Proof Assistant Based Verification of Programs

Haitao Hu

May 19, 2023

Introduction to proof assistant

1. Interactive proof editor
2. Automated/Refined proofs
3. Libraries
4. Agda and Coq

Introduction to Type theory

1. We use $A, B, \cdots$ to denote a *type*.
2. A judgement is $a : A$, where a is element of type A. Informal, if A is a proposition, then we may say A has a proof a.
3. Given two proposition A, B, from BHK-interpretation (Brouwer-Heyting-Kolmogorov) :
   - $c : A \wedge B$ follows directly from $a : A$ and $b : B$.
   - $c : A \vee B$ follows directly from $a : A$ or $b : B$.
   - $c : A \to B$ follows directly from $b : B$ under the assumption that $a : A$.

   But how does c look like ? we do not know...

Curry-Howard isomorphism

1. CHI is a one-by-one correspondence between *simply typed lambda calculus* and *propositional logic*.
2. It gives c a precise definition:
   - $(a, b) : A \times B$ follows directly from $a : A$ and $b : B$.
   - $\texttt{inl}\ a : A + B$ follows directly from $a : A$.
   - $\texttt{inr}\ b : A + B$ follows directly from $b : B$.
   - $\lambda x.\, b[x/a] : A \to B$ follows directly from $b : B$ under the assumption that $a : A$.
   - ...
3. A program term in STLC is indeed a proof term in PL.

Proposition as types

1. To prove a proposition A, one must find a proof, which is equivalent to finding an element a such that $a : A$.
2. Some terminologies when we use type theory to do proof:
   - how to form new types of this kind, via **formation rules**.
   - how to construct elements of that type, via **constructors** or **introduction rules**.
   - how to use elements of that type, via **eliminators** or **elimination rules**.
   - a **computation rule**, which expresses how an eliminator acts on a constructor.
   - an optional **uniqueness principle**, which expresses uniqueness of maps into or out of that type.

A example

1. Prove $(A \wedge B \to C) \to (A \to B \to C)$, this is we have to find a $c : (A \wedge B \to C) \to (A \to B \to C)$.
2. Suppose we have a constructor $\wedge$-intro : $A \to B \to A \wedge B$.
3. Then we have

$$\lambda f.\, (\lambda x, y.\, f\ (\wedge\text{-intro}\ x\ y)) : (A \wedge B \to C) \to (A \to B \to C)$$

Dependent Type theory

1. CHI is just a correspondence for non-dependent type. i.e, $A \to B$.
2. Given a type family $P : A \to \mathcal{U}$, then we may have a dependent function type $\Pi_{a:A} P(a)$.
3. Dependent type give us a interpretation for $\forall$ and $\exists$.
   - dependent function type is $\forall$. So predicate is actually something that generates new proposition for given a object.
   - dependent product type $\Sigma_{a:A} P(a)$ is $\exists$.
4. Propositional logic $+ \forall + \exists =$ first order logic.
5. Agda is a dependent type language.

Formalization of security flow analysis

1. Source code : `https://github.com/m4p1e/FoS`
2. The two principles of security flow analysis:
   - a variable is only data-dependent on the variables such that have lower security level i.e. $\mathtt{x}_H = \mathtt{y}_L$.
   - a variable is only control-dependent on the variables such that have lower security level i.e. $\mathtt{if}\ (\mathtt{z}_L)\ \{\mathtt{x} = \mathtt{y}\}$.
3. We give a definition of simple language :

$$
\begin{aligned}
\langle P \rangle \quad &::= \langle \mathit{stmt} \rangle \\
\langle \mathit{stmt} \rangle \quad &::= \text{skip} \\
&\quad\mid\ \langle \mathit{var} \rangle := \langle \mathit{rexpr} \rangle \\
&\quad\mid\ \text{if } \langle \mathit{bexpr} \rangle \text{ then } \langle \mathit{stmt} \rangle \text{ else } \langle \mathit{stmt} \rangle \\
&\quad\mid\ \text{while } \langle \mathit{bexp} \rangle \text{ loop } \langle \mathit{stmt} \rangle \\
&\quad\mid\ \langle \mathit{stmt} \rangle \ ;\ \langle \mathit{stmt} \rangle
\end{aligned}
$$

Formalization of security flow analysis

1. Given variables $x_1, x_2, \cdots, x_m$ and function $\text{Sec}_v : \text{var} \to \mathbb{N}$. For each x_i, the security level of x_i is $\text{Sec}_v(x_i)$.
2. Then we can define function $\text{Sec}_{ne} : \text{nexpr} \to \mathbb{N}$ by pattern matching
   - if e is constant, then $\text{Sec}_{ne}(e) = 0$;
   - if e is a variable, then $\text{Sec}_{ne}(e) = \text{Sec}_v(e)$.
   - if e is $(e_1 + e_2)$, then $\text{Sec}_{ne}(e) = \text{Sec}_{ne}(e_1) \sqcup \text{Sec}_{ne}(e_2)$.
   - if e is $(e_1 - e_2)$, then $\text{Sec}_{ne}(e) = \text{Sec}_{ne}(e_1) \sqcup \text{Sec}_{ne}(e_2)$.
   - if e is $(e_1 * e_2)$, then $\text{Sec}_{ne}(e) = \text{Sec}_{ne}(e_1) \sqcup \text{Sec}_{ne}(e_2)$.

   where $\sqcup$ is lowest upper bound operator.

Formalization of security flow analysis

1. The function Sec_{ne} look like something below in Agda.

$$\mathrm{sec}_{ne} : \mathrm{sec}_v \to \mathrm{nexp} \to \mathbb{N}$$
$$\mathrm{sec}_{ne}\ sec'_v\ (\text{n-const}\ x) = 0 \text{ -- } ?$$
$$\mathrm{sec}_{ne}\ sec'_v\ (\text{n-var}\ x) = sec'_v\ x$$
$$\mathrm{sec}_{ne}\ sec'_v\ (\text{n-add}\ e_1\ e_2) = \mathrm{sec}_{ne}\ sec'_v\ e_1 \sqcup \mathrm{sec}_{ne}\ sec'_v\ e_2$$
$$\mathrm{sec}_{ne}\ sec'_v\ (\text{n-sub}\ e_1\ e_2) = \mathrm{sec}_{ne}\ sec'_v\ e_1 \sqcup \mathrm{sec}_{ne}\ sec'_v\ e_2$$
$$\mathrm{sec}_{ne}\ sec'_v\ (\text{n-mul}\ e_1\ e_2) = \mathrm{sec}_{ne}\ sec'_v\ e_1 \sqcup \mathrm{sec}_{ne}\ sec'_v\ e_2$$

2. Similarly, we can define Sec_{be} : bexpr $\to \mathbb{N}$ and Sec_{re} : rexpr $\to \mathbb{N}$.

Formalization of security flow analysis

1. Finally, we may define function $\text{Sec}_{st} : \text{stmt} \to \mathbb{N}$.
   - if st is $x := e$, then $\text{Sec}_{st}(st) = \text{Sec}_v(x) \sqcup \text{Sec}_{re}(e)$.
   - if st is $\texttt{if}\ e\ \texttt{then}\ st_1\ \texttt{else}\ st_1$, then $\text{Sec}_{st}(st) = \text{Sec}_{re}(e) \sqcap \text{Sec}_{st}(st_1) \sqcap \text{Sec}_{st}(st_2)$.
   - if st is $\texttt{while}\ e\ \texttt{loop}\ st_1$, then $\text{Sec}_{st}(st) = \text{Sec}_{re}(e) \sqcap \text{Sec}_{st}(st_1)$
   - if st is $st_1; st_2$, then $\text{Sec}_{st}(st) = \text{Sec}_{st}(st_1) \sqcap \text{Sec}_{st}(st_2)$.

   where $\sqcap$ is greatest lower bound operator. But something bad happens, what is the security level of `skip` ?
   - If $\text{Sec}_{st}(\texttt{skip})$ is 0, then every conditional statement $\texttt{if}\ e\ \texttt{then}\ st_1\ \texttt{else}\ st_1$ such that st_1 or st_2 includes `skip` will be unsafe.
   - Thus, $\text{Sec}_{st}(\texttt{skip}) := \top$ such that every number $n < \top$.
2. We need to introduce a new type of number $\hat{\mathbb{N}} = \mathbb{N} \cup \{\top\}$.

Formalization of security flow analysis

1. The function Sec_{st} look like something below in Agda.

$$
\begin{aligned}
&\mathsf{sec}_{st} : \mathsf{sec}_{v} \rightarrow \mathsf{stmt} \rightarrow \hat{\mathbb{N}} \\
&\mathsf{sec}_{st}\ sec'_{v}\ \mathsf{skip} = \top \\
&\mathsf{sec}_{st}\ sec'_{v}\ (x := e) = n{\leq}\top\ (\mathsf{sec}\ sec'_{v}\ x\ \ \mathsf{sec}_{re}\ sec'_{v}\ e) \\
&\mathsf{sec}_{st}\ sec'_{v}\ (\mathsf{if}\ e\ \mathsf{then}\ st_1\ \mathsf{else}\ st_2) = \\
&\quad n{\leq}\top\ (\mathsf{sec}_{be}\ sec'_{v}\ e) \sqcap^{g} \mathsf{sec}_{st}\ sec'_{v}\ st_1 \sqcap^{g} \mathsf{sec}_{st}\ sec'_{v}\ st_2 \\
&\mathsf{sec}_{st}\ sec'_{v}\ (\mathsf{while}\ e\ \mathsf{loop}\ st) = \\
&\quad n{\leq}\top\ (\mathsf{sec}_{be}\ sec'_{v}\ e) \sqcap^{g} \mathsf{sec}_{st}\ sec'_{v}\ st \\
&\mathsf{sec}_{st}\ sec'_{v}\ (st_1\ ;\ st_2) = \mathsf{sec}_{st}\ sec'_{v}\ st_1 \sqcap^{g} \mathsf{sec}_{st}\ sec'_{v}\ st_2
\end{aligned}
$$

where $\top : \hat{\mathbb{N}}$ and $n{\leq}\top : \mathbb{N} \rightarrow \hat{\mathbb{N}}$ are introduction rules of $\hat{\mathbb{N}}$.

Formalization of security flow analysis

1. Now we can build a checker upon early defined security level functions by defining a function $\texttt{accpet} : \mathsf{stmt} \to \mathsf{Bool}$.
    - If $\texttt{accpet}(st) = \text{true}$, then st is safe.
    - If $\texttt{accpet}(st) = \text{false}$, then st is unsafe.
2. The two principles of security flow analysis:
    - a variable is only data-dependent on the variables such that have lower security level i.e. $\mathtt{x}_H = \mathtt{y}_L$.
    - a variable is only control-dependent on the variables such that have lower security level i.e. $\texttt{if}\ (\mathtt{z}_L)\ \{\mathtt{x} = \mathtt{y}\}$.

Safety Checker

1. We define `accept` by pattern matching. For briefly, we use Sec for all objects.
   - if st is `skip`, then $\texttt{accept}(st) = \text{true}$.
   - if st is $x := e$, then $\texttt{accept}(st) = \text{Sec}(e) \leq^b \text{Sec}(x)$.
   - if st is `if` e `then` st_1 `else` st_1,
     - if $\text{Sec}(e) \leq \text{Sec}(st_1) \sqcap^g \text{Sec}(st_2)$, then $\texttt{accept}(st) = \text{true}$.
     - otherwise, $\texttt{accept}(st) = \text{false}$.
   - if st is `while` e `loop` st_1,
     - if $\text{Sec}(e) \leq \text{Sec}(st_1)$, then $\texttt{accept}(st) = \text{true}$.
     - otherwise, $\texttt{accept}(st) = \text{false}$.
   - if st is $st_1; st_2$, then $\texttt{accept}(st) = \texttt{accept}(st_1) \wedge \texttt{accept}(st_2)$.

   where $\leq^b$ is boolean less than operator.
2. The corresponding code at `https://github.com/m4p1e/FoS/blob/main/Proof.agda#L224`

Proof of security flow analysis

1. Our goal : is that checker always output right answer ?
2. The equivalent proposition is

   For any program P, $\texttt{accept}(P) = \text{true}$ if and only if P is safe.

   But what does safe mean ?
3. Operational semantics tell us how does a program run.
4. Observational semantics are the computational effects in operational semantics that are visible to a third observer.
   - memory layout
   - input and output trace
   - syscalls
   - environment
   - ...

Proof of security flow analysis

1. We say two memory M_1, M_2 : var $\to$ value are distinguishable in security level ℓ, if

   for any variable x such that $\mathrm{Sec}(x) \leq l$, we have $M_1(x) = M_2(x)$.

   Simply, written $M_1 =_\ell M_2$.
2. If a program P satisfies following conditions
   - given two initial memory M_1 and M_2 such that $M_1 =_\ell M_2$;
   - there are two final memory M_1' and M_2' such that $\{M_1\}\ P\ \{M_1'\}$ and $\{M_2\}\ P\ \{M_1'\}$;
   - then $M_1' =_\ell M_2'$,

   then P is safe. This property is also called no-interference.

Proof of security flow analysis

1. The main theorem looks like something below in Agda.

$$
\begin{aligned}
\text{Theorem} : \; & \{\ell : \mathbb{N}\} \\
& \to (s_1 : \text{state}) \to (s_1{}' : \text{state}) \\
& \to (s_2 : \text{state}) \to (s_2{}' : \text{state}) \\
& \to s_1 \; [\equiv \ell \,] \; s_2 \\
& \to (st : \text{stmt}) \\
& \to \text{accept } st \; sec'_v \equiv \text{true} \\
& \to \{\; s_1 \;\} \; st \; \{\; s_1{}' \;\} \\
& \to \{\; s_2 \;\} \; st \; \{\; s_2{}' \;\} \\
& \to s_1{}' \; [\equiv \ell \,] \; s_2{}'
\end{aligned}
$$

Roadmap of Proof

1. Give big-step evaluation rules for expressions. (Proof.agda#L248-L318)
2. Give operational semantics for our language. (Proof.agda#L321-L362)
3. Prove some lemmas, then organize them to main theorem.
   - Lemma-1: If $\mathrm{Sec}(st) > \ell$ and $\mathtt{accept}(st) = \mathrm{true}$, then $s =_\ell s'$ for any $\{s\}\ st\ \{s'\}$. (Proof.agda#L676)
   - Lemma-2: If $s_1 =_\ell s_2$ and $[\![\, e \,]\!]\ s_1 \neq [\![\, e \,]\!]\ s_2$, then $\mathrm{Sec}(e) > l$. (Proof.agda#L525)
   - ...

Summary

1. Good structures bring good readability and less-pain proving.
2. Don't overly care about these details.
3. If you want to know why does your proof work, Martain Löf type theory (MLTT) or Homotopy type theory (HoTT) is all you need.

Present and Future

1. PL: languages for expressing mathematics
2. SE: managing large codebases
3. Compilers + distributed computing: speed
4. Machine learning: automated proof search
5. HCI: usable by working mathematicians/computer scientists
6. Graphics: visualization

Thank you for listening!