Turning proof assistants into programming assistants

ST Winter Meeting, 3 Feb 2015

Magnus Myréen

CHALMERS
UNIVERSITY OF TECHNOLOGY

Why?

Why combine proof- and programming assistants?

Why proofs? Testing cannot show absence of bugs.

Some care very much about bugs.

(Applicable to specialist code only…)

What is the specification of ***Microsoft Word***?

But what about bugs in compilers,
library routines, OS?

Why?

Why combine proof- and programming assistants?

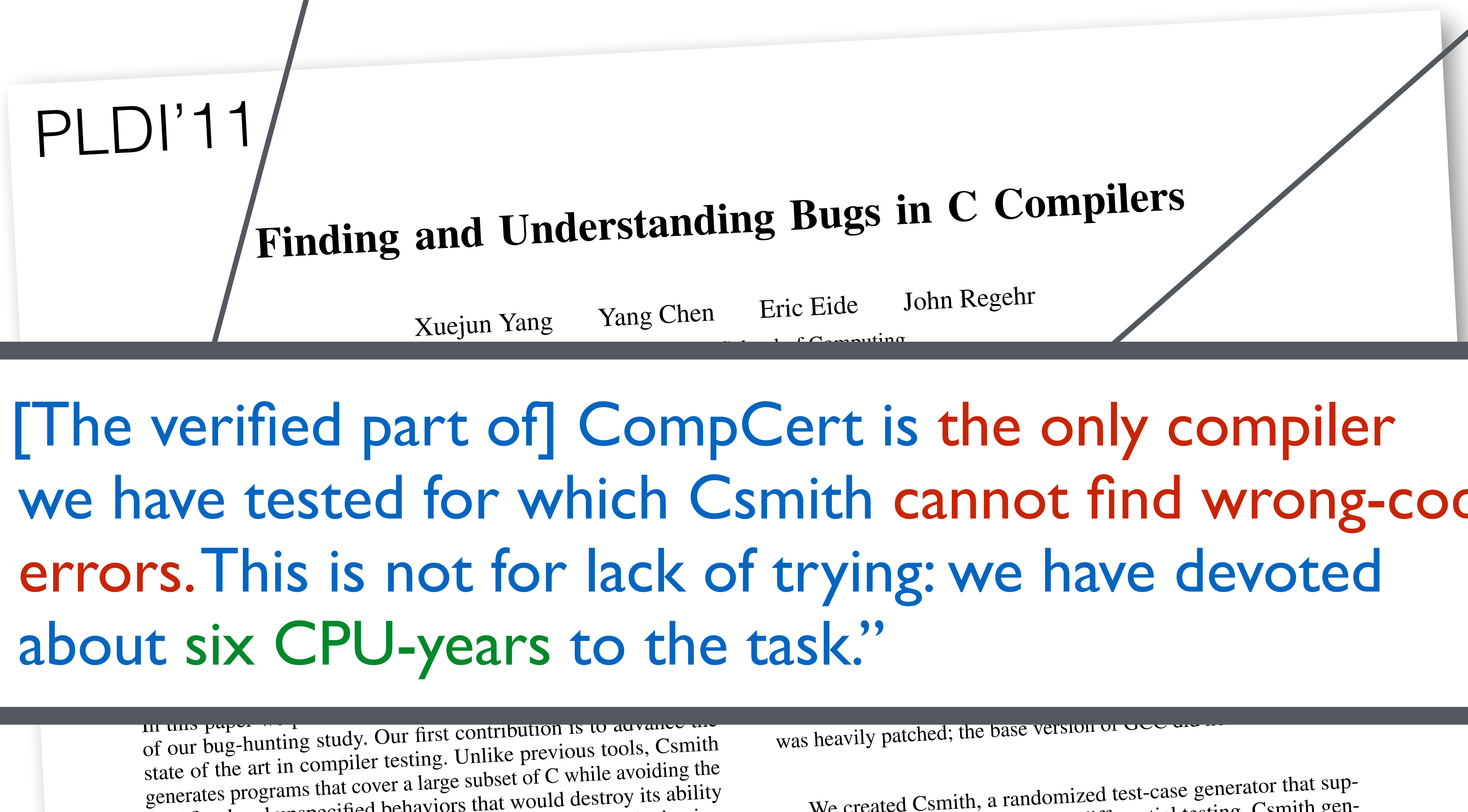
If proof assistants were convenient programming environments, then proofs might become more commonplace.

- Unit proofs, instead of unit tests?
- Proving some key properties of algorithm implementations?

Not necessarily full functional correctness…

Trusting your toolchain

"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

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

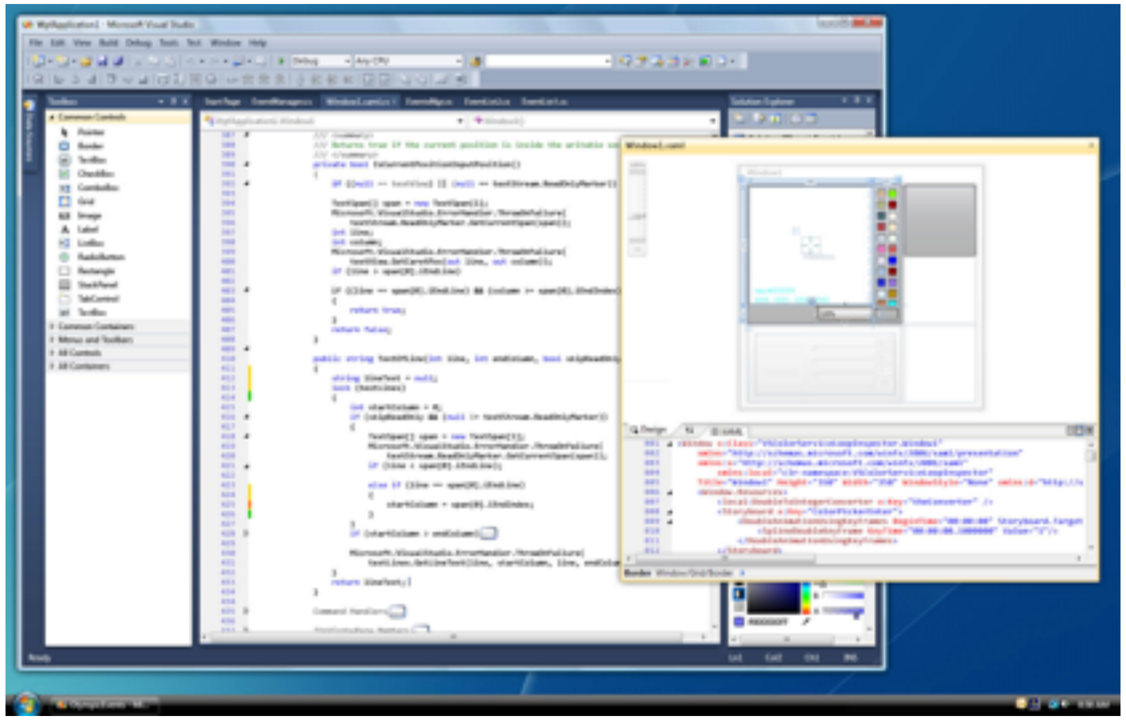
was heavily patched; the base version of GCC

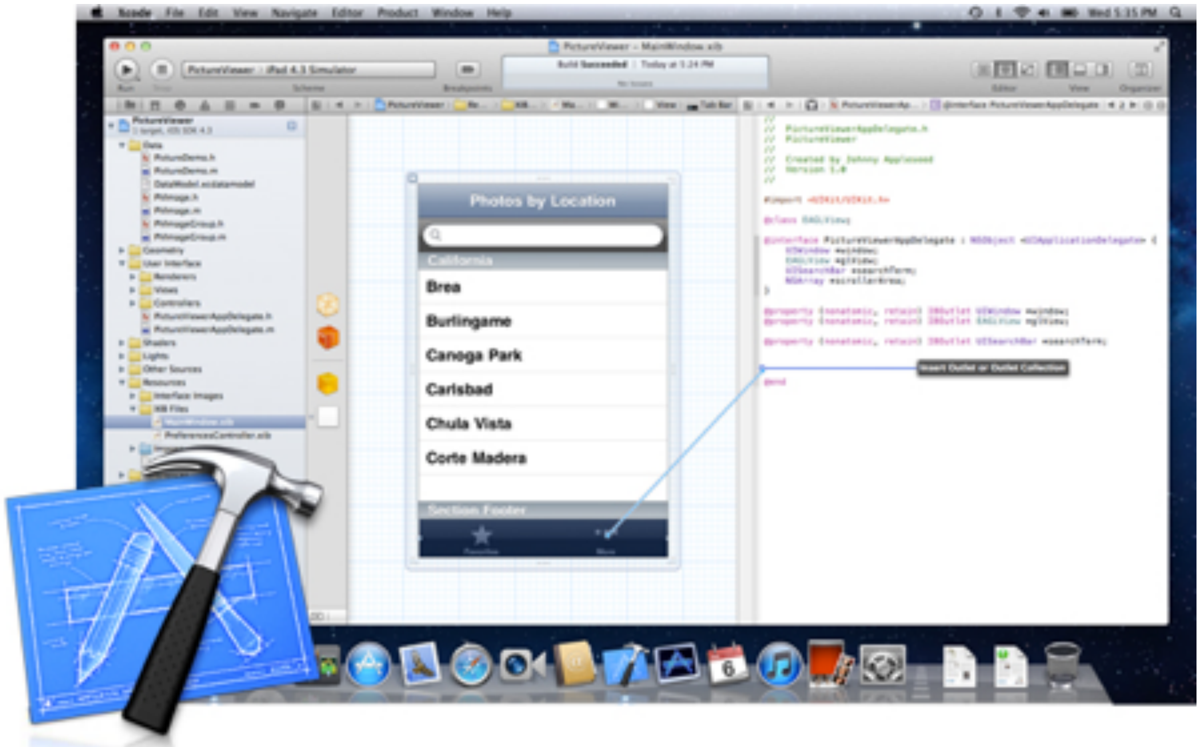
We created Csmith, a randomized test-case generator that sup-

"[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."

Programming assistants

Visual Studio, Xcode, Eclipse



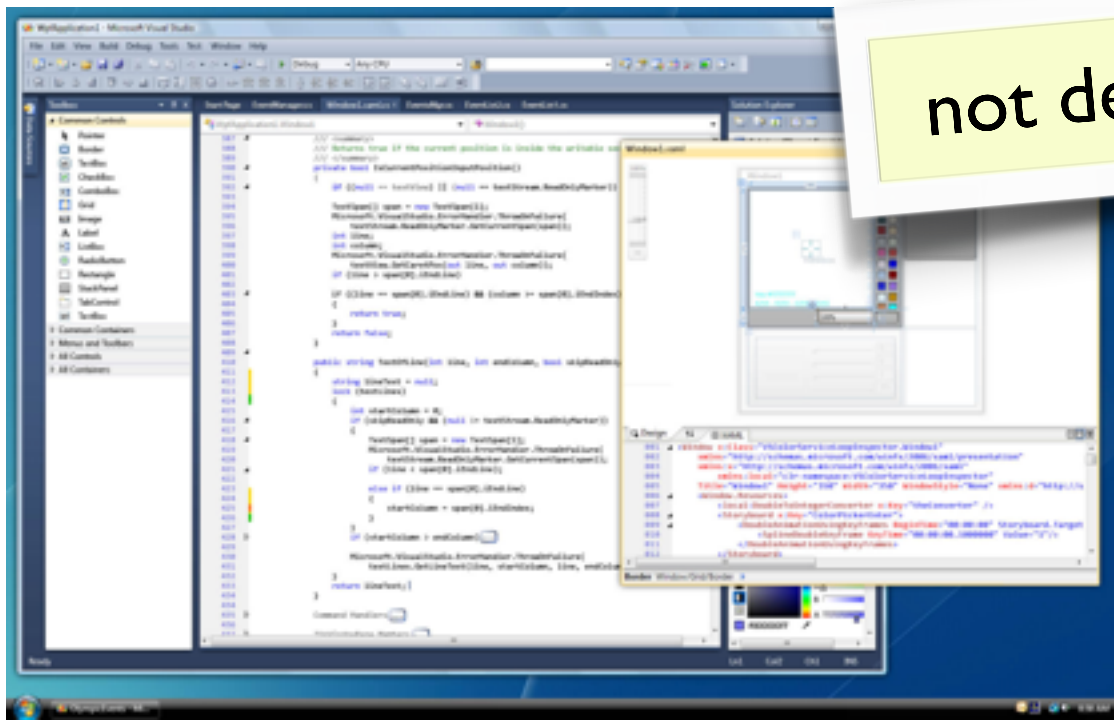


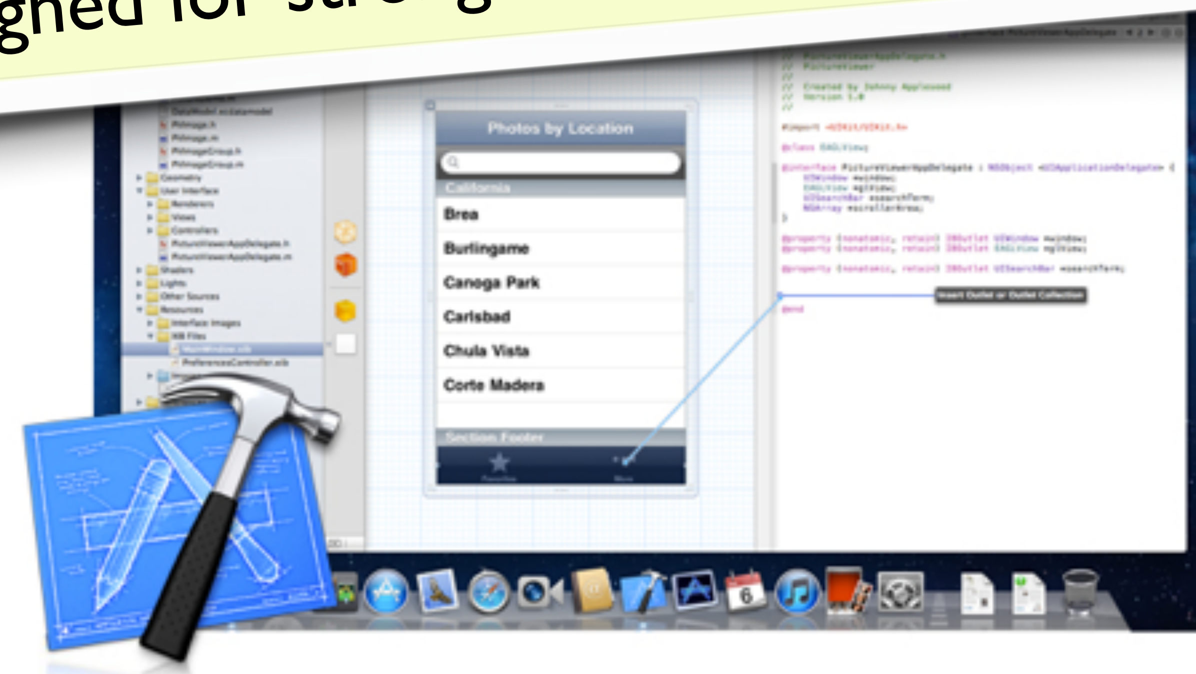
- a helpful program editor
- helps test and refactor code
- debugger
- some can even do complex static analysis

Programming assistants

Visual Studio, Xcode, Eclipse

not designed for strong semantic guarantees





High-assurance code development?

- can be used:
  1. write code in programming assistant
  2. verify code using other tools

what about development life cycle?

Producing high-assurance code

Approaches:

Source code verification (traditional)

e.g. annotate code with assertions and (automatically) prove that program respects the assertions, i.e. never fails

Verification of compiler output (bottom up)

e.g. translation of low-level code (e.g. machine code) into higher-level representation (functions in logic).

Correct-by-constriction (top down)

synthesis of implementations from high-level specifications (e.g. functions in logic)

Trustworthy code

But is the source code good enough for

expressing the specification and

implementation strategy in the same text?

… but that’s what compilers do!

Correct-by-constriction (top down)

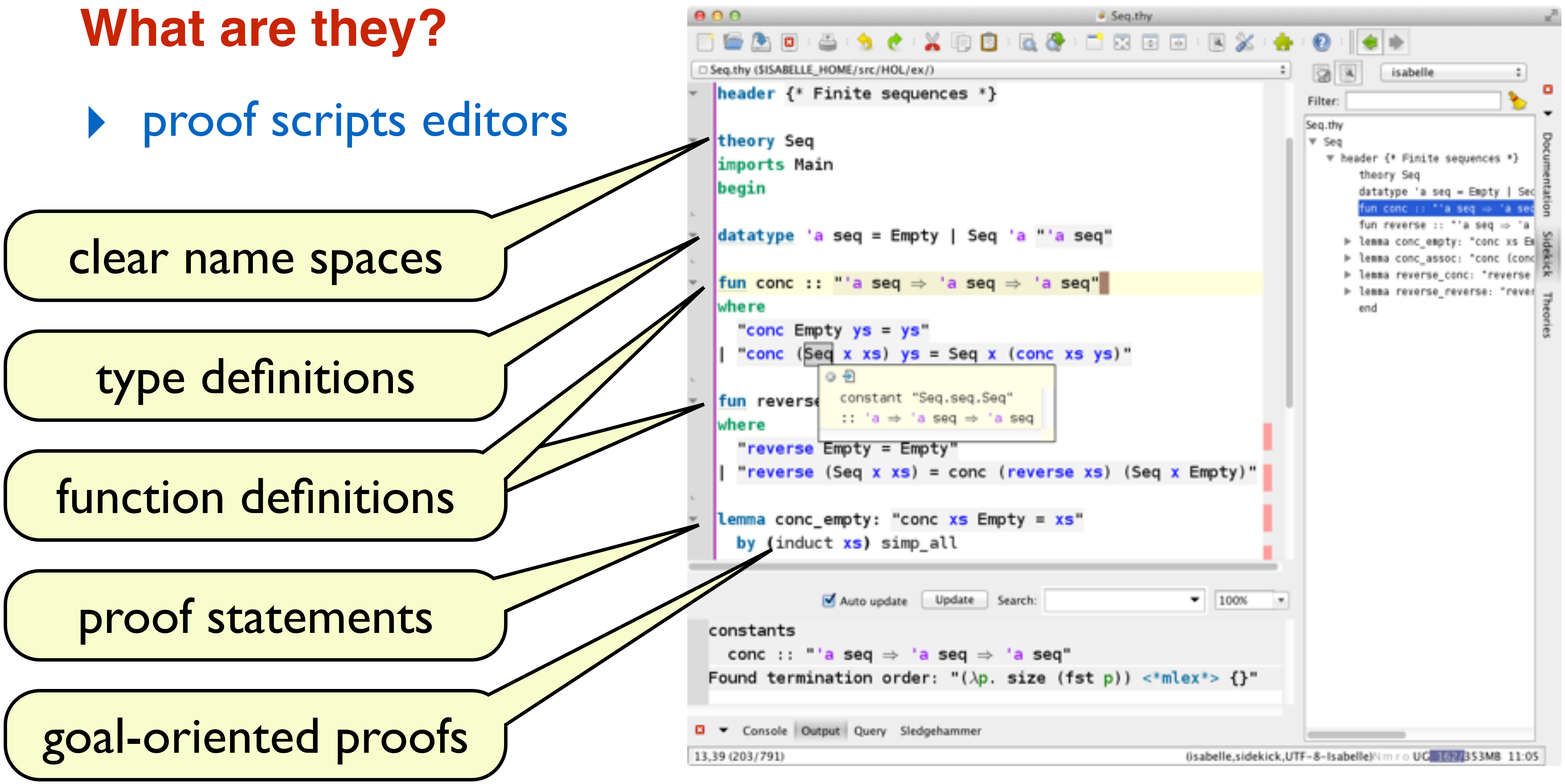
synthesis of implementations from high-level specifications (e.g. functions in logic)

Proof assistants

General-purpose proof assistants: HOL4, Isabelle/HOL, Coq, ACL2…

What are they?

- proof scripts editors

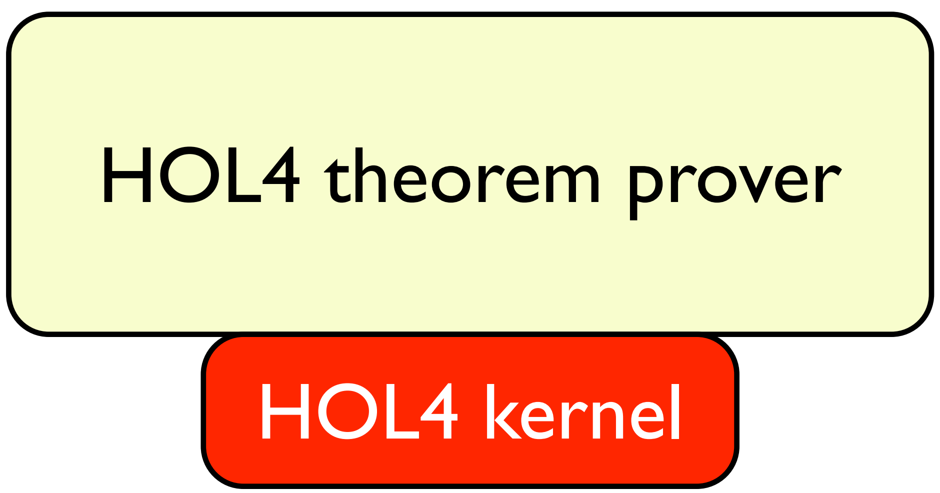


- important feature: ***proof assistants are programmable*** (not shown)

Trustworthy?

Proof assistants are designed to be trustworthy.

HOL4 is a fully expansive theorem prover:



All proofs expand at runtime into primitive inferences in the HOL4 kernel.

The kernel implements the axioms and inference rules of higher-order logic.

Thus all HOL4 proofs are *formal* proofs.

Landmarks

Modern provers are scale well:

Major maths proofs

- Odd Order Theorem, Gonthier et al.
- Kepler Conjecture, Hales et al.
- Four-Colour Theorem, Gonthier

Major code verification proofs

- Correctness of OS microkernel, Klein et al. (NICTA)
- CompCert optimising C compiler, Leroy et al. (INRIA)

These proofs are 100,000+ lines of proof script.

compositional development!

Proof assistants

A closer look:

bottom up-ish

- Correctness of OS microkernel, Klein et al. (NICTA)

  Verified a *deep embedding* of 10,000 *C code*
  w.r.t. a very detailed *semantics of C*
  and a high-level functional specification.
  Proofs also extended down to *machine code* (I helped).

top down

- CompCert optimising C compiler

  Compiler written as *function in logic* (not a deep embedding)
  Correctness theorems proved about this function.
  Function *exported to Ocaml* using an unverified code generator.

Proof assistants

Used as generators of code

inside the prover:

functions in logic

↓

theorem about functions

no proof - - - → Ocaml code

no formal semantics

- CompCert optimising C compiler

  Compiler written as *function in logic* (not a deep embedding)

  Correctness theorems proved about this function.

  Function *exported to Ocaml* using an <u>unverified code generator</u>.

top down

Proof assistants

Are they programming assistants?

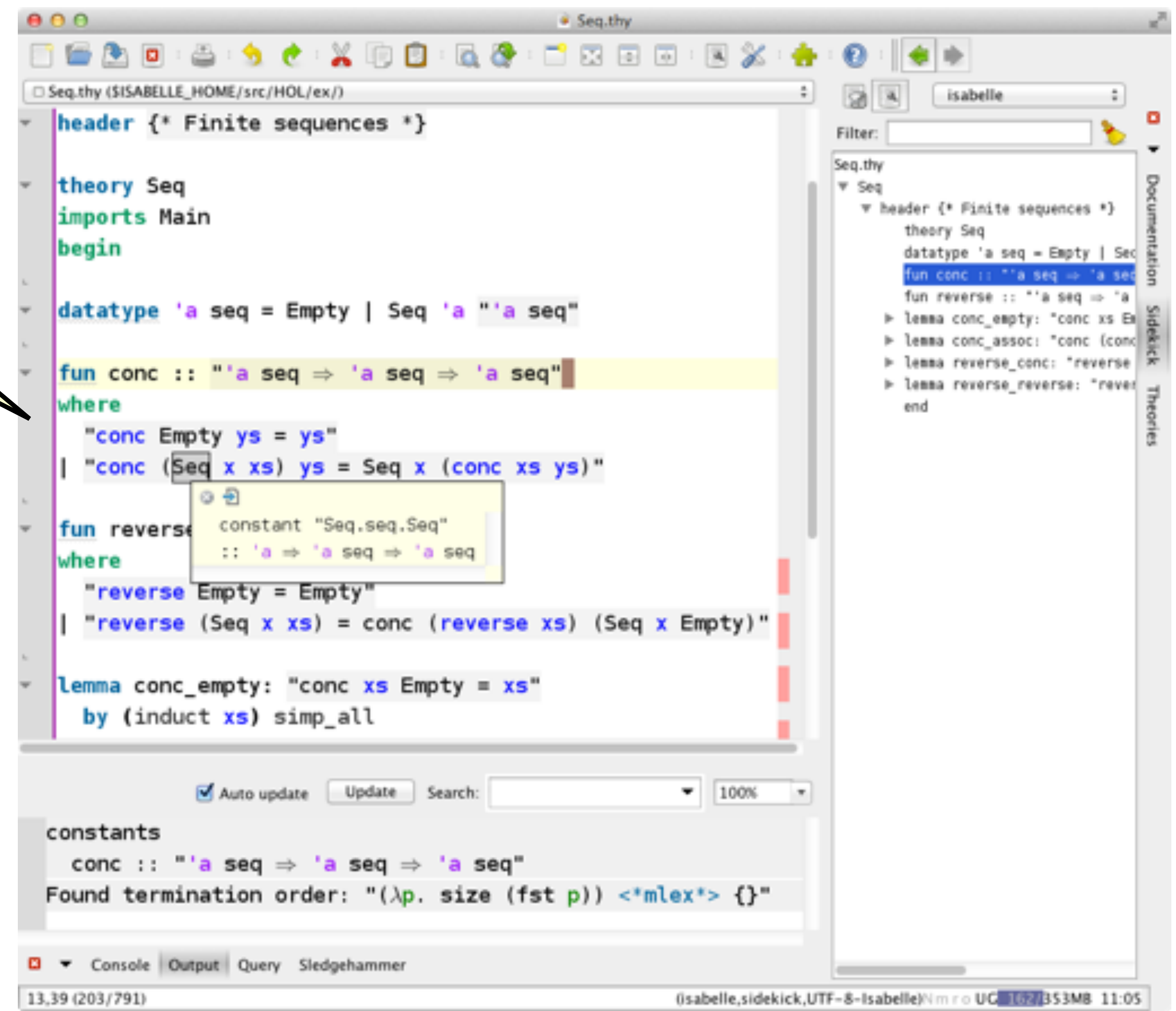
Comparison

proof scripts contain functional programs

… that can be exported to programming languages (Ocaml, SML, Haskell, Scala)

But here: code and spec not necessarily the same.

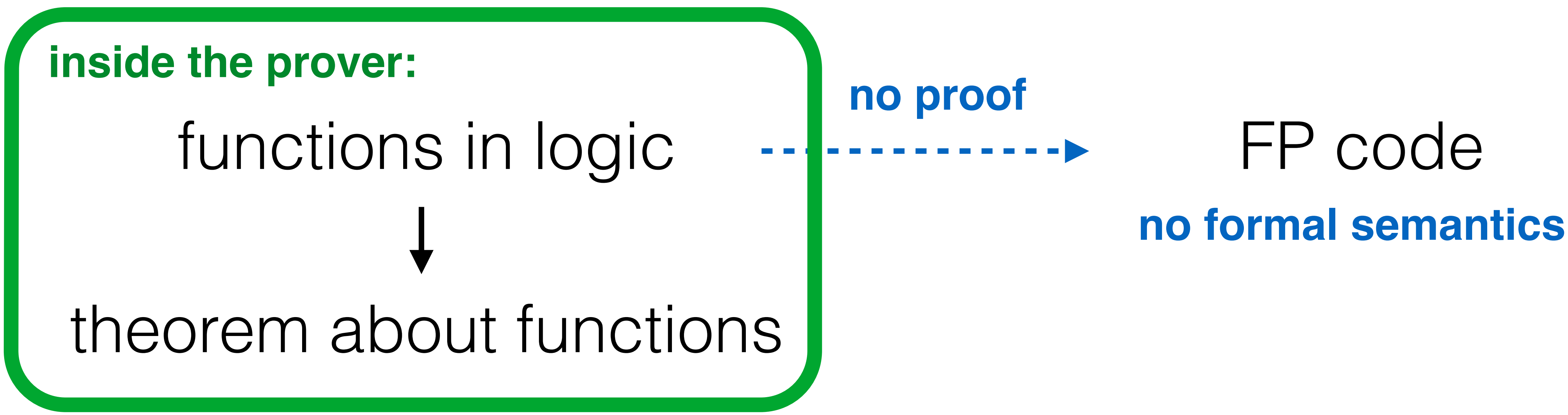
'Code' can be ***abstract***, non-executable.



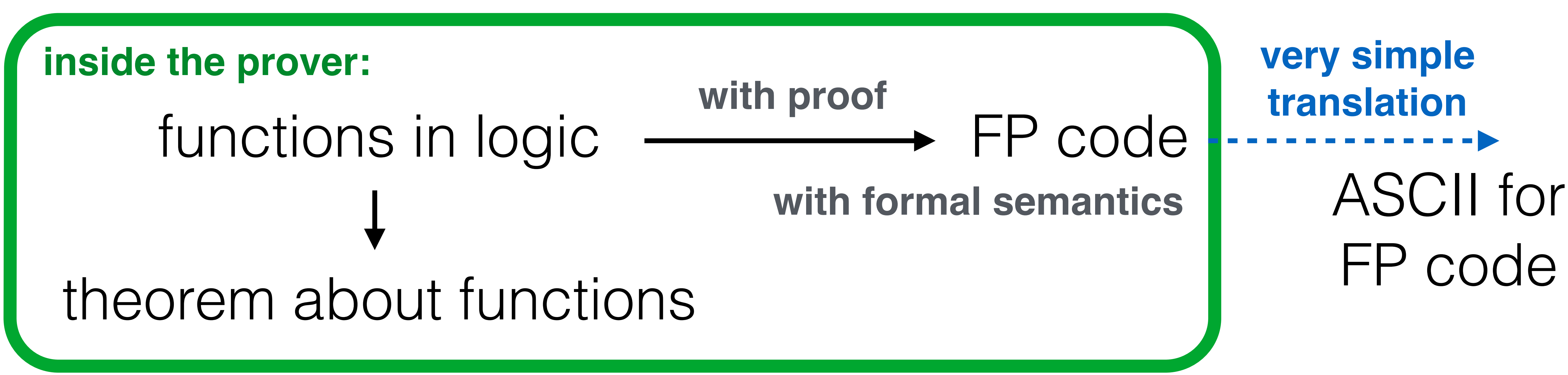
(Isabelle/HOL has nice automation for finding counter examples.)

Trustworthy?

Not to the high standards of fully expansive provers…



A better solution:



Code generation as a trustworthy step

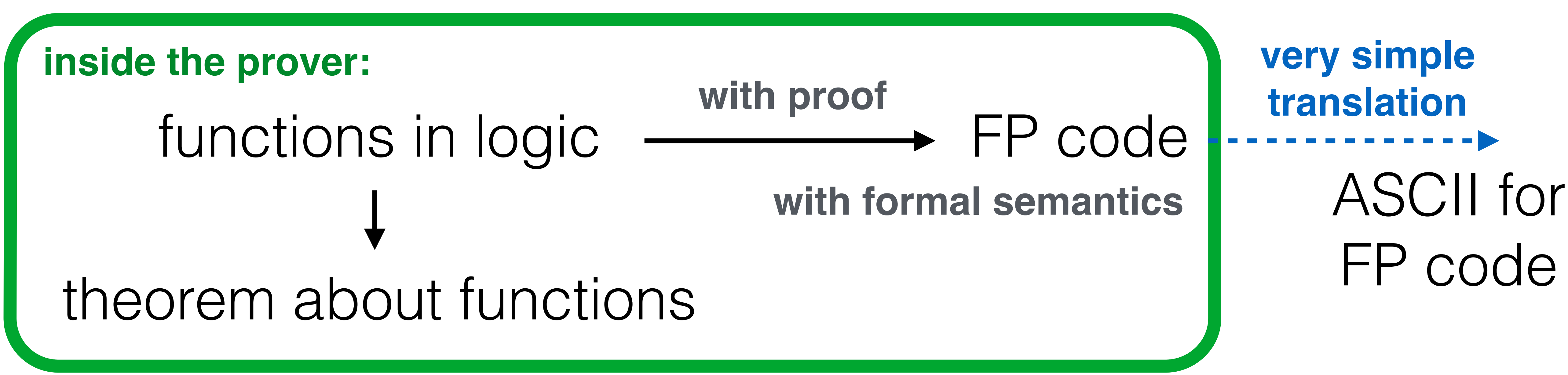
At ICFP'12 (and a JFP'14 paper):

Showed that we can automate proof-producing code generation for FP programs written in HOL4.

The target is **CakeML**, a (large) subset of Standard ML.

… but do we trust Poly/ML to implement **CakeML** according to our semantics?

A better solution:



Going to machine code

Code generation from functions in logic
directly to concrete machine code.

From my PhD thesis: Given function f,

$$f(r_1) \;=\; \text{if } r_1 < 10 \text{ then } r_1 \text{ else let } r_1 = r_1 - 10 \text{ in } f(r_1)$$

our *compiler* generates ARM machine code:

```
E351000A      L:  cmp r1,#10
2241100A          subcs r1,r1,#10
2AFFFFFC          bcs L
```

and automatically proves a certificate HOL theorem:

$$\vdash \; \{\, \mathsf{R1}\ r_1 * \mathsf{PC}\ p * \mathsf{s} \,\}$$
$$p : \texttt{E351000A 2241100A 2AFFFFFC}$$
$$\{\, \mathsf{R1}\ f(r_1) * \mathsf{PC}\ (p{+}12) * \mathsf{s} \,\}$$

Going to machine code

Code generation from functions in logic
directly to concrete machine code.

Has been used to build ***non-trivial applications***:

e.g. a fully verified machine-code implementation of
a **Lisp read-eval-print loop** (with ***dynamic compilation***)

Disadvantage of the approach:

The source *functions in logic* must be stated in a
very constrained format (only tail-rec, only specific types etc.).

Better: going via ML and compilation

We ***can be less restrictive*** using our ***verified compiler*** (POPL'14)

inside the prover:

functions in logic → **with proof** → FP code **with formal semantics**

↓

theorem about functions

applying a verified *compilation* function ↓

ARM, x86, MIPS machine code **with formal semantics** → **very simple translation** → binary

i.e. we can use the compiler's correctness theorem

Interest

We are getting closer to a reality of using proof assistants as program development platforms…

Rockwell Collins

- large avionics/defence contractor in the US
- keen to use this technology
- two concrete projects in mind

NICTA

- developers of the seL4 verified OS microkernel
- keen to build verified user code
- connect everything up to produce complete system with formal guarantees

I/O needed

Problem: real applications need I/O

CakeML has only very basic ***putc*** and ***getc*** char I/O…

Solution (my current work):

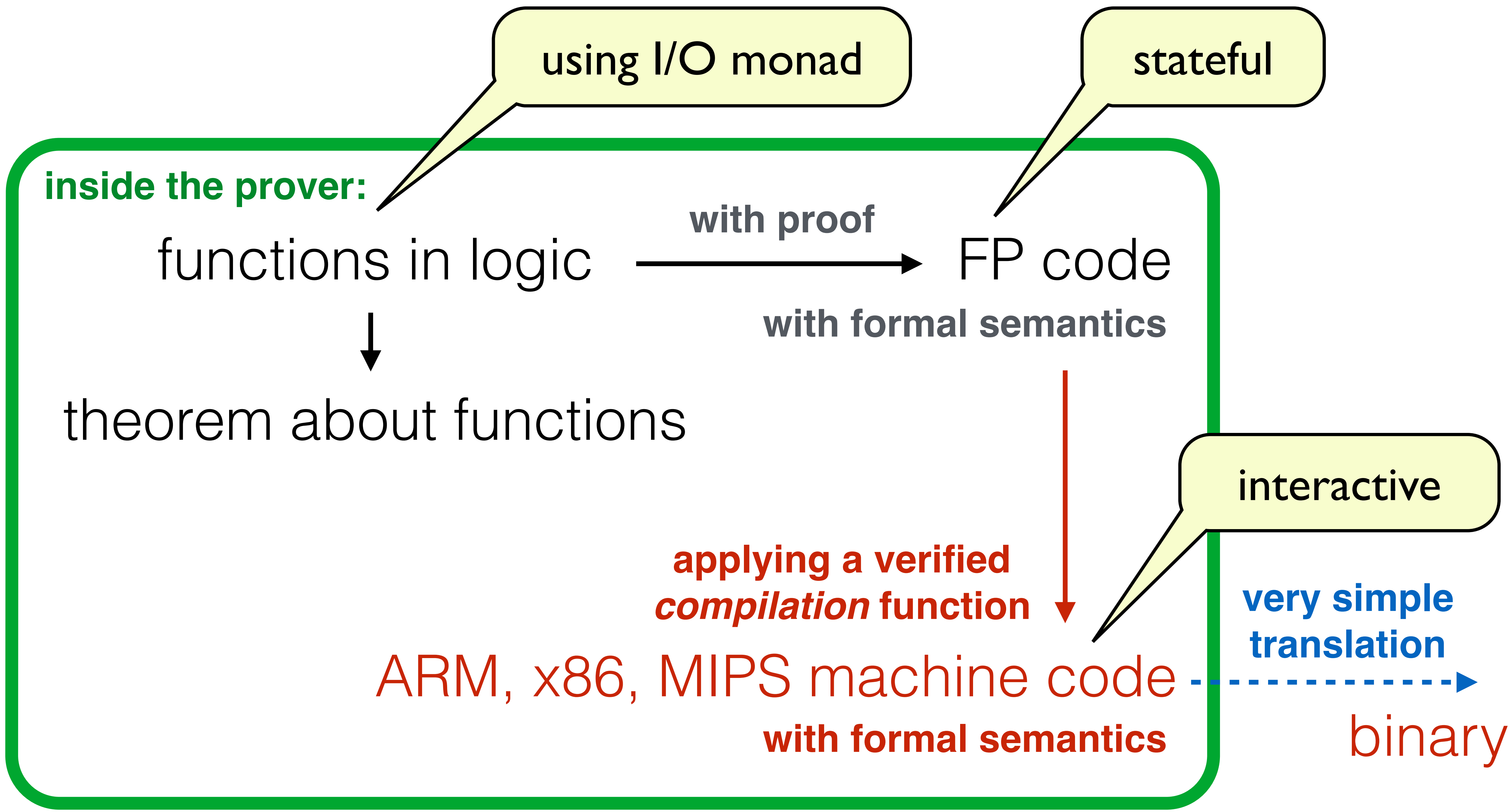
- the next version of the compiler will have I/O through a simple foreign function interface (FFI)

  works through mutable byte arrays that are shared with C

  formally: in the semantics, I/O is modelled by an oracle function (oracle state = rest of the world)

- the new version ***will also include optimisations*** (proper register allocation, better closure conversion, multi-argument function opt)

Going via ML and compilation (revisited)



… but still not good enough

CakeML has ***automatic memory management***…

The correctness theorem allows it to always exit with "not enough memory".

Execution time unpredictable…

In the long run: need language without a GC. **Go?**

or sublanguage of CakeML

Summary

State-of-the-art:

Proof scripts contain functional programs.

Proof automation for data refinement, testing etc.

Can generate (without proofs) FP code.

I've showed that *this can be done with proofs.*

Verified compilation from FP to machine code.

Future vision:

Proof assistants should be able to *automatically produce verified binaries* from FP-style definitions.

Usable in real high-assurance applications.

Questions?

Collaborators:

Ramana Kumar
(Uni. Cambridge)

Scott Owens
(Uni. Kent)