Towards machine-checked proofs

MPRI 2.4

François Pottier

2024

MPRI 2.4 Towards machine-checked proofs

François Pottier

Mechanization

Coq in a nutshell

Syntax with binders

de Bruijn

Nominal

Why formalize programming languages?

To obtain precise definitions of programming languages.

To obtain rigorous proofs of soundness for tools such as

- interpreters,
- compilers,
- type systems ("well-typed programs do not go wrong"),
- type-checkers and type inference engines,
- static analyzers (e.g. abstract interpreters),
- program logics (e.g. Hoare logic, separation logic),
- deductive program provers (e.g. verification condition generators).

Challenge 1: Scale

Hand-written proofs have difficulty scaling up:

- From minimal calculi (λ, π) and toy languages (IMP, MiniML) to large real-world languages such as Java, C, JavaScript, ...
- From textbook compilers to multi-pass optimizing compilers producing code for real processors.
- From textbook abstract interpreters to scalable and precise static analyzers such as Astrée.

Challenge 2: Trust

Hand-written proofs are seldom trustworthy.

- Authors struggle with huge LaTeX documents.
- Reviewers give up on checking huge but boring proofs.
- Proof cases are omitted because they are “obvious”.
- It is difficult to maintain hand-written proofs as definitions evolve.

Opportunity: machine-assisted proof

Mechanized theorem proving has made great progress.

Landmark examples in mathematics:

- the 4-color th.: Haken & Appel (1976), Gonthier & Werner (2005);
- the Feit-Thompson theorem: Gonthier *et al.* (2013);
- Kepler's conjecture: Hales *et al.* (2015).

Programming language theory is a good match for proof assistants:

- discrete objects (trees); no reals, no analysis, no topology...
- large definitions; proofs with many similar cases;
- syntactic techniques (induction); few deep mathematical concepts.

The POPLmark challenge

In 2005, Aydemir *et al.* challenged the POPL community:

> *How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?*

Today, over 20% of the papers at POPL come with such an appendix.

Proof assistants

An interactive proof assistant offers:

- A formal specification language,
  in which definitions are written and theorems are stated.
- A set of commands for building proofs,
  either automatically or interactively.
- Often, an independent, automated proof checker,
  so the above commands do not have to be trusted.

Barendregt and Wiedijk,
The Challenge of Computer Mathematics, 2005.

Popular proof assistants include Coq, Agda, HOL4, Isabelle/HOL...

Computations and functions

Coq offers a pure functional programming language in the style of ML, with recursive functions and pattern-matching.

```
Fixpoint factorial (n: nat) :=
  match n with
  | O   => 1
  | S p => n * factorial p
  end.

Fixpoint concat (A: Type) (xs ys: list A) :=
  match xs with
  | nil => ys
  | x :: xs => x :: concat xs ys
  end.
```

The language is total: all functions terminate. This is enforced by requiring every recursive call to be decreasing w.r.t. the subterm ordering.

Mathematical logic

Propositions can be expressed in this language. They have type **Prop**.

```
Definition divides (a b: N) := exists n: N, b = n * a.

Theorem factorial_divisors:
  forall n i, 1 <= i <= n -> divides i (factorial n).

Definition prime (p: N) :=
  p > 1 /\ (forall d, divides d p -> d = 1 \/ d = p).

Theorem Euclid:
  forall n, exists p, p >= n /\ prime p.
```

The standard logical connectives and quantifiers are available.

Inductive types

An inductive type is a data type.

It is equipped with a finite number of constructors.

Its inhabitants are generated by repeated application of the constructors.

```
Inductive nat: Type :=
| O: nat
| S: nat -> nat.

Inductive list: Type -> Type :=
| nil:  forall A, list A
| cons: forall A, A -> list A -> list A.
```

E.g., the inhabitants of `nat` are `O`, `S O`, `S (S O)`, etc.

This is well suited to describing the syntax of a programming language.

Inductive predicates

An inductive predicate is equipped with a finite number of constructors, and is generated by repeated application of the constructors.

```
Inductive even: nat -> Prop :=
| even_zero:
    even 0
| even_plus_2:
    forall n, even n -> even (S (S n)).
```

On paper, this is typically written in the form of inference rules:

$$\frac{}{\texttt{0}\text{ is even}} \qquad \frac{n\text{ is even}}{\texttt{S}\ (\texttt{S}\ n)\text{ is even}}$$

The inhabitants of the type `even n` can be thought of as derivation trees whose conclusion is `even n`.

Binding and α-equivalence

Most programming languages provide constructs that bind variables:

- function abstractions (in terms): $\lambda x.t$
- local definitions (in terms): $\mathit{let}\ x = t\ \mathit{in}\ t$
- quantifiers (in types): $\forall\alpha.\alpha \rightarrow \alpha$

α-equivalence is a relation that allows renamings of bound variables, e.g.:

$$\lambda x.\, x + 1 \equiv_\alpha \lambda y.\, y + 1 \qquad\qquad \forall\alpha.\, \alpha\ \mathit{list} \equiv_\alpha \forall\beta.\, \beta\ \mathit{list}$$

On paper, usually,

- one does not clearly define what is a term;
- one confuses α-equivalence $\equiv_\alpha$ with equality $=$.

Representations of syntax

How should syntax with binding be mathematically defined, on paper on in a proof assistant?

Several representations come to mind:

- de Bruijn notation – used in this course (de Bruijn, 1972);
- equivalence classes – the nominal approach (Pitts, 2006);
- (parametric) higher-order abstract syntax (Chlipala, 2008);
- the locally nameless representation (Charguéraud, 2009);
- the solutions of the POPLmark challenge (2005) involve 8 different representations, and there are more.

One should choose a representation for which the proof assistant has good support.

de Bruijn indices

A simple idea: don't use names.

Instead, use pointers from variables back to their binding site.

A second idea: use relative pointers, encoded as natural integers.

- 0 denotes the nearest enclosing λ,
  i.e., the most recently bound variable;
- 1 denotes the next enclosing λ, and so on.

$\lambda x.x$ is $\lambda 0$.

$\lambda f.\lambda x.f\ x$ is $\lambda\lambda(1\ 0)$.

Why is this a good idea?

de Bruijn syntax has several strengths:

- it is easily defined;
- it is inductive – terms are trees, no quotient is required;
- it is canonical – α-equivalence is just equality.

Its drawbacks are well-known, too:

- terms are more difficult to read – a printer may be needed;
- definitions and theorems can seem difficult to write and read – mostly a matter of habit?

λ-terms in de Bruijn's notation

The syntax of λ-calculus is simple:

$$t ::= x \mid \lambda t \mid t\ t \qquad \text{where } x \in \mathbb{N}$$

In Coq:

```
Inductive term :=
| Var: nat -> term
| Lam: term -> term
| App: term -> term -> term.
```

Suggested exercises

Exercise: In OCaml, implement conversions between the nominal representation and de Bruijn's representation, both ways.

Exercise: In OCaml, implement an exhaustive enumeration of the λ-terms of size *s* and with at most *n* free variables. (Let variables have size 0; let λ-abstractions and applications contribute 1.)

Exercise: Use this exhaustive enumeration to test that the above conversions are inverses of each other.

MPRI 2.4 Towards machine-checked proofs

François Pottier

Mechanization
Coq in a nutshell
Syntax with binders
de Bruijn
Nominal

Substitution

— *Substitution is the éminence grise of the λ-calculus.*
Abadi, Cardelli, Curien, Lévy, Explicit substitutions, 1990.

Substitutions

Let a substitution σ be a total function of variables $\mathbb{N}$ to terms $\mathbb{T}$.

It can also be thought of as an infinite sequence $\sigma(0) \cdot \sigma(1) \cdot \ldots$

Let *id* be the identity substitution: $id(x) = x$.

- $0 \cdot 1 \cdot 2 \cdot \ldots$

Let $+i$ be the lift substitution: $(+i)(x) = x + i$.

- $i \cdot (i+1) \cdot (i+2) \cdot \ldots$

Let $t \cdot \sigma$ be the cons substitution that maps 0 to t and $x + 1$ to $\sigma(x)$.

- $t \cdot \sigma(0) \cdot \sigma(1) \cdot \ldots$

id can in fact be viewed as sugar for $0 \cdot (+1)$.

Substitution application and composition

Can we define $t[\sigma]$, the application of the substitution σ to the term t?

It should satisfy the following laws:

$$\begin{aligned} x[\sigma] &= \sigma(x) \\ (\lambda t)[\sigma] &= \,? \\ (t_1\ t_2)[\sigma] &= t_1[\sigma]\ t_2[\sigma] \end{aligned}$$

where $\Uparrow\sigma$ stands for $0 \cdot (\sigma\,;+1)$

and the (left-to-right) composition of two substitutions $\sigma_1\,;\sigma_2$ should satisfy:

$$(\sigma_1\,;\sigma_2)(x) = (\sigma_1(x))[\sigma_2]$$

These equations are mutually recursive, so do not form a valid definition.

This can be worked around by first inductively defining $t[+1]$ ("lift").
— in fact, one must inductively define $t[\Uparrow^i(+1)]$.

Then, we define $\sigma\,;+1$, whence $\Uparrow\sigma$, whence $t[\sigma]$ ("subst").

de Bruijn algebra

The following equations are sound, that is, valid:

$$
\begin{array}{rclcrcl}
(\lambda t)[\sigma] & = & \lambda(t[0 \cdot (\sigma \,;\, +1)]) & \quad & id \,;\, \sigma & = & \sigma \\
(t_1\ t_2)[\sigma] & = & t_1[\sigma]\ t_2[\sigma] & & \sigma \,;\, id & = & \sigma \\
0[t \cdot \sigma] & = & t & & (\sigma_1 \,;\, \sigma_2) \,;\, \sigma_3 & = & \sigma_1 \,;\, (\sigma_2 \,;\, \sigma_3) \\
(+1) \,;\, (t \cdot \sigma) & = & \sigma & & (t \cdot \sigma_1) \,;\, \sigma_2 & = & t[\sigma_2] \cdot (\sigma_1 \,;\, \sigma_2)
\end{array}
$$

Furthermore, they are complete (Schäfer *et al.*, 2015).

That is, if an equation based on the following grammar is valid, then it logically follows from the above equations.

$$
\begin{array}{rcl}
t & ::= & 0 \mid \lambda t \mid t\ t \mid t[\sigma] \mid T \\
\sigma & ::= & +1 \mid t \cdot \sigma \mid \sigma \,;\, \sigma \mid \Sigma
\end{array}
$$

Schäfer *et al.* also prove that validity is decidable.

de Bruijn algebra

Decidability means that the machine can answer questions for us.

Does $t[id] = t$ hold? Yes.

Does $t[\sigma_1][\sigma_2] = t[\sigma_1 \,;\, \sigma_2]$ hold? Yes.

Does $t[+1][u \cdot id] = t$ hold? Yes.

— In nominal style, this would be written: $t[u/x] = t$ if $x \notin fv(t)$.
— de Bruijn style is less familiar, but has no side condition.

And so on, and so forth.

For proofs of the first two equations above, see Schäfer *et al.*, Fact 6.

Yet, we do not really care about these proofs – a machine can find them.

Coq tactics for de Bruijn algebra

The Coq library Autosubst offers two tactics:

- `autosubst` proves an equation between terms or substitutions;
- `asimpl` simplifies a goal in which a term or substitution appears.

λ-terms with AutoSubst

The syntax of λ-calculus can be declared as follows:

```
Inductive term :=
| Var:          var -> term
| Lam:  {bind term} -> term
| App: term -> term -> term.
```

AutoSubst defines `var` as a synonym for `nat`
and `{bind term}` as a synonym for `term`.

AutoSubst defines substitution application, composition, etc., for us.

See `DemoSyntaxReduction`.

AutoSubst key notations

$t \cdot \sigma$	`t .: sigma`	substitution "cons"
$+i$	`ren (+i)`	the substitution $+i$
id	`ids`	the identity substitution
$t[\sigma]$	`t.[sigma]`	substitution application
$\sigma_1 ; \sigma_2$	`sigma1 >> sigma2`	substitution composition
$\Uparrow \sigma$	`up sigma`	taking a substitution under a binder
$\Uparrow^n \sigma$	`upn n sigma`	taking a substitution under n binders
$t.[u \cdot id]$	`t.[u/]`	substituting u for 0 in t

"lift" as end-of-scope

Suppose we are building a term in a context where n variables exist and we wish to refer to a subterm t that was built in a context where one of our variables, say x, where $0 \leq x < n$, did not exist.

We cannot just refer to t. Instead, we must use $t[\Uparrow^x (+1)]$.

Ugly, low-level index arithmetic? No: read it as an end-of-scope mark.

Adopt a nicer meta-level notation for it, say "*unbind x in t*".

The equation discussed earlier is now written $(\textit{unbind}\ 0\ \textit{in}\ t)[u \cdot id] = t$.

A related, object-level end-of-scope construct, "abdmal", has been studied by Hendriks and van Oostrom (2003).

Calculi of explicit substitutions

Similarly, we have viewed substitution application as a meta-level operation. There is no syntax for it in the λ-calculus.

In the $\lambda\sigma$-calculus, however, there is syntax for substitutions and substitution application, and a set of small-step reduction rules that explain how substitutions interact with λ-abstractions and applications.

Abadi, Cardelli, Curien, Lévy, Explicit substitutions, 1990.

Curien, Hardin, Lévy, Confluence properties of weak and strong calculi of explicit substitutions, 1992.

Binding and α-equivalence

Most programming languages provide constructs that bind variables:

- function abstractions (in terms): $\lambda x.t$
- local definitions (in terms): $let\ x = t\ in\ t$
- quantifiers (in types): $\forall\alpha.\alpha \to \alpha$

α-equivalence is a relation that allows renamings of bound variables, e.g.:

$$\lambda x.\, x + 1 \equiv_\alpha \lambda y.\, y + 1 \qquad\qquad \forall\alpha.\, \alpha\ list \equiv_\alpha \forall\beta.\, \beta\ list$$

α-equivalence can be defined as follows (Pitts, 2006):

$$\lambda x.t \equiv_\alpha \lambda y.\binom{x}{y}t \qquad \text{if } y \notin fv(\lambda x.t)$$

where $\binom{x}{y}$ swaps all occurrences of x and y

Implicit α-equivalence

On paper, it is customary to confuse α-equivalence $\equiv_\alpha$ with equality $=$.

This plays a role, for instance, in the definition of System F.

This is the traditional rule for type-checking a function application:

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\; e_2 : \tau'}$$

If α-equivalence was explicit, the rule would be written as follows:

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau \equiv_\alpha \tau_2}{\Gamma \vdash e_1\; e_2 : \tau'}$$

In simply-typed λ-calculus, this issue does not arise, as there are no quantifiers in types: α-equivalence and equality of types coincide.

Explicit α-equivalence

In principle, one should distinguish between:

- trees versus equivalence classes of trees;
- equality $=$ versus α-equivalence $\equiv_\alpha$.

This sounds easy enough, but leads to subtleties when defining mathematical functions that consume or produce trees... such as:

- program transformations, which produce and consume syntax trees;
- proofs, which produce and consume derivation trees.

Functions on equivalence classes

To define a function f from $\mathbb{T}/\equiv_\alpha$ to $\mathbb{T}/\equiv_\alpha$,
it suffices to first define a relation F between $\mathbb{T}$ and $\mathbb{T}$,
and to require two conditions:

- every tree is α-equivalent to some tree in the domain of F:

$$\forall t \in \mathbb{T} \quad \exists t', u' \in \mathbb{T} \quad t \equiv_\alpha t' \wedge t' \; F \; u'$$

  – note: the domain of F need not be $\mathbb{T}$
  – "without loss of generality, let us assume that x does not occur in ..."

- F is compatible with α-equivalence:

$$\forall t, t', u, u' \in \mathbb{T} \quad t \; F \; u \wedge t' \; F \; u' \wedge t \equiv_\alpha t' \Rightarrow u \equiv_\alpha u'$$

  – note: F need not be deterministic (single-valued)
  – nondeterminism is fine as long as all choices yield α-eq. results
  – "let us pick a name x outside of ..."

Free variables

The classic definition of the set of the free variables of a λ-term:

$$
\begin{aligned}
fv(x) &= \{x\} \\
fv(\lambda x.t) &= fv(t) \setminus \{x\} \qquad \text{– no requirement on } x \\
fv(t_1\ t_2) &= fv(t_1) \cup fv(t_2)
\end{aligned}
$$

A total function from $\mathbb{T}$ to sets of names.

Condition 1 is vacuously satisfied (the relation is defined everywhere).

Condition 2 requires checking the following equality:

$$fv(\lambda x.t) = fv(\lambda y.\binom{x}{y}t) \qquad \text{where } y \notin fv(\lambda x.t)$$

This follows from the fact that fv is equivariant, i.e., commutes with swaps:

$$fv(\pi\, t) = \pi\, fv(t)$$

and from the fact that neither x nor y appear in the set $fv(\lambda x.t)$.

Thus, fv gives rise to a total function from $\mathbb{T}/\equiv_\alpha$ to sets of names.

Capture-avoiding substitution

The classic definition of capture-avoiding substitution:

$$
\begin{array}{rcll}
x[u/x] & = & u & \\
y[u/x] & = & y & \text{if } y \neq x \\
(\lambda z.t)[u/x] & = & \lambda z.\, t[u/x] & \text{if } z \notin fv(u) \cup \{x\} \qquad \text{– avoid capture!} \\
(t_1\ t_2)[u/x] & = & t_1[u/x]\ t_2[u/x] &
\end{array}
$$

A partial function from $\mathbb{T}$ to $\mathbb{T}$.

Condition 1 holds, as only a finite number of choices for z are forbidden.

Condition 2 requires checking:

$$\lambda z.\, t[u/x] \equiv_\alpha \lambda z'.\, t'[u/x] \qquad \text{where } z, z' \notin fv(u) \cup \{x\} \text{ and } \lambda z.t \equiv_\alpha \lambda z'.t'$$

which follows, again, from the fact that substitution is equivariant.

Thus, this gives rise to a total function from $\mathbb{T}/\equiv_\alpha$ to $\mathbb{T}/\equiv_\alpha$.

Naïve substitution

Naïve substitution does not have the side condition $z \notin fv(u) \cup \{x\}$.

It is a total function from $\mathbb{T}$ to $\mathbb{T}$,
but does not satisfy condition 2,
hence does not give rise to a function from $\mathbb{T}/{\equiv_\alpha}$ to $\mathbb{T}/{\equiv_\alpha}$.

$$
\begin{array}{rcll}
(\lambda y.\, x + y)[2 \times y / x] & = & \lambda y.\, 2 \times y + y & \text{– naïve} \\
(\lambda y.\, x + y)[2 \times y / x] & = & & \\
(\lambda z.\, x + z)[2 \times y / x] & = & \lambda z.\, 2 \times y + z & \text{– capture-avoiding}
\end{array}
$$

The nominal approach in proof assistants

The nominal approach is prevalent in informal (paper) proofs.

It is implemented in Nominal Isabelle (Urban, 2008).

- Urban and Narboux (2008) present typical proofs about operational semantics.

It is not well supported in Coq, perhaps for engineering reasons.

- Cohen (2013) shows how to use quotients in Coq (when they exist) and how to construct them (up to certain axioms or hypotheses).