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 1: proving the soundness of Milawa

Milawa

A self-verifying theorem prover

Jared Davis — PhD work

Jitawa

verified

LISP

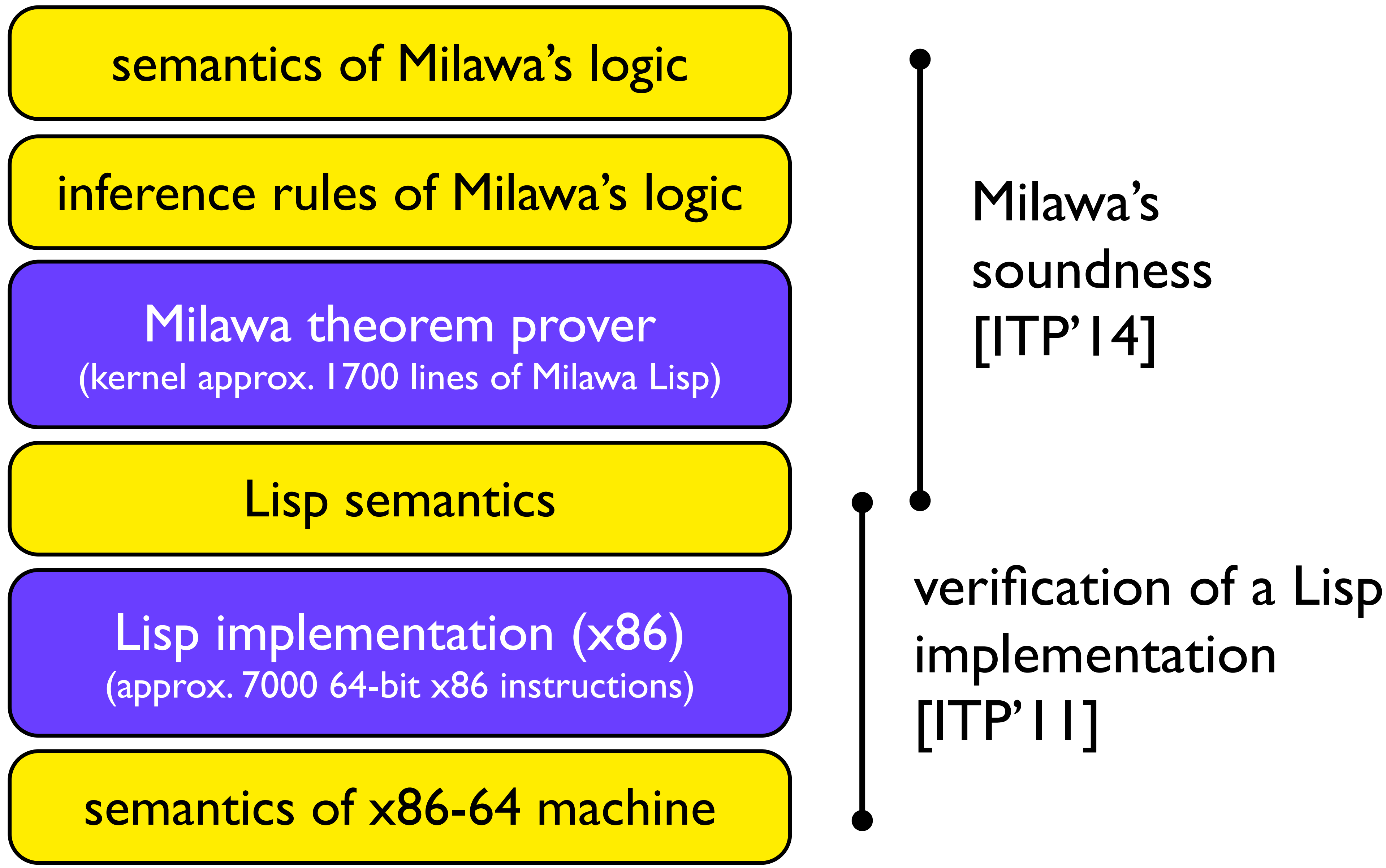
A verified runtime for a verified theorem prover

Magnus Myreen, Jared Davis — ITP'11

Proving Milawa sound

Milawa

Jitawa verified LISP



work by Jared Davis

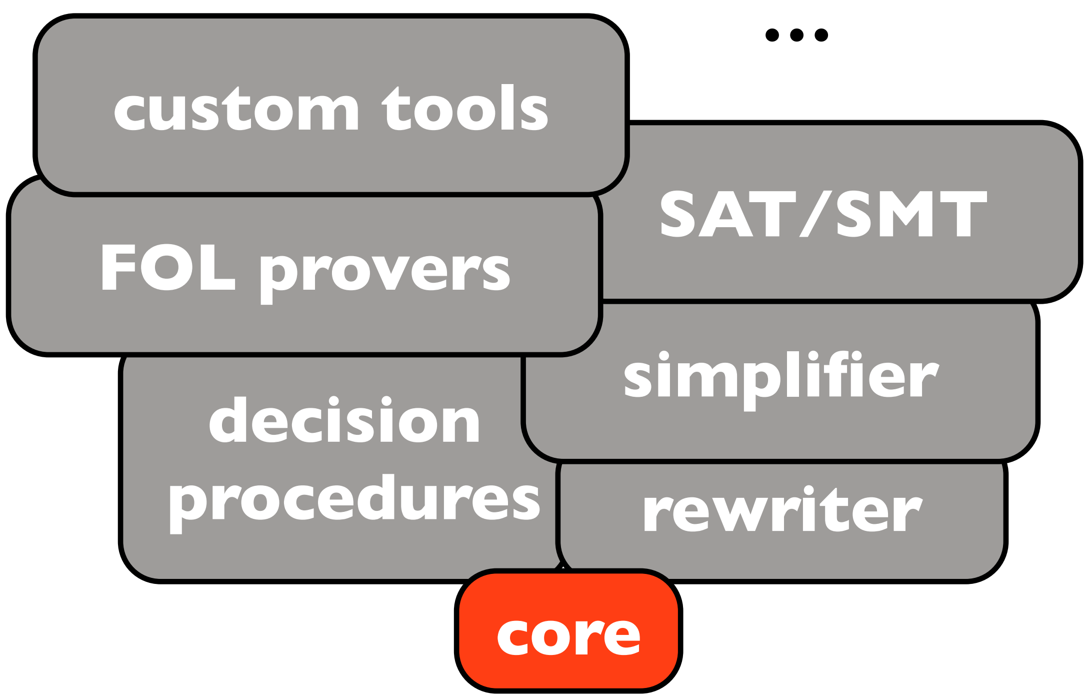
A very short introdution

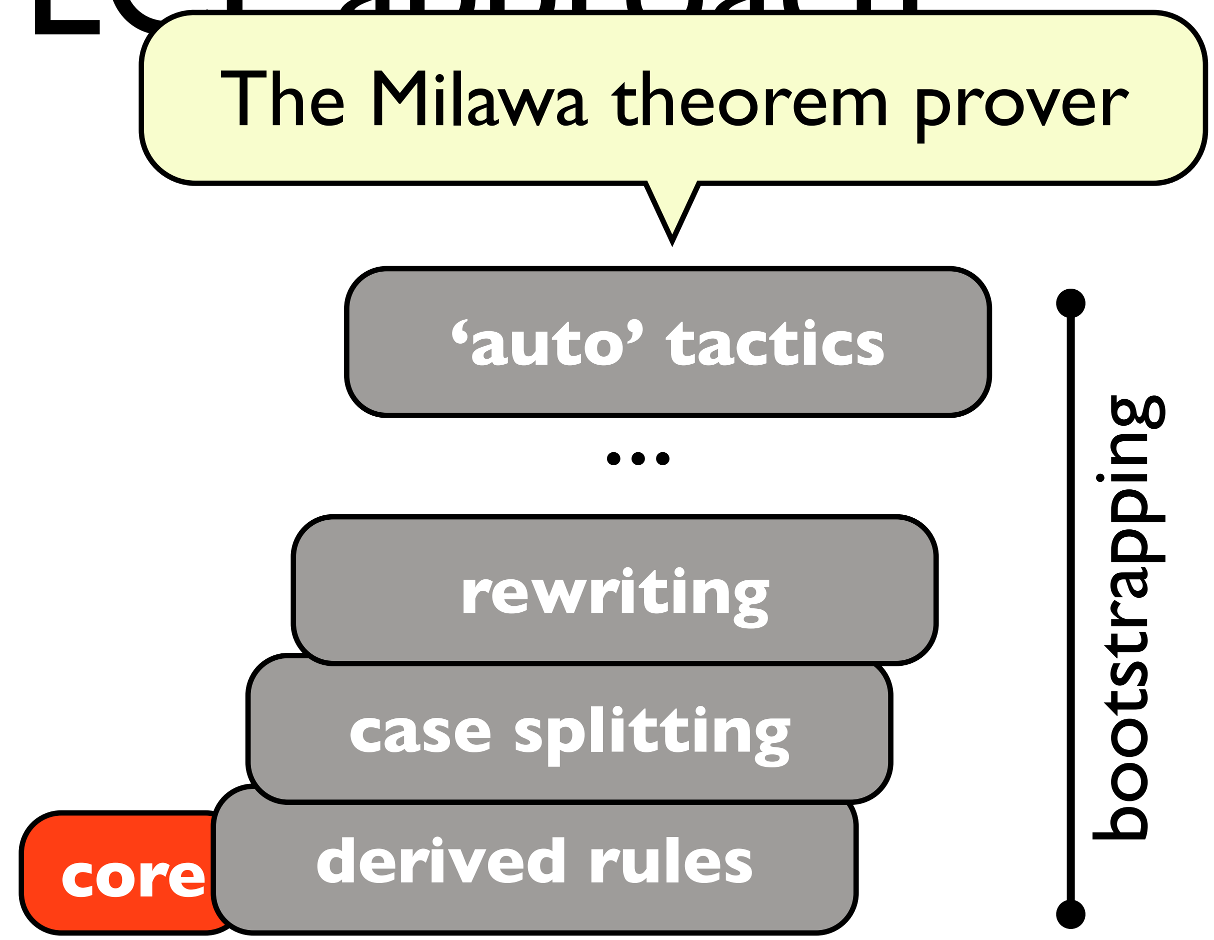
Milawa

- 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.

work by Jared Davis

Comparison with LCF approach





LCF-style approach

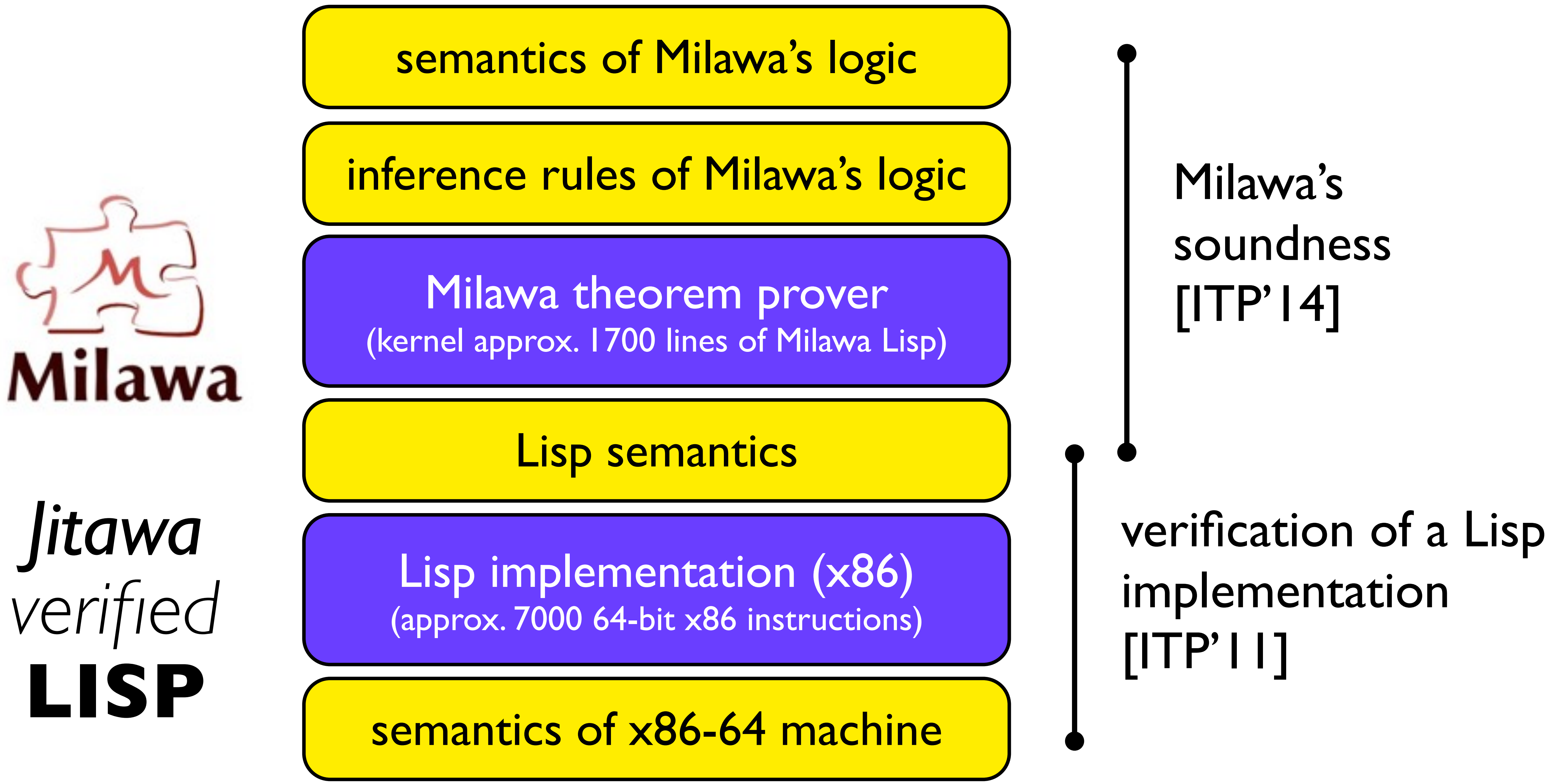
- all proofs pass through the core's primitive inferences
- extensions steer the core

the Milawa approach

- all proofs must pass the core
- the core proof checker can be replaced at runtime

Brief Milawa demo

Proving Milawa sound



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 $sexp$ $sexp$	S-expression
$prim$	$::=$	If | Equal | Not | Symbolp | Symbol_less	
	|	Natp | Add | Sub | Less | Consp | Cons	
	|	Car | Cdr | Rank | Ord_less | Ordp	
$func$	$::=$	PrimitiveFun $prim$	primitive functions
	|	Fun $string$	user-defined
$term$	$::=$	Const $sexp$	constant S-expression
	|	Var $string$	variable
	|	App $func$ ($term$ list)	function application
	|	LamApp ($string$ list) $term$ ($term$ list)	λ formals body actuals
$formula$	$::=$	$\neg formula$	negation
	|	$formula \vee formula$	disjunction
	|	$term$ = $term$	term equality

Context

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 $\rightarrow$ $sexp$)

parameters

semantic interpretation

syntax of body

$func\_body$	::=	Body $term$	concrete term (e.g. recursive function)
	|	Witness $term$ $string$	property, element name
	|	None	no function body given

Semantics

$$(\models_\pi p) \quad = \quad \text{formula\_ok}_\pi\ p\ \wedge\ \forall i.\ \text{eval\_formula}\ i\ \pi\ p$$

syntax makes sense

truth value

$$
\begin{array}{lcl}
\text{eval\_formula}\ i\ \pi\ (\neg p) & = & \neg(\text{eval\_formula}\ i\ \pi\ p) \\
\text{eval\_formula}\ i\ \pi\ (p \vee q) & = & \text{eval\_formula}\ i\ \pi\ p \vee \text{eval\_formula}\ i\ \pi\ q \\
\text{eval\_formula}\ i\ \pi\ (x \mathrel{\text{=}} y) & = & (\text{eval\_term}\ i\ \pi\ x = \text{eval\_term}\ i\ \pi\ y)
\end{array}
$$

$$
\begin{array}{lcl}
\text{eval\_term}\ i\ \pi\ (\text{Const}\ c) & = & c \\
\text{eval\_term}\ i\ \pi\ (\text{Var}\ v) & = & i(v) \\
\text{eval\_term}\ i\ \pi\ (\text{App}\ f\ xs) & = & \text{eval\_app}\ (f, \text{map}\ (\text{eval\_term}\ i\ \pi)\ xs, \pi) \\
\text{eval\_term}\ i\ \pi\ (\text{LambdaApp}\ vs\ x\ xs) & = & \text{let}\ ys = \text{map}\ (\text{eval\_term}\ i\ \pi)\ xs\ \text{in} \\
 & & \quad \text{eval\_term}\ [vs \mapsto ys]\ \pi\ x
\end{array}
$$

$$
\begin{array}{lcl}
\text{eval\_app}\ (\text{PrimitiveFun}\ p, args, \pi) & = & \text{eval\_primitive}\ p\ args \\
\text{eval\_app}\ (\text{Fun}\ name, args, \pi) & = & \text{let}\ (\_, \_, interp) = \pi(name)\ \text{in} \\
 & & \quad interp(args)
\end{array}
$$

$$
\begin{array}{lcl}
\text{eval\_primitive Add}\ [\text{Val}\ 2, \text{Val}\ 3] & = & \text{Val}\ 5 \\
\text{eval\_primitive Add}\ [\text{Val}\ 2, \text{Sym}\ \texttt{"a"}] & = & \text{Val}\ 2 \\
\text{eval\_primitive Cons}\ [\text{Val}\ 2, \text{Sym}\ \texttt{"a"}] & = & \text{Dot}\ (\text{Val}\ 2)\ (\text{Sym}\ \texttt{"a"})
\end{array}
$$

Well-formedness of context

Semantics only makes sense for well-formed contexts.

For every entry,

$$\pi(name) \;=\; (formals, \mathsf{Body}\ body, interp)$$

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:

$$\forall i.\;\; interp(\mathsf{map}\ i\ formals) \;=\; \mathsf{eval\_term}\ i\ \pi\ body$$

Similarly for the other function types.

(a few of the) Inference rules

$$\frac{\vdash_\pi a \lor (b \lor c)}{\vdash_\pi (a \lor b) \lor c} \text{ (associativity)}$$

facts about Lisp primitives

$$\frac{a \in \mathsf{milawa\_axioms}}{\vdash_\pi a} \text{ (basic axiom)}$$

function definition in context

body of function

$$\frac{\pi(name) = (formals, \mathsf{Body}\ body, interp)}{\vdash_\pi \mathsf{App}\ (\mathsf{Fun}\ name)\ (\mathsf{map\ Var}\ formals)\ \mathsf{=}\ body}$$

defining equation

Soundness of logic

Soundness of inference rules:

$$\forall \pi\ p.\ \ \textsf{context\_ok}\ \pi \wedge (\vdash_\pi p) \Longrightarrow (\models_\pi p)$$

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

Soundness of definition mechanism:

$$\begin{array}{l}\forall \pi\ name\ formals\ body. \\ \quad \textsf{context\_ok}\ \pi \wedge \textsf{definition\_ok}\ (name, formals, body, \pi)\ \Longrightarrow \\ \quad \textsf{context\_ok}\ (\pi[name \mapsto (formals, body, \textsf{new\_interp}\ \pi\ name\ formals\ body)])\end{array}$$

- 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

```
(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

$\mathsf{App\ Define}\ [\mathsf{Const}\ (\mathsf{Sym}\ \texttt{"LOOKUP-SAFE"}), \mathsf{Const}\ (\ldots), \mathsf{Const}\ (\ldots)]$

Steps towards an easier verification

When

$$\mathsf{App\ Define}\ [\mathsf{Const}\ (\mathsf{Sym}\ \texttt{"LOOKUP-SAFE"}), \mathsf{Const}\ (\ldots), \mathsf{Const}\ (\ldots)]$$

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

$$\begin{array}{rl}
\text{function name:} & \texttt{"LOOKUP-SAFE"} \\
\text{parameter list:} & \texttt{"A"}, \texttt{"X"} \\
\text{function body:} & \mathsf{If}\ (\mathsf{App}\ (\mathsf{PrimitiveFun\ Consp})\ [\mathsf{Var}\ \texttt{"X"}]) \\
& \quad (\mathsf{If}\ (\mathsf{App}\ (\mathsf{PrimitiveFun\ Equal})\ [\ldots]) \\
& \quad\quad (\mathsf{If}\ (\mathsf{App}\ (\mathsf{PrimitiveFun\ Consp})\ [\ldots]\ (\ldots)\ (\ldots)) \\
& \quad\quad (\mathsf{App}\ (\mathsf{Fun}\ \texttt{"LOOKUP-SAFE"})\ [\ldots])) \\
& \quad (\mathsf{Const}\ (\mathsf{Sym}\ \texttt{"NIL"}))
\end{array}$$

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]

$$
\begin{array}{rcl}
\text{lookup\_safe}\ a\ x & = & \text{if consp}\ x\ \text{then} \\
& & \quad \text{if}\ a = \text{car}\ (\text{car}\ x)\ \text{then} \\
& & \quad\quad \text{if consp}\ (\text{car}\ x)\ \text{then} \\
& & \quad\quad\quad \text{car}\ x \\
& & \quad\quad \text{else cons}\ (\text{car}\ (\text{car}\ x))\ (\text{cdr}\ (\text{car}\ x)) \\
& & \quad \text{else lookup\_safe}\ a\ (\text{cdr}\ x) \\
& & \text{else Sym}\ \texttt{"NIL"}
\end{array}
$$

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

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

name in deep embedding

Lisp semantics

shallow embedding

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:

$$\begin{array}{l}\exists ans\ k\ output\ ok.\\ \quad \text{milawa\_main}\ cmds\ \text{init\_state} = (ans, (k, output, ok))\ \wedge\\ \quad (ok \Longrightarrow (ans = \text{Sym}\ \texttt{"SUCCESS"})\ \wedge\\ \qquad\qquad \text{let}\ result = \text{compute\_output}\ cmds\ \text{in}\\ \qquad\qquad\quad \text{every\_line line\_ok}\ result\ \wedge\\ \qquad\qquad\quad output = \text{output\_string}\ result)\end{array}$$

where

$$\begin{array}{lcl}\text{line\_ok}\ (\pi, l) & = & (l = \texttt{"NIL"})\ \vee\\ & & (\exists n.\ (l = \texttt{"(PRINT (}n\ \ldots\ \texttt{))"})\wedge \text{is\_number}\ n)\ \vee\\ & & (\exists \phi.\ (l = \texttt{"(PRINT (THEOREM }\phi\texttt{))"})\wedge \text{context\_ok}\ \pi \wedge \models_\pi \phi)\end{array}$$

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:

$$\{\,\mathsf{init\_state}\ input * \mathsf{pc}\ pc * \langle \mathsf{terminates\_for}\ input \rangle\,\}$$
$$pc : \mathsf{code\_for\_entire\_jitawa\_implementation}$$
$$\{\,\mathsf{error\_message} \vee \exists output.\ \langle ([], input) \xrightarrow{\mathsf{exec}} (output, \mathsf{true}) \rangle * \mathsf{final\_state}\ output\,\}$$

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.

$$\{\,\mathsf{init\_state}\ \mathit{input} * \mathsf{pc}\ \mathit{pc} * \langle \mathsf{terminates\_for}\ \mathit{input}\rangle\,\}$$
$$\mathit{pc} : \mathsf{code\_for\_entire\_jitawa\_implementation}$$
$$\{\,\mathsf{error\_message} \lor \exists \mathit{output}.\ \langle([\,], \mathit{input}) \xrightarrow{\mathsf{exec}} (\mathit{output}, \mathsf{true})\rangle * \mathsf{final\_state}\ \mathit{output}\,\}$$

list of numbers

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.

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*.

$$\forall input\ pc.$$

$$\{\,\text{init\_state}\ (\text{milawa\_implementation} \mathbin{+\!\!+} \texttt{"(milawa-main '}\mathit{input}\texttt{)"}) * \text{pc}\ pc\,\}$$

$$pc : \text{code\_for\_entire\_jitawa\_implementation}$$

$$\begin{aligned}\{\,\text{error\_message} \lor (&\text{let } result = \text{compute\_output}\ (\text{parse}\ input)\ \text{in}\\ &\langle \text{every\_line line\_ok}\ result\rangle\ *\\ &\text{final\_state}\ (\text{output\_string}\ result \mathbin{+\!\!+} \texttt{"SUCCESS"}))\,\}\end{aligned}$$

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 booststrap process:

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

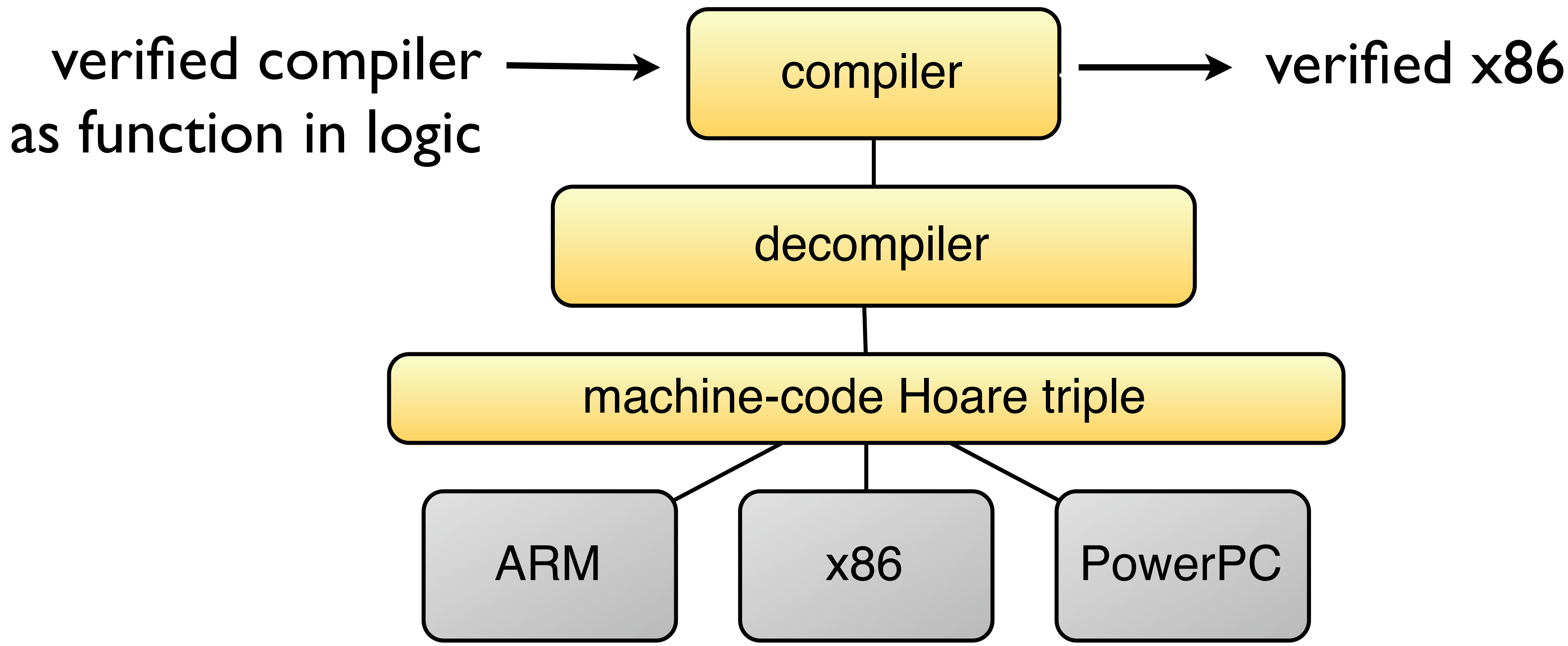
Jitawa's compiler performs almost no optimisations.

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:



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``

in Lisp (eval '(compile compile)) ?

derives a theorem:

compile COMPILE = compiler-as-machine-code

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

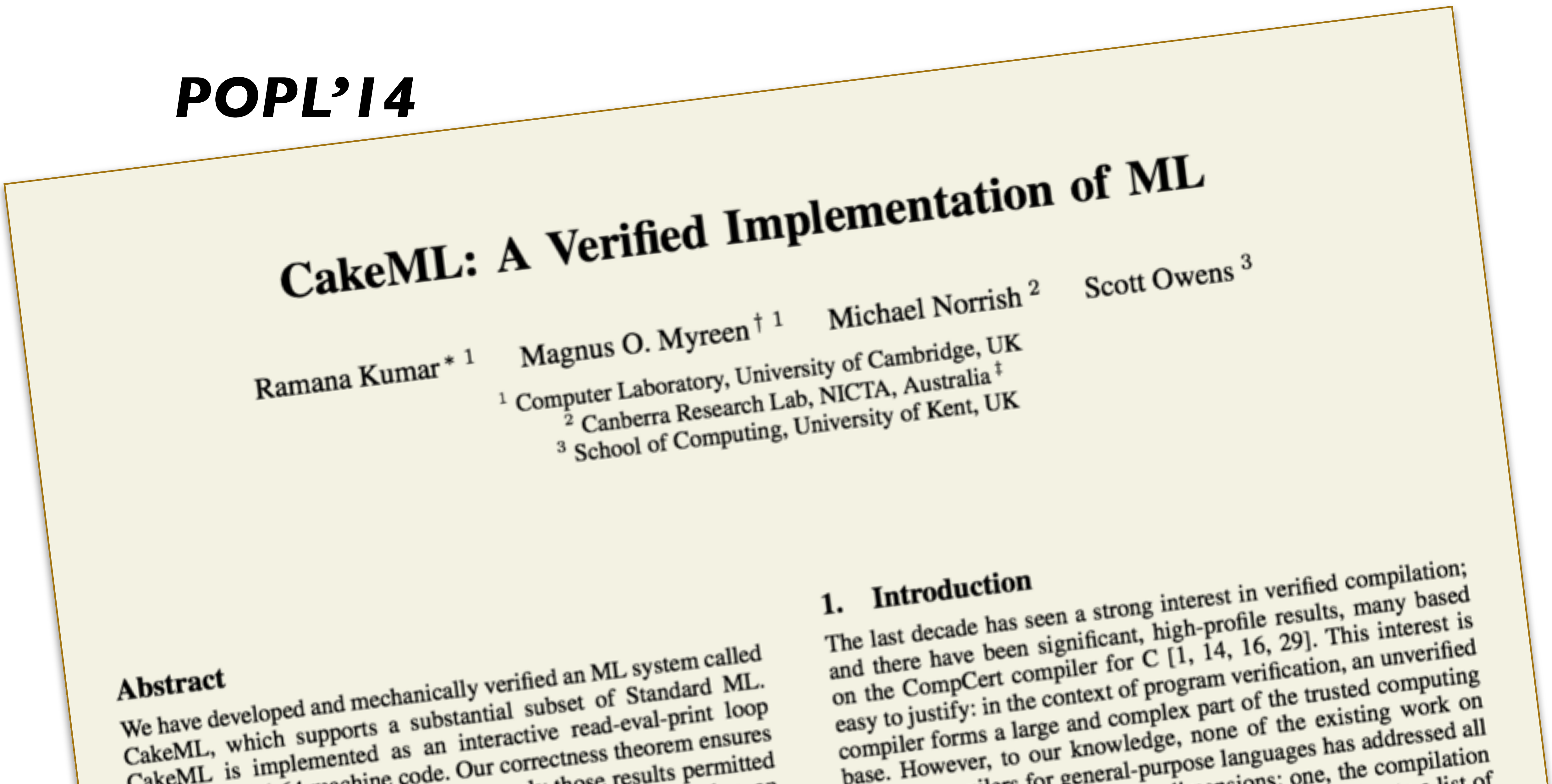
Ramana Kumar
(Uni. Cambridge)

Magnus Myreen
(Uni. Cambridge)

Michael Norrish
(NICTA, ANU)

Scott Owens
(Uni. Kent)

POPL'14



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

Abstract

We have developed and mechanically verified an ML system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop ...

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 ...

Summary

The top-level theorem:

relates the logic's semantics
with the execution of the x86 machine code.

Questions?

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