

# *Computational Reflection, Theorem Provers and Verification*

## Lecture 2:

### *Milawa: A Reflective Theorem Prover*

---

Milawa: A Reflective Theorem Prover

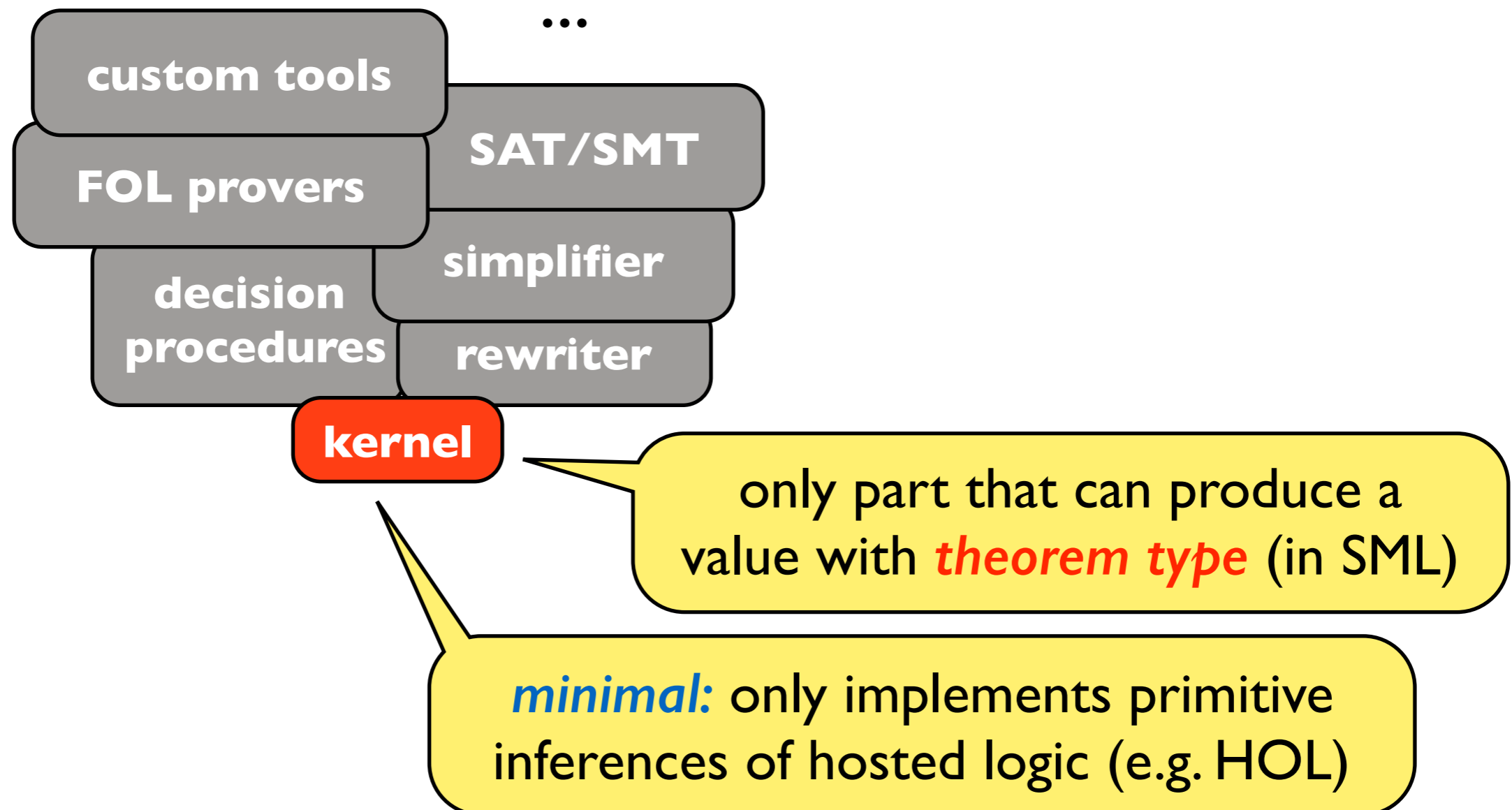
Marktoberdorf Summer School MOD 2019

Magnus O. Myreen, Chalmers University of Technology

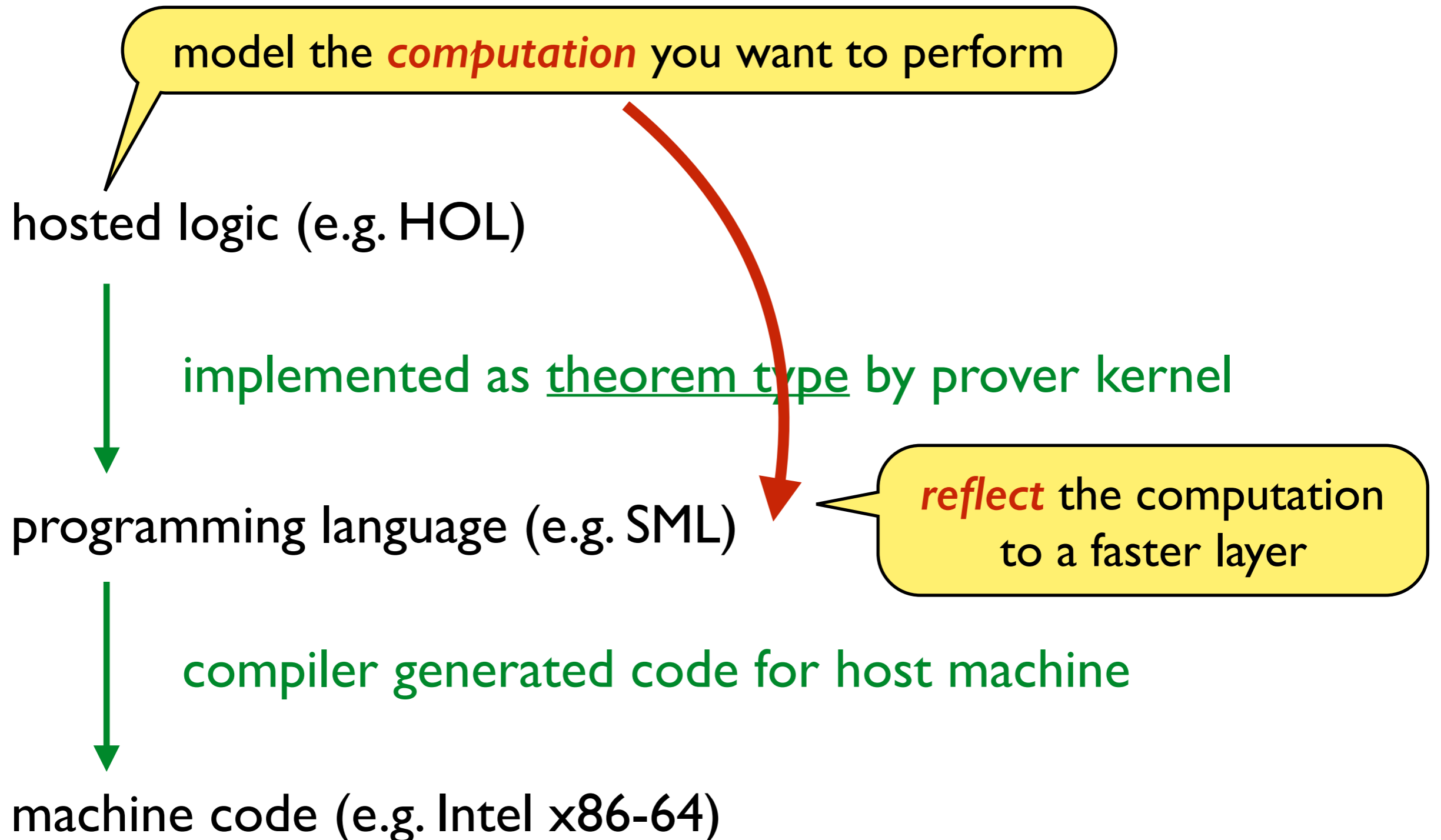
From yesterday

# Soundness-centric design

## LCF-style prover design:



# Idea for going fast



# Today's lecture: Milawa

Milawa designed to have best of both worlds:

- ✓ Soundness story as strong as LCF-style provers
- ✓ ... and at the same time: speed of non-LCF-style kernel implementations

Milawa has a logic similar to ACL2 — motivation is to show ACL2 could have LCF-style soundness story.

---

Today: *Part 1:* what is Milawa

*Part 2:* proving Milawa sound in HOL4

# Milawa is Jared Davis' PhD work



Check out his 550-page PhD thesis:

*A Self-Verifying Theorem Prover.*

The University of Texas at Austin.  
December, 2009

<https://www.kookamara.com/jared/dissertation.pdf>

<https://www.kookamara.com/jared/>

# Tutorial on Lisp

# Lisp

Values in Lisp are called s-expressions:

*In the Lisp we consider here, s-expressions are one of:*

*a natural number*

e.g. 0 1 500

*a symbol (think of this as a string)*

e.g. a hello nil + natp ord-<

*an ordered pair of s-expressions*

e.g. (a . b)

# More Lisp syntax

*List syntax such as*

```
(a b c)
```

*is short for:*

```
(a . (b . (c . nil)))
```

*Lisp quotes*

```
'54
```

*is short for*

```
(quote 54)
```

*which is short for* (quote . (54 . nil))

# Function applications

*Functions applications always written in prefix-form*

```
(defun fib (n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
          (fib (- n 2))))))
```

# Example: testing list membership

## *Use of car and cdr*

```
(defun memp (a xs)
  (if (consp xs)
      (or (equal a (car xs))
          (memp a (cdr xs)))
      'nil))
```

convention to have suffix p on functions that return true (t) or false (nil)

car returns first part of cons pair

cdr returns first part of cons pair

returns quoted nil because want to return nil rather than evaluate nil

# More Lisp

*We use Lisp with primitives:*

CONS CAR CDR CONSP NATP SYMBOLP EQUAL  
+ - < SYMBOL-< IF OR

*Macros:*

AND LIST LET LET\* COND FIRST SECOND  
THIRD FOURTH FIFTH DEFUN



note that defun is a macro

*And special functions:*

DEFINE PRINT ERROR FUNCALL

# Defun and Define

*Common definition pattern:*

```
(defun inc (n) (+ n 1))
```

*... is a macro for:*

```
(define 'inc '(n) '(+ n 1))
```

... and body!

parameters, ...

note that if we leave off the quotes then we can *dynamically* provide the function name, ...

# Dynamic function calls: Funcall

This Lisp has *no function values*.

However, it has dynamic function calls:

```
> (defun inc (n) (+ n 1))
```

```
NIL
```

```
> (funcall 'inc 5)
```

```
6
```

```
> (funcall (car '(inc dec)) 5)
```

```
6
```

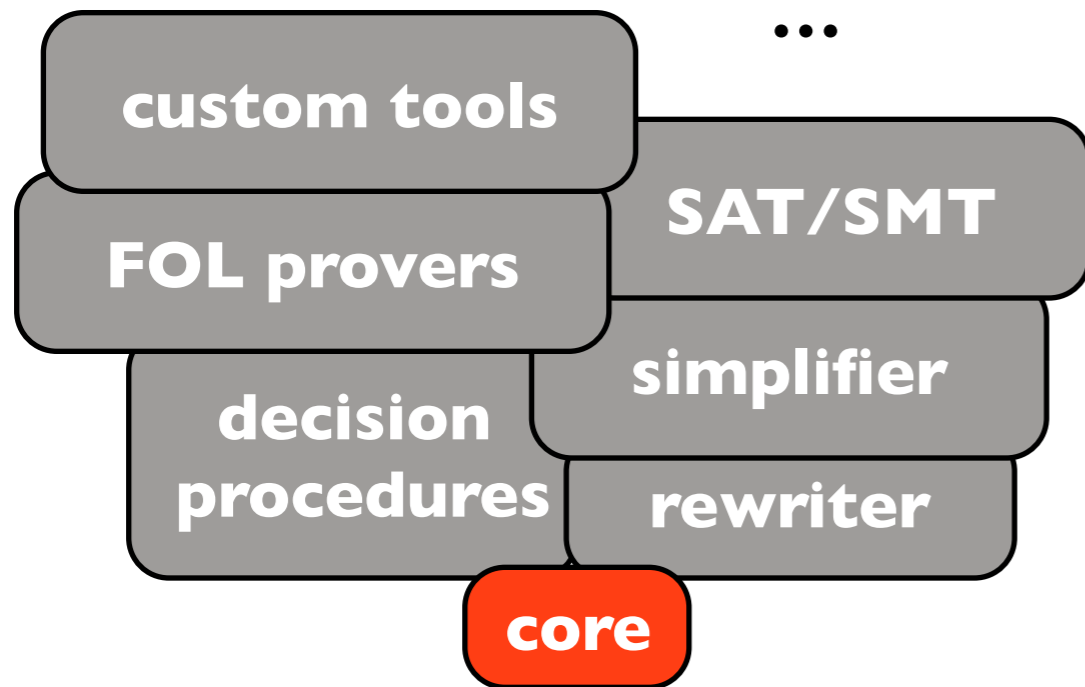


we can compute which function to call



Milawa

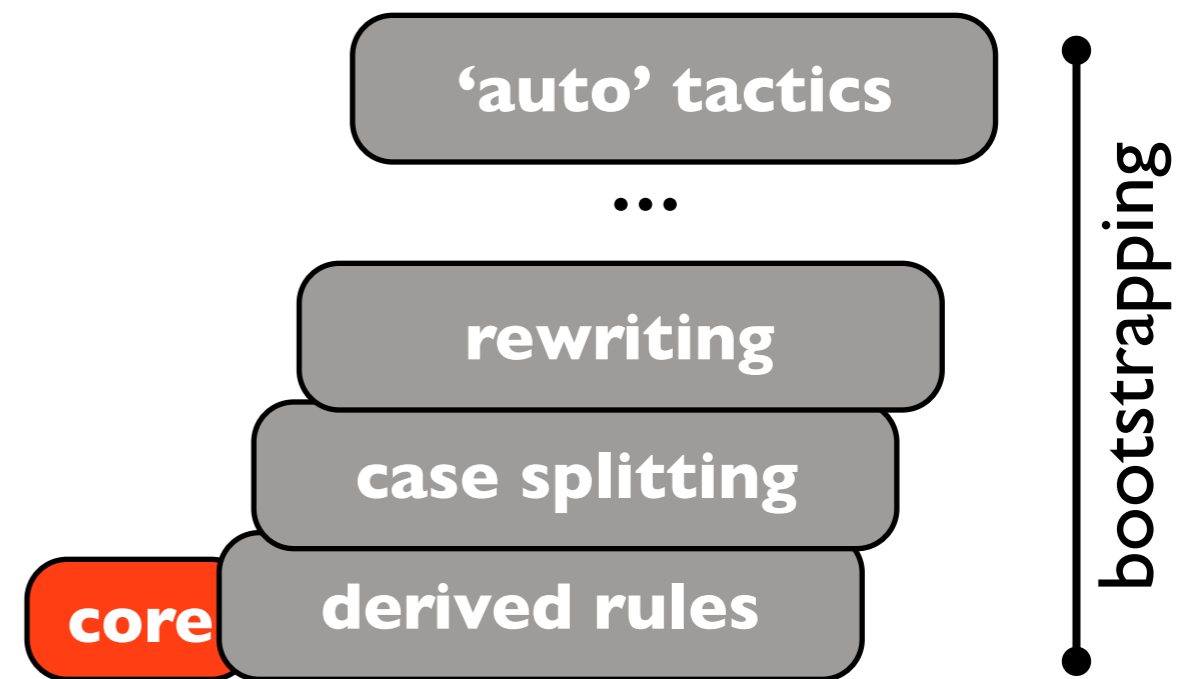
# Comparison with LCF approach



## LCF-style approach

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

This stack is called Milawa



## the Milawa approach

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

# Bootstrapping Milawa

Output from Milawa's bootstrap proof

starts with very basic definitions and lemmas

```
(PRINT (1 VERIFY THEOREM-SUBSTITUTE-INTO-NOT-PEQUAL))
```

```
(PRINT (2 VERIFY THEOREM-NOT-T-OR-NOT-NIL))
```

```
(PRINT (3 DEFINE NOT))
```

```
(PRINT (4 VERIFY NOT))
```

```
(PRINT (5 DEFINE IF))
```

```
(PRINT (6 VERIFY IF))
```

```
(PRINT (7 VERIFY THEOREM-NOT-T-OR-NOT-NIL))
```

```
...
```

```
(PRINT (4611 VERIFY (INSTALL-NEW-PROOFP-LEVEL2.PROOFP I))
```

```
(PRINT (4612 SWITCH (LEVEL2.PROOFP I))
```

```
(PRINT (4613 VERIFY (BUST-UP-LOGIC.FUNCTION-ARGS-EXPENSIVE I))
```

```
...
```

```
(PRINT (15685 VERIFY (INSTALL-NEW-PROOFP-LEVEL11.PROOFP I))
```

```
(PRINT (15686 SWITCH (LEVEL11.PROOFP I))
```

```
...
```

```
SUCCESS
```

up to this point the original core is used

this event switches to a new extended core

the extended core is used from now onwards

10 core extensions during bootstrap

# Running Milawa

Demo of the Jitawa Lisp implementation running Milawa's first 100 events.

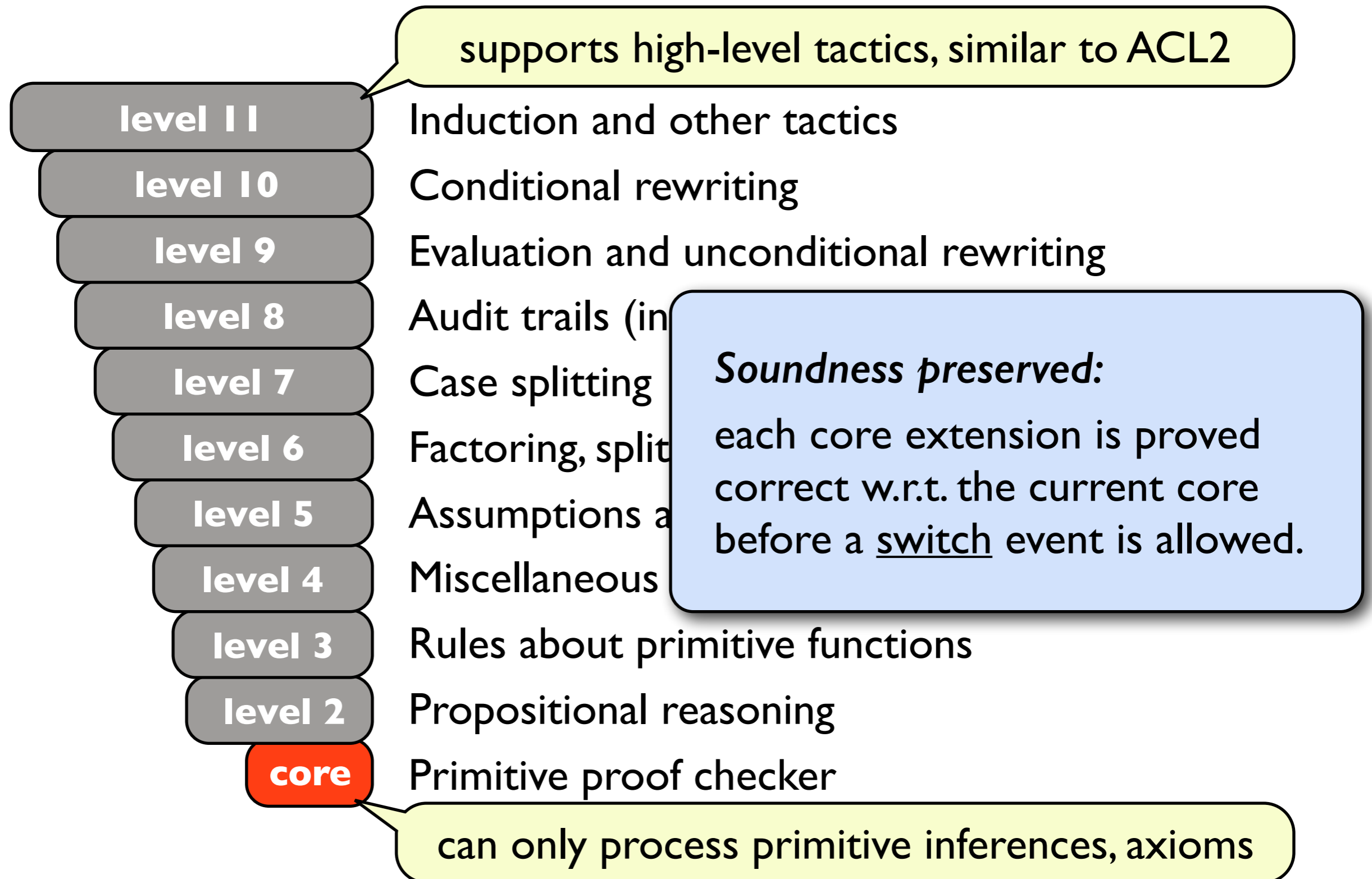
Download Jitawa from:

<https://www.cl.cam.ac.uk/~mom22/jitawa/>

Some documentation about the Lisp language:

<https://www.cl.cam.ac.uk/~mom22/jitawa/jitawa-manual.html>

# Milawa's core extensions



# Basic architecture

Milawa processes *a list of events*.

*Each event* is one of:

`(verify name formula proof)`

Prove a theorem.

`(def name formals body measure proofs)`

Define a recursive function and prove it terminates.

`(witness name bound-var free-vars body)`

Define a witness function (to emulate a quantifier).

`(switch name)`

Use a new proof-checking function

checks proofs using *current*  
proof-checking function

changes the current proof-checking function

# Proof checker functions

They walk an s-expression representing a proof tree.

The function returns a boolean (in Lisp: `t` or `nil`).

*The initial proof checker* is called `| LOGIC.PROOFPP |`

It requires each step of in the tree to be a primitive inference step.

*Subsequent proof checkers* allow longer leaps.

# Milawa's logic

more details on later slides in Part 2

Prop. Schema  $\frac{}{\neg A \vee A}$

Contraction  $\frac{A \vee A}{A}$

Expansion  $\frac{A}{B \vee A}$

Associativity  $\frac{A \vee (B \vee C)}{(A \vee B) \vee C}$

Cut  $\frac{A \vee B \quad \neg A \vee C}{B \vee C}$

Instantiation  $\frac{A}{A/\sigma}$

w.r.t. ordinals up to  $\epsilon_0$

Induction

Reflexivity Axiom

$$x = x$$

Equality Axiom

$$x_1 = y_1 \rightarrow x_2 = y_2 \rightarrow x_1 = x_2 \rightarrow y_1 = y_2$$

Referential Transparency

$$x_1 = y_1 \rightarrow \dots \rightarrow x_n = y_n \rightarrow f(x_1, \dots, x_n) = f(y_1, \dots, y_n)$$

Beta Reduction

$$((\lambda x. x \beta) t) \equiv \beta / [x \leftarrow t]$$

evaluation of any lisp primitive applied to constants

Base Evaluation

$$\text{e.g., } 1 + 2 = 3$$

Lisp Axioms

$$\text{e.g., } \text{consp}(\text{cons}(x, y)) = t$$

56 axioms describing properties of Lisp primitives

# Switch command

User provides name of new proof checker.

Switches to a new proof checker.



soundness critical!

# Switch command (cont.)

Before switching to use `new-proofp` as the new checker, *Milawa checks that user has proved a theorem* of the form:

for any proof accepted by the new checker ...

$\forall x, \text{axioms}, \text{thms}, \text{atbl} :$

`(new-proofp  $x$  axioms thms atbl)`

$\implies$

$\left( \begin{array}{l} \exists p : (\text{logic.appealp } p) \\ \quad \wedge (\text{logic.proofp } p \text{ } \textit{axioms} \textit{ thms} \textit{ atbl}) \\ \quad \wedge (\text{logic.conclusion } p) = (\text{logic.conclusion } x) \end{array} \right)$

... there must exist a proof  $p$  that passes the initial proof checker `logic.proofp`

# Switch command (cont.)

**Actual check is closer to:**

$\forall \phi, \text{axioms}, \text{thms}, \text{atbl} :$

$(\text{logic.provablep } \phi \text{ } \text{axioms thms atbl})$

$=$

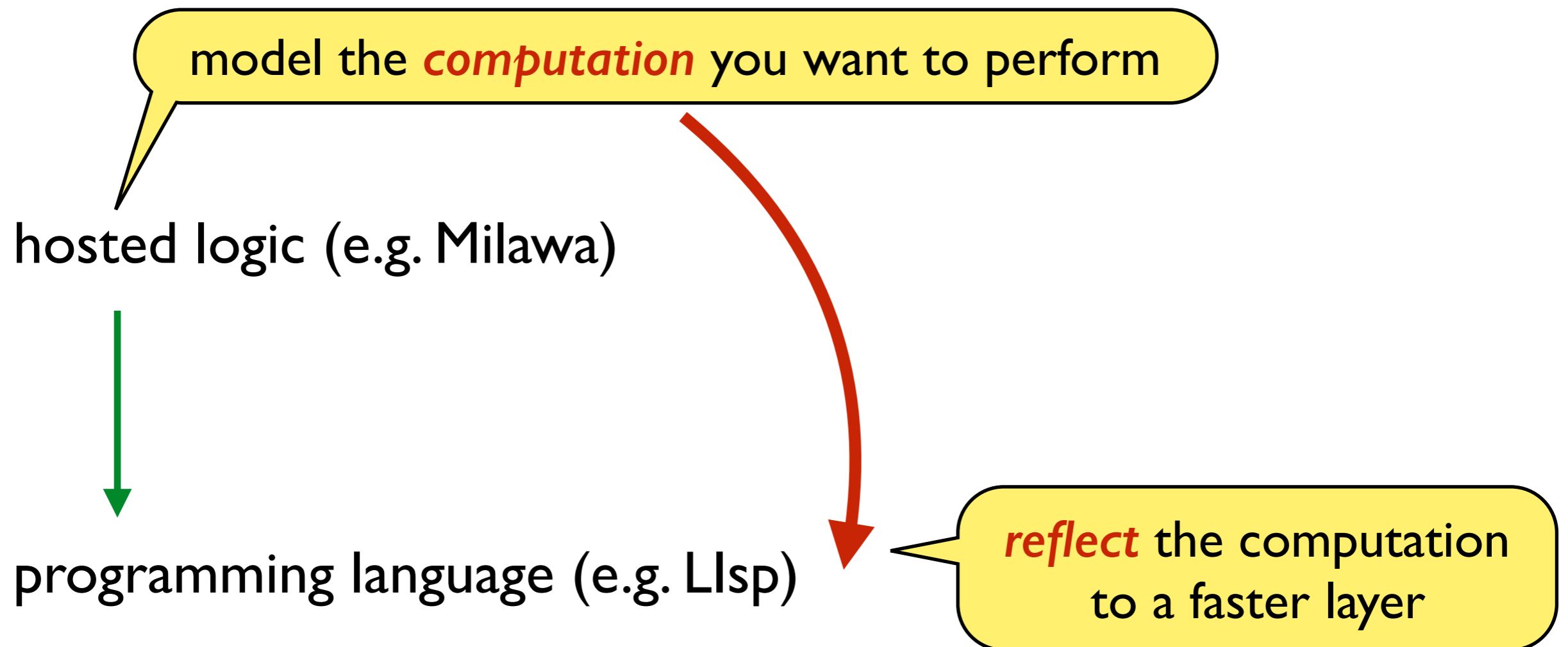
let  $wit = (\text{logic.provable-witness } \phi \text{ } \text{axioms thms atbl})$

in

$\left( \begin{array}{l} (\text{logic.appealp } wit) \\ \wedge (\text{logic.proofp } wit \text{ } \text{axioms thms atbl}) \\ \wedge (\text{logic.conclusion } wit) = \phi \end{array} \right)$

# Where's the reflection?

**Note:** user provides *logic function* for checking proofs and Milawa runs the function *outside of the logic*.



# Browsing the code of Milawa

The core is here:

<https://github.com/HOL-Theorem-Prover/HOL/blob/develop/examples/theorem-prover/milawa-prover/core.lisp>

Points of interest:

- ▶ The main function `milawa-main` at the bottom
- ▶ Milawa defines its (initial) proof checker *in both the hosted logic and in the Lisp programming language outside the logic.*
- ▶ The implementation of `|CORE.ADMIT-SWITCH|`

# Proving Milawa sound

We design our provers to be sound.

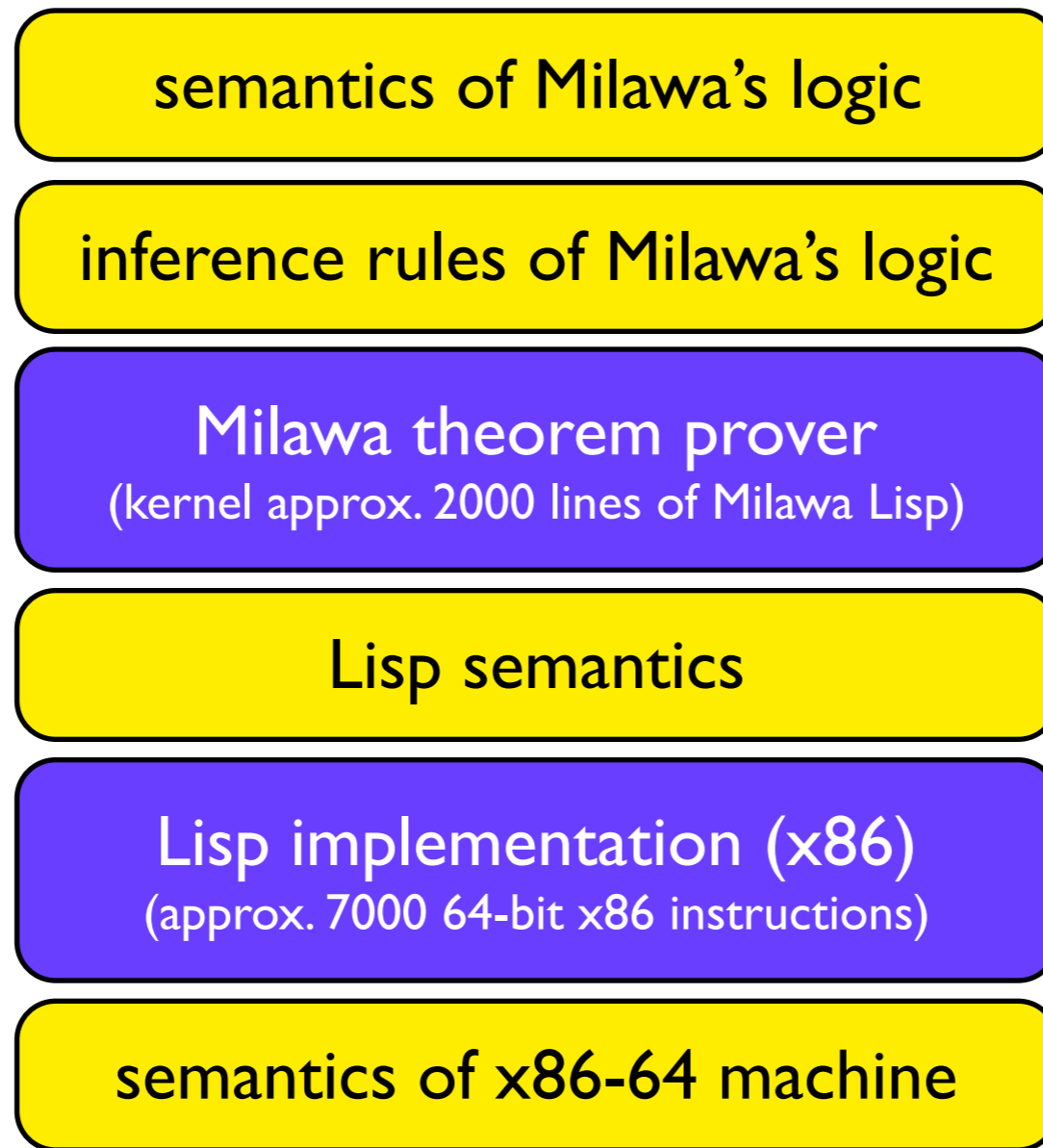
We verify programs with them.

Why not prove the theorem provers sound?

# Proving Milawa sound



*Jitawa*  
*verified*  
**LISP**



Correctness of  
Milawa prover  
(rest of lecture)

Correctness of  
Lisp system  
(next lecture)

# 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 of Lisp impl.

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

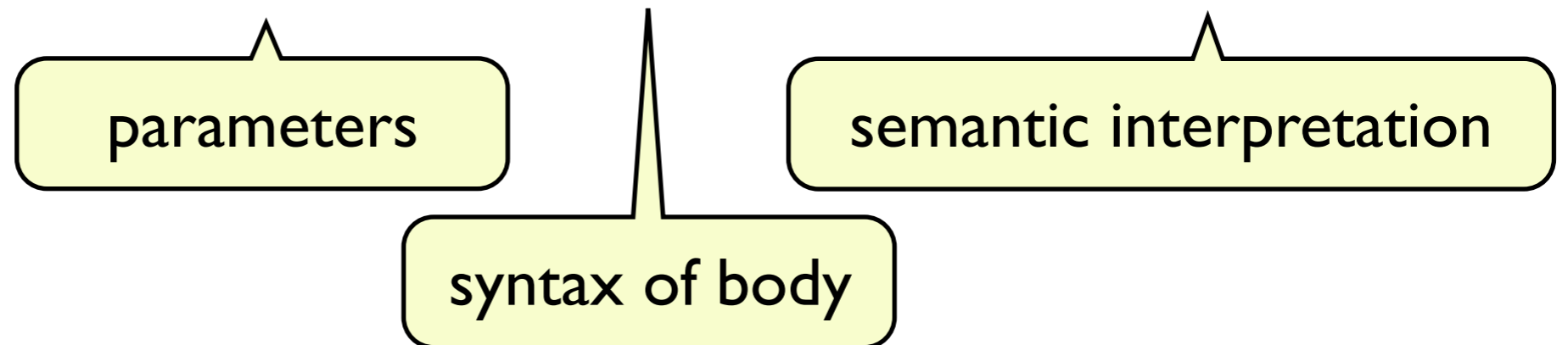
# Syntax

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



*func\_body* ::= Body term  
          | Witness term string  
          | None

concrete term (e.g. recursive function)  
property, element name  
no function body given

# Semantics

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

syntax makes sense

truth value

$$\text{eval\_formula } i \pi (\neg p) = \neg(\text{eval\_formula } i \pi p)$$

$$\text{eval\_formula } i \pi (p \vee q) = \text{eval\_formula } i \pi p \vee \text{eval\_formula } i \pi q$$

$$\text{eval\_formula } i \pi (x = y) = (\text{eval\_term } i \pi x = \text{eval\_term } i \pi y)$$

$$\text{eval\_term } i \pi (\text{Const } c) = c$$

$$\text{eval\_term } i \pi (\text{Var } v) = i(v)$$

$$\text{eval\_term } i \pi (\text{App } f \ xs) = \text{eval\_app } (f, \text{map } (\text{eval\_term } i \pi) \ xs, \pi)$$

$$\text{eval\_term } i \pi (\text{LambdaApp } vs \ x \ xs) = \text{let } ys = \text{map } (\text{eval\_term } i \pi) \ xs \text{ in} \\ \text{eval\_term } [vs \mapsto ys] \pi \ x$$

# Semantics (cont.)

$\text{eval\_app} (\text{PrimitiveFun } p, \text{args}, \pi) = \text{eval\_primitive } p \text{ args}$   
 $\text{eval\_app} (\text{Fun } name, \text{args}, \pi) = \text{let } (-, -, \text{interp}) = \pi(name) \text{ in } \text{interp}(\text{args})$

semantic interpretation

$\text{eval\_primitive Add} [\text{Val } 2, \text{Val } 3] = \text{Val } 5$   
 $\text{eval\_primitive Add} [\text{Val } 2, \text{Sym } "a"] = \text{Val } 2$   
 $\text{eval\_primitive Cons} [\text{Val } 2, \text{Sym } "a"] = \text{Dot } (\text{Val } 2) (\text{Sym } "a")$

# Well-formedness of context

Semantics only makes sense for well-formed contexts.

For every entry,

$$\pi(\textit{name}) = (\textit{formals}, \text{Body } \textit{body}, \textit{interp})$$

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:

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

Similarly for the other function types.

**Verification of Milawa to  
continue in the next lecture ...**

# Summary

## *This lecture:*

- ✓ Introduced Lisp including special functions such as **define** and **funcall**
- ✓ Milawa's small kernel for a first-order Lisp logic
- ✓ Milawa's **switch** command which swaps the proof checker for a user-provided one
- ✓ First steps in verification of Milawa's kernel

*Questions?*