Communicating and trusting formal proofs

The ProofCert project

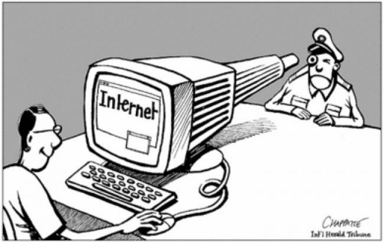
Dale Miller

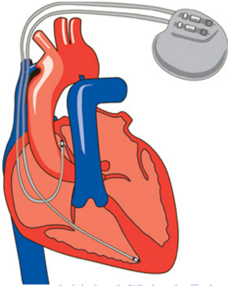
Inria Saclay & LIX, École Polytechnique
Palaiseau, France

23 February 2017

An ERC funded project involving many people in the Parsifal team for the last five years.

What can we trust?





In software correctness: Trust proofs

With software systems, there are so many things to trust.

- compilers
- printers and parsers
- verification condition generators
- type checkers, type inference, abstract interpretation
- theorem provers

All this seems overwhelming. Our challenge here:

provide the framework so that we can at least trust proofs.

We restriction our of attention to *formal proofs*, generated and checked by computer tools.

Security problems often result from programming errors

There are certainly cryptographic errors (eg, keys too short, poor random number generation, side-channel attacks).

Most security problems result from programming errors.

- Buffer overflows
- Incorrect memory management
- Typing errors: find a way convert, say, a string to a wallet.

An important application of proofs is to program correctness.

The current situation with formal proofs

Most proof production and checking is technology based.

If you change the version number of a prover, it may not recognized its earlier proofs.

Some bridges are now being built between different provers, but these are affected by two version numbers.

In them we can trust

de Bruijn, Huet, Paulson, Boyer, Moore

In them we can trust

de Bruijn, Huet, Paulson, Boyer, Moore

Obvious, this model of trust does not scale!

The vision: The network *is* the prover

Goal: Permit the formal methods community to become a network of communicating provers.

Proof certificates: documents that circulate and denote proofs.

Approach: Provide formal definitions of "proof evidence" so that proof certificates can be checked by *trusted checkers*.

But: There is a wide range of "proof evidence."

- proof scripts for steering a theorem prover to a proof
- resolution refutations, natural deduction, tableaux, etc
- winning strategies, simulations

The need for frameworks

Three central questions:

- How can we manage so many "proof languages"?
- Will we need just as many proof checkers?
- How does this improve trust?

Computer scientists have seen this kind of problem before.

The need for frameworks

Three central questions:

- How can we manage so many "proof languages"?
- Will we need just as many proof checkers?
- How does this improve trust?

Computer scientists have seen this kind of problem before.

We develop *frameworks* to address such questions.

- lexical analysis: finite state machines / transducers
- language syntax: grammars, parsers, attribute grammars, parser generators
- programming languages: denotational and operational semantics

A note about logic programming

Prototype implementations of some of these frameworks are often written using logic programming.

- The first Prolog programs (by Colmerauer, 1972) were parsers.
- The Centaur project (G. Kahn, et al, Centaur project, 1988) provided a uniform implementation of Structured Operational Semantics via a Prolog engine.
- Specifying the operational semantics of the λ-calculus and the π-calculus are strong points of λProlog.

A note about logic programming

Prototype implementations of some of these frameworks are often written using logic programming.

- The first Prolog programs (by Colmerauer, 1972) were parsers.
- The Centaur project (G. Kahn, et al, Centaur project, 1988) provided a uniform implementation of Structured Operational Semantics via a Prolog engine.
- Specifying the operational semantics of the λ-calculus and the π-calculus are strong points of λProlog.

Production systems are seldom written using logic programming.

- The specifications get fixed (prototyping not needed).
- Specifications are analyzed so that the flexible execution model provided by unification and backtracking are not needed.

A framework for proof evidence: First pick the logic

Church's 1940 Simple Theory of Types (STT) is a good choice for the syntax of formulas.

Understood well for both classical and intuitionistic logics.

Propositional, first-order, and higher-order logics are easily identifiable sublogics of STT.

Many other logics can adequately be encoded into STT: eg, equational, modal, temporal, etc. Also type systems such as dependently typed λ-terms.

STT is a popular choice in various implemented systems.

There is likely to always be a frontier of research that involves logics that do not fit well into a fixed framework. C'est la vie.

Earliest notion of formal proof

Frege, Hilbert, Church, Gödel, etc, made extensive use of the following notion of proof:

> *A proof is a list of formulas, each one of which is either an* axiom *or the conclusion of an* inference rule *whose premises come earlier in the list.*

While granting us trust, there is little useful structure here.

The first programmable proof checker

LCF/ML (1979) viewed proofs as slight generalizations of such lists.

ML provided types, abstract datatypes, and higher-order programming in order to increase confidence in proof checking.

Many provers today (HOL, Coq, Isabelle) follow LCF principles.

More recent advances: Atoms and molecules of inference

Atoms of inference

- Gentzen's **sequent calculus** (1935) first provided these: introduction, identity, and structural rules.
- Girard's **linear logic** refined our understanding of these further.
- To account for first-order structure, we also need **fixed points** and **equality**. (eg. Baelde, Gacek, McDowell, M, Tiu)

Rules of Chemistry

- **Focused proof systems** show us that some atoms stick together while other atoms form boundaries.

Molecules of inference

- Collections of atomic inference rules that stick together form synthetic inference rules.

Features enabled for proof certificates

Simple checkers can be implemented.

Only the atoms of inference and the rules of chemistry (both small and closed sets) need to be implemented in a checker of certificates.

Certificates support a wide range of proof systems.

The molecules of inference can be engineered into a wide range of inference rules.

Certificates are based (ultimately) on proof theory.

Immediate by design.

Proof details can be elided.

Search using atoms will match search in the space of molecules: that is, the checker will not invent new molecules.

An analogy between two frameworks: SOS and FPC

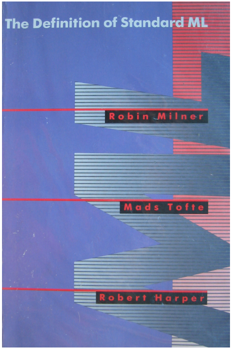
Structural Operational Semantics (SOS)

1. There are many programming languages.
2. SOS can define the semantics of many of them.
3. Logic programming can provide prototype interpreters.
4. Compliant compilers can be built based on the semantics.

An analogy between two frameworks: SOS and FPC

Structural Operational Semantics (SOS)

1. There are many programming languages.
2. SOS can define the semantics of many of them.
3. Logic programming can provide prototype interpreters.
4. Compliant compilers can be built based on the semantics.



An analogy between two frameworks: SOS and FPC

Structural Operational Semantics (SOS)

1. There are many programming languages.
2. SOS can define the semantics of many of them.
3. Logic programming can provide prototype interpreters.
4. Compliant compilers can be built based on the semantics.

Foundational Proof Certificates (FPC)

1. There are many forms of proof evidence.
2. FPC can define the semantics of many of them.
3. Logic programming can provide prototype checkers.
4. Compliant checkers can be built based on the semantics.

Clerks and experts: the office workflow analogy

Imagine an accounting office that needs to check if a certain mound of financial documents (provided by a **client**) represents a legal tax transaction (as judged by the **kernel**).

Experts look into the mound and extract information and

- *decide* which transactions to dig into and
- *release* their findings for storage and later reconsideration.

Clerks take information released by the experts and perform some computations on them, including their *indexing* and *storing*.

Focused proofs alternate between two phases: *positive* (experts are active) and *negative* (clerks are active).

The terms *decide*, *store*, and *release* come from proof theory.

A proof certificate format defines workflow and the duties of the clerks and experts.

Proof checking and proof reconstruction

The clerks can perform (determinate) computation.

Proof *reconstruction* might be needed when invoking not-so-expert experts (or ambiguous tax forms).

Non-deterministic computation is part of the mix: non-determinism is an important resource that is useful for proof-compression.

The *LKneg* proof system

Use invertible rules where possible.

$$\frac{\vdash \cdot ; B}{\vdash B}\ \mathit{start} \qquad \frac{\vdash \Delta, L; \Gamma}{\vdash \Delta; L, \Gamma}\ \mathit{store} \qquad \frac{}{\vdash \Delta, A, \neg A; \cdot}\ \mathit{init}$$

$$\frac{\vdash \Delta; \Gamma}{\vdash \Delta; \mathit{false}, \Gamma} \quad \frac{\vdash \Delta; B, C, \Gamma}{\vdash \Delta; B \vee C, \Gamma} \quad \frac{}{\vdash \Delta; \mathit{true}, \Gamma} \quad \frac{\vdash \Delta; B, \Gamma \quad \vdash \Delta; C, \Gamma}{\vdash \Delta; B \wedge C, \Gamma}$$

Here, A is an atom, L a literal, Δ a multiset of literals, and Γ a list of formulas. Sequents have two *zones*.

This proof system provides a decision procedure (resembling conjunctive normal forms).

A small (constant sized) certificate is possible.

The *LKneg* proof system

Use invertible rules where possible.

$$\frac{\vdash \cdot; B}{\vdash B}\ \textit{start} \qquad \frac{\vdash \Delta, L; \Gamma}{\vdash \Delta; L, \Gamma}\ \textit{store} \qquad \frac{}{\vdash \Delta, A, \neg A; \cdot}\ \textit{init}$$

$$\frac{\vdash \Delta; \Gamma}{\vdash \Delta; \textit{false}, \Gamma} \qquad \frac{\vdash \Delta; B, C, \Gamma}{\vdash \Delta; B \vee C, \Gamma} \qquad \frac{}{\vdash \Delta; \textit{true}, \Gamma} \qquad \frac{\vdash \Delta; B, \Gamma \quad \vdash \Delta; C, \Gamma}{\vdash \Delta; B \wedge C, \Gamma}$$

Here, A is an atom, L a literal, Δ a multiset of literals, and Γ a list of formulas. Sequents have two *zones*.

This proof system provides a decision procedure (resembling conjunctive normal forms).

A small (constant sized) certificate is possible.

Consider proving $(p \vee C) \vee \neg p$ for large C.

The *LKpos* proof system

Non-invertible rules are used here.

$$\frac{\vdash B; \cdot; B}{\vdash B}\ start \qquad \frac{\vdash B; \mathcal{N}, \neg A; B}{\vdash B; \mathcal{N}; \neg A}\ restart \qquad \frac{}{\vdash B; \mathcal{N}, \neg A; A}\ init$$

$$\frac{\vdash B; \mathcal{N}; B_i}{\vdash B; \mathcal{N}; B_1 \vee B_2} \qquad \frac{}{\vdash B; \mathcal{N}; true} \qquad \frac{\vdash B; \mathcal{N}; B_1 \quad \vdash B; \mathcal{N}; B_2}{\vdash B; \mathcal{N}; B_1 \wedge B_2}$$

Here, A is an atom and $\mathcal{N}$ is a multiset of negated atoms. Sequents have three *zones*.

The $\vee$ rule *consumes* some external information or some non-determinism.

An *oracle string*, a series of bits used to indicate whether to go left or right, can be a proof certificate.

A proof in *LKpos*

Let C have several alternations of conjunction and disjunction.

Let $B = (p \vee C) \vee \neg p$.

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\overline{\vdash B; \neg p; p}\ \mathit{init}}{\vdash B; \neg p; p \vee C}\ *}{\vdash B; \neg p; (p \vee C) \vee \neg p}\ *}{\vdash B; \cdot\ ; \neg p}\ \mathit{restart}}{\vdash B; \cdot\ ; (p \vee C) \vee \neg p}\ *}{\vdash B}\ \mathit{start}
$$

The subformula C is avoided. Clever choices $*$ are injected at these points: right, left, left. We have a small certificate and small checking time. In general, these certificates may grow large.

Combining the *LKneg* and *LKpos* proof systems

Introduce two versions of conjunction, disjunction, and their units.

$$t^-, t^+, f^-, f^+, \vee^-, \vee^+, \wedge^-, \wedge^+$$

The inference rules for negative connectives are invertible.

These polarized connectives also exist in linear logic.

Introduce the two kinds of sequent, namely,

$\vdash \Theta \Uparrow \Gamma$: for invertible (negative) rules (Γ a list of formulas)

$\vdash \Theta \Downarrow B$: for non-invertible (positive) rules (B a formula)

LKF : a focused proof systems for classical logic

$$\frac{}{\vdash \Theta \Uparrow \Gamma, t^-} \qquad \frac{\vdash \Theta \Uparrow \Gamma, B \qquad \vdash \Theta \Uparrow \Gamma, B'}{\vdash \Theta \Uparrow \Gamma, B \wedge^- B'} \qquad \frac{\vdash \Theta \Uparrow \Gamma}{\vdash \Theta \Uparrow \Gamma, f^-} \qquad \frac{\vdash \Theta \Uparrow \Gamma, B, B'}{\vdash \Theta \Uparrow \Gamma, B \vee^- B'}$$

$$\frac{}{\vdash \Theta \Downarrow t^+} \qquad \frac{\vdash \Theta \Downarrow B_1 \qquad \vdash \Theta \Downarrow B_2}{\vdash \Theta \Downarrow B_1 \wedge^+ B2} \qquad \frac{\vdash \Theta \Downarrow B_i}{\vdash \Theta \Downarrow B_1 \vee^+ B_2}$$

Init	Store	Release	Decide
$\frac{}{\vdash \neg A, \Theta \Downarrow A}$	$\frac{\vdash \Theta, C \Uparrow \Gamma}{\vdash \Theta \Uparrow \Gamma, C}$	$\frac{\vdash \Theta \Uparrow N}{\vdash \Theta \Downarrow N}$	$\frac{\vdash P, \Theta \Downarrow P}{\vdash P, \Theta \Uparrow \cdot}$

P is a positive formula; N is a negative formula;
A is an atom; C positive formula or negative literal

Results about LKF

Let B be a propositional logic formula and let $\hat{B}$ result from B by placing $+$ or $-$ on t, f, $\wedge$, and $\vee$ (there are exponentially many such placements).

Theorem. [Liang & M, TCS 2009]

- If B is a tautology then every polarization $\hat{B}$ has an LKF proof.
- If some polarization $\hat{B}$ has an LKF proof, then B is a tautology.

The different polarizations do not change *provability* but can radically change the *proofs*.

Also:

- Negative (non-atomic) formulas are treated linearly (never weakened nor contracted).
- Only positive formulas are contracted (in the Decide rule).

Example: deciding on a simple clause

Assume that Θ contains the formula $a \wedge^+ b \wedge^+ \neg c$ and that we have a derivation that Decides on this formula.

$$\dfrac{\dfrac{}{\vdash \Theta \Downarrow a}\,Init \quad \dfrac{}{\vdash \Theta \Downarrow b}\,Init \quad \dfrac{\dfrac{\vdash \Theta, \neg c \Uparrow \cdot}{\vdash \Theta \Uparrow \neg c}\,Store}{\vdash \Theta \Downarrow \neg c}\,Release}{\dfrac{\vdash \Theta \Downarrow a \wedge^+ b \wedge^+ \neg c}{\vdash \Theta \Uparrow \cdot}\,Decide}\,\wedge^+$$

This derivation is possible iff Θ is of the form $\neg a, \neg b, \Theta'$. Thus, the "macro-rule" is

$$\frac{\vdash \neg a, \neg b, \neg c, \Theta' \Uparrow \cdot}{\vdash \neg a, \neg b, \Theta' \Uparrow \cdot}$$

Example: Resolution as a proof certificate

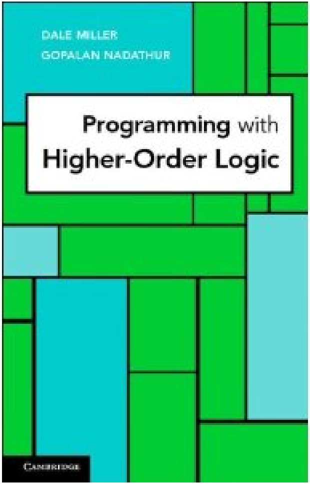
- A *clause:* $\forall x_1 \ldots \forall x_n [L_1 \vee \cdots \vee L_m]$
- C_3 is a *resolution* of C_1 and C_2 if we chose the mgu of two complementary literals, one from each of C_1 and C_2, etc.
- If C_3 is a resolvent of C_1 and C_2 then $\vdash \neg C_1, \neg C_2 \Uparrow C_3$ has a short proof (decide depth 2 or less).

Translate a refutation of $C_1, \ldots, C_n$ into a (focused) sequent proof with small holes:

$$\dfrac{\dfrac{\Xi}{\vdash \neg C_1, \neg C_2 \Uparrow C_{n+1}} \qquad \dfrac{\dfrac{\vdots}{\vdash \neg C_1, \ldots, \neg C_n, \neg C_{n+1} \Uparrow \cdot}}{\vdash \neg C_1, \ldots, \neg C_n \Uparrow \neg C_{n+1}}\ \mathit{Store}}{\vdash \neg C_1, \ldots, \neg C_n \Uparrow \cdot}\ \mathit{Cut}$$

Here, Ξ can be replaced with a "hole" bounded by depth 2.

Reference proof checking in λProlog



Logic programming can check proofs in sequent calculus.

Proof reconstruction requires unification and (bounded) proof search.

The λProlog programming language [M & Nadathur, 1986, 2012] also include types, abstract datatypes, and higher-order programming.

From inference rules to λProlog clauses

We first "instrument" the inference rules with terms denoting proof certificates and add premises that invoke "clerks" and "experts".

$$\frac{\Xi_1 \vdash \Theta \Uparrow \Gamma, A \qquad \Xi_2 \vdash \Theta \Uparrow \Gamma, B \qquad \wedge\text{clerk}(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0 \vdash \Theta \Uparrow \Gamma, A \wedge^- B}$$

$$\frac{\Xi_1 \vdash \Theta \Downarrow B_i \qquad \vee\text{expert}(\Xi_0, \Xi_1, i)}{\Xi_0 \vdash \Theta \Downarrow B_1 \vee^+ B_2}$$

Turning inference rules sideways yields logic programs.
Soundness of checking is reduced to soundness of the logic programming implementation.
The formal definition of "proof evidence" involves

- describing the structure of the certificate terms Ξ and
- providing the definition of the clerk and expert predicates.

An FPC: Checking by conjunctive normal form

```
type lit                      index.
type cnf                       cert.

andNeg_kc         cnf  cnf  cnf.
orNeg_kc               cnf  cnf.
false_kc               cnf  cnf.
release_ke             cnf  cnf.
initial_ke                  cnf  lit.
decide_ke              cnf  cnf  lit.
store_kc               cnf  cnf  lit.
```

The token `cnf` is just passed around during the checking. The only items that are stored are literals and they are all indexed the same, using `lit`.

An FPC: Checking binary resolution

```
type idx                    int -> index.
type lit                           index.
kind resol                          type.
type resol    int -> int -> int -> resol.
type dl               list int    -> cert.
type ddone                          cert.
type rdone                          cert.
type rlist            list resol -> cert.
type rlisti   int ->  list resol -> cert.

orNeg_kc     (dl L) _  (dl L).
false_kc     (dl L) (dl L).
store_kc     (dl L) C  lit (dl L).
decide_ke    (dl [I]) (idx I) (dl []).
decide_ke    (dl [I,J]) (idx I) (dl [J]).
decide_ke    (dl [J,I]) (idx I) (dl [J])
all_kc       (dl L) (x\ dl L).
true_ke      (dl L).
some_ke      (dl L) _  (dl L).
andPos_ke    (dl L) _  (dl L) (dl L).
release_ke   (dl L) (dl L).
initial_ke   (dl L) _.
decide_ke    (dl L) _ ddone.
initial_ke   ddone _.

false_kc  (rlist R) (rlist R).
store_kc  (rlisti K R) _ (idx K) (rlist R).
true_ke   rdone.
decide_ke (rlist [])  (idx I) rdone.
cut_ke    (rlist [(resol I J K) |R]) CutForm (dl [I,J]) (rlisti K R).
```

The ProofCert project: recent results

The FPC framework for first-order (classical and intuitionistic) logics.

Defined various proof certificate formats:

- Classical: resolution, expansion trees, matings, CNF, etc.
- Intuitionistic: natural deduction, various typed λ-calculi.
- Also: Frege systems, equality reasoning, etc.
- Also: proof systems for modal logics.

Implemented a reference kernel in λProlog (Teyjus / ELPI)

The intuitionistic checker can "host" the classical kernel.

Automated *elaboration* of proof certificates (and proof outlines) into maximally explicit certificates. These are checkable by a very simple (OCaml) checker.

The ProofCert project: next steps

Move from logic to arithmetic: include induction and coinduction.

Provide certificates for model checking.

Develop certificates for various modal and temporal logics.

Treat parallelism in proof structures.

Standards and adoption. Theorem proving competitions.

Performant checkers

The ProofCert project: still further ahead

Design of libraries and marketplaces of theorems and proofs

Develop an approach to theories: set theories, type theories, etc.

Integrating counter-examples / counter-models

Once proofs are checked, how can we read / browse them?

Interdisciplinary effort: Can we lift the underlying issues of *reputation*, *reproducibility*, and *trust* from the (easy) domain of proof to support social media and journalism?

Thank you. Questions?

What about LF, LFSC, Dedukti?

LF: The logical framework of Harper, Honsell, and Plotkin [1987, 1993] (a.k.a. $\lambda\Pi$ and dependently typed λ-calculus).

It seems straightforward to provide FPC definitions of LF, LFSC (LF with side conditions), and LF modulo (Dedukti).

Alone LF does not seem to have the right "atoms of inference."

- Canonical normal forms provide only one structuring of proofs.
- These lack an analytic notion of classical reasoning and sharing.
- Also lacking is a natural treatment of parallel proof steps.

Dedukti has serious implementations and can check output from several serious provers. No FPC checker can claim that (yet).