Proofs and refutations in the formal verification of software systems

Alberto Momigliano

GNCS, Montecatini

15 Febbraio, 2018
Il progetto:

- Certificazione di verificatori automatici del software basati su clausole di Horn con vincoli.
- Partecipanti: Gianluca Amato, Fabio Fioravanti, Maria Chiara Meo, Francesca Scozzari [CHIETI], Alberto Pettorossi, Maurizio Proietti [ROMA], Mauro Ferrari [VARESE], Camillo Fiorentini, AM [MILANO].
- Ma non é quello che Emanuele ha appena descritto?
- Appunto: quindi vi faccio una panoramica di alto livello su alcuni aspetti della relazione tra dimostrazione e controesempi nei metodi formali.
The software crisis (1)

- **The Pentium FDIV (1994):**
  - it affected the floating point unit of early Intel Pentium processors. The processor might return incorrect binary floating point when dividing a number.
  - Intel recalled the defective processors and announced “a pre-tax charge of 475 million against earnings, ostensibly the total cost associated with replacement of the flawed processors.”

- **Ariane 5 Flight 501 (1996):**
  - It took the European Space Agency 10 years and 7 billion dollars to produce a giant rocket capable of hurling a pair of three-ton satellites into orbit.
  - All it took for the rocket to explode was the on-board guidance software trying to stuff a 64-bit number into a 16-bit space and causing an uncaught arithmetic overflow.
...and many, many other instances, see http://www.cse.psu.edu/~gxt29/bug/softwarebug.html

Ah, the good old times where bugs were a problem for big companies, not for the little guys...

Enter **security** bugs:
  - Heartbleed
  - Meltdown and Spectre (OK, those are hardware problems)
  - ...

Not my cup of tea, so let’s stick to software bugs
Formal methods

- The branch of CS that tries to address this issues:
  - **Lightweight** formal methods: specifying critical properties of a system and focusing on finding errors, rather than prove correctness.
    “Spec n Check” is the mantra, up to . . .
  - **Full** correctness: Specify all properties of interest of an entire system and perform a complete proof of correctness

- Trade-off: latter gives the ultimate guarantee, but it’s extremely labour intensive and therefore costly, the former is just a validation, but it is automatic and cost-effective.

- Formal verification proofs are shallow, essentially boring, but very complex due to the large number of cases to distinguish.

- An excellent candidate to **mechanization** and in fact, this has been pursued since the 80’s — huge literature and increasing success story.
Story of my life

- For nearly 20 years I’ve been working in formal verification via interactive theorem provers, where the user drives the proof assistant in the search for a proof, solving the more mechanical tasks.

- Although systems such as Isabelle/HOL, Coq etc. are very powerful and sophisticated, it is a very demanding, often frustrating, day job.
For nearly 20 years I’ve been working in formal verification via interactive theorem provers, where the user drives the proof assistant in the search for a proof, solving the more mechanical tasks.

Although systems such as Isabelle/HOL, Coq etc. are very powerful and sophisticated, it is a very demanding, often frustrating, day job.

Especially when the theorem I’m trying to prove is, ehm, wrong.
For nearly 20 years I’ve been working in formal verification via interactive theorem provers, where the user drives the proof assistant in the search for a proof, solving the more mechanical tasks.

Although systems such as Isabelle/HOL, Coq etc. are very powerful and sophisticated, it is a very demanding, often frustrating, day job.

Especially when the theorem I’m trying to prove is, ehm, wrong. I mean, almost right:

- statement is too strong/weak
- there are minor mistakes in the spec I’m reasoning about
For nearly 20 years I’ve been working in formal verification via interactive theorem provers, where the user drives the proof assistant in the search for a proof, solving the more mechanical tasks.

Although systems such as Isabelle/HOL, Coq etc. are very powerful and sophisticated, it is a very demanding, often frustrating, day job.

Especially when the theorem I’m trying to prove is, ehm, wrong. I mean, almost right:

- statement is too strong/weak
- there are minor mistakes in the spec I’m reasoning about

A failed proof attempt not the best way to debug those kind of mistakes

That’s why I’m inclined to give testing a try, in particular property-based testing.
On Lakatos’ “Proofs and Refutations”

- People more authoritative than me have argued that this dynamic (or to give Hegel his due), this *dialectic*, is typical of mathematical development in general.

- Water-tight deductions from well-defined premises are the (perhaps temporary) end-points of an evolutionary process in which the constituent concepts are initially ill-defined and open-ended but become sharper in the context of a “debate”

- **Counterexamples** are not a final refutations for a theory, but a principled way to refine conjectures and proofs thereof.
My interest is not **program** verification in general, but **semantics engineering**: the study of the **meta**-theory of programming languages (PL).

Designing a PL is far from trivial:

"The truth of the matter is that putting languages together is a very tricky business. When one attempts to combine language concepts, unexpected and counterintuitive interactions arise. At this point, even the most experienced designers intuition must be buttressed by a rigorous definition of what the language means. Of course, this is what programming language semantics is all about "

- J. Reynolds, 1990
On PL’s semantics

▶ One shiny example: the definition of **SML** (Milner, Tofte, Harper and MacQueen, 1997): a complete formalization of the semantics of a major functional PL.
On PL’s semantics

- One shiny example: the definition of SML (Milner, Tofte, Harper and MacQueen, 1997): a complete formalization of the semantics of a major functional PL

- In the other corner (infamously) PHP:

  "There was never any intent to write a programming language. I have absolutely no idea how to write a programming language, I just kept adding the next logical step on the way". (Rasmus Lerdorf, on designing PHP)

- In the middle: lengthy prose documents (viz. the Java Language Specification), whose internal consistency is but a dream...
PL designs and implementations can go wrong.

Type Safety of Java (20 years ago)

**Java is Type Safe — Probably**

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
email: sd and se @doc.ic.ac.uk

**Abstract.** Amidst rocketing numbers of enthusiastic Java programmers and internet applet users, there is growing concern about the security of executing Java code produced by external, unknown sources. Rather than waiting to find out empirically what damage Java programs do, we aim to examine first the language and then the environment looking for points of weakness. A proof of the soundness of the Java type system is a first, necessary step towards demonstrating which Java programs won’t compromise computer security.

We consider a type safe subset of Java describing primitive types, classes, inheritance, instance variables and methods, interfaces, shadowing, dynamic method binding, object creation, null and arrays. We argue that for this subset the type system is sound, by proving that program execution preserves the types, up to subclasses/subinterfaces.
PL designs and implementations can go wrong.

Type Safety of Java (20 years ago)

Java is Type Safe  Probably

Sophia Drossopoulou  Michael Bach

Abstract. Andrews and internet of concepts and the esoteric world of executables. Given that we aim to write programs that do not support dynamic typing, Java is more than just a language, it is a framework.

Vijay Saraswat

A&T Research, 180 Park Avenue, Florham Park NJ 07932

Java is not type-safe, though it was intended to be. Java object may read and modify fields (and invoke methods) private to another object. It may invoke operations not even defined for that

Java is not type-safe. It may read and modify internal Java Virtual Machine (JVM) data structures. It may invoke methods (core dumps). Thus, Java security,
PL designs and implementations can go wrong.

Type Safety of Java and Scala (20 years later)

Java is not type-safe, the type system is not very expressive, and the Java Virtual Machine (JVM) is not type-safe. A Java object may read a non-private field of another object, causing a security violation. It may invoke operations not even defined for that object, e.g., JVM crashes (core dumps). Thus, Java security, which depends strongly on the JVM, is not as secure as it should be.
Computing tools can go wrong.

▶ Suppose, you write a spec, you implement a program, say in C, and prove full correctness. Are you happy?
▶ Well, you need to trust the compiler (and the pretty-printer, and the run time system, etc.)
▶ Big mistake: GCC and LLVM had over 195 bugs [Vu et al PLDI’14]:
▶ **Compcert** the only compiler where no bugs were found, but it’s been written for certification, where the proof is an order of magnitude bigger than the compiler itself.
OK then, what can we prove about PL’s semantics?
- type soundness
- correctness of compiler transformations
- (strong) normalization/cut elimination
- simulation, non-interference . . .

Such proofs are quite standard, but notoriously fragile:
- a lot of overhead
- challenging to keep track of details
- hard to understand interaction between different features
- difficulties increase with size

In practice, they are “write-only”, and thus often “PhD student-powered”, when not left to the reader.
OK then, what can we prove about PL’s semantics?
- type soundness
- correctness of compiler transformations
- (strong) normalization/cut elimination
- simulation, non-interference...

Such proofs are quite standard, but notoriously fragile:
- a lot of overhead
- challenging to keep track of details
- hard to understand interaction between different features
- difficulties increase with size

In practice, they are “write-only”, and thus often “PhD student-powered”, when not left to the reader

Yeah. Right.
Even in this sub-area, computer assisted verification still is
lot of work (even if you’re not burned out like I am)
unhelpful if system has a bug — only worthwhile if we already
“know” the system is correct, not in the design phase!
Even in this sub-area, computer assisted verification still is
- lots of work (even if you’re not burned out like I am)
- unhelpful if system has a bug — only worthwhile if we already “know” the system is correct, not in the design phase!

(Partial) “model-checking” approach to the rescue:
- searches for counterexamples
- produces helpful counterexamples for incorrect systems
- unhelpfully diverges for correct systems
- little expertise required
- fully automatic, CPU-bound

Testing for PL meta-theory:
- Represent object system in a logical framework.
- Specify properties it should have.
- System searches (exhaustively/randomly) for counterexamples.
- Meanwhile, user can try a direct proof (or go to the pub)
Isn’t testing the very thing theorem proving want to replace?

Theorem proving is hard, especially in the design phase, when mistakes are common.

We’re talking here about small faults in specifications, which would break the aforementioned complicated but shallow proofs.

Debugging specifications and theorems via failed proof attempts is way too expensive.

Rather: test a conjecture before attempting to prove it and/or test a subgoal (a lemma) inside a proof.
Testing in combination with theorem proving is probably less controversial now than sometimes ago, when Dijkstra’s ghost loomed large(r):

“Program testing can at best show the presence of errors, but never their absence”. [Notes On Structured Programming, 1970]

“None of the program in this monograph, needless to say, has been tested on a machine” [Introduction to A Discipline of Programming, 1980]

In fact, when Isabelle/HOL broke the ice adopting random testing some 15 years ago, many followed suit:

- a la QuickCheck: Agda (04), PVS (06), Coq (15)
- exhaustive/smart generators (Isabelle/HOL (12))
- model finders (Nitpick, again in Isabelle/HOL (11))
A light-weight validation approach merging two well known ideas:

1. automatic generation of test data, against
2. executable program specifications.

Brought together in QuickCheck (Claessen & Hughes ICFP 00) for Haskell

The programmer specifies in a small logical DSL properties that functions should satisfy

QuickCheck tries to falsify the properties by trying a large number of randomly generated cases.
let rec rev ls = 
  match ls with 
  | [] -> [] 
  | x :: xs -> append (rev xs, [x])

let prop_revRevIsOrig (xs:int list) = 
  rev (rev xs) = xs;;

do Check.Quick prop_revRevIsOrig ;;
>> Ok, passed 100 tests.

let prop_revIsOrig (xs:int list) = 
  rev xs = xs 
do Check.Quick prop_revIsOrig ;;

>> Falsifiable, after 3 tests (5 shrinks) (StdGen (518275965,.. [1; 0]
(not in chronological order)

1. Haskell harness for PBT tools (with Guglielmo Fachini).
2. A reconstruction of PBT in proof-theoretical terms (with Dale Miller and Rob Blanco).
3. The $\alpha$Check tool (with James Cheney and Matteo Pessina).
1. The Haskell thing

Aim: set up a Haskell environment to validate PL’s meta-theory:

- Taking **binders** seriously (no strings!) and declaratively:
- Varying the **testing** strategies (and the tools) from random to enumerative (QC, SmallCheck, LazySmallCheck, Feat);
- Limiting the efforts needed to **configure** and use all the relevant libraries;
  - limiting the manual definition of complex generators
  - producing counterexamples in reasonable time (five minutes)
- See our paper at SEFM17 for details and experimental results
2. PBT and the proof-theory of logic programming

- Functional approaches to PBT are rediscovering logic (programming): unification/mode analysis Coq’s QC, Randomized CLP in PLT-Redex.

- If we take a proof-theoretic view of LP, good things start to happen

- The approach here is to view logic programming computations as search for proofs in a focused sequent calculus.

- Searching for a cex is searching for a proof of a polarized formula like: \( \exists x : \tau \left[ P(x) \land^+ \neg Q(x) \right] \).

- Intuition: generation (\( P \)) is hard, testing (\( Q \)) is but a deterministic computation.

- Added bonus of this view: we can change the underlying logic and do PBT in different setting, for example linear logic.
3. $\alpha$Check

- Our recently released tool: https://github.com/aprolog-lang
- On top of $\alpha$Prolog, a simple extension of Prolog with nominal abstract syntax.
- Equality coincides with $\equiv_\alpha$, support for freshness, abstractions and the $\forall$ quantifier.
- Consider specifications of the form

$$\forall \vec{X}. A_1 \land \cdots \land A_n \supset A$$

- A *counterexample* is a ground substitution $\theta$ such that

$$\mathcal{M} \models \theta(A_1) \land \cdots \land \mathcal{M} \models \theta(A_n) \land \mathcal{M} \not\models \theta(A)$$

- $\alpha$Check searches *exhaustively* for those counterexample with iterative deepening search strategy.
Implementation with \textit{NAF}

- A check \( \forall \vec{a} \forall \vec{X}. A_1 \land \cdots \land A_n \supseteq A \) is basically a bounded query:

\[
\neg \exists \vec{a}. A_1 \land \cdots \land A_n \land \text{gen}(X_1) \land \cdots \land \text{gen}(X_n) \land \neg(A)
\]

- Search for complete (up to the bound) proof trees of all hypotheses
- Instantiate all remaining variables \( X_1 \ldots X_n \) occurring in \( A \) with \textit{exhaustive generator} predicates for all base types, automatically provided by the tool.
- Then, see if conclusion fails using \textit{negation-as-failure}.
- Easy peasy. Still, \textit{NAF} is messy (semantically and computationally); another more declarative technique is available, not discussed here.
Case studies carried out

- Several $\lambda$-calculi (MiniML with references, $\lambda$-zap) where the properties of interests are related to type preservation, up to equivalence algorithms in dependant type theory.
- Code in the “wild”: exercise code whose validity is not known, save for having stood some unit testing.
- The listmachine benchmark by Appel & Leroy in the area of compiler correctness.
- Type System for Secure Flow Analysis — A mild extension of Volpano et al.’s type system as formalized in Nipkow and Klein’s Concrete Semantics
Future Work

- Search for deeper known bugs
  - “value” restriction in ML with references and let-polymorphous.
- Tackle coinductive specs, typically in process calculi:
  - Two process that are pairwise similar but not bisimilar
- Application to the GNCS project to validate the operational semantics of the object level languages we use.
- Look into sub-structural PBT.
Conclusions

- Computer-assisted formal verification of software correctness is crucial in many areas, but potentially a lot of work.
- Testing, in particular PBT, can be an effective complement, prior or during an attempted proof.
- Our work points to the efficacy of PBT in the sub-area of the mechanization of the meta-theory of PL.
- A plurality of approaches, stemming from functional to logic programming, seem a plus.
Thanks!