Programming languages and software verification

or: Theorem provers are a P.L. researcher's best friends

Xavier Leroy

INRIA Paris-Rocquencourt

Programming Languages Mentoring Workshop 2013

A landscape of research in P.L.

Design

- Programming paradigms, language features, how to blend them.
- Still very much a black art.
- Limited impact of P.L. research
  (some exceptions: functional programming, reactive programming).

A landscape of research in P.L.

Design

Implementation

- Compilers, virtual machines, run-time systems, ...
- Compiler optimizations & the underlying static analyses
- Parallelization, distribution.

A landscape of research in P.L.

Design

Implementation

Principles and foundations

- Formal semantics
- Program equivalences, algebraic laws, program logics
- Type structure

A landscape of research in P.L.

Design

Implementation

Principles and foundations

Verification of programs

- Showing that a program does what it should (correctness)
- . . . or at least that it does not do anything bad (safety, security).
- Very dependent on the P.L. used and its semantics.

A landscape of software verification

Shiny new languages

Type systems
(basic type and memory safety)
(polymorphism, reuse, . . .)

Model checking
(overapproximation of reachable states)

Widely-used unsafe languages

Deductive verification
(a.k.a. program proof)
(pre- and post-conditions, logical invariants)

Static analysis
(overapproximation of data)

Programmer-supplied logical specifications — *Type declarations* — *Fully automatic*

A landscape of software verification

Shiny new languages

Widely-used unsafe languages

Type systems
(basic type and memory safety)
(polymorphism, reuse, . . .)

Model checking
(overapproximation of reachable states)

Deductive verification
(a.k.a. program proof)
(pre- and post-conditions, logical invariants)

Static analysis
(overapproximation of data)

Programmer-supplied logical specifications

Type declarations

Fully automatic

A landscape of software verification

Shiny new languages

Type systems
(basic type and memory safety)
(polymorphism, reuse, . . .)

Model checking
(overapproximation of reachable states)

Deductive verification
(a.k.a. program proof)
(pre- and post-conditions, logical invariants)

Static analysis
(overapproximation of data)

Widely-used unsafe languages

Programmer-supplied logical specifications — *Type declarations* — *Fully automatic*

The many uses of theorem provers

Automatic theorem provers: (Z3, Vampire, Alt-Ergo, ...)

(Pushbutton, but minimalistic first-order logic.)

- In program provers: to discharge the generated verification conditions
- In static analyzers and related tools: as generic solvers
  (e.g. SLAM/SDV, Terminator, Pex, etc. from MSR, using Z3)

The many uses of theorem provers

Automatic theorem provers: (Z3, Vampire, Alt-Ergo, ...)
(Pushbutton, but minimalistic first-order logic.)

Interactive theorem provers: (Coq, Isabelle, HOL, ACL2, ...)
(Richer specification language, but user writes big parts of the proof)

- To formalize a P.L. and verify specific programs.
  (e.g. the seL4 verified microkernel)
- To formalize a P.L. and prove properties of all programs.
  (E.g. soundness of a type system or a program logic, correctness of program transformations, etc. Also called "metatheory".)

A glimpse of the Coq ITP

A rich specification language (called "Gallina"), including:

- Ordinary mathematics

```
Theorem Fermat's_last:
  forall (x y z n: nat),
  n >= 3 /\ x > 0 /\ y > 0 /\ z > 0 ->
  pow x n + pow y n <> pow z n.
```

A glimpse of the Coq ITP

A rich specification language (called “Gallina”), including:

- Ordinary mathematics
- Recursive function definitions ≈ Haskell, ML

```
Fixpoint length {A: Type} (l: list A): nat :=
  match l with
  | nil => 0
  | hd :: tl => length tl + 1
  end.
```

A glimpse of the Coq ITP

A rich specification language (called "Gallina"), including:

- Ordinary mathematics
- Recursive function definitions $\approx$ Haskell, ML
- Inductive predicates $\approx$ inference rules

$$\frac{E \vdash a : \tau' \to \tau \qquad E \vdash b : \tau'}{E \vdash a\ b : \tau}$$

```
Inductive hastype:
    typenv -> term -> expr -> Prop :=
  ...
  | ty_app: forall E a b t t',
      hastype E a (Arrow t' t) ->
      hastype E b t' ->
      hastype E (App a b) t
```

A glimpse of the Coq ITP

A rich specification language (called "Gallina"), including:

Interactive proof using tactics (the text adventure game)

- User types commands ("tactics") that solve the current goal or transform it into subgoals.
- Limited automation (arithmetic, equational reasoning).
- Under the hood, Coq builds a proof term that is rechecked at the end.

Demo: the Coq ITP in action

A simplistic example of compiler verification:
proving that a compiler always generate machine code that implements the semantics of the source program.

Source language (arithmetic expressions)		*Target language* (stack calculator)
abstract syntax	$\xrightarrow{\text{compiler}}$	abstract syntax
semantics	$\longleftrightarrow$ semantic preservation theorem	semantics

A few notable projects using ITPs

Jinja (Nipkow, Klein, Lochbilher, et al; Munich)
Java-light, the JVM, the Java concurrency model, soundness of bytecode verifier, compiler correctness.

The seL4 verified microkernel (Klein et al, NICTA)
8,000 lines of C, fully verified for correctness & security.

The CompCert verified C compiler (Leroy et al, NICTA)
Just like the demo, but for a 15-pass optimizing compiler from most of ANSI C to ARM/PowerPC/x86.

Metatheory of Standard ML (Crary and Harper, CMU)

A few notable projects using ITPs

The POPLmark challenge (Pierce et al, U.Penn)
Comparing ITPs and formalization styles on the metatheory of $F_{<:}$.

VellVM (Zhao, Zdancewic et al, U.Penn.)
Formalizing the LLVM intermediate representation and passes.

Verified Software Toolchain (Appel et al, Princeton)
Concurrent separation logic for C.

JSCert (Gardner et al, Imperial & Inria)
Formal semantics, program logic, certified analyses for Javascript.

A change in P.L. research practices?

P.L. research and many other areas of CS have a love-hate relationship with mathematical proofs:

- Often a necessity to make research credible.
- Big but shallow proofs (many cases) $\rightarrow$ boring and sketchy.

> *Proofs written by computer scientists often feel like the author is trying to program the reader.*
>
> *(John Mitchell)*

> *The proofs of the remaining 18 cases are similar and make extensive use of the hypothesis that [. . .]*
>
> *(author shall remain anonymous)*

A change in P.L. research practices

> *How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?*
>
> *(The POPLmark challenge, Aydemir et al, TPHOLs 2005)*

8 years later: we're on our way! (20% of POPL 2012 papers).

+ Stronger, more trustworthy results.
+ Can attack bigger, more realistic formalizations.
+ Makes papers crisper and easier to read (and write!).
+ Gives a second chance to students who dislike mathematics.

– Very time consuming.
– Somewhat addictive.
– Proof engineering is hard.

Want to learn more?

A must-read: *Software Foundations* by B. Pierce et al.

> *This electronic book is a one-semester course on Software Foundations — the mathematical theory of programming and programming languages — suitable for graduate or upper-level undergraduate students. It develops basic concepts of functional programming, logic, operational semantics, lambda-calculus, and static type systems, using the Coq proof assistant.*
>
> *The main novelty of the course is that the development is formalized and machine-checked: the text is literally a script for the Coq proof assistant. It is intended to be read hand-in-hand with the accompanying Coq source file in an interactive session with Coq. All the details of the lectures are fully developed in Coq, and the exercises are designed to be worked using Coq.*

In closing. . .

Judicious use of automatic or interactive theorem provers can take your P.L. research to new heights.

Go forth and mechanize!