Guest Lecture: Introduction to HOL

Interactive Theorem Proving with Dependent Types (FID3217)

Magnus O. Myreen, CSE, Chalmers, June 2024

HOL provers are ITPs …

Guest Lecture: Introduction to HOL

Interactive Theorem Proving with Dependent Types (FID3217)

… but HOL provers do not have dependent types.

Magnus O. Myreen, CSE, Chalmers, June 2024

Why learn about HOL?

Here's one reason:

Both Apple and AWS have independently started projects on verification of low-level code. Both chose Isabelle/HOL.

Why didn't they choose Coq / Lean / Agda?

Lecture outline:

History of HOL ITPs

My work in a HOL ITP

A closer look at HOL4 (demos)

Lecture outline:

History of HOL ITPs

My work in a HOL ITP

A closer look at HOL4 (demos)

Motivation

How can I know my software satisfies a spec?

You can ***prove*** that it satisfies the spec.

How do I know my proof isn't flawed?

By strictly following the rules of a ***formal logic***,
you can be sure the proof is sound.

What is a formal logic and how can I be sure I follow its rules?

By strictly following the rules of a *formal logic*,
you can be sure the proof is sound.

What is a formal logic and how can I be sure I follow its rules?

A *formal logic* is a formal system with limited vocabulary and *exact syntactic rules* for deducing new facts from other facts in the system.

You can be sure to follow its rules if you use software, called *interactive theorem provers* (ITPs), to create and check your proofs.

Wait… How can I *trust* the correctness of these ITPs?

ITPs are very defensively programmed.

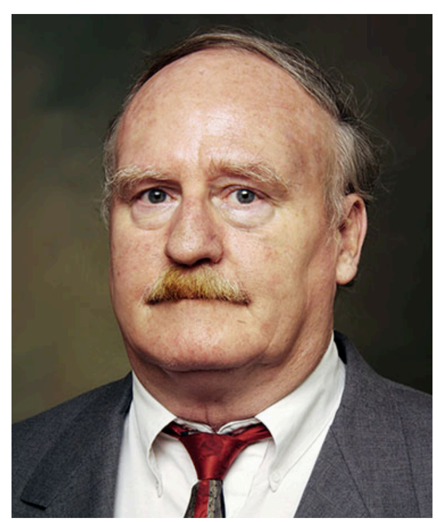
Late 1960s & Early 1970s

Historically significant early ITPs:

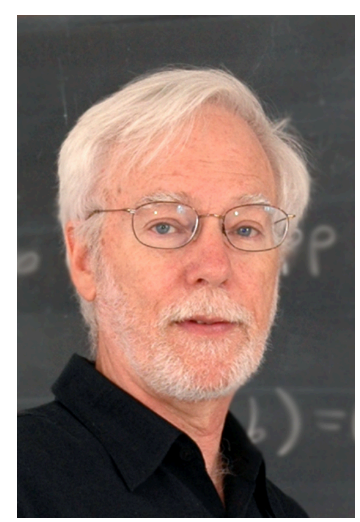
Automath

Nicolaas Govert
de Bruijn

Boyer-Moore Theorem Prover



Bob Boyer



J Moore

LCF

Robin Milner

Early 1970s

first practical system that used the Curry–Howard correspondence

Lisp programming language as a logic, strong simplifier, automatic induction

LCF = logic of computable functions (logic by Dana Scott)

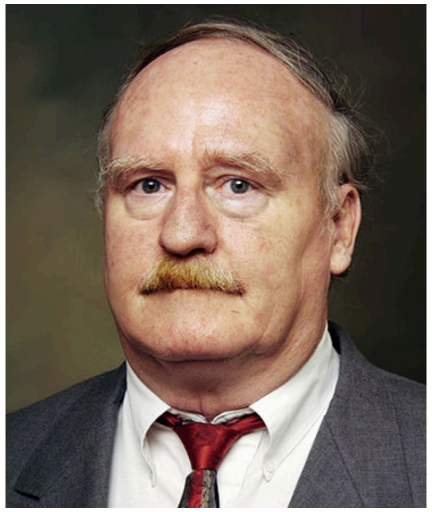
Historically significant early TPs:

Automath

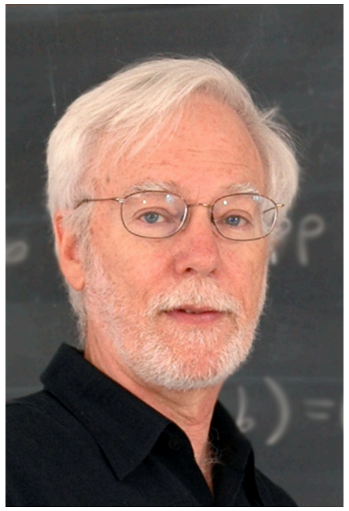
Boyer-Moore Theorem Prover

LCF

Nicolaas Govert de Bruijn



Bob Boyer



J Moore

Robin Milner

Robin Milner

who later took an interest in cryptography

Standford LCF

team: Robin Milner and Whitfield Diffie

Diffie taught Milner Lisp

key features: goal manager and powerful simplifier

shortcomings:

(1) size of proofs was limited by memory

(2) fixed set of proof commands

Edinburgh LCF (1973 onwards)

Robin Milner, Lockwood Morris, Malcolm Newey

Milner tackled shortcomings (1) and (2)

Robin Milner

shortcomings:

(1) size of proofs was limited by memory

(2) fixed set of proof commands

Edinburgh LCF (1973 onwards)

Robin Milner, Lockwood Morris, Malcolm Newey

Milner tackled shortcomings (1) and (2)

System should only remember results of proofs (→ 1)

User should be able to program new tactics (→ 2)

Key idea: abstract data type **thm**: predefined values were axioms and operations over **thm** were inference rules of the logic

strict type checking ensured that all values of type **thm** are axioms or follow by inference rules

Implementation: a new programming language, called ML

Implementation: a new programming language, called ML

ML = Meta Language

Robin Milner

In 1975, Morris and Newey moved away

→ Chris Wadsworth and Mike Gordon joined the effort

Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages

POPL'78

A Metalanguage for Interactive Proof in LCF*

M. Gordon, R. Milner
University of Edinburgh

L. Morris
Syracuse University

M. Newey
Australian National University

C. Wadsworth
University of Edinburgh

Introduction

LCF (Logic for Computable Functions) is a proof generating system consisting of an interactive programming language ML (MetaLanguage) for conducting proofs in PPλ (Polymorphic Predicate

computing system) of ML and PPλ began over three years ago at Edinburgh; for about two years the system has been usable, and its development is now virtually complete. Recently it has been used in various studies concerning formal semantics:

Implementation: a new programming language, called ML

ML = Meta Language

In 1975, Morris and Newey moved away

→ Chris Wadsworth and Mike Gordon joined the effort

Cambridge LCF

Mike Gordon (and Milner) moved to Cambridge

Larry Paulson was hired as a postdoc in early 1980s

Larry and Gérard Huet produced an ML compiler that sped up LCF by factor for 20

behind Caml

Larry significantly improved many parts of Cambridge LCF

Larry Paulson

The HOL theorem prover

Mike Gordon

Mike was doing hardware verification in LCF

LCF's foundations in domain theory were overkill

Ben Moskowski (then a postdoc) showed Mike how Mike's hardware descriptions could be encoded nicely in higher-order logic (HOL)

Church's simple type theory (extended with polymorphic types)

→ Mike cloned Cambridge LCF and adjusted the thm type to implement HOL

HOL provers:

HOL88, HOL90, HOL4 and also Proof Power, Isabelle/HOL, HOL Light

Break for questions!

I like lots of questions.

Lecture outline:

History of HOL ITPs

My work in a HOL ITP

A closer look at HOL4 (demos)

Prior to my PhD

Mike hired Anthony Fox as a postdoc

Anthony continued Mike's hardware verification

Ambitious project: prove functional correctness of ARM processor down to RTL level

ARM6 RTL design was in the public domain

By product: an extensive definition of the how ARM machine code executes (ISA specification)

Can Anthony's ARM model be used?

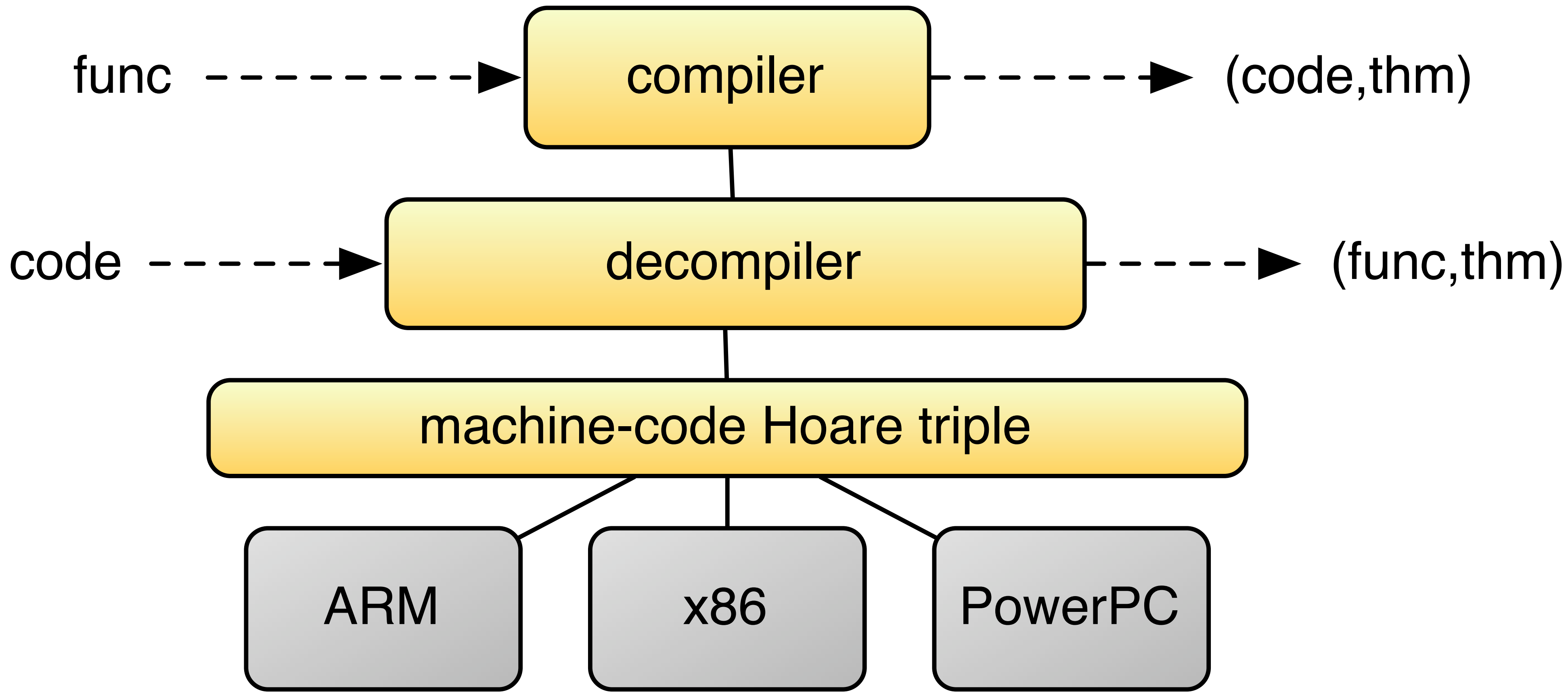
His tooling produced theorems that describe ARM,
e.g. ARM instruction add r0,r0,r0 is described by:

encoding of add r0,r0,r0

```
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state =
    0xE0800000w) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state =
     ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
     (ARM_WRITE_REG 0w
      (ARM_READ_REG 0w state + ARM_READ_REG 0w state) state))
```

My PhD (2005-08)

During my PhD, I developed the following infrastructure:



Decompiler illustrated

Example: Given some hard-to-read (ARM) machine code,

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

The decompiler produces a readable HOL4 function:

$$
\begin{aligned}
f(r_0, r_1, m) &= \text{let } r_0 = 0 \text{ in } g(r_0, r_1, m) \\
g(r_0, r_1, m) &= \text{if } r_1 = 0 \text{ then } (r_0, r_1, m) \text{ else} \\
&\qquad\quad \text{let } r_0 = r_0{+}1 \text{ in} \\
&\qquad\quad \text{let } r_1 = m(r_1) \text{ in} \\
&\qquad\quad\quad g(r_0, r_1, m)
\end{aligned}
$$

Decompiler illustrated (cont.)

Decompiler automatically proves a certificate, which states that f describes the effect of the ARM code:

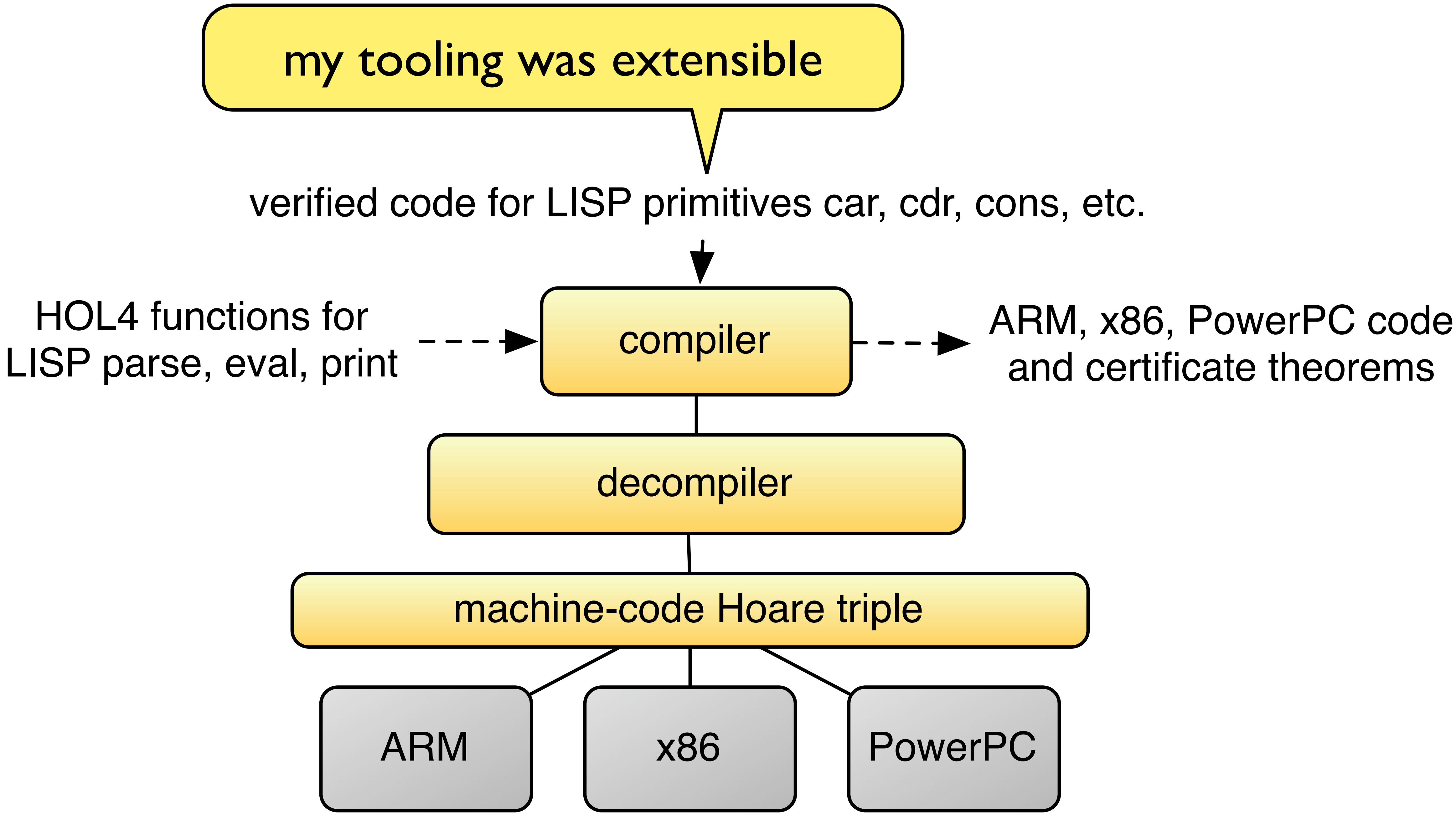
$$f_{pre}(r_0, r_1, m) \Rightarrow$$

$$\{\, (\mathrm{R0}, \mathrm{R1}, \mathrm{M}) \text{ is } (r_0, r_1, m) * \mathrm{PC}\ p * \mathrm{S} \,\}$$

$$p : \texttt{E3A00000 E3510000 12800001 15911000 1AFFFFFB}$$

$$\{\, (\mathrm{R0}, \mathrm{R1}, \mathrm{M}) \text{ is } f(r_0, r_1, m) * \mathrm{PC}\ (p + 20) * \mathrm{S} \,\}$$

My PhD (2005-08)



It was a lot of fun

Example: paper gives a definition of `pascal-triangle`, for which:

```
(pascal-triangle '((1)) '6)
```

returns:

```
((1 6 15 20 15 6 1)
 (1 5 10 10 5 1)
 (1 4 6 4 1)
 (1 3 3 1)
 (1 2 1)
 (1 1)
 (1))
```

The verified code was run on several platforms:

Nintendo DS lite (ARM)

MacBook (x86)

old MacMini (PowerPC)

My PhD (2005-08)

Most important lesson learnt:

Developing ***custom automation***

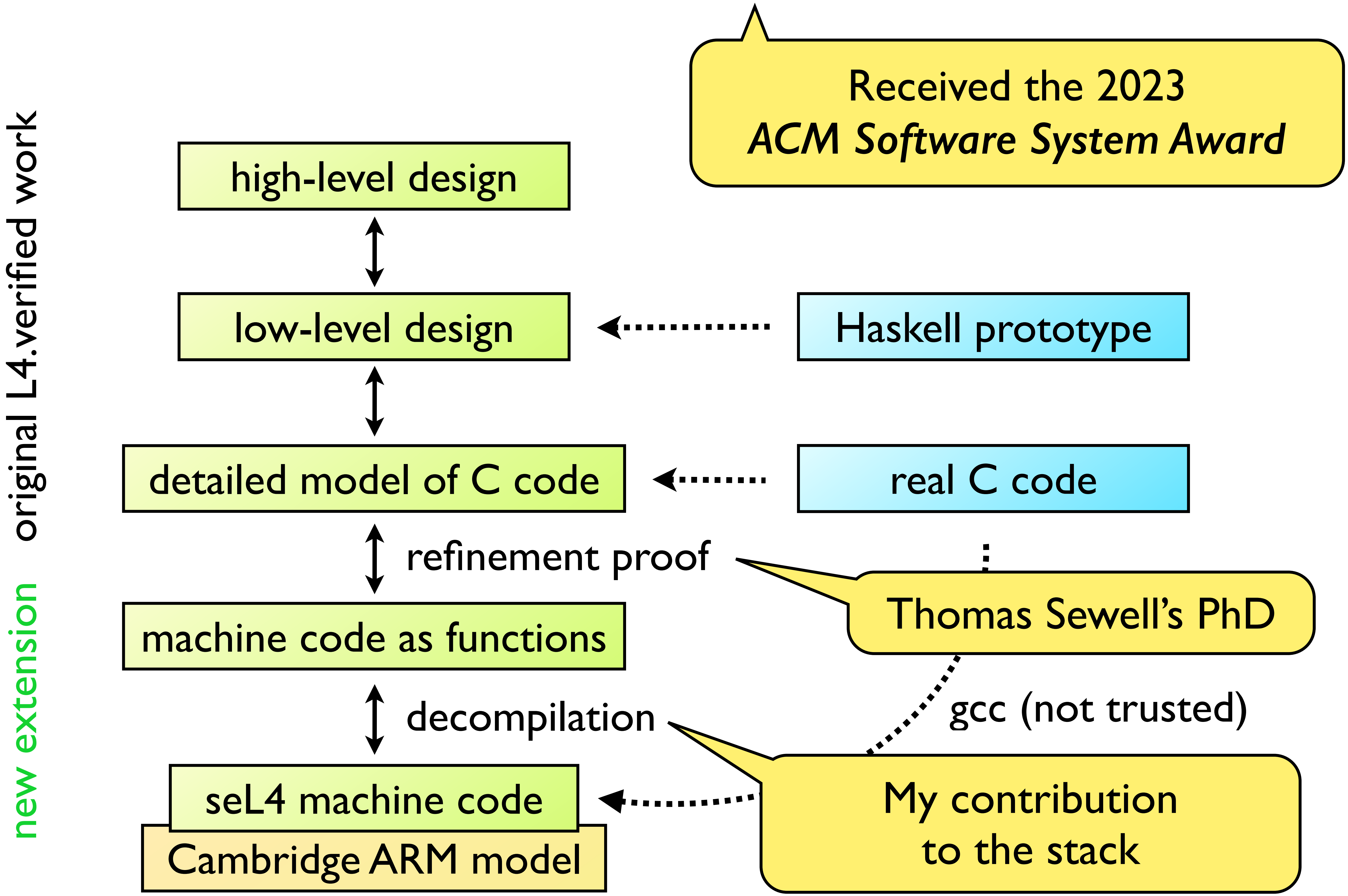
and mixing that with ***interactive proving***

leads to

high quality results (quickly)

and a lot of fun.

Verified seL4 OS



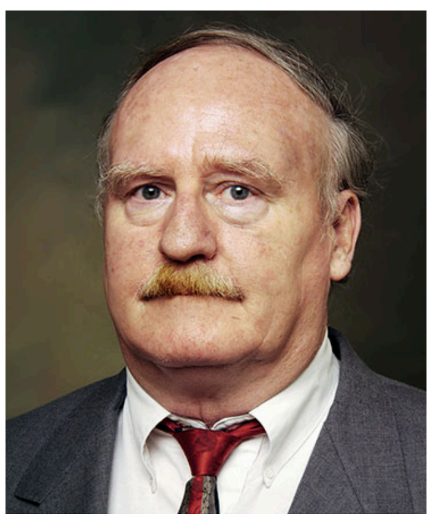
Break for questions!

I like lots of questions.

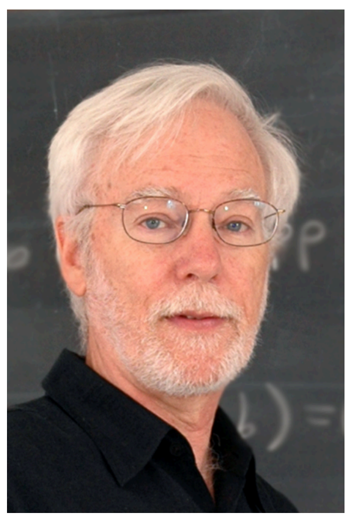
Connection to Boyer Moore

Boyer-Moore Theorem Prover - - - - - -▶ **Milawa**

Email: can I try running Milawa on your verified Lisp?



Bob Boyer

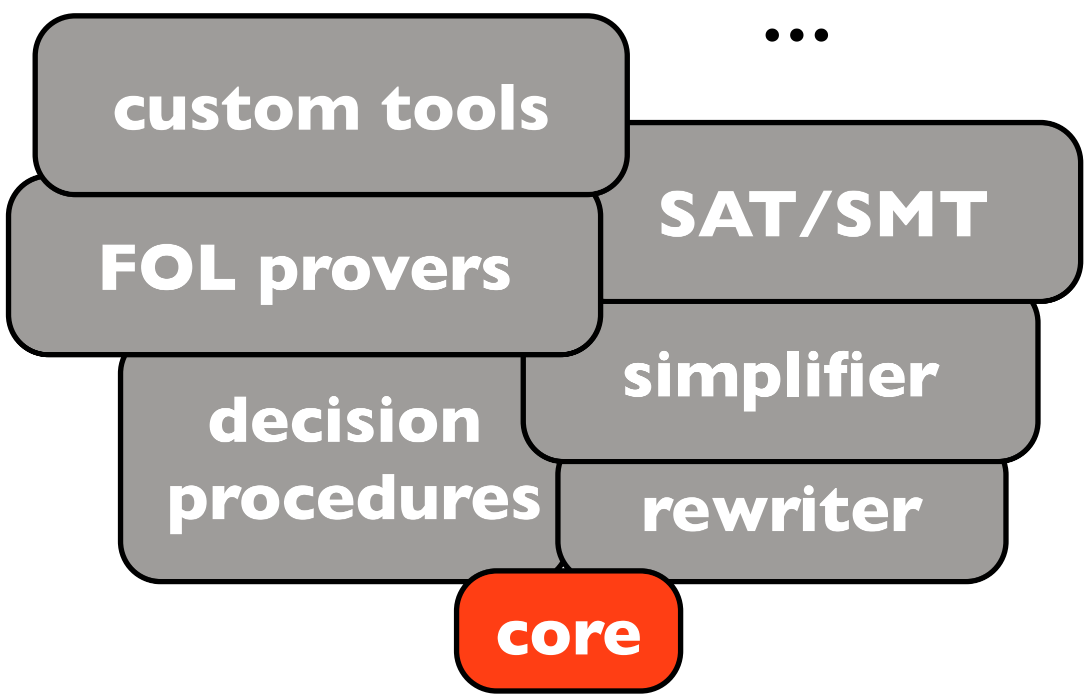


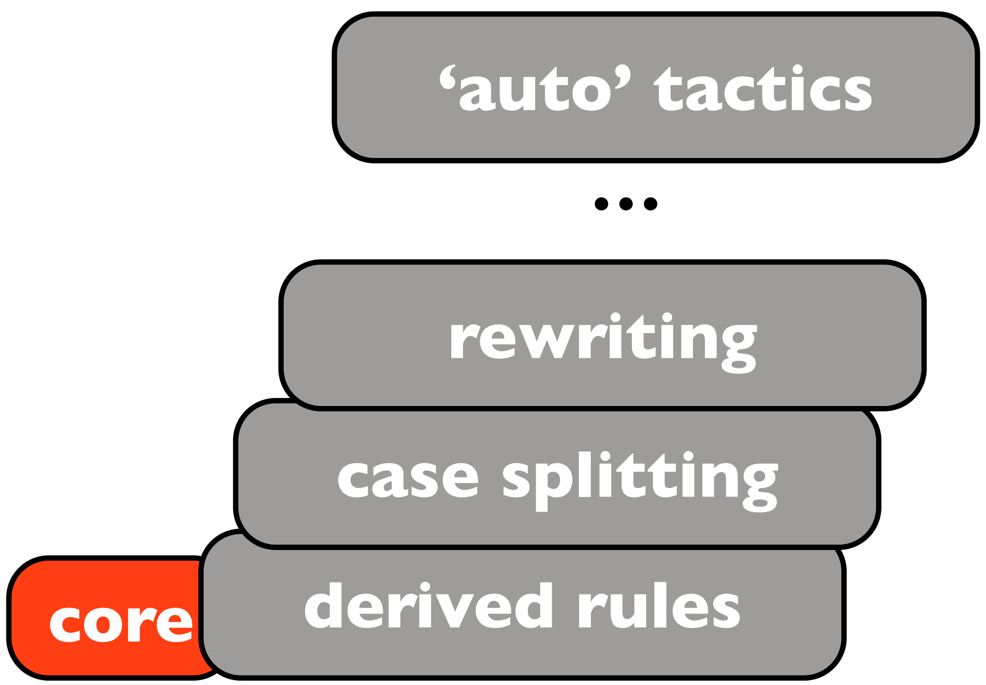
J Moore

Jared Davis
(formerly Moore's PhD student)

work by Jared Davis

LCF vs Milawa





LCF-style approach

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

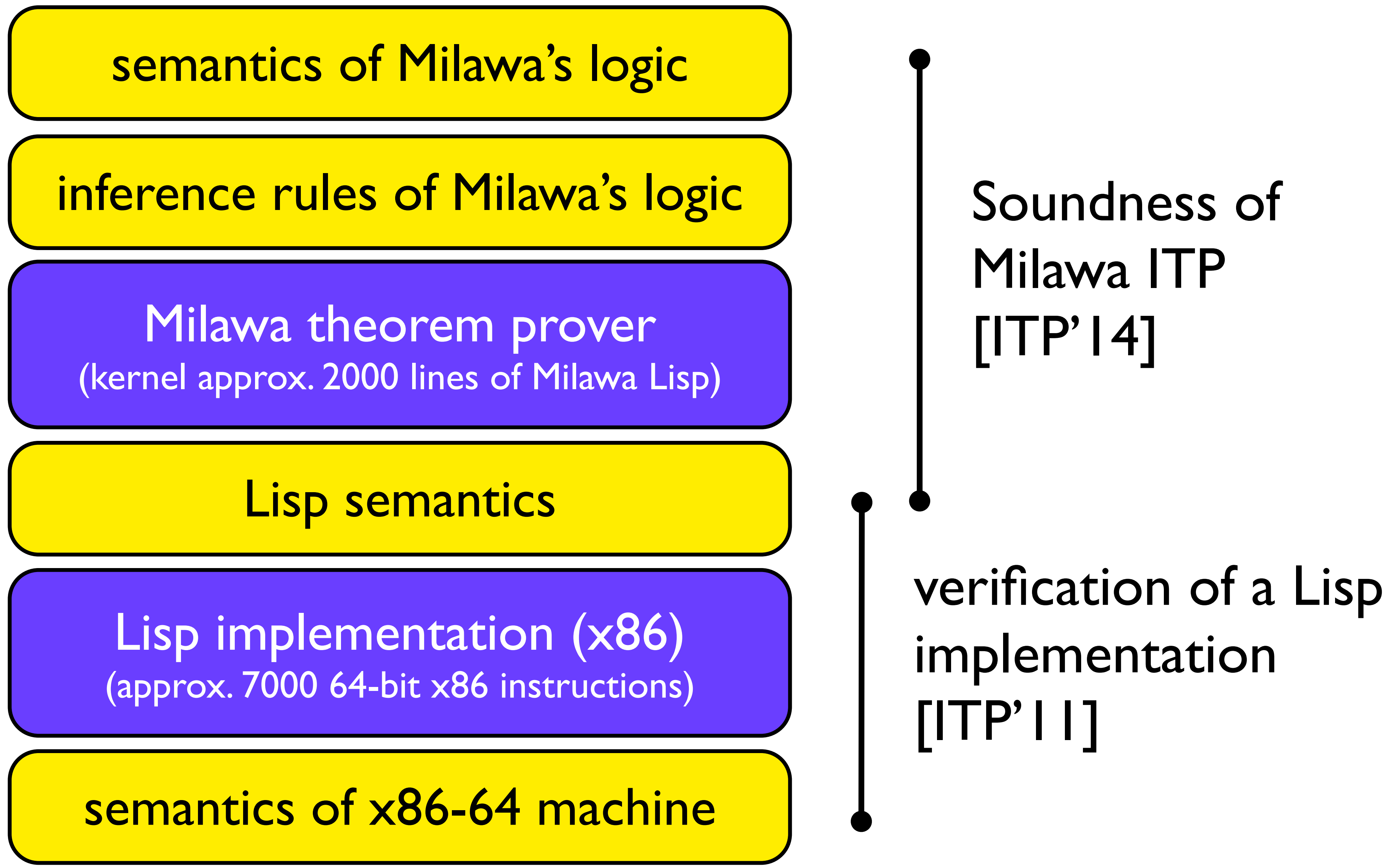
the Milawa approach

- all proofs must pass the core
- the core can be replaced by a new one at runtime

I proved Milawa sound

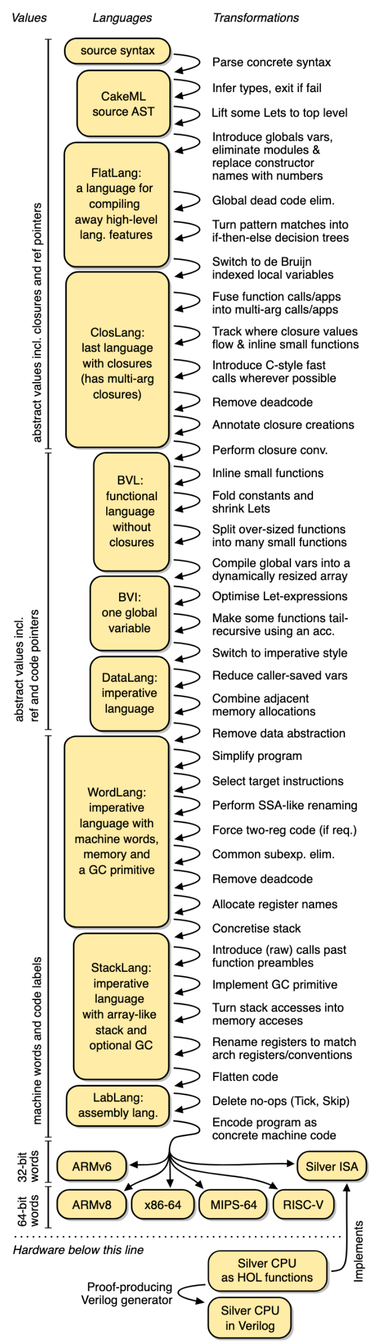
Milawa

Jitawa verified **LISP**



The CakeML project

Cambridge and **Ke**nt **ML**



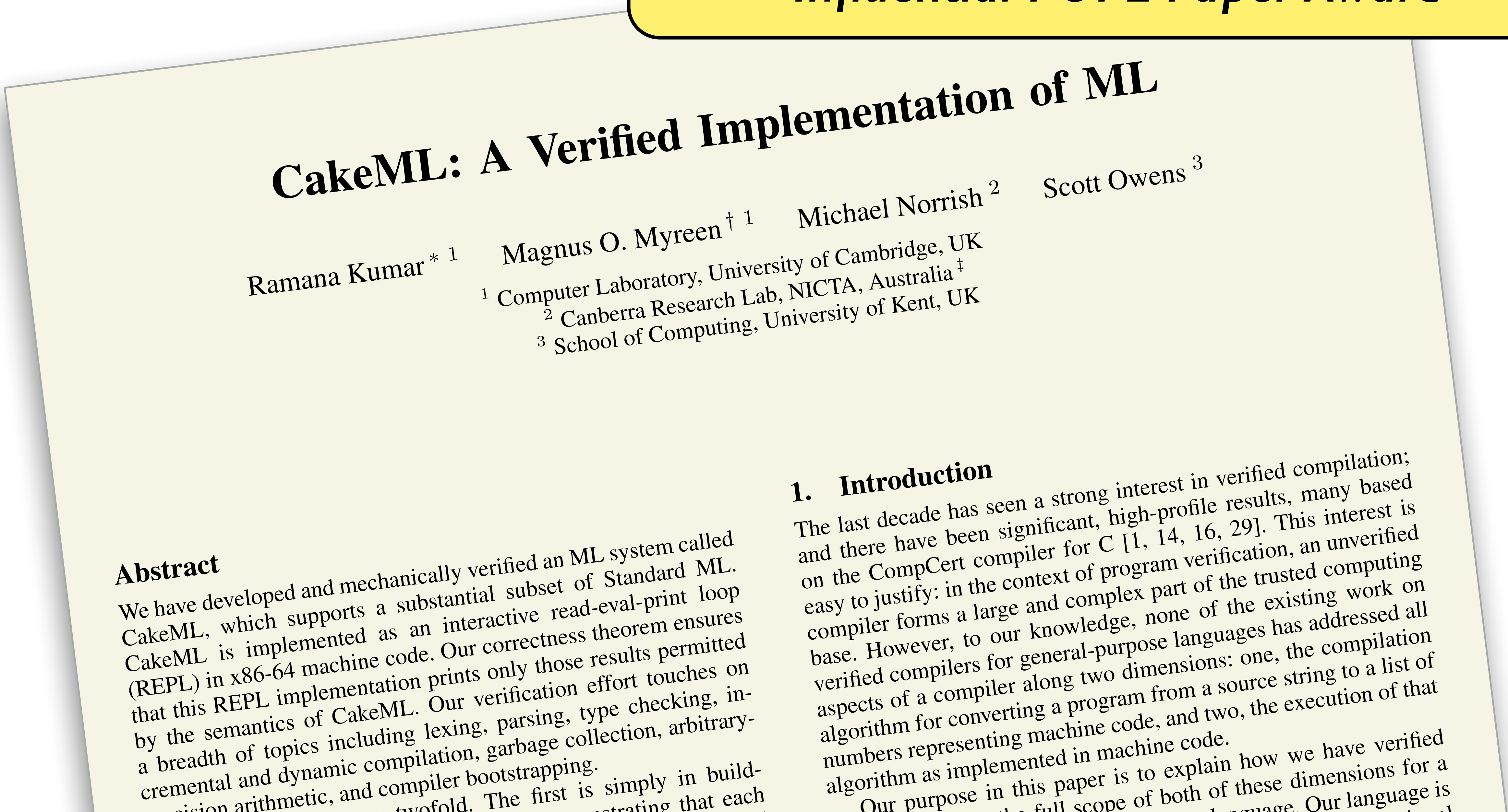
Has produced a significant verified compiler for ML

CakeML's First Major Result

36 years after original ML paper

POPL'14

Received the 2024 *ACM SIGPLAN Most Influential POPL Paper Award*



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 (REPL) in x86-64 machine code. Our correctness theorem ensures that this REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on a breadth of topics including lexing, parsing, type checking, incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping.

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 verified compilers for general-purpose languages has addressed all aspects of a compiler along two dimensions: one, the compilation algorithm for converting a program from a source string to a list of numbers representing machine code, and two, the execution of that algorithm as implemented in machine code.

Our purpose in this paper is to explain how we have verified

Proving a HOL prover sound

Candle: A Verified Implementation of HOL Light

Oskar Abrahamsson ✉
Chalmers University of Technology, Gothenburg, Sweden

Magnus O. Myreen ✉
Chalmers University of Technology, Gothenburg, Sweden

Ramana Kumar ✉
London, UK

Thomas Sewell ✉
University of Cambridge, UK

ITP'22

Abstract

This paper presents a fully verified interactive theorem prover for higher-order logic, more specifically: a fully verified clone of HOL Light. Our verification proof of this new system results in an end-to-end correctness theorem that guarantees the soundness of the entire system down to the machine code that executes at runtime. Our theorem states that every exported fact produced by this machine-code program is valid in higher-order logic. Our implementation consists of a read-eval-print loop (REPL) that executes the CakeML compiler internally. Throughout this work, we have strived to make the

Motivation continued

Wait… How can I *trust* the correctness of these ITPs?

ITPs are very defensively programmed.

Some are even proved to be sound.

Proved with an unverified ITP?

Yes, but see Yang el al. [PLDI'11]

Break for questions!

I like lots of questions.

Lecture outline:

History of HOL ITPs

My work in a HOL ITP

A closer look at HOL4 (demos)

Trust story

Coq

Proving produces ***proof terms*** that are checked by a ***trusted proof checker***.

HOL provers

Proving produces ***values of type*** thm using a ***trusted LCF-style kernel***.

One benefit:
Proofs are not kept around.
Proofs don't occupy space.

HOL logic

HOL logic is really simple

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

Kernel of the HOL light theorem prover

Break for questions!

I like lots of questions.

Demo

Example taken from lecture on compiler verification.

Syntax

Source:

```
exp = Num num
    | Var name
    | Plus exp exp
```

Target 'machine code':

```
inst = Const name num
     | Move name name
     | Add name name name
```

Target program consists of list of `inst`

Source semantics (big-step)

Big-step semantics as relation ↓ defined by rules, e.g.

```
                              lookup s in env finds v
-------------------------     -----------------------
(Num n, env) ↓ n              (Var s, env) ↓ v


(x1, env) ↓ v1          (x2, env) ↓ v2
--------------------------------------
   (Plus x1 x2, env) ↓ v1 + v2
```

called "big-step": each step ↓ describes complete evaluation

Target semantics (small-step)

"small-step": transitions describe parts of executions

We model the state as a mapping from names to values here.

```
step (Const s n) state = state[s ↦ n]
step (Move s1 s2) state = state[s1 ↦ state s2]
step (Add s1 s2 s3) state = state[s1 ↦ state s2 + state s3]

steps [] state = state
steps (x::xs) state = steps xs (step x state)
```

Compiler function

```
compile (Num k) n = [Const n k]

compile (Var v) n = [Move n v]

compile (Plus x1 x2) n =
   compile x1 n ++ compile x2 (n+1) ++ [Add n n (n+1)]
```

generated code stores result in register name (n) given to compiler

Relies on variable names in source to match variables names in target.

Uses names above n as temporaries.

Correctness statement

Proved using proof assistant — demo!

```
∀x env res.
  (x, env) ↓ res ⇒
  ∀state k.
    (∀i v. (lookup env i = SOME v) ⇒ (state i = v) ∧ i < k) ⇒
    (let state' = steps (compile x k) state in
       (state' k = res) ∧
       ∀i. i < k ⇒ (state' i = state i))
```

For every evaluation in the source …

for target state and k, such that …

k greater than all var names and state in sync with source env …

… in that case, the result res will be stored at location k in the target state after execution

… and lower part of state left untouched.

Code for the demo:

```
open HolKernel Parse boolLib bossLib stringTheory combinTheory
     arithmeticTheory finite_mapTheory pairTheory;

val _ = new_theory "demo";

Type name = ``:num``;


(* -- SYNTAX -- *)

(* source *)

Datatype:
  exp = Num num
      | Var name
      | Plus exp exp
End

(* target *)

Datatype:
  inst = Const name num
       | Move name name
       | Add name name name
End

(* -- SEMANTICS -- *)

(* source *)

Inductive eval:
  (T
   ⇒
   eval (Num n, env) n)
  ∧
  ((FLOOKUP env s = SOME v)
   ⇒
   eval (Var s, env) v)
  ∧
  (eval (x1,env) v1 ∧ eval (x2,env) v2
   ⇒
   eval (Plus x1 x2, env) (v1+v2))
End

(* target *)

Definition step_def:
  step (Const s n) state = (s =+ n) state ∧
  step (Move s1 s2) state = (s1 =+ state s2) state ∧
  step (Add s1 s2 s3) state = (s1 =+ state s2 + state s3) state
End

Definition steps_def:
  steps [] state = state ∧
  steps (x::xs) state = steps xs (step x state)
End

(* -- COMPILER -- *)

Definition compile_def:
  compile (Num k) n = [Const n k] ∧
  compile (Var v) n = [Move n v] ∧
  compile (Plus x1 x2) n =
    compile x1 n ++ compile x2 (n+1) ++ [Add n n (n+1)]
End

(* verification proof *)

Theorem steps_append[simp]:
  ∀xs ys state. steps (xs ++ ys) state = steps ys (steps xs state)
Proof
  Induct \\ fs [steps_def]
QED

Theorem eval_ind = eval_ind |> Q.SPEC ‘λ(x,y) z. P x y z’
                            |> SIMP_RULE (srw_ss()) [FORALL_PROD] |> GEN_ALL;

Theorem compile_correct:
  ∀x env res.
    eval (x, env) res ⇒
    ∀k state.
      (∀i v. (FLOOKUP env i = SOME v) ⇒ (state i = v) ∧ i < k) ⇒
      let state' = steps (compile x k) state in
        (state' k = res) ∧
        ∀i. i < k ⇒ (state' i = state i)
Proof
  ho_match_mp_tac eval_ind \\ rpt strip_tac
  \\ fs [compile_def,steps_def,step_def,APPLY_UPDATE_THM]
  \\ last_x_assum $ drule_then strip_assume_tac \\ simp []
  \\ last_x_assum $ qspecl_then [‘k+1’,‘steps (compile x k) state’] mp_tac
  \\ impl_tac >- (rw [] \\ res_tac \\ fs [])
  \\ strip_tac \\ simp []
QED

val _ = export_theory();
```

Break for questions!

I like lots of questions.

Other demos

Operational semantics for Haskell-like language.

The n-bit word type in HOL.

Break for questions!

I like lots of questions.

Lecture outline:

History of HOL ITPs

My work in a HOL ITP

A closer look at HOL4 (demos)

End of lecture