Automatic Verification & Interactive Theorem Proving

AILA 2017
XXVI incontro dell'Associazione Italiana di Logica e sue Applicazioni

Padova, 25-28 settembre 2017

Andrea Asperti

DISI - Department of Informatics: Science and Engineering
University of Bologna
Mura Anteo Zamboni 7, 40127, Bologna, ITALY
andrea.asperti@unibo.it

Formal Verification

A somewhat trivial activity

You need:

- a formal language to express proofs (typically a sequence of logical steps)
- the ability to check step-by-step its correctness

Very similar to type checking (Curry-Howard analogy).

Main Properties

$$\underbrace{p : P}_{p \text{ is a proof of } P}$$

- usually decidable
- complexity may vary (in practice, low)

theorem proving = searching in the proof space

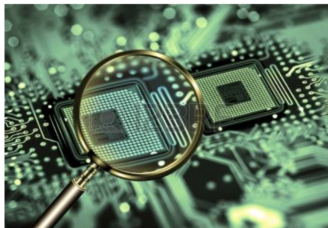
p : P	decidable
? : P	semi-decidable

The kernel

In interactive theorem provers, the unit in charge of verification is a **small** component called kernel

The kernel is the core of the system, in a same way as the cpu is the core of computers.

Somewhere inside your laptop there is a cpu...



The programming environment

The interesting part of a computer is not the cpu, but the software system that allows you to exploit the cpu.

Similarly, in an interactive prover, the interesting (and complex) part is not the kernel, but the set of software components mediating between the user and the kernel, assisting the user in the process of writing formal proofs.

A common misconception

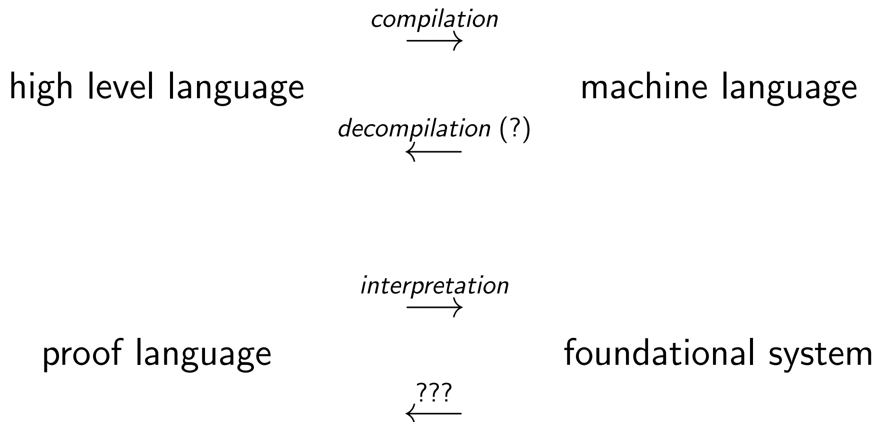
In the '50s, (almost) nobody believed in the possibility to write long assembly programs

Then, people invented high level programming languages, operating systems, and so on.

```
00000000    push   ebp
00000001    mov    ebp, esp
00000003    movzx  ecx, [ebp+arg_0]
00000007    pop    ebp
00000008    movzx  dx, cl
0000000C    lea    eax, [edx+edx]
0000000F    add    eax, edx
00000011    shl    eax, 2
00000014    add    eax, edx
00000016    shr    eax, 8
00000019    sub    cl, al
0000001B    shr    cl, 1
0000001D    add    al, cl
0000001F    shr    al, 5
00000022    movzx  eax, al
00000025    retn
```

In formal verification, we have exactly the same situation: you cannot expect to explicitly write long low-level formal proofs, but you can create an environment easing the prolixity and idiosynchrasies of foundational languages.

important analogy



what we learned

- verifying is easy
- the foundational system does not particularly matter
- interactive provers provide abstractions over the kernel

The proof language: Procedural vs. Declarative style

Redundancy in logical rules

$$\frac{P_1 \quad \dots \quad P_n}{C}\ \boxed{\text{logical rule}}$$

Redundancy in logical rules

procedural

$$\frac{P_1 \quad \ldots \quad P_n}{C}\ \boxed{\text{logical rule}}$$

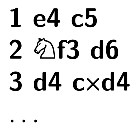
Redundancy in logical rules

declarative

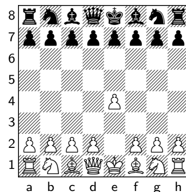
$$\frac{P_1 \quad \dots \quad P_n}{C}\ \text{logical rule}$$

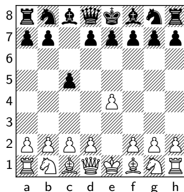
An analogy with chess

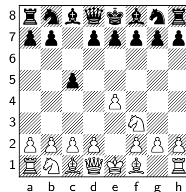
procedural

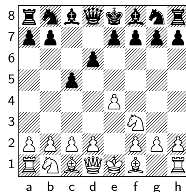


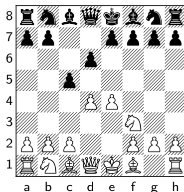
declarative

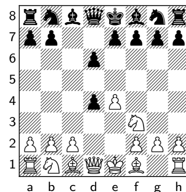












Moves (procedural) vs Positions (declarative)

example

Suppose to have the goal

$$(G) \quad n \neq 0 \rightarrow \exists m, S(m) = n$$

We want to proceed by cases on n, exploiting the following case analysis principle for natural numbers

$$(*) \quad \frac{P(0) \qquad \forall x, P(S(x))}{P(t)}$$

procedural style

We give a command like

cases n

This has the following effect:

1. the system understands by the type of n that the principle to use is (*) in the previous slide
2. (*) is a *schema* and P must be instantiated; the system derives P abstracting the goal (G) w.r.t. n:

$$P = \lambda n, n \neq 0 \rightarrow \exists m, S(m) = n$$

3. the system instantiate P to build the premises of (*), hence reducing (G) to the new goals

$$(G_1) \quad 0 \neq 0 \rightarrow \exists m, S(m) = 0$$
$$(G_2) \quad \forall x, S(x) \neq 0 \rightarrow \exists m, S(m) = S(x)$$

declarative style

$(G_1)\ \ 0 \neq 0 \to \exists m, S(m) = 0$

$(G_2)\ \ \forall x, S(x) \neq 0 \to \exists m, S(m) = S(x)$

by case analysis from G_1 and G_2 we get

$(G)\ \ n \neq 0 \to \exists m, S(m) = n$

declarative style

$(G_1)\ \ 0 \neq 0 \rightarrow \exists m, S(m) = 0$

$(G_2)\ \ \forall x, S(x) \neq 0 \rightarrow \exists m, S(m) = S(x)$

by case analysis from G_1 and G_2 we get

$(G)\ \ n \neq 0 \rightarrow \exists m, S(m) = n$

nice, but having to write it when you can just write

cases n

is a bit disappointing.

pro and cons

procedural

- compact
- unreadable (not meant to be read)
- more suited to backward reasoning

Progenitor: LCF $\rightarrow$ HOL, Isabelle, Coq, ...

declarative

- verbose
- more readable
- suits well with forward reasoning

Progenitor: Mizar $\rightarrow$ HOL mizar-mode, Isar, ...

Research directions

Research directions

Main Goal: reduce the formalization effort

- improved techniques:
  - separation logic (local reasoning)
  - nominal techniques (management of names)
  - ...
- **shift the burden of the proof from user to machine**
  - having a more coarse-grained description of the proof

  means different things from the procedural and declarative point of view

Granularity in procedural style

Proof steps are logical commands (tactics)

We augment granularity by exploiting

more powerful tactics

Examples from Chess:

- openings (e.g. Alekhine's defense)
- variants (e.g. four pawns attack)

- exploit computation (reflection)
- domain-specific (may require configurability)
- automation not so crucial (all or nothing)

Granularity in declarative style

Proof steps are intermediate propositions

We augment granularity by

reducing the number of intermediate goals

Strongly relies on automation to fill the gaps (**hammering**):

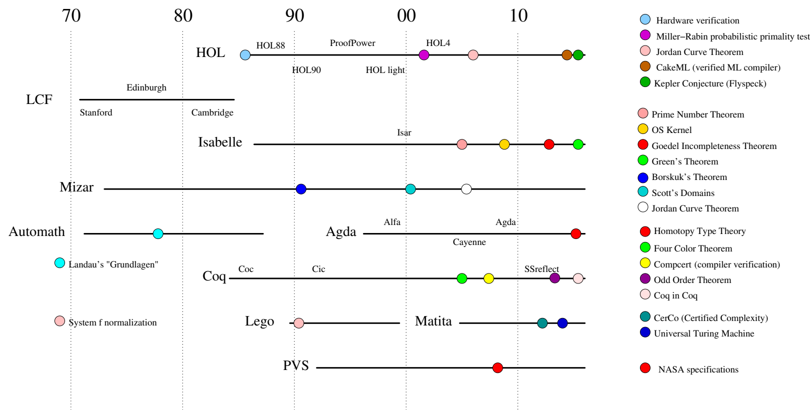
Blanchette, Kaliszyk, Paulson, Urban, JFR 2016

The main ingredients underlying this approach are efficient automatic theorem provers that can cope with hundreds of axioms, suitable translations of richer logics to their formalisms, heuristic and learning methods that select relevant facts from large libraries, and methods that reconstruct the automatically found proofs inside the proof assistants.

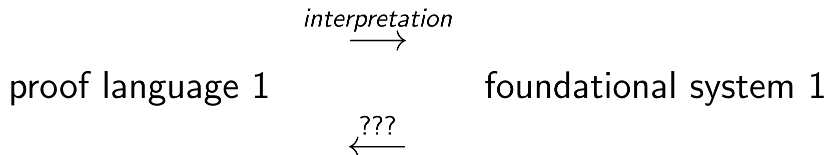
Warning: fragile!!

A babel of systems

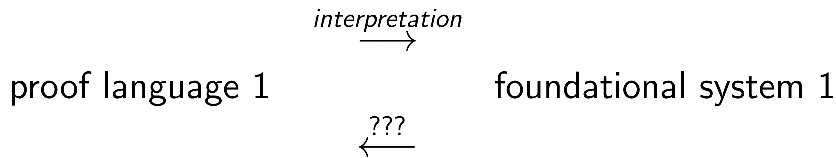
historical picture (many systems and results are missing)

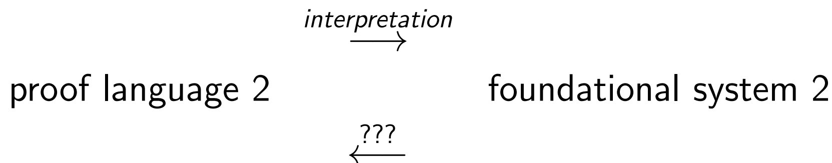


no inter-operability

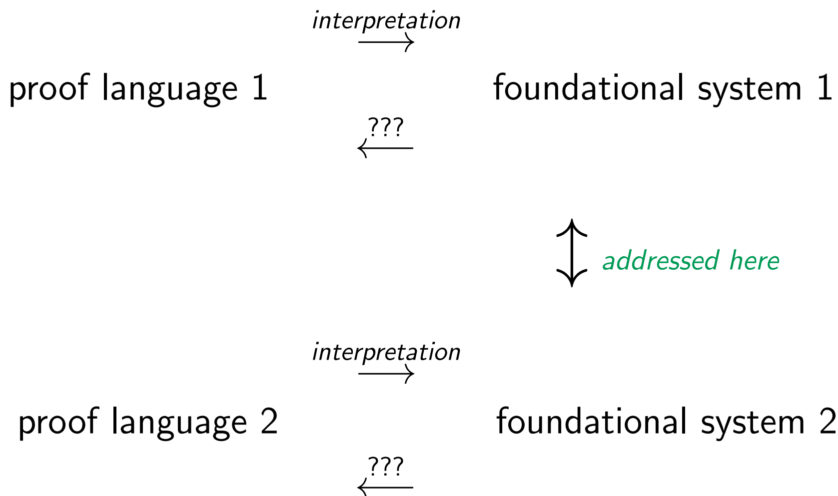


no inter-operability

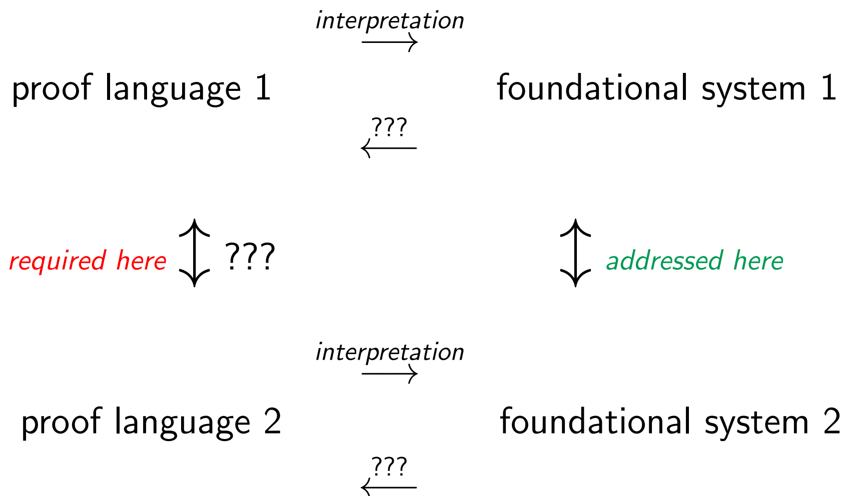




no inter-operability



no inter-operability



No sensible way to compare systems

small kernel? ✓
automation? ✗
reflection? ✓

...

likes honey?

...

Conclusions

Conclusions

- it works! we can formalize and check complex results

Conclusions

- it works! we can formalize and check complex results
- it is a time expensive activity; the cost-benefit ratio is still high

Conclusions

- it works! we can formalize and check complex results
- it is a time expensive activity; the cost-benefit ratio is still high
- formal verification has a steep learning curve; it is a highly specialized activity sensibly more complex than programming

Conclusions

- it works! we can formalize and check complex results
- it is a time expensive activity; the cost-benefit ratio is still high
- formal verification has a steep learning curve; it is a highly specialized activity sensibly more complex than programming
- (just between us) a bit boring and not particularly rewarding (a solipsistic activity)

my personal viewpoint

Do you feel more confident in a mathematical result if it has been formally checked?

Do you feel more confident in a mathematical result if it has been formally checked?

Yes, definitely

my personal viewpoint

Do you feel more confident in a mathematical result if it has been formally checked?

Yes, definitely

Do you feel safer if you know that some of the software components of a critical system have been formally checked?

my personal viewpoint

Do you feel more confident in a mathematical result if it has been formally checked?

Yes, definitely

Do you feel safer if you know that some of the software components of a critical system have been formally checked?

No: too many imponderable parameters