Utrecht University

How to believe a verified program?

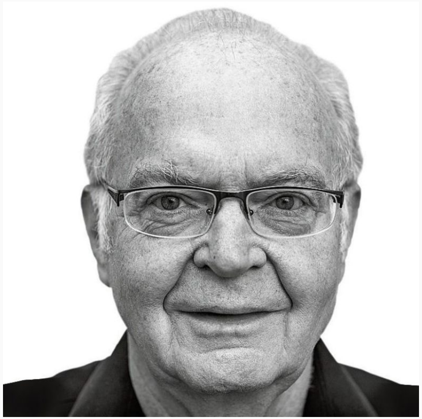
Wouter Swierstra

Utrecht University

Why do any formal verification?

"Program testing can be used to show the presence of bugs, but never to show their absence!" — Edsger W. Dijkstra

Why do only formal verification?



"Beware of bugs in the above code; I have only proved it correct, not tried it." — Donald Knuth

Question?

How to believe a (formally) verified program?

Inspired by Randy Pollack's *How to believe a machine-checked proof?*

Question?

How to believe a (formally) verified program?

Inspired by Randy Pollack's *How to believe a machine-checked proof?*

- When is a program correct?
- Why can there be bugs in verified code?
- What can we do to ensure a verified program really works?
- Can we do do better?

When is a program correct?

Program verification in a nutshell

A verified program consists of three parts

· A computer program;

· A specification of this program in some formal logic;

· A formal proof establishing the program satisfies its specification.

Programs

A computer program

- How do we know what this program means? What is its semantics?
- How can we be sure that the language implementation is correct?
- What else can go wrong?

Specifications

A specification of this program in some formal logic

- What is logic is our spec written in?
- How can we be sure that the logic’s implementation is correct?
- How do we know there is no bug in the spec?

Proofs

A formal proof establishing the program satisfies its specification.

- What is such a proof?
- If the proof is found (semi)automatically, do we believe the computer?
- Formal proofs are not fit for human consumption...
- How can we believe a machine-checked proof?

Programming and proving

A lot of my own work has focussed on programming languages based on *type theory*.

The **programming language** and the **formal logic** are the same!

This, on paper, reduces the number of these questions to resolve.

But still plenty of issues remain...

Programming and proving

A lot of my own work has focussed on programming languages based on *type theory*.

The **programming language** and the **formal logic** are the same!

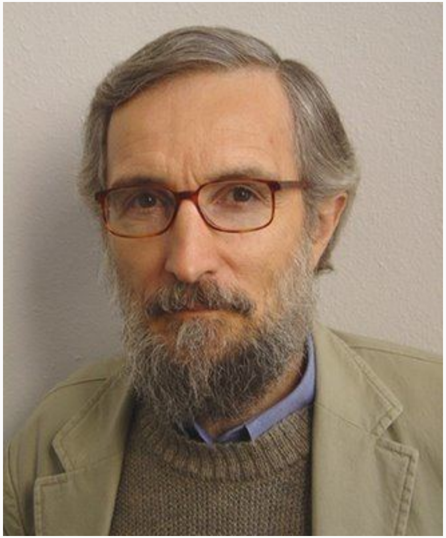
This, on paper, reduces the number of these questions to resolve.

But still plenty of issues remain...

...both in theory and in practice.

Why are there bugs in verified programs?

Theoretical problems in type theory



“Set : Set“

— Per Martin Löf in *A Theory of Types* (1971)

Theoretical problems in type theory

- Set : Set - or rather, the type of all types is a type.
- This was shown to be inconsistent by Jean Yves Girard (1972) and later simplified by Tonny Hurkens (1995) - using a variation of Russell's paradox regarding the set of all sets.

Theoretical problems in type theory

- Set : Set - or rather, the type of all types is a type.
- This was shown to be inconsistent by Jean Yves Girard (1972) and later simplified by Tonny Hurkens (1995) - using a variation of Russell's paradox regarding the set of all sets.

- **Good news** - this is relatively easy to fix using a universe hierarchy, stating that Set : Set_1, $\text{Set}_1 : \text{Set}_2$, etc.

Theoretical problems in type theory

- Set : Set - or rather, the type of all types is a type.
- This was shown to be inconsistent by Jean Yves Girard (1972) and later simplified by Tonny Hurkens (1995) - using a variation of Russell's paradox regarding the set of all sets.
- **Good news** - this is relatively easy to fix using a universe hierarchy, stating that Set : Set_1, Set_1 : Set_2, etc.
- **More good news** - it is highly unlikely you'll run into this problem accidentally.

Implementation problems

Solving this problem in theory, does not mean it is solved in practice...

Implementation problems

Solving this problem in theory, does not mean it is solved in practice...

In Agda, we can check that Set : Set_1

```
x : Set₁
x = Set
```

Implementation problems

Solving this problem in theory, does not mean it is solved in practice...

In Agda, we can check that Set : Set_1

```
x : Set₁
x = Set
```

But what about:

```
WTF : Set₀
WTF = Set₁₈₄₄₆₇₄₄₀₇₃₇₀₉₅₅₁₆₁₅
```

Implementation problems

Solving this problem in theory, does not mean it is solved in practice...

In Agda, we can check that Set : Set_1

```
x : Set₁
x = Set
```

But what about:

```
WTF : Set₀
WTF = Set₁₈₄₄₆₇₄₄₀₇₃₇₀₉₅₅₁₆₁₅
```

Hint: 18,446,744,073,709,551,615 = 2^{63} - 1

Totality and soundness

To ensure the soundness of Agda’s logic, programs in Agda are *total function*:

- no missing cases;
- only structurally recursive definitions;

A program is guaranteed to terminate in finite time.

Another theoretical problem…

Even these guarantees are not enough…

```
data D : Set where
  lam : (D → D) → D

app : D → D → D
app (lam f) x = x
```

Another theoretical problem...

Even these guarantees are not enough...

```
data D : Set where
  lam : (D → D) → D


app : D → D → D
app (lam f) x = x
```

What about non-terminating untyped lambda terms?

```
ω : D
ω = lam (λ x → app x x)


Ω : D
Ω = app ω ω
```

Implementation consequences...

We have managed to write a non-recursive term that diverges - making the logic unsound.

Agda uses a ‘positivity checker’ to rule out such data types.

Sure this is only another problem of only theoretical interest?

Implementation consequences…

We have managed to write a non-recursive term that diverges - making the logic unsound.

Agda uses a ‘positivity checker’ to rule out such data types.

Sure this is only another problem of only theoretical interest?

Porting this code to Haskell and causes the **compiler** to **diverge** (previously) or **crash** (nowadays):

```
ghc: panic! (the 'impossible' happened)
  (GHC version 8.2.1 for x86_64-unknown-linux):
    Simplifier ticks exhausted
  When trying UnfoldingDone x_alB
  To increase the limit, use -fsimpl-tick-factor=N (default 100)
```

The GHC inliner is overly agressive in optimising (non-recursive) code…

Type theory - a language for programs & proofs – in theory...

Type theory ideally a language for both programs and proofs.

Yet most languages focus on interesting *theory* rather than writing compilers.

Type theory - a language for programs & proofs – in theory…

Type theory ideally a language for both programs and proofs.

Yet most languages focus on interesting *theory* rather than writing compilers.

Most systems do not have a fully fledged backend…

Instead, they transpile to other (high level) languages, such as OCaml, Haskell, Scheme, or Javascript – and piggyback on the work of others.

Problems with transpilation

This transpilation—or *extraction*—can often be tweaked and customised.

Generated code often needs to be post-processed to integrate smoothly with other developments.

This is an excellent source of bugs in verified code:

- Extraction in Coq deletes all proofs;
- Customising extraction may invalidate proofs;
- Compilation may change a program’s meaning.

Proof erasure

In Coq, proof arguments (in `Prop`) cannot influence computations.

Hence they can be safely removed from the generated code, without changing the program's outcome.

This is a Good Thing—we should not compute propositions if they are computationally irrelevant.

Proof erasure

In Coq, proof arguments (in `Prop`) cannot influence computations.

Hence they can be safely removed from the generated code, without changing the program's outcome.

This is a Good Thing—we should not compute propositions if they are computationally irrelevant.

Now consider the following example:

```
Definition safeHead {A : Set} (xs : list A)  : xs <> nil -> list A.
```

Proof erasure

We can generate Haskell code for such a program easily enough…

```
[1 of 1] Compiling Main             ( Main.hs, interpreted )
Ok, one module loaded.
*Main> safeHead []
```

Proof erasure

We can generate Haskell code for such a program easily enough...

```
[1 of 1] Compiling Main             ( Main.hs, interpreted )
Ok, one module loaded.
*Main> safeHead []
```

Running our `safeHead` function now produces a run-time error!

```
*** Exception: absurd case
CallStack (from HasCallStack):
  error, called at Main.hs:17:11 in main:Main
```

What happened to our safe & verified definition?

Custom extraction - what might fail...

```
Inductive Bit : Type := O | I.

Definition encodeBool  (b : bool) : Bit :=
  match b with
  | true => I
  | false => O
  end.

Lemma encodeTrue : encodeBool true = I. reflexivity. Qed.

Extraction Language Haskell.
Extract Inductive bool => "Bool" [ "False" "True" ].
Recursive Extraction encodeBool.
```

Custom extraction - what might fail...

Now fire up GHCi and test our code:

```
[1 of 1] Compiling Main             ( Main.hs, interpreted )
Ok, one module loaded.
*Main> encodeBool True
0
```

Custom extraction - what might fail...

Now fire up `GHCi` and test our code:

```
[1 of 1] Compiling Main             ( Main.hs, interpreted )
Ok, one module loaded.
*Main> encodeBool True
0
```

What went wrong?

Custom extraction - what might fail...

Now fire up GHCi and test our code:

```
[1 of 1] Compiling Main             ( Main.hs, interpreted )
Ok, one module loaded.
*Main> encodeBool True
0
```

What went wrong?

The extraction mapped Coq's `false` to Haskell's `True`...

Compilation may change semantics

- Erasing proofs may change a function’s strictness…
- ‘Optimisations’ such as mapping natural numbers to `Int32` may introduce overflow bugs;
- Inlining, deleting singletons, and other transformations change a program’s performance—for better or for worse.
- Compilation may even change function types, such as removing singleton arguments.

Many of these points are not specific to Coq.

Recent work better integration between the ‘proof assistant’ and ‘programming language’ – such as `agda2hs` – is a step in the right direction.

What can we do to ensure a verified program really works?

Certified compilation for smart contracts

- PlutusTx is a dialect of Haskell for writing smart contracts on the Cardano blockchain;
- Each smart contract written in a subset of Haskell;
- Each contract is compiled & deployed on the the blockchain (immutable once committed)
- These programs have to be extremely well-behaved!
- Is the code deployed on the blockchain correct?

Certified compilation for smart contracts

- PlutusTx is a dialect of Haskell for writing smart contracts on the Cardano blockchain;
- Each smart contract written in a subset of Haskell;
- Each contract is compiled & deployed on the the blockchain (immutable once committed)
- These programs have to be extremely well-behaved!
- Is the code deployed on the blockchain correct?

How do we know there’s no bug in the Plutus compiler?

Specifying the compiler: translation relations

The heart of the compiler is a series of optimisation passes, manipulating Plutus's Intermediate Representation (PIR).

- The compiler outputs the before and after AST associated with every pass.
- For each compiler pass, we define a *translation relation*, characterising that pass.
- We write $t_{pre} \triangleright t_{post}$ when the AST t_{pre} may be transformed into an (optimised) target AST t_{post}.
- Each such relation can be defined inductively in Coq, independent of the compiler implementation.
- We can check each run of the compiler against its spec.

Example: inlining

$$\frac{\Gamma(x) = t' \qquad \Gamma \vdash t' \triangleright t}{\Gamma \vdash x \triangleright t}\ [\text{Inline-Var}_1]$$

$$\frac{}{\Gamma \vdash x \triangleright x}\ [\text{Inline-Var}_2]$$

$$\frac{\Gamma \vdash t_1 \triangleright t_1' \qquad (x, t_1), \Gamma \vdash t_2 \triangleright t_2'}{\Gamma \vdash \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \triangleright \textbf{let}\ x = t_1'\ \textbf{in}\ t_2'}\ [\text{Inline-Let}]$$

$$\frac{\Gamma \vdash t_1 \triangleright t_1' \qquad \Gamma \vdash t_2 \triangleright t_2'}{\Gamma \vdash t_1\ t_2 \triangleright t_1'\ t_2'}\ [\text{Inline-App}]$$

$$\frac{\Gamma \vdash t_1 \triangleright t_1'}{\Gamma \vdash \lambda x.t_1 \triangleright \lambda x.t_1'}\ [\text{Inline-Lam}]$$

Translation relations - semantics preservation?

Proving the compiler preserves the semantics in each pass is now ‘easy’…

For each pass of the compiler, we need to show:

$$\forall p, q.\ \ p \rhd q \Rightarrow p \equiv q$$

Where we consider $p \equiv q$ when the two programs are contextually equivalent.

But we can prove this – independent of the inliner heuristics, compiler’s magic numbers, etc.

Certification

How can we check that a program was compiled correctly?

- We run the compiler, generating intermediate ASTs;
- We check that subsequent compiler pass has behaved according to its spec;
- We prove once and for all that each compiler pass is semantics preserving.

Hence we can be sure that the generated code behaves the same as the input program.

Proof transport

Contextual equivalence of the source program p and generated code q is a very powerful property.

$$\forall C. \quad C[p] \cong C[q]$$

In particular, assume we prove a property P of our program in our favourite proof assistant, automated theorem prover, or just on pen and paper.

Proof transport

Contextual equivalence of the source program p and generated code q is a very powerful property.

$$\forall C. \quad C[p] \cong C[q]$$

In particular, assume we prove a property P of our program in our favourite proof assistant, automated theorem prover, or just on pen and paper.

As there is no context that distinguishes p and q, we know that in particular:

```
λ x → if P (☐ (x)) then true else false
```

In other words, the target code satisfies P too!

We can *transport* proofs *independently* of the technology used (provided the property P is decidable).

Can we do better?

What assumptions are we making?

When we believe a verified program works, we are assuming at least:

- The correctness of the hardware, operating system, and software to render the program and proof files;
- The correctness of the parser to read in these files, translating them to some internal representation.
- The correctness of the compiler to translate program code to executable machine instructions with the same behaviour.
- The correctness of the proof checker, the underlying hardware, and everything in between.

What assumptions are we making?

When we believe a verified program works, we are assuming at least:

- The correctness of the hardware, operating system, and software to render the program and proof files;
- The correctness of the parser to read in these files, translating them to some internal representation.
- The correctness of the compiler to translate program code to executable machine instructions with the same behaviour.
- The correctness of the proof checker, the underlying hardware, and everything in between.

If you think about it, it’s a miracle computers work at all!

Is there any hope?

It seems infeasible to try and Verify All The Things!

The whole technology stack is ‘*too complex to believe by direct understanding*’.

Is there any hope?

It seems infeasible to try and Verify All The Things!

The whole technology stack is '*too complex to believe by direct understanding*'.

But the situation is not quite so bad:

- Compile using different compilers…
- Run on different operating systems…
- On different architectures…
- On different times of day…

It is highly unlikely that these are all wrong in the same way.

Empirical evidence is also evidence!

The limitations of verification

Even verified software can break:

- Writing formal specifications is hard! Who says there's no mistake in the spec?
- Verified compilers or serious back-ends for proof assistants are far too rare.
- No program exists in isolation – how can we be sure our verified code will be called the way we expect?
- No program is executed in isolation: the network may drop, a database can go down, etc.

The limitations of verification

Even verified software can break:

- Writing formal specifications is hard! Who says there's no mistake in the spec?
- Verified compilers or serious back-ends for proof assistants are far too rare.
- No program exists in isolation – how can we be sure our verified code will be called the way we expect?
- No program is executed in isolation: the network may drop, a database can go down, etc.

Pulling the plug will cause a verified program to fail too.

Perspectives for trustworthy verified programs

Type theory is a language for programs and proofs.

- If we have a specification for a given function in our type theory – why not turn this into a test?
- Can we (automatically) derive a (semi)decision procedure `check : A → B → Bool` given a specification `S : A → B → Set`? Tools like QuickChick are heading in this direction.
- If so, we can enrich any extracted code with *contracts* that the code we actually run behaves as expected.
- If we erase proofs, such as in the `safeHead` example, we will still get a dynamic failure – but it is clear who is at fault.

Perspectives for trustworthy verified programs

Formal verification is not enough.

- We write unit tests to check the functional correctness of one piece of code.
- Formal verification can ensure that code is *always* well behaved.

Perspectives for trustworthy verified programs

Formal verification is not enough.

- We write unit tests to check the functional correctness of one piece of code.
- Formal verification can ensure that code is *always* well behaved.
- But testing typically also focuses on *integration* testing – does the entire system behave expected?
- So where is the holistic approach to verification?

Perspectives for trustworthy verified programs

If there is one verification technology that can tackle this – it has to be type theory!

We can program & prove everything:

- we can model the types of other languages;
- write generic programs to marshall data back and forth;
- check data integrity dynamically and fail predictably when we encouter errors.

We may not be able to rule out *all* failures - but we enforce the inevitable dynamic checks are always made.

Conclusions

- Verified programs can still ‘go wrong’ – for countless reasons.

Conclusions

- Verified programs can still ‘go wrong’ – for countless reasons.
- Software today is so complex, writing flawless code with an empty trusted code base is impossible.

Conclusions

- Verified programs can still ‘go wrong’ – for countless reasons.
- Software today is so complex, writing flawless code with an empty trusted code base is impossible.
- If we embrace this imperfection, there is still much fun to be had.

References & inspiration

- Randy Pollack's *How to Believe a Machine-Checked Proof*
- Stephen Dolan's *Counterexamples in Type Systems*
- Jacco Krijnen, Manuel Chakravarty, Gabriele Keller, Wouter Swierstra, *Translation certification for smart contracts*
- Wouter Swierstra, *xmonad in Coq: programming a window manager in a proof assistant*

Questions?