Specification, Proof and Correctness

Derek Sorensen

Computer Laboratory, University of Cambridge, UK

5 February 2024

Proof, Formal and Otherwise

- Mathematical proof
- Formal proof
- Proof in Formal Methods
- Zero-Knowledge Proofs

Mathematical Proof

- **A convincing argument**
- Notion of "rigor" developed in the 1920s
- Hilbert's program

Suppose $\sqrt{2}$ is rational.

That is, $\sqrt{2} = \frac{p}{q}$ for some $p \in \mathbb{Z}$ and $q \in \mathbb{N}$.

We can assume the fraction is in lowest terms.
That is, p and q share no common factors.

Then $\sqrt{2}\,q = p$

$2q^2 = p^2$

So p^2 is a multiple of 2,
therefore p must be a multiple of 2.

Write $p = 2m$.

Then $2q^2 = (2m)^2$

$2q^2 = 4m^2$

$q^2 = 2m^2$.

So q^2 is a multiple of 2,
therefore q is a multiple of 2.

Thus p and q share a common factor.
This is a contradiction!
Thus $\sqrt{2}$ is irrational.

Formal Proof

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash A}{\Gamma,\ x : A \vdash} \qquad \frac{\Gamma \vdash \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B \qquad A \simeq_{\beta\eta} B}{\Gamma \vdash t : B}$$

$$\frac{\Gamma \vdash A \qquad \Gamma,\ x : A \vdash B}{\Gamma \vdash \Pi x : A.\ B} \qquad \frac{\Gamma,\ x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\ t : \Pi x : A.\ B} \qquad \frac{\Gamma \vdash t : \Pi x : A.\ B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B\{x := u\}}$$

$$\frac{\Gamma \vdash A \qquad \Gamma,\ x : A \vdash B}{\Gamma \vdash \Sigma x : A.\ B} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B\{x := t\}}{\Gamma \vdash (t, u) : \Sigma x : A.\ B} \qquad \frac{\Gamma \vdash t : \Sigma x : A.\ B}{\Gamma \vdash \pi_1(t) : A}$$

$$\frac{\Gamma \vdash t : \Sigma x : A.\ B}{\Gamma \vdash \pi_2(t) : B\{x := \pi_1(t)\}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbb{P}} \qquad \frac{\Gamma \vdash A : \mathbb{P}}{\Gamma \vdash \underline{A}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathcal{U}} \qquad \frac{\Gamma \vdash A : \mathcal{U}}{\Gamma \vdash \underline{A}}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t \equiv_A u : \mathbb{P}} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathbf{refl}_{\equiv}\ t : \underline{t \equiv_A t}}$$

$$\frac{\Gamma,\ x : A,\ p : \underline{t \equiv_A x} \vdash P \qquad \Gamma \vdash q : \underline{t \equiv_A t'} \qquad \Gamma \vdash u : P\{x := t,\ p := \mathbf{refl}_{\equiv}\ t\}}{\Gamma \vdash J_{\equiv}(P,\ q,\ u) : P\{x := t',\ p := q\}}$$

$$(\lambda x : A.\ t)\ u \simeq_{\beta\eta} t\{x := u\} \qquad \lambda x : A.\ t\ x \simeq_{\beta\eta} t$$

$$\pi_1(t, t') \simeq_{\beta\eta} t \qquad \pi_2(t, t') \simeq_{\beta\eta} t' \qquad (\pi_1(t), \pi_2(t)) \simeq_{\beta\eta} t$$

Formal Proof

$$
\begin{aligned}
& \mathsf{A}[70:\tau_0, 70:\tau_1] \mid \mathsf{B}[30:\tau_0, 10:\tau_1] \\
\xrightarrow{\mathsf{A}:\mathsf{dep}(70:\tau_0, 70:\tau_1)} \; & \mathsf{A}[70:\{\tau_0,\tau_1\}] \mid \mathsf{B}[30:\tau_0, 10:\tau_1] \mid \{70:\tau_0, 70:\tau_1\} \\
\xrightarrow{\mathsf{B}:\mathsf{swap}(30, \tau_0, \tau_1)} \; & \mathsf{A}[70:\{\tau_0,\tau_1\}] \mid \mathsf{B}[0:\tau_0, 31:\tau_1] \mid \{100:\tau_0, 49:\tau_1\} \\
\xrightarrow{\mathsf{B}:\mathsf{swap}(21, \tau_1, \tau_0)} \; & \mathsf{A}[70:\{\tau_0,\tau_1\}] \mid \mathsf{B}[30:\tau_0, 10:\tau_1] \mid \{70:\tau_0, 70:\tau_1\} \\
\xrightarrow{\mathsf{A}:\mathsf{rdm}(30:\{\tau_0,\tau_1\})} \; & \mathsf{A}[30:\tau_0, 30:\tau_1, 40:\{\tau_0,\tau_1\}] \mid \mathsf{B}[30:\tau_0, 10:\tau_1] \mid \{40:\tau_0, 40:\tau_1\} \\
\xrightarrow{\mathsf{B}:\mathsf{swap}(30, \tau_0, \tau_1)} \; & \mathsf{A}[30:\tau_0, 30:\tau_1, 40:\{\tau_0,\tau_1\}] \mid \mathsf{B}[0:\tau_0, 27:\tau_1] \mid \{70:\tau_0, 23:\tau_1\} \\
\xrightarrow{\mathsf{A}:\mathsf{rdm}(30:\{\tau_0,\tau_1\})} \; & \mathsf{A}[82:\tau_0, 47:\tau_1, 10:\{\tau_0,\tau_1\}] \mid \mathsf{B}[0:\tau_0, 27:\tau_1] \mid \{18:\tau_0, 6:\tau_1\}
\end{aligned}
$$

Figure 1: Interactions between two users and an AMM.

Proof in Formal Verification

1. Interactive theorem provers (Isabelle/HOL, Coq, Lean *etc.*)
   - Mathematical proof and specification
   - Proofs (almost always) by induction, computational in nature
2. Model Checkers
   - Brute force proof over a discrete data type
   - Custom specification language

Proof in Formal Verification

```
Proof.
    intros ? ? ? ? ? ? t_x_data t_y_data ? ? ? ? ? ? successful_txn msg_is_trade
        token_in_tx token_out_ty tx_neq_ty r_x'.
    destruct t_x_data as [stor_tx t_x_data].
    destruct t_x_data as [x_geq0 t_x_data].
    destruct t_x_data as [rate_rx rx_geq_0].
    (* first, prove that r_x' < r_x while r_z = r_z' for all other rates *)
    assert (r_x' <= r_x /\
        forall t,
        t <> t_x ->
        get_rate t (stor_rates cstate') = get_rate t (stor_rates cstate))
    as change_lemma.
    {   intros.
        is_sp_destruct.
        rewrite msg_is_trade in successful_txn.
        pose proof (trade_entrypoint_check_pf cstate chain ctx msg_payload
                cstate' acts successful_txn)
        as token_in_qty.
        do 3 destruct token_in_qty as [* token_in_qty].
        destruct token_in_qty as [token_in_qty rates_exist].
        destruct rates_exist as [rate_in_exists rate_out_exists].
        (* get the new rates *)
        rewrite <- token_out_ty in tx_neq_ty.
        rewrite <- msg_is_trade in successful_txn.
        rewrite msg_is_trade in successful_txn.
        pose proof (trade_update_rates_formula_pf cstate chain ctx msg_payload cstate' acts successful_txn) as updating_rates.
        destruct updating_rates as [_ updating_rates].
        destruct updating_rates as [calc_new_rate_x other_rates_equal].
        (* split into cases *)
        split.
        - unfold r_x'. unfold get_rate.
```

Zero-Knowledge Proofs

Produce e.g. **ZK-SNARK** to prove **knowledge of data**.

- Lurk:
  "*Correct execution of Lurk expressions can be proved in zero-knowledge.*"
- Leo-lang pairs formal and zero-knowledge proof

Specification, Formal and Otherwise

- Theorem statement
- Formal statement/specification
- Statement/specification in specification language
- Execution trace/Data (?)

Correct Specifications

A proof is only as good as its statement/specification.

Specification Engineering

Features of specification engineering:

- Variables
- `If ... then ...` blocks
- Functions
- Hooks
- Assertions

Specification Engineering

Borda.spec

Borda > Borda.spec

```
/*
Vote is the only state-changing function.
A vote can only affect the voter and the selected candidates, and has no effect on other addresses.
    ∀address c, c ≠ {f, s, t}.
    { c_points = points(c) ∧ b = voted(c) }  vote(e, f, s, t)  { points(c) = c_points ∧ ( voted(c) = b V c = e.msg.sender ) }
*/
rule noEffect(method m) {
    address c;
    env e;
    uint256 c_points = points(c);
    bool c_voted = voted(c);
    if (m.selector == sig:vote(address, address, address).selector) {
        address f;
        address s;
        address t;
        require( c != f  &&  c != s  &&  c != t );
        vote(e, f, s, t);
    }
    else {
        calldataarg args;
        m(e, args);
    }
    assert ( voted(c) == c_voted || c  == e.msg.sender ) &&
            points(c) == c_points, "unexpected change to others points or voted";
}
```

Specifications in Theorem Provers

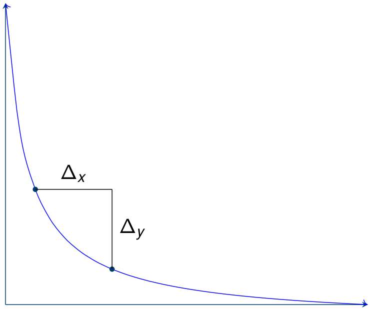
Features of specifications in theorem provers:

- Much less engineering, more like prose
- Quantifiers: `forall` and `exists`
- Arbitrary specification
- Has access to formalized mathematics, but still constrained by computation
- Still extremely challenging

Smart Contract Specification

AMM specification:

- Pricing
- Efficient market
- Liquidity providers
- Rewards distribution
- Governance
- *etc.*



Undone Science

__There is no formal way of establishing that a specification captures a concept__, but we expect to have gained from using the proof methodology because __(hopefully) a specification is easier to understand than a program__, so that "convincing oneself" that a specification captures a concept is less error-prone than a similar process applied to a program. (Liskov and Zilles 1974)

Undone Science

1. We need to tackle the meta-analysis of correct specifications.
2. Formal analysis of specification correctness which does *not* devolve into a tower of meta-(...-)meta-specifications.

Take (pragmatic) inspiration from mathematics

- In ITPs, programs are **well-defined mathematical objects**
- Specifications gain meaning **in context** with other specifications