

Computational Reflection, Theorem Provers and Verification

Lecture I:

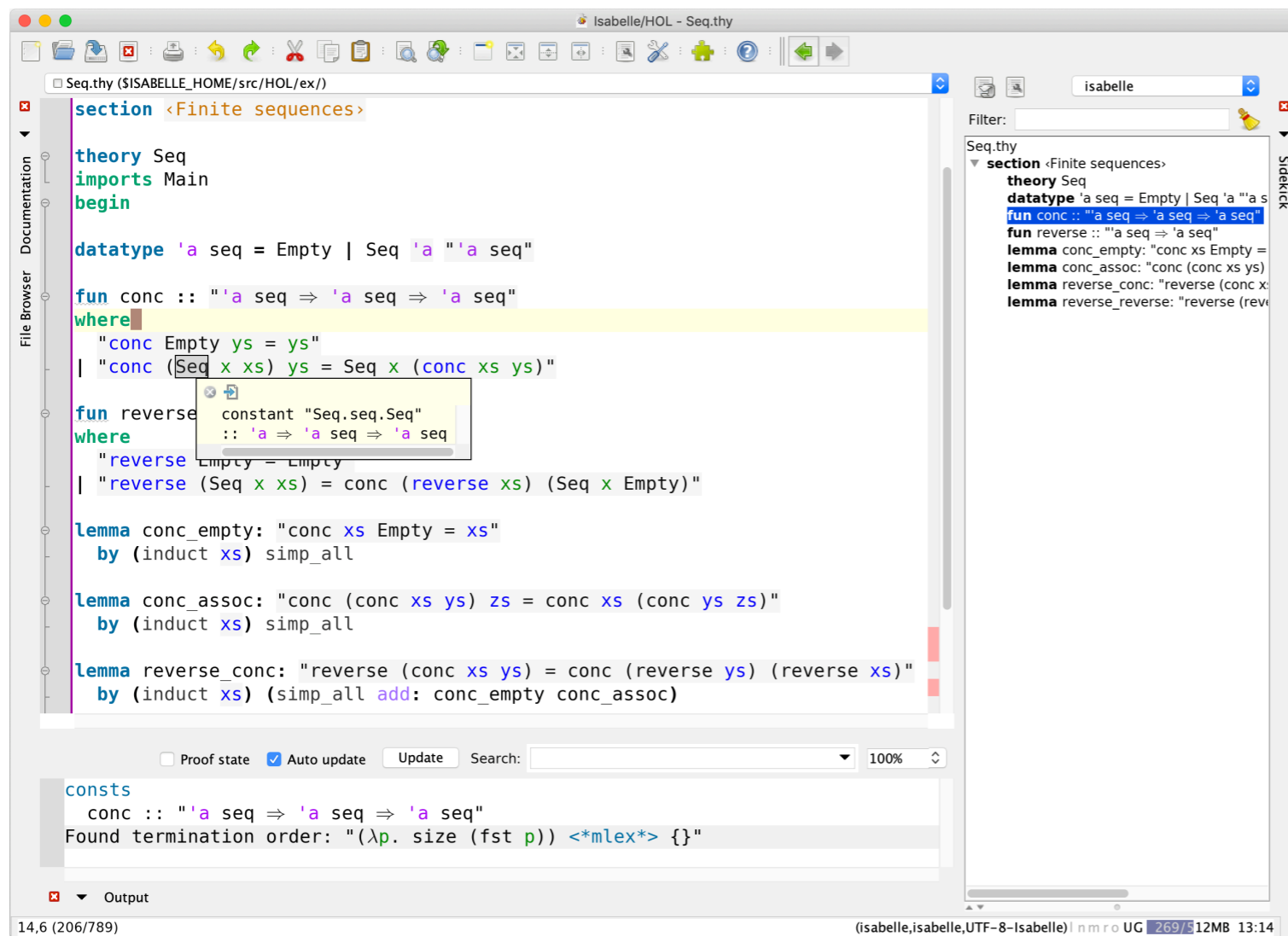
*Towards Faster Interactive
Theorem Provers*

Marktoberdorf Summer School MOD 2019

Magnus O. Myreen, Chalmers University of Technology

What are Interactive Theorem Provers?

Answer: general-purpose provers for powerful logics (e.g. HOL)



The screenshot shows the Isabelle/HOL interactive theorem prover interface. The main window displays a theory file named 'Seq.thy' with the following content:

```
section <Finite sequences>

theory Seq
imports Main
begin

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq => 'a seq => 'a seq"
where
  "conc Empty ys = ys"
| "conc (Seq x xs) ys = Seq x (conc xs ys)"

fun reverse
where
  "reverse Empty = Empty"
| "reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)"

lemma conc_empty: "conc xs Empty = xs"
  by (induct xs) simp_all

lemma conc_assoc: "conc (conc xs ys) zs = conc xs (conc ys zs)"
  by (induct xs) simp_all

lemma reverse_conc: "reverse (conc xs ys) = conc (reverse ys) (reverse xs)"
  by (induct xs) (simp_all add: conc_empty conc_assoc)
```

The right-hand pane shows the 'isabelle' search results for the current theory, listing the datatype, functions, and lemmas defined in the file.

Interactive:
most proofs require human guidance (through goal-oriented “tactics”)

Automatic:
they provide powerful automation (rewriting, decision procedures, FOL solvers, etc)

Interactive Theorem Provers (ITPs)

Example ITPs:

for classical higher-order logic (HOL), i.e. Church's simply typed lambda calculus:

HOL4

Isabelle/HOL

HOL Light

ProofPower

...

for constructive logics with dependent types:

Coq

Lean

Matita

(Agda)

(Idris)

...

for first-order logics:

ACL2

Nqthm

...

Projects using ITPs

CLI stack (Nqthm)

Four-Color Theorem (Coq)

CompCert (Coq)

Verification of AAMP7 processor (ACL2)

Verification of seL4 microkernel (Isabelle/HOL)

Proof of Kepler Conjecture (HOL Light)

CakeML (HOL4)

DeepSpec (on going in Coq)

All (unverified) compilers have bugs

“ Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. ”

PLDI'11

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr

“ [The verified part of] CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. ”

In this paper, we present the results of our bug-hunting study. Our first contribution is to advance the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the unspecified behaviors that would destroy its ability

was heavily patched; the base version of GCC used

We created Csmith, a randomized test-case generator that supports C99 and C11. Csmith generates test cases that

Soundness emphasised

When you see:

“... developed in the HOL4 theorem prover”

“... formalised in Isabelle/HOL”

“... proved correct in Coq”

then you know: *proofs mechanically checked against formal logic.*

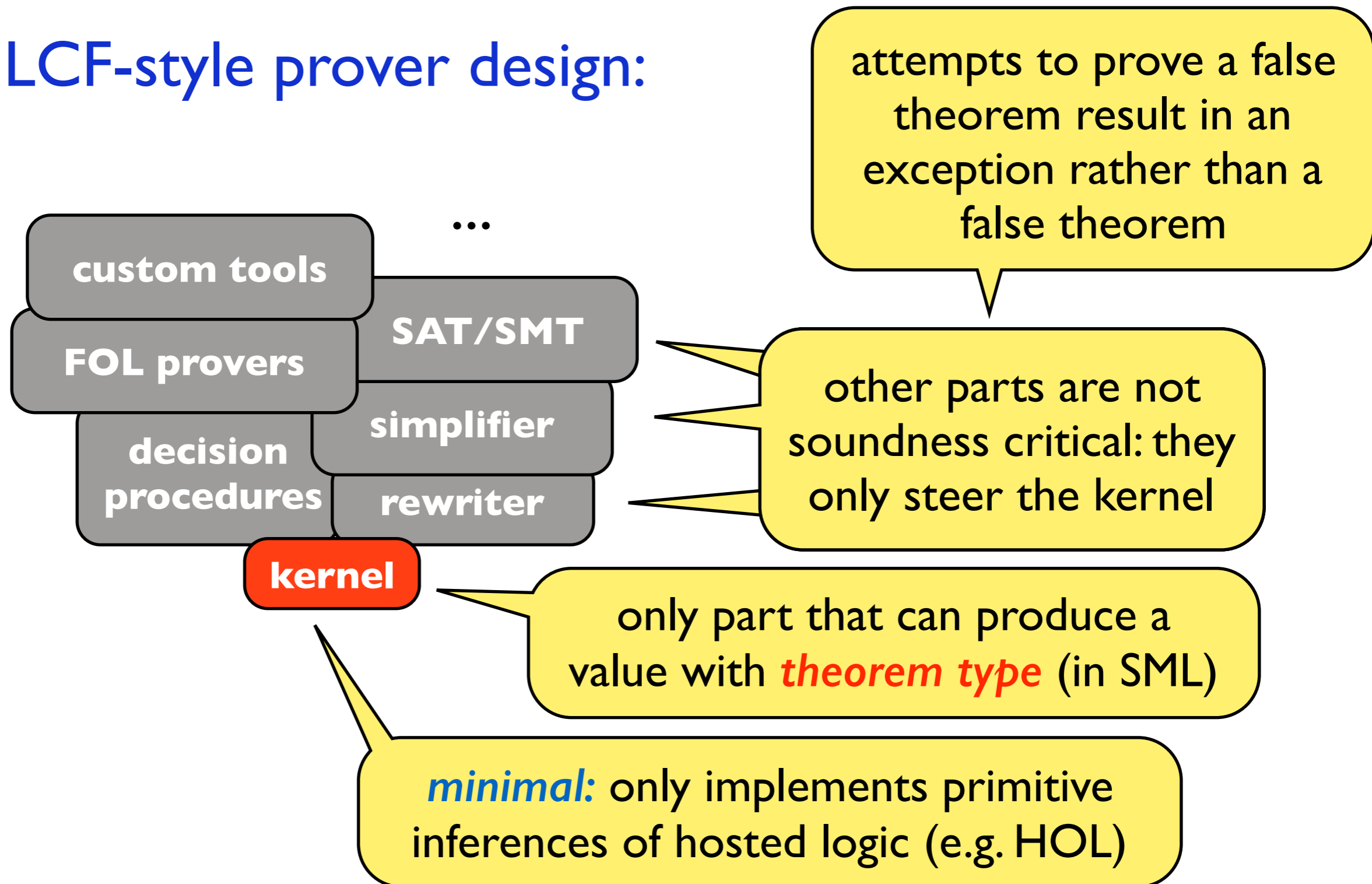


HOL or CiC in this case

Motivation

Soundness-centric design

LCF-style prover design:



Layers of implementation

hosted logic (e.g. HOL)



implemented as theorem type by prover kernel

programming language (e.g. SML)

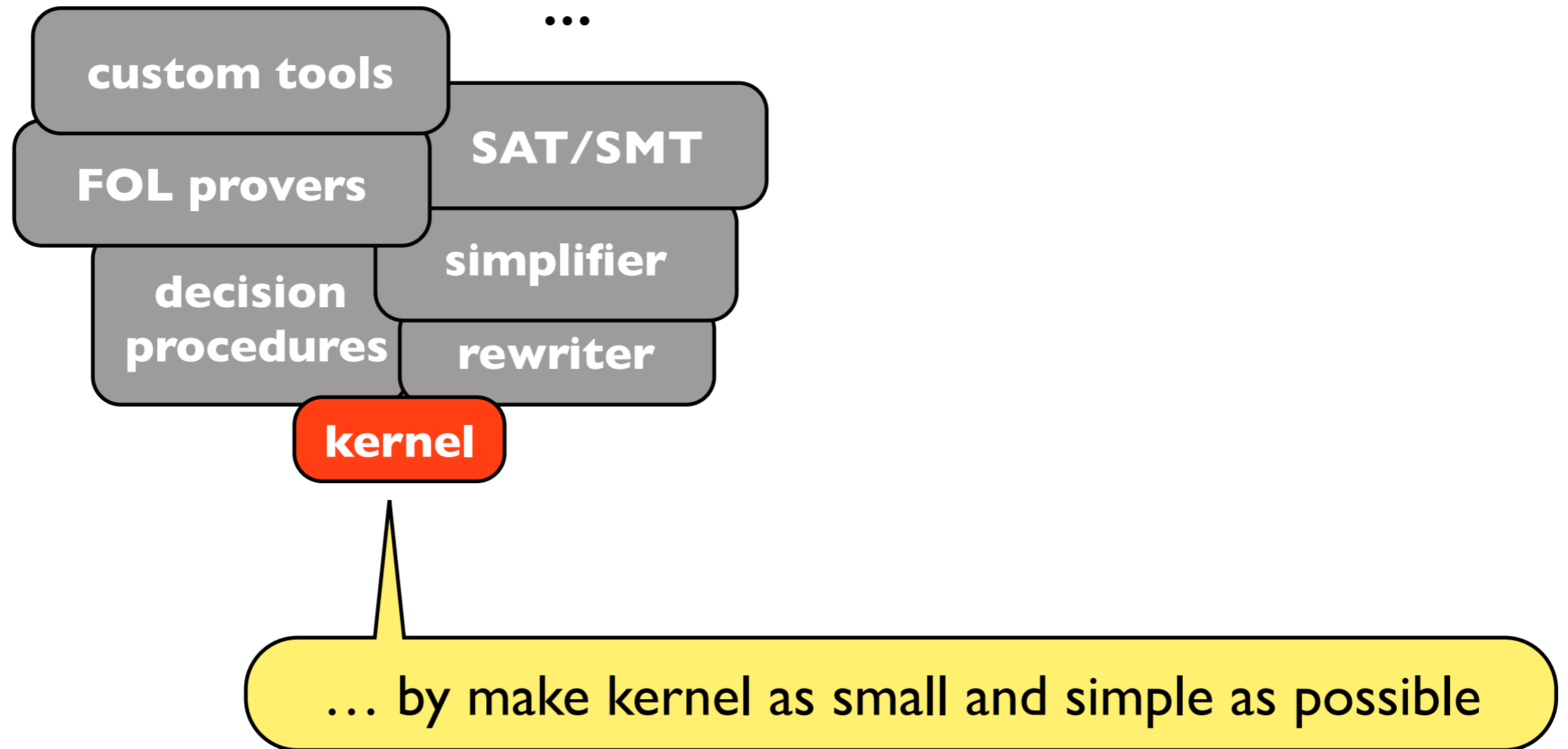


compiler generated code for host machine

machine code (e.g. Intel x86-64)

Demo showing theorem type: HOL4 exposes SML level

As trustworthy as possible ...



Check out *HOL Light's kernel (only 544 sloc):*

<https://github.com/jrh13/hol-light/blob/master/fusion.ml>

Aside: Syntax of HOL

Types of HOL:

- type variables:
- type constants/operators:
- function types:

Examples:

α
 $\text{int} \quad \text{real} \quad \alpha \quad \text{list}$
 $\tau \rightarrow \pi$

Terms of HOL:

- variables:
- constants:
- lambda-abstraction:
- function application:

Examples:

v
 $\text{PERM} \quad + \quad \wedge$
 $\lambda v. t$
 $t_1 \quad t_2$

Sequents: $\vdash (\forall x. \exists y. P \ x \ y) \iff \exists f. \forall x. P \ x \ (f \ x)$

Aside: Syntax of HOL

Types of HOL:

- type variables:
- type constants/operators:
- function types:

defined by inference (conservative ext);
the only primitive types are **bool** and **ind**

Terms of HOL:

- variables:
- constants:
- lambda-abstraction:
- function application:

only primitive constant is = and choice
others defined by inference (conservative ext)

only binder in the logic!

What about logical connectives, quantifiers etc.?

Aside: HOL Connectives and Quantifiers

They are defined (by inference) as abbrevs for others

$$\vdash \mathbf{T} \iff ((\lambda x. x) = (\lambda x. x))$$

$$\vdash (\mathbf{\forall}) = (\lambda P. P = (\lambda x. \mathbf{T}))$$

$$\vdash (\mathbf{\wedge}) = (\lambda p\ q. (\lambda f. f\ p\ q) = (\lambda f. f\ \mathbf{T}\ \mathbf{T}))$$

$$\vdash \mathbf{COND} =$$

$$(\lambda t\ t_1\ t_2.$$

$$\varepsilon x. ((t \iff \mathbf{T}) \Rightarrow (x = t_1)) \wedge ((t \iff \mathbf{F}) \Rightarrow (x = t_2)))$$

There are two standard axioms:

$$\vdash P\ x \Rightarrow P\ ((\varepsilon)\ P)$$

$$\vdash \exists f. \mathbf{ONE_ONE}\ f \wedge \neg \mathbf{ONTO}\ f$$

Aside: HOL Inference rules

Handful of simple inference rules:

- **assume, refl**
- **beta conversion**
- **antisym**
- **eq_mp**
- **term/type instantiation**
- **comb**

<https://github.com/CakeML/cakeml/blob>

<https://raw.githubusercontent.com/jrh13/hol-light/master/rulestml>

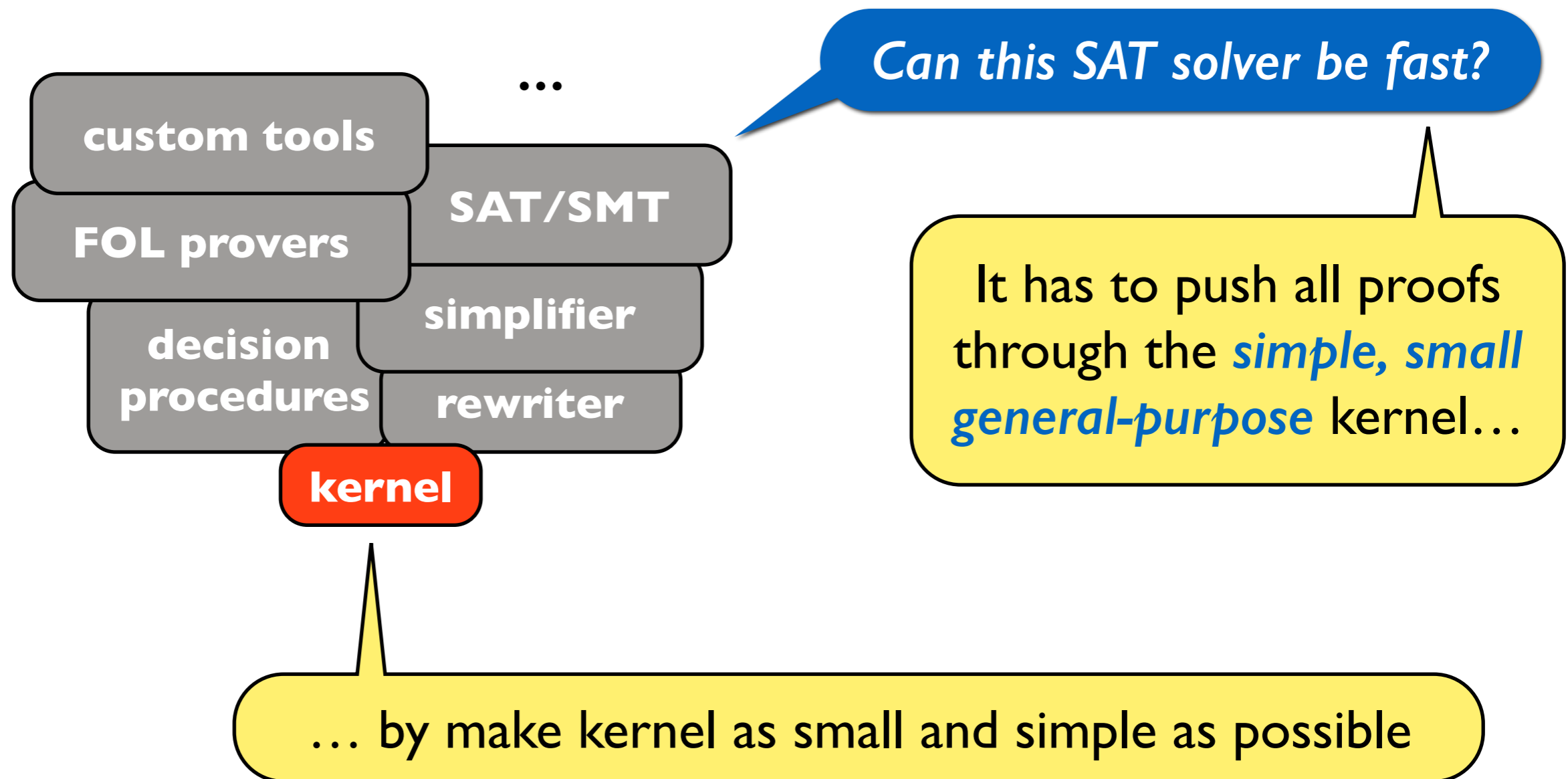
Safety net: one cannot make definitions that introduce unsoundness :-)

Definition principles:

- **constant specification**
- **type definition**

conservative extensions:
one can remove the definitions and still prove the same sequent

What about **speed**?



Check out *HOL Light's kernel (only 544 sloc)*:

<https://github.com/jrh13/hol-light/blob/master/fusion.ml>

Topic of these lectures

Lecture 1:

*Towards **Faster** Interactive Theorem Provers*

Can we have: (1) soundness of LCF-style design and
(2) speed of ad hoc implementations?

Answer: Yes! We need: *Computation,
Reflection,
Verification*

Topic of these lectures

Touches on topics:

- ✓ Formalisation of logic
- ✓ Programming language semantics
- ✓ Theorem prover and compiler verification

Computation,

expressed inside the ITP

Reflection,

for moving code to faster “layer”

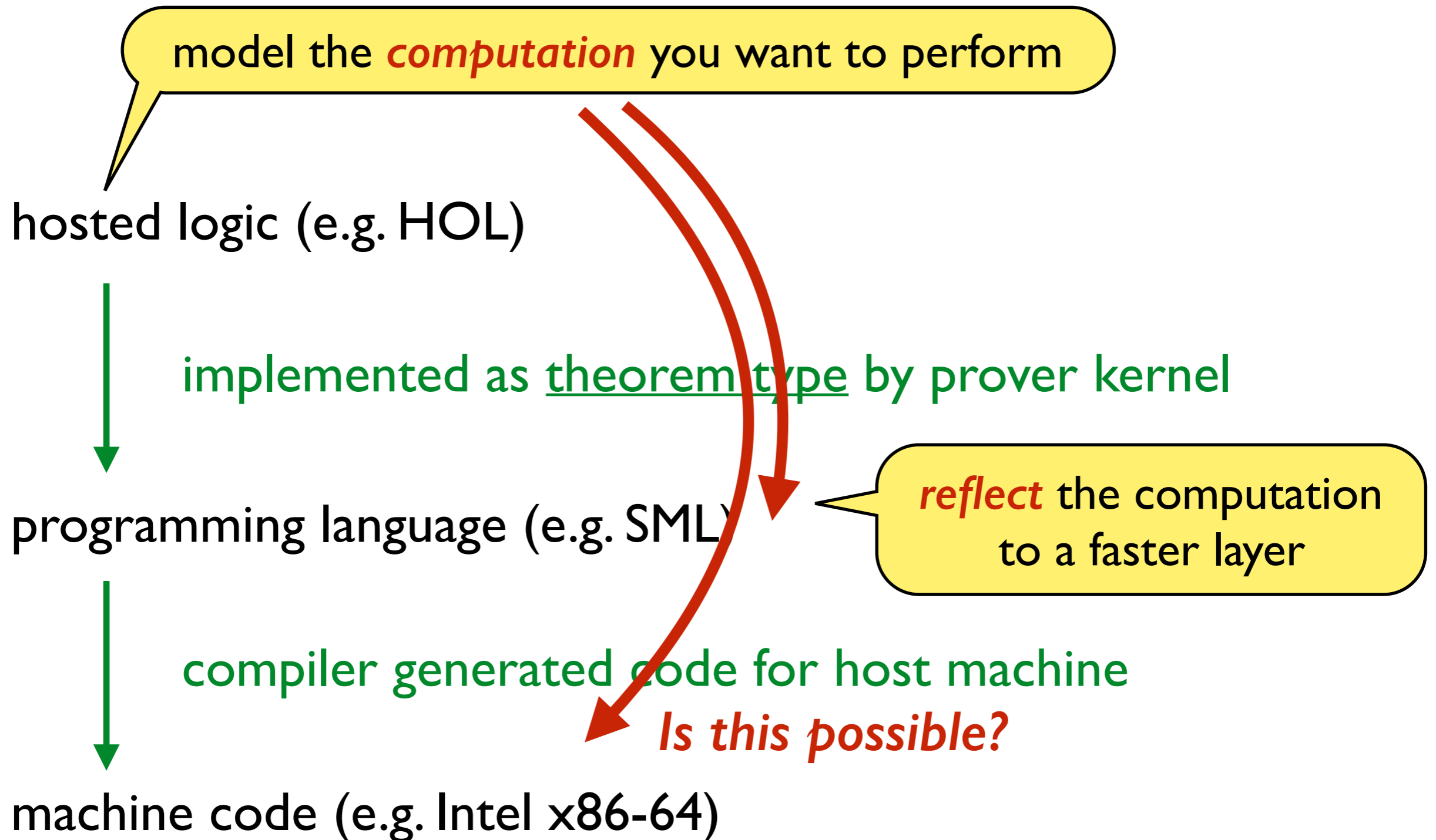
Verification

in order to ensure soundness

These lectures

1. Towards Faster Interactive Theorem Provers
2. Milawa: A Reflective Theorem Prover
3. Verification of Milawa down to Machine Code
4. Compiler Bootstrapping and Verified HOL ITP

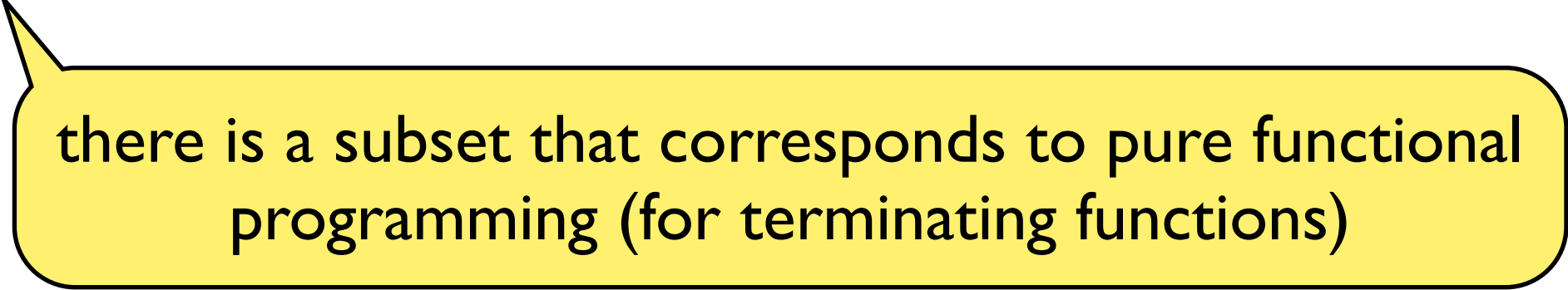
Idea for going faster



Computation in ITP

Computation and ITP

All major ITPs base their logic on some form of lambda calculus (or Lisp)

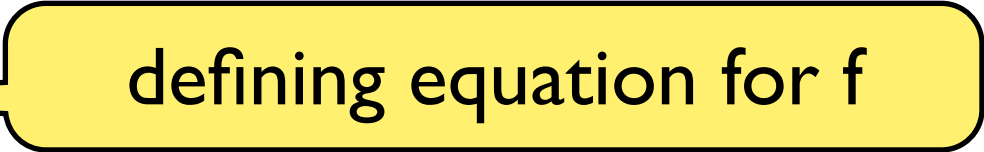


there is a subset that corresponds to pure functional programming (for terminating functions)

Why is non-termination an issue?

Suppose the logic allowed definitions of non-terminating functions such as

$$f\ n = f\ n + 1$$



defining equation for f

then $f\ n - f\ n = f\ n + 1 - f\ n$

and $0 = 1$

Example: Odd

A non-computable definition:

$$\text{Odd } n = \exists y. n = 2 \times y + 1$$

Which can be used in proofs:

$$\frac{\frac{\dots}{7 = 2 \times 3 + 1}}{\exists y. 7 = 2 \times y + 1} \begin{array}{l} \exists\text{-intro} \\ \text{def} \end{array}$$

Odd 7

Example: Odd (continued)

As a functional program:

$$\begin{aligned} \text{odd } 0 &= \text{F} \\ \text{odd } (\text{SUC } n) &= \neg \text{odd } n \end{aligned}$$

Connection to Odd:

$$\text{odd } n = \text{T} \Rightarrow \text{Odd } n$$

Improved proof:

$$\frac{\frac{\frac{\forall x. \text{odd } x = \text{T} \Rightarrow \text{Odd } x}{\text{thm}}}{\text{odd } 7 = \text{T} \Rightarrow \text{Odd } 7} \forall\text{-elim}}{\text{Odd } 7} \text{thm} \quad \frac{\frac{\overline{\text{T} = \text{T}}}{\text{odd } 7 = \text{T}} \text{refl}}{\text{odd } 7 = \text{T}} \text{comp}}{\text{odd } 7 = \text{T} \Rightarrow \text{Odd } 7} \Rightarrow\text{-elim}$$

Example: Odd (continued)

One can derive computable equations from definition of Odd:

$$\text{Odd } 0 = \text{F} \quad \wedge \quad \forall n. \text{Odd } (\text{SUC } n) = \neg \text{Odd } n$$

... and thus the definition of odd could have been avoided.

What are proofs by computation?

Answer: they are proofs by rewriting/reduction

$$\begin{aligned} & \text{odd (SUC (SUC (SUC } \emptyset))) = \\ & \neg \text{ odd (SUC (SUC } \emptyset)) = \\ & \neg \neg \text{ odd (SUC } \emptyset) = \\ & \neg \neg \neg \text{ odd } \emptyset = \\ & \neg \neg \neg F = \\ & \neg \neg T = \\ & \neg F = \\ & T \end{aligned}$$

By default these are “interpreted” by the kernel.

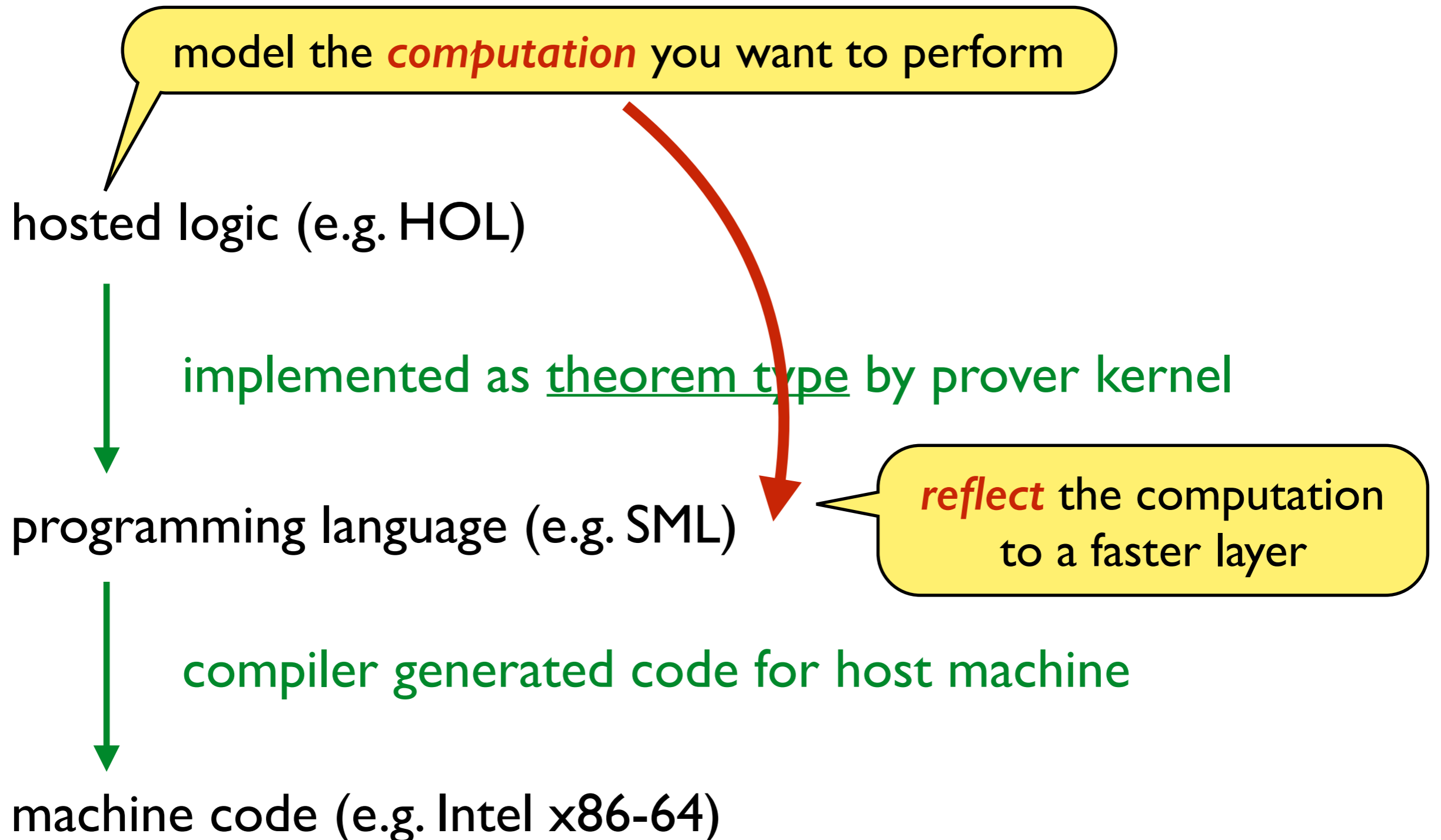
Reflection + Computation

Reflection

When I write “reflection” I mean:

A function/expression is represented (sufficiently) equivalently at another level of implementation.

Reminder:



A simpler example

inside the logic:

deep embedding

shallow embedding

reflect an expression to a different representation

(terminology will be explained soon)

hosted logic (e.g. HOL)

implemented as theorem type by prover kernel

programming language (e.g. SML)

Deciding simple propositions

Example: we want to prove *by computation* properties such as:

$$\neg(x \wedge \neg x)$$

We can define a syntax for such expressions:

prop ::= Var \mathbb{N} | Not prop | Conj prop prop

And can represent the property above as:

Not (Conj (Var 0) (Not (Var 0)))

Terminology

Example: we want to prove *by computation* properties such as:

$\neg(x \wedge \neg x)$

this is a *shallow embedding*

this is a variable of the host logic

syntax for such expressions:

prop ::= Var \mathbb{N} | Not prop | Conj prop prop

this is a constructor of the prop datatype

the property

this is a *deep embedding*

Not (Conj (Var 0) (Not (Var 0)))

Connection between deep and shallow

We define a function in the logic that interprets the deep embeddings:

$$\begin{aligned} \text{eval } env \text{ (Var } v) &= env \ v \\ \text{eval } env \text{ (Not } p) &= \neg \text{ eval } env \ p \\ \text{eval } env \text{ (Conj } p \ q) &= \text{eval } env \ p \wedge \text{eval } env \ q \end{aligned}$$

Example application:

$$\text{eval } (\lambda v. x) \text{ (Not (Conj (Var 0) (Not (Var 0))))} = \neg(x \wedge \neg x)$$

Deciding truth

Some functional programming in the logic:

$$\begin{aligned} \text{truth } 0 \text{ env } p &= \text{Some } (\text{eval } \text{env } p) \\ \text{truth } (\text{Suc } k) \text{ env } p &= \text{case } (\text{truth } k \text{ env}[k \mapsto \text{T}] p, \text{truth } k \text{ env}[k \mapsto \text{F}] p) \text{ of} \\ &| (\text{Some } \text{T}, \text{Some } \text{T}) \rightarrow \text{Some } \text{T} \\ &| (\text{Some } \text{F}, \text{Some } \text{F}) \rightarrow \text{Some } \text{F} \\ &| _ \rightarrow \text{None} \end{aligned}$$

$$\begin{aligned} \text{next_var } (\text{Var } v) &= v + 1 \\ \text{next_var } (\text{Not } p) &= \text{next_var } p \\ \text{next_var } (\text{Conj } p \ q) &= \max (\text{next_var } p) (\text{next_var } q) \\ \text{decide } p &= \text{truth } (\text{next_var } p) (\lambda x. \text{F}) p \end{aligned}$$

true for any env

Correctness theorem:

$$\begin{aligned} \text{decide } p = \text{Some } \text{T} &\Rightarrow \text{eval } \text{env } p \\ \text{decide } p = \text{Some } \text{F} &\Rightarrow \neg \text{eval } \text{env } p \end{aligned}$$

Using it for proof

A proof that uses computation twice:

Can we get this to go fast?

$$\frac{\frac{\frac{\frac{}{\forall e p. \text{decide } p = \text{Some } T \Rightarrow \text{eval } e p}}{\text{decide } t = \text{Some } T \Rightarrow \text{eval } e t} \text{thm}}{\text{decide } t = \text{Some } T \Rightarrow \text{eval } e t} \forall\text{-elim}}{\frac{\frac{\frac{}{\text{eval } e t}}{\neg(x \wedge \neg x)} \text{comp}}{\text{decide } t = \text{Some } T \Rightarrow \text{eval } e t} \Rightarrow\text{-elim}}{\text{decide } t = \text{Some } T \Rightarrow \text{eval } e t} \text{comp}}{\text{decide } t = \text{Some } T \Rightarrow \text{eval } e t} \Rightarrow\text{-elim}$$

where e abbreviates $\lambda v. x$

and t abbreviates $\text{Not } (\text{Conj } (\text{Var } 0) (\text{Not } (\text{Var } 0)))$

Speed of a slow interpreter

inside the logic:

proof by rewriting and beta-reduction:

decide (Not (Conj (Var 0) (Not (Var 0)))) = SOME T

hosted logic (e.g. HOL)



implemented as theorem type by prover kernel

programming language (e.g. SML)



Speed of a slow interpreter

inside the logic:

proof by rewriting and beta-reduction:

decide (Not (Conj (Var 0) (Not (Var 0)))) = SOME T

hosted logic (e.g. HOL)

reflect the computation
to a faster layer

implemented as theorem type by prover kernel

programming language (e.g. SML)

Warning: uses questionable
extensions of kernel

State of the art

Milawa and ACL2

ACL2 and Milawa are systems that host logics that are very similar to *a pure subset of Lisp*.

Milawa is a proof-of-concept prover:

- ✓ has the strongest soundness story regarding fast computation

Next two lectures are about Milawa.

ACL2 is an industrial strength ITP, which *lacks an LCF-style design*, but has strong soundness story for computation due to the logic's proximity with Lisp.

Computation in Coq

Coq offers 3 ways of doing computation:

compute, an interpreter that supports evaluation (with different evaluation orders: call-by-value, lazy, ...)

vm_compute relies on compilation to the bytecode of a virtual machine, delivers performance comparable to that of the OCaml bytecode compiler.

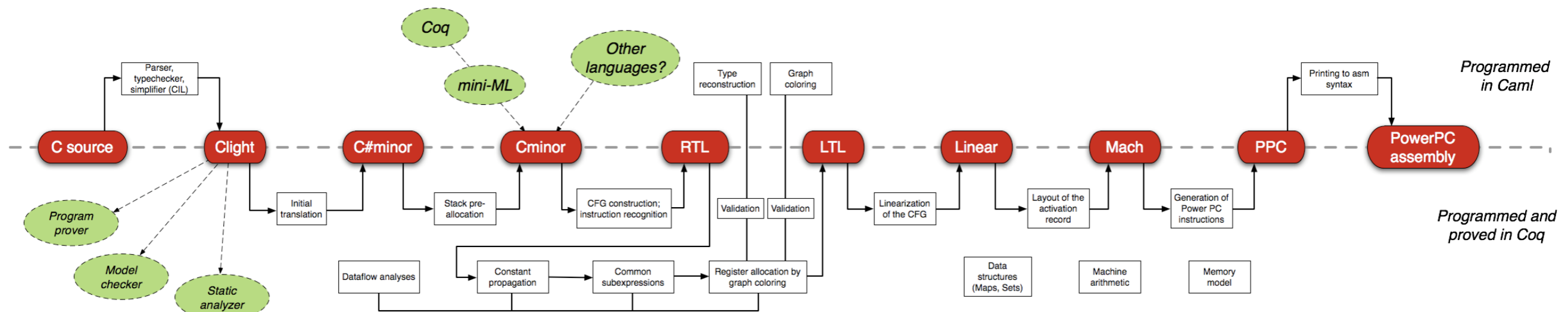
native_compute, uses OCaml native-code compiler to produce even higher performance.

Xavier Leroy has interesting blog post on this:

<http://gallium.inria.fr/blog/coq-eval/>

Compute \neq Extraction

CompCert C compiler needs to be extracted to OCaml to be run.



Leroy et al. Source: <http://compcert.inria.fr/>

Compiles C source code to assembly

Has good performance numbers

Isabelle/HOL (and other HOLs)

Coq users call with code extraction

Isabelle/HOL has well developed *code generator* too

Including for *imperative features*, search “Imperative HOL”

Possible to:

1. define functions in logic
2. generate code for them
3. evaluate a closed term outside of the logic
4. get a theorem based on 3. in the logic

However, theorem will be tagged
“you are trusting non-kernel code”

Summary

This lecture:

- ✓ Introduced Interactive Theorem Provers (ITPs) and their soundness-centric design (LCF-style design)
- ✓ Looked at computation inside an ITP
- ✓ Introduced the concept of reflection for computation
- ✓ Summarised the state of the art

Questions?