Programming in type theory, or:

How Coq became my favorite programming language

Xavier Leroy (Collège de France)

Workshop in Honour of Thierry Coquand’s 60th Birthday, 2022-08-24

Early days

Starting my graduate studies

Two suns : Caml (functional language) and Coq (proof assistant).

Types in programming languages

A central theme in 1990's P.L. research.

Goals for a P.L. type system :

- Guarantee integrity of data structures.
- Find bugs.
- Express some of the program structure.

Non-goal : termination. Turing-completeness was a must-have.

Some work on typing (other) effects.

The mysterious system called Coq

Technically : OK

(Calculus of Constructions $\approx F_\omega$ on steroids)

Conceptually : mysterious

(I had no background in constructive logic back then...)

Practically : unclear

(What were the intended uses for this Coq system ?)

My first attempt to use Coq (circa 1991)

```
Inductive Set regexp =
    Empty: regexp
  | Epsilon: regexp
  | Char: nat -> regexp
  | Seq: regexp -> regexp -> regexp
  | Alt: regexp -> regexp -> regexp
  | Star: regexp -> regexp.
```

My first attempt to use Coq (circa 1991)

```
Inductive Set regexp =
    Empty: regexp
  | Epsilon: regexp
  | Char: nat -> regexp
  | Seq: regexp -> regexp -> regexp
  | Alt: regexp -> regexp -> regexp
  | Star: regexp -> regexp.

Definition nullable : regexp -> Prop =
  [r:regexp](<Prop>Match r with
      (* Empty *)     False
      (* Epsilon *)   True
      (* Char c *)    [c:char] False
      (* Seq r1 r2 *) [r1,r2:regexp][nu1,nu2:Prop](nu1 /\ nu2)
      (* Alt r1 r2 *) [r1,r2:regexp][nu1,nu2:Prop](nu1 \/ nu2)
      (* Star r1 *)   [r1:regexp][nu1:Prop] True).
```

My first actual use of Coq (1998)

Specifying and proving properties of a JavaCard bytecode verifier (≈ a type-checker for virtual machine code).

Part of a general movement towards mechanized metatheory of programming languages.

Type Inference Verified: Algorithm $\mathcal{W}$ in Isabelle/HOL ⋆

WOLFGANG NARASCHEWSKI and TOBIAS NIPKOW
Technische Universität München, Institut für Informatik, 80290 München, Germany.
e-mail: {narasche,nipkow}@in.tum.de

Certification of a Type Inference Tool for ML: Damas–Milner within Coq

CATHERINE DUBOIS
Université d'Évry Val d'Essonne. e-mail: Catherine.Dubois@lami.univ-evry.fr

VALÉRIE MÉNISSIER-MORAIN⋆
Université Paris 6. e-mail: Valerie.Menissier@lip6.fr

Mechanized Metatheory for the Masses: The PoplMark Challenge

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1], Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

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

Mechanizing the metatheory of programming languages

On paper : mostly definitions by inference rules.

In Coq : mainly inductive predicates, with proofs by structural induction, inversion, and Prolog-style search.

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1 . t_2 : T_1 \rightarrow T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

(B. C. Pierce, 2002)

```
Reserved Notation "Gamma '⊢' t '∈' T"
            (at level 101,
             t custom stlc, T custom stlc at level 0).

Inductive has_type : context → tm → ty → Prop :=
  | T_Var : ∀ Gamma x T₁,
      Gamma x = Some T₁ →
      Gamma ⊢ x \in T₁
  | T_Abs : ∀ Gamma x T₁ T₂ t₁,
      x ⊢> T₂ ; Gamma ⊢ t₁ \in T₁ →
      Gamma ⊢ \x:T₂, t₁ \in (T₂ → T₁)
  | T_App : ∀ T₁ T₂ Gamma t₁ t₂,
      Gamma ⊢ t₁ \in (T₂ → T₁) →
      Gamma ⊢ t₂ \in T₂ →
      Gamma ⊢ t₁ t₂ \in T₁
```

(B. C. Pierce et al, since 2008)

Programming in Coq

An eye opener : verification of OCaml's AVL sets library

(Filliâtre and Letouzey, ESOP 2004)

Functors for Proofs and Programs

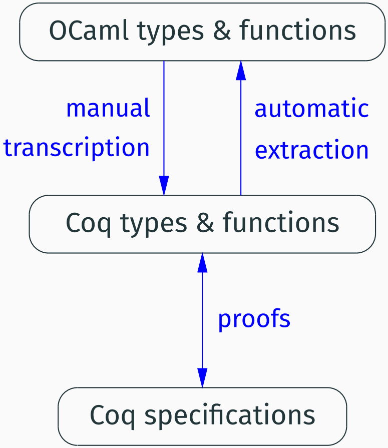
Jean-Christophe Filliâtre and Pierre Letouzey

LRI – CNRS UMR 8623
Université Paris-Sud, France
{filliatr,letouzey}@lri.fr

Abstract. This paper presents the formal verification with the Coq proof assistant of several applicative data structures implementing finite sets. These implementations are parameterized by an ordered type for the elements, using functors from the ML module system. The verification follows closely this scheme, using the newly Coq module system. One of the verified implementation is the actual code for sets and maps from the Objective Caml standard library. The formalization refines the informal specifications of these libraries into formal ones. The process of verification exhibited two small errors in the balancing scheme, which have been fixed and then verified. Beyond these verification results, this article illustrates the use and benefits of modules and functors in a logical framework.

An eye opener : verification of OCaml's AVL sets library

(Filliâtre and Letouzey, ESOP 2004)




```
type t = Empty
       | Node of t * elt * t * int
```

```
Inductive raw :=
| Empty: raw
| Node: raw -> elt -> raw -> int -> raw.
Definition t :=
  { r: raw | bst r /\ avl r }.
```

```
Lemma add_spec: forall x y s,
  In y (add x s) <-> In y s \/ eq y x.
```

An eye opener : verification of OCaml's AVL sets library

(Filliâtre and Letouzey, ESOP 2004)

The birth of a methodology : Coq as a proof assistant and as a functional programming language.

Found two balancing bugs in the OCaml implementation (correct results but wrong complexity).

Prompted a welcome simplification of the `compare` function (the “same fringe problem”) :

- Original implementation : complicated traversal, termination argument unclear.
- Revised implementation : using zippers as iterators ; all recursions are structural.

Compiler verification

How to establish that a compiler is free of miscompilation bugs ?
Prove a semantic preservation property :

> *When executed, the generated compiled code behaves as prescribed by the semantics of the source program.*

An old idea :

- McCarthy and Painter (1967) : arithmetic expressions, paper proof.
- Milner and Weyrauch (1972) : arithmetic expressions, LCF proof.
- Rittri (1992), Hardin et al (1998) : functional abstract machines (SECD, CAM, etc), paper proofs.
- Grégoire and Leroy (2002) : functional abstract machine, Coq proof.

Compiler verification mechanized in Stanford LCF (1972)

3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch

Computer Science Department
Stanford University

Abstract

We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is presented only in outline. However, we present in full the expression-compiling part of the algorithm. More than half of the complete proof has been machine checked, and we anticipate no difficulty with the remainder. We discuss our experience in conducting the proof, which indicates that a large part of it may be automated to reduce the human contribution.

Even proof scripts look familiar…

APPENDIX 2: command sequence for McCarthy-Painter lemma

```
GOAL ∀e sp,Iswfse e::MT(compe e,sp)≡svof(sp)|((MSE(e,svof sp))&pdof(sp)),
     ∀e,Iswfse e::Iswft(compe e)≡TT,
     ∀e,Iswfse e::(count(compe e)=0)≡TT;

TRY 1 INDUCT 56;
 TRY 1 SIMPL;
 LABEL INDHYP;
 TRY 2 ABSTR;
  TRY 1 CASES wfsefun(f,e);
  LABEL TT;
   TRY 1 CASES type e=_N;
    TRY 1 SIMPL BY ,FMT1,,FMSE,,FCOMPE,,FISWFT1,,FCOUNT;
    TRY 2;SS-,TT;SIMPL,TT;QED;
    TRY 3 CASES type e=_E;
      TRY 1 SUBST ,FCOMPE;
      SS-,TT;SIMPL,TT;USE BOTH3 -;SS+,TT;
      INCL-,1;SS+-;INCL--,2;SS+-;INCL---,3;SS+-;
       TRY 1 CONJ;
        TRY 1 SIMPL;
         TRY 1 USE COUNT1;
          TRY 1;
          APPL ,INDHYP+2,arg1of e;
          LABEL CARG1;
          SIMPL-;QED;
          TRY 2 USE COUNT1;
           TRY 1;
```

Scaling up : the CompCert verified C compiler

Same kind of compiler verification, just more realistic :

- Source language : most of C.
- Target language : assembly code for real processors.
- Produces efficient enough code $\rightarrow$ some optimizations.

Same methodology as in Filliâtre and Letouzey :
program and prove the compiler in Coq.

A methodology : Coq as a programming language and a prover

Write the program as Coq datatypes and functions, in "hyper-pure" functional style.

- No imperative programming; use monads for all effects.
- All functions terminate (structural or well-founded recursion).

Prove the expected properties of these functions.

- The program is an object of Coq's logic.
- No need for a separate program logic!

Generate executable OCaml code by automatic extraction.

- Erases most of the specs, proofs, and termination arguments.
- Can link with hand-written OCaml code for I/O, etc.

Programming a compiler in hyper-pure functional style

Doable with a few tricks that can be presented as monads.

Error reporting : no exceptions; use the error monad.

```
Inductive mon A := OK (res: A) | Error (err: error_message).
```

Algorithms whose termination is difficult to prove :
can use “fuel”, or Capretta’s delay type.

```
Definition mon A := nat -> option A.
```

```
CoInductive mon A := Now (res: A) | Later (d: mon A).
```

Programming a compiler in hyper-pure functional style

In-place update of arrays, graphs, ... : (state monad)
use functional data structures + state-passing functions.

```
Definition mon A := state -> A * state.
```

Can use dependent types to express interesting properties of the imperative computation, such as monotonic state.

```
Definition mon A :=
    forall (s: state), A * { s': state | s' >= s }.
```

Can even embed a Hoare-style program logic!

```
Definition Hoare (A: Type) (Pre: state -> Prop)
                           (Post: A -> state -> Prop) :=
    forall (s: state),
    Pre s -> { v: A & s': state | Post v s' }.
```

Efficient, extensional data structures

Efficient functional data structures

Lists are not good enough! Need more efficient data structures

- for execution after extraction to OCaml;
- for computation within Coq, typically
  for program logics embedded in Coq, like VST and Iris.

When verifying a given program, variables names are known, so general theorems such as

$$\texttt{get}\ x\ (\texttt{set}\ y\ v\ m) = \texttt{get}\ x\ m \qquad \text{if } x \neq y$$

can become mere computations

$$\texttt{get}\ \texttt{"foo"}\ (\texttt{set}\ \texttt{"bar"}\ v\ m) \xrightarrow{*} \texttt{get}\ \texttt{"foo"}\ m$$

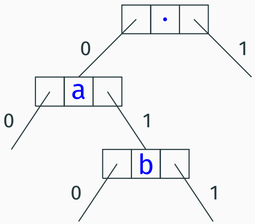
The main data structures of CompCert

Integers and floating-point numbers

Any base-2 representation is fast enough, but not Peano integers.

Finite maps for environments, functional arrays, graphs, …

CompCert mainly uses binary tries indexed by base-2 positive integers ($\equiv$ lists of bits).




```
Inductive tree A :=
  | Leaf
  | Node (l: tree A)
         (x: option A)
         (r: tree A).
```

Finite sets, union-find, priority queues for static analyses.

Extensional equality for sets and maps

Just as with functional extensionality, proofs are simpler when

- finite sets having the same elements are (Leibniz-)equal ;
- finite maps that map equal keys to equal data are equal.

This is false for implementations based on binary search trees.
For instance, the set $\{1, 2\}$ has two BST representations :

Consequently, properties such as $A \cup B = B \cup A$ are not identities, only setoid equalities.

Extensionality via well-formedness constraints

Lists of integers are not an extensional representation of sets (since $[1; 2] \neq [2; 1]$), but sorted lists are.

```
Definition intset := { l: list Z | Sorted Z.lt l }.
```

Binary tries are not an extensional representation of maps since the empty map has multiple representations :

```
Leaf  ≠ Node Leaf None Leaf
      ≠ Node Leaf None (Node Leaf None Leaf)
```

However, well-formed binary tries (not containing `Node Leaf None Leaf`) are extensional.

```
Definition map A := { t: tree A | wf t }.
```

Problems with subset types

```
Definition map A := { t: tree A | wf t }.
```

The proposition `wf t` must have unique proofs

- Often, a Boolean equality works : `wf t` is `wf_dec t = true`.
- More generally : use a “mere proposition”.

Problems with subset types

```
Definition map A := { t: tree A | wf t }.
```

The proposition `wf t` must have unique proofs

- Often, a Boolean equality works : `wf t` is `wf_dec t = true`.
- More generally : use a “mere proposition”.

Subset types often compute inefficiently within Coq

- The proof term for `wf t` grows uncontrollably.
- Not an issue after extraction (proof erasure).

The attack of the huge proof term

```
Fixpoint t_set x v t := ...

Lemma wf_set: forall x v t, wf t -> wf (t_set x v t).

Definition set x v m :=
  let (t, w) := m in exist (t_set x v t) (wf_set x v t w).
```

Successively adding N values $v_1, \ldots, v_N$ to key 1 results in a small binary tree `Node Leaf (Some` v_N`) Leaf` and a proof of size N

$$\texttt{wf\_set}\ 1\ v_N\ (\ \ldots\ (\texttt{wf\_set}\ 1\ v_1\ \texttt{wf\_Leaf})\ \ldots\ )$$

If `wf_set` is opaque, this proof is in normal form but takes time $O(N)$ for convertibility checks or just for garbage collection.

(Making `wf_set` transparent usually makes things worse.)

Extensionality via canonical representations

In some lucky cases, we can build representations that are canonical : every abstract object has a unique representation.

Example : binary natural numbers.

Lists of bits are not a canonical representation (can always add leading zero bits), but the following representation is :

```
Inductive positive :=
  | xH               (* 1 *)
  | xO (p: positive) (* 2p *)
  | xI (p: positive) (* 2p+1 *).

Inductive N := N0 | Npos (p: positive).
```

Canonical binary tries (A. W. Appel and X. Leroy, 2022)

A similar approach leads to a canonical representation of binary tries, where every map has a unique representation.

```
Inductive tree' A :=                        (* nonempty maps *)
  | Node001: tree' A -> tree' A
  | Node010: A -> tree' A
  | Node011: A -> tree' A -> tree' A
  | Node100: tree' A -> tree' A
  | Node101: tree' A -> tree' A -> tree' A
  | Node110: tree' A -> A -> tree' A
  | Node111: tree' A -> A -> tree' A -> tree' A.

Inductive tree A :=                         (* all maps *)
  | Empty: tree A
  | Nodes: tree' A -> tree A.
```

Concluding remarks

On programming in type theory

Hyper-pure functional programming works fine now that we have

- monads to express effects, including nontermination ;
- efficient functional data structures.

The combination with dependent types gives tremendous power to reason about programs.

> *Type theory is the mother of all program logics.*

Coq was one of the first systems to demonstrate this approach. Great work, Thierry !

On data structures and equality

It’s not just mathematicians who need quotient types that work!

Programmers, too, would like data structures (with multiple concrete representations for an abstract value) to behave well with respect to equality…

(Canonical representations and subset types generally don’t suffice to get efficient data structures.)

I still hope that solutions to this problem will come out of the work on homotopy type theory or observational type theory.

In the meantime, it would be nice to have efficient computations within Coq on subset types $\{x : A \mid P\}$ where P is a mere proposition.