{exec}} (\text{output}, \text{true}) \rangle \ast \text{final	extunderscore state } \text{output} \} 
\]
Theorem: Milawa is sound down to x86

∀input pc.
{ init_state (milawa_implementation ++ ",(milawa-main 'input)")) * pc pc }
p: 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")) }

7 Quirks, bugs and other points of interest
We ran into some surprises during the proof.

13
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")) }
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 }
pce : 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")) }
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.

\[ \forall \text{input } pc. \]
\[ \{ \text{init_state (milawa_implementation ++ "(milawa-main 'input)")} \} \] \[ \times pc \] \[ pc : \text{code_for_entire_jitawa_implementation} \]
\[ \{ \text{error_message \lor (let result = compute_output (parse input) in} \]
\[ \langle \text{every_line line_ok result} \rangle \] \[ \times \]
\[ \langle \text{final_state (output_string result ++ "SUCCESS")} \rangle \} \] 

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

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

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