Implementing Constructive Logics with OCaml

Siva Somayyajula Chirag Bharadwaj

December 6, 2016

Contents

1 Introduction

We survey constructive logics in terms of logics and their corresponding proof (by refinement) systems. We first consider OCaml as a candidate for a polymorphic programming logic—in particular, we demonstrate that techniques in "lightweight dependent-type programming" allow us to encode the Peano Axioms as OCaml types with corresponding evidence terms as witnesses to their validity. Furthermore, we attempt at encoding full Peano Arithmetic and discuss the possibility of extracting OCaml terms from proofs-by-refinement in this logic. Using this knowledge, we discuss the design of a framework for building proof assistants for refinement logics in OCaml as well as the actual implementation of propositional logic. We will also discuss the hypothetical implementation of more sophisticated (first-order) systems—in particular, first-order logic with equality and Peano Arithmetic. In doing so, we hope to highlight the deep computational meaning of propositions and proofs and how that aids mathematicians and computer scientists alike in their work.

2 Theory of Polymorphic Programming Logics

2.1 Realizability Interpretation

The key insight into constructive logic that gave rise to computational interpretations of logic like propositions-as-types/programs-as-proofs and proof by refinement was the notion of *realizability*. Stephen Kleene demonstrated that a proposition provable in a constructive logic can be *realized* by a mathematical object dual to the original proof. We can elaborate on this using the notion that constructive proofs have *computational content*, which makes realizers programs. With this, we have a programmatic interpretation of logics that allows us to investigate constructive logics using computational systems known as *type theories*. We will thus look at logics of increasing expressivity and attempt to emulate them in OCaml, while cognizant of the limitations of its type system. As a result, we will conclude that OCaml is actually a useful *polymorphic programming logic*.

2.2 Propositional Logic

To review, in lecture, we demonstrated that the simply typed lambda calculus augmented with products and sums is identifiable with propositional logic. In other words, atomic propositions correspond to the base types, and propositions constructed by logical connectives correspond to types constructed by various type constructors. Furthermore, the proof of a proposition is dual to a program of the corresponding type. For example, fix a set of base types $B = \{\mathbf{int}, \mathbf{bool}\}$ corresponding to propositions P and Q, respectively. Then the proposition $P \wedge Q \Rightarrow P$ has a proof $\mathrm{fst}_{\mathbf{int}\times\mathbf{bool}}$, whose type is $\mathbf{int} \times \mathbf{bool} \to \mathbf{int}$. For completeness, we provide a type for falsity and negation below.

```
type void = {none:'a.'a}
type 'a not = 'a -> void
```

2.3 Second-order Propositional Logic

While the above system is sufficient for ordinary propositional logic, such a system lacks universal quantification over propositions. For example, it is impossible to prove $\forall P.\ P \Rightarrow P$. Indeed, one would have to write a separate proof for every possible proposition. Dually, one cannot write a universal identity function in the simply typed lambda calculus! In our previous example, one would have to write one for both **int** and **bool**.

For logicians, this motivated the development of a second-order propositional logic that introduces a universal quantifier for propositions; and for computer scientists who desired less redundant code, parametric polymorphism, which introduces a universal quantifier for types. In fact, the type theory dual specifically to this logic (without existential quantification over propositions) is System F, discovered independently by Jean-Yves Girard and John Reynolds [6].

However, the remainder of our discussion will be with respect to OCaml's type system, which implements a subset of System F called *rank-1 polymorphism*. This requires all type quantification to be done at the head of the type to keep type inference decidable [6]. Of course, this means that OCaml can only implement a subset of a second-order propositional logic, but that is sufficient for our needs. Going off of our previous example, the proposition $\forall P.\ P \Rightarrow P$ has a proof `fun x -> x` whose type is `'a -> 'a`. OCaml omits the universal quantifier and instead places a syntactic restriction on type variables to recognize them.

2.4 First-order Logic

The type-theoretic interpretation of first-order logic is *dependent type theory*, which interprets universal quantification $\forall x.\ P(x)$ as the *dependent function type* $\Pi_{x:A}\ P(x)$ whose terms are functions with return types that may depend on the input term. Intuitively, proofs involving universal quantifiers must construct evidence of $P(x)$ for every possible x, hence the need for a function. Furthermore, existential quantification corresponds to the *dependent pair type* $\sum_{x:A} P(x)$ whose terms are pairs where the type of the second component may depend on the first. That is, proofs involving existential quantifiers require one to construct evidence for a single value x (existence) that satisfies the proposition paired with the proof of $P(x)$.

To further discuss first-order logic, we need to clarify the dichotomy of types and terms in the logical sense. To elaborate, it is clear that OCaml lacks dependent types, so one cannot encode first-order propositions as OCaml types. However, this does *not* mean we cannot produce realizers of first-order propositions in OCaml! In fact, this is possible because OCaml's term language has function and pair terms that are the same for dependent types. In other words, a realizer for a first-order proof in OCaml would have a type dual to the original proposition in some dependently-typed language, although its "real" OCaml type is not. Here's an example straight from lecture: the proof of $\exists x.\ (P(v) \Rightarrow C) \Rightarrow (\forall x.\ P(x) \Rightarrow C)$ is given by `fun (x, f) f' -> f (f (f' x))`, which is a valid OCaml term.

2.5 The Equality Problem

First-order logic *with equality* is implemented by *intuitionistic type theory*, which is dependent type theory augmented with an *equality type* constructor: for terms a and b of type A, $a = b$ is a type whose values are proofs that a and b are equal. Thus $a = b$ is inhabited if and only

if a is in fact equal to b, and its only canonical value is reflexivity: $\text{refl} : \Pi_{a:A}\ a = a$—that is, $\forall a.\ a = a$.

Unfortunately, this is where vanilla OCaml stops getting away with having a deficient type system, because there are currently no values that can be interpreted as proofs of equality between terms. As a result, one cannot realize first-order proofs with equality in OCaml. However, we may recover propositional equality on types via a constructive definition of equality, which we will show obeys first-order axioms.

Note that equality in first-order logic must satisfy the following axioms.

1. Reflexivity: $\forall a.\ a = a$
2. Functional substitutivity: for all functions f, $\forall a, b.\ a = b \Rightarrow f(a) = f(b)$
3. (Second-order) Leibniz's laws
   (a) The indiscernibility of identicals: $\forall P.\ \forall a, b.\ a = b \Rightarrow P(a) \Leftrightarrow P(b)$
   (b) The identity of indiscernibles: $\forall P.\ \forall a, b.\ P(a) \Leftrightarrow P(b) \Rightarrow a = b$

Consequences of these laws are symmetry and transitivity. To encode these in OCaml, Yallop and Kiselyov's [7] define constructive equality as: $a = b \Leftrightarrow \forall P.\ P(a) \Rightarrow P(b)$. Dually, types `a` and `b` are equal iff for all type constructors `t`, there exists an injective function `a t -> b t`. Below is the signature of the equality module they give, modified for our use.

```
(* The type for unary atomic propositions *)
module type PROP = sig
  type 'a t
end

module type EQ = sig
  (* Type-level (propositional) equality *)
  type ('a, 'b) eq
  (* Reflexivity *)
  val refl : ('a, 'a) eq
  (* Symmetry *)
  val sym : ('a, 'b) eq -> ('b, 'a) eq
  (* Transitivity *)
  val trans : ('a, 'b) eq -> ('b, 'c) eq -> ('a, 'c) eq
  (* Substitution for functions (computationally, a coercion) *)
  val coerce : ('a, 'b) eq -> 'a -> 'b
end

module WeakEq : EQ with type ('a, 'b) eq = ('a -> 'b) * ('b -> 'a)

module StrongEq : sig
  include EQ
  (* The indiscernibility of identicals *)
  module Lift : functor (P : PROP) -> sig
    val f : ('a, 'b) eq -> ('a P.t, 'b P.t) eq
```

```
  end
  (* The identity of indiscernibles *)
  module Inject : functor (P : PROP) -> sig
    val f : ('a P.t, 'b P.t) eq -> ('a, 'b) WeakEq.eq
  end
end
```

However, there is a fundamental limitation in their implementation: they must differentiate between strong (Leibnizian, given by `StrongEq`) and weak equality (type isomorphism given by `WeakEq`) because strong equality specifically cannot recover the identity of indiscernibles, whereas isomorphism can [7]. This is due to the fact that type constructor injectivity is in general not deducible—for example, if the type constructor is `type 'a t = int`, then it is not injective, so it varies based on the specific constructor. That is, the set of all types represented by `'a` must be assigned to `int`, which is only a surjection. This will change the way we encode Peano's axioms, as we discuss below.

While type-level equality is useful in software engineering, we can actually recover full propositional term equality à la intuitionistic type theory by restricting our universe of terms and finding ways to encode said terms as types.

2.6 Type-level Naturals with Singletons

If we restrict ourselves to the natural numbers as a domain of discourse, turns out we may encode them as types using *singletons*, a mechanism Kiselyov introduced [3]. A *singleton* is a type that has only one non-bottom inhabitant. Thus, consider the following definitions.

```
type z = Z
type 'n s = S of 'n
```

The only non-bottom value `z` has is `Z`, `z s` has `S Z`, `z s s` has `S (S Z)`, and so on. As a result, we (almost) have a type of natural numbers indexed by value. To unify them, we will use *generalized algebraic data types*.

```
type _ nat =
  | Z : z nat
  | S : 'n nat -> 'n s nat
```

In effect, the type of `'a nat` depends on its value i.e. `Z` has type `z nat`, `S Z` has `z s nat`, `S (S Z)` has `z s s nat`, etc. As a result, we get dependent types over the naturals in OCaml: $\Pi_{\texttt{x:nat}}$ `T(x)` $\Leftrightarrow$ `'x nat -> T('x)` and $\sum_{\texttt{x:nat}}$ `T(x)` $\Leftrightarrow$ `'x nat * T('x)`. One might inaccurately conclude that this discovery invalidates our discussion about the inability to give an OCaml type for a term corresponding to a first-order proof. We have not recovered *full* dependent types; in effect, our domain of discourse is limited to the natural numbers.

2.7 Encoding Peano's Axioms

We may encode the Peano axioms in OCaml using the equality types with `nat`.

1. $0 \in \mathbb{N}$

   ```
   let zero : z nat = Z
   ```

2. $\forall x.\ S(x) \in \mathbb{N}$

```
let succ : 'x nat -> 'x s nat =
  fun x -> S x
```

3. $\forall x.\ \neg(S(x) = 0)$

```
let succ_neq_zero : 'x nat -> ('x s, z) StrongEq.eq not =
  fun x eq -> invalid_arg "impossible"
```

4. $\forall x, y.\ S(x) = S(y) \Rightarrow x = y$

```
let succ_eq : 'x nat -> 'y nat -> ('x s, 'y s) StrongEq.eq -> ('x, 'y) WeakEq.eq =
  fun _ _ ->
    let module I = StrongEq.Inject(struct type 'a t = 'a s end) in
    I.f
```

   In other words, the successor function is an injection, and since we already have evidence that type constructors are injective, we reuse the proof.

5. Second-order induction axiom: $\forall P.\ P(0) \wedge (\forall x.\ P(x) \Rightarrow P(S(x))) \Rightarrow \forall x.\ P(x)$. Note that quantification over type constructors (dually, n-ary predicates) requires the use of functors.

```
module Induction (P : PROP) = struct
  let rec f :
    type x n. z P.t -> (x nat -> x P.t -> x s P.t) -> n nat -> n P.t =
      fun b i -> function
        | Z -> b
        | S nn -> i nn (f b i nn)
end
```

   However, attempting to compile this in OCaml will not work even though it is type-correct—this is simply a limitation in how OCaml performs *polymorphic recursion*. In particular, it fails to instantiate the type variable `x` with the type of `nn` in the call to the inductive case. While we cannot use an induction axiom, this does not stop us from actually doing inductive proofs, as we will see below. That being said, this reveals a rather deep duality between induction and recursion, i.e. the computational content of an inductive argument is a recursive function.

2.8 Towards Peano Arithmetic

It turns out the singleton encoding is powerful enough to simulate type-level addition, although in a slightly unconventional way. We use the encoding given by Rosetta Code [5], but change it to match our encoding of natural numbers.

```
type (_, _, _) plus =
  (* forall x. 0 + x = x *)
  | PZero : 'x nat -> (z, 'x, 'x) plus
  (* x + y = z => x + S(y) = S(z) *)
  | PSucc : ('x, 'y, 'z) plus -> ('x, 'y s, 'z s) plus
```

In essence, we are defining addition as a relation. Let us prove a simple property, like $\forall n.\, 0 + n = n$.

```
let rec plus_zero_eq_n : type n. n nat -> (z, n, n) plus = function
  | Z   -> PZero Z
  | S n -> PSucc (plus_zero_eq_n n)
```

Notice that this differs slightly from [5] due to the change in the definition for `plus`. As noted before, we are still allowed to perform recursive proofs since the polymorphic recursion here is inferable by the OCaml typechecker.

3 Refinement Logic

Now that we have finished our discussion of OCaml as a polymorphic programming logic, we will make these ideas explicit in a set of proof calculi collectively known as *refinement logic*. On the surface, refinement logics resemble other such proof styles as Hilbert's style—but what sets it apart is that inference rules specifically encode how to *realize* a logical formula in a given *term language*. As such, this is the logical conclusion of our discussion—"promoting" a polymorphic programming logic to a full proof system. Since we demonstrated that OCaml terms may realize propositional and first-order formulae, we will implement propositional logic in OCaml and discuss the hypothetical implementation of higher systems like Peano Arithmetic.

3.1 Theory

We must establish some definitions before any further discussion on refinement logic [2]. Given some term language (e.g. OCaml) and formula language (e.g. the language of propositional formulae), we may associate any formula A with a term variable x using the *declaration* $x : A$. Then, the core structure of proofs is the *sequent*—a list of declarations paired with a *conclusion*, like so $x_1 : A_1, \ldots, x_n : A_n \vdash C$. Such a sequent is meant to express that the conclusion is valid assuming the *hypotheses* i.e. the declarations to the left of the turnstile. We may *refine* a sequent by transforming it into a list of subgoal sequents, and then use them to recursively realize the conclusion with an evidence term. A pair of such transformations is an *inference rule*, so a proof calculus is then a set of correct refinement rules, in the sense that after refining a sequent, one should be able to produce an evidence term for it.

A proof assistant is then an algorithm that, given an input sequent, allows the user to refine a sequent into subgoals until every branch can no longer be transformed further. The resultant tree-like structure of sequents is called a *proof*, from which an *extract term* that realizes the original conclusion may be generated from each rule applied.

3.2 Implementation

Given this information, we implemented a proof system development framework called `refined logic` in OCaml. Using the core structures it exports, one can write a proof system for any logic with extract terms in any term language. We summarize the main modules below.

- `base.ml(i)`: implements the core structures required for a refinement proof system. In particular, it exports a signature `PROOF_SYSTEM` which, quite literally, describes a proof system—a set of (correct) refinement rules, as well as a functor `Logic` that provides all the necessary data types and functions for writing refinement rules. `Logic` is parameterized by signatures `TERM_LANG`, which describes a language for extract terms, and `FORM_LANG`, which is for formulas in the given logic.
- `caml.ml(i)`: exports a signature `CAML_LIKE` which describes any `TERM_LANG` that is a typed lambda calculus with products and sums as well as an implementation for OCaml.
- `prop_logic.ml(i)`: exports a `PROOF_SYSTEM` for propositional logic called `PropLogic` with extract terms in any `CAML_LIKE` language. Uses rules from [2].
- `interactive.ml(i)`: exports a functor `ProofAssistant` that generates an interactive environment for proving theorems given a `PROOF_SYSTEM`.
- `prop_main.ml`: instantiates a `ProofAssistant` with `PropLogic` with extract terms in `OCaml` and runs it for the user.

To run `refined logic`, first make sure that you have OCaml's package manager OPAM installed. Then, make sure you have the current version of OCaml installed, and if you have not already, install Jane Street's `core` and the `ppx_import` package. Then, unzip the provided source code. Set the current directory to `refined-logic`, and then run any of the two following commands.

- `make prop`: builds an executable for the propositional logic proof assistant described in `prop_main.ml`
- `make clean`: cleans the directory of compiler objects

To run the proof assistant for propositional logic `./prop <sequent>` where sequents are of the form `x_1:A_1,...,x_n:A_n |- C` where `x_i` must be valid OCaml variables and `A_i` and `C` must be valid propositional formulae. We describe our syntax for propositional logic below:

- Atomic propositions use the same syntax as OCaml variables e.g. `P`, `Q`, etc.
- The logical connectives are `/\`, `\/`, $\sim$, and `=>`, for conjunction, disjunction, negation, and implication respectively.
- `False` denotes the proposition of falsity
- Parentheses may be used to group terms e.g. `(A \/ B) /\ ~ C => D`

3.3 Examples

We will now demonstrate examples of proving propositions in `refined logic`. Here is a session where we prove $\neg(P \vee Q) \implies \neg P$.

```
$ ./prop "|- ~(P \/ Q) => ~P"
|- ~(P \/ Q) => ~P by impliesR
  v1 : ~(P \/ Q) |- ~P by notR
    v1 : ~(P \/ Q), v2 : P |- false by notL v1
      v2 : P, v1 : ~(P \/ Q) |- P \/ Q by orR1
        v2 : P, v1 : ~(P \/ Q) |- P by axiom v2
(fun v1 -> (fun v2 -> (Obj.magic(v1(`L v2)))))
```

The final extract term is printed at the end. To verify that it has the correct type, we can paste it into UTOP.

```
utop # (fun v1 -> (fun v2 -> (Obj.magic(v1(`L v2)))));;
- : ([> `L of 'a ] -> 'b) -> 'a -> 'c = <fun>
```

This type can be specialized to `([ `L of 'a | `R of 'b ] -> bot) -> 'a -> bot`, as shown below.

```
utop # type bot = {none:'a.'a};;
type bot = { none : 'a. 'a; }
utop # ((fun v1 -> (fun v2 -> (Obj.magic(v1(`L v2))))) :
  ([`L of 'a | `R of 'b] -> bot) -> 'a -> bot);;
- : ([ `L of 'a | `R of 'b ] -> bot) -> 'a -> bot = <fun>
```

Now we will prove a more complicated proposition: $P \vee Q \Rightarrow (P \Rightarrow R) \Rightarrow (Q \Rightarrow R) \Rightarrow R$.

```
$ ./prop "|- (P \/ Q) => ((P => R) => ((Q => R) => R))"
 |- (P \/ Q => ((P => R) => ((Q => R) => R))) by impliesR
  v1 : P \/ Q |- ((P => R) => ((Q => R) => R)) by impliesR
    v1 : P \/ Q, v2 : (P => R) |- ((Q => R) => R) by impliesR
      v1 : P \/ Q, v2 : (P => R), v3 : (Q => R) |- R by orL v1
        v2 : (P => R), v3 : (Q => R), v5 : P |- R by impliesL v2
          v3 : (Q => R), v5 : P, v2 : (P => R) |- P by axiom v5
          v3 : (Q => R), v5 : P, v6 : R |- R by axiom v6
        v2 : (P => R), v3 : (Q => R), v4 : Q |- R by impliesL v3
          v2 : (P => R), v4 : Q, v3 : (Q => R) |- Q by axiom v4
          v2 : (P => R), v4 : Q, v7 : R |- R by axiom v7
(fun v1 -> (fun v2 -> (fun v3 ->
  (match v1 with
     `L v5 -> ((fun v6 -> v6) (v2 v5))
   | `R v4 -> ((fun v7 -> v7) (v3 v4))))))
```

Indeed, if we paste this extract into UTOP, we get the desired type.

```
utop # (fun v1 -> (fun v2 -> (fun v3 ->
  (match v1 with
     `L v5 -> ((fun v6 -> v6) (v2 v5))
   | `R v4 -> ((fun v7 -> v7) (v3 v4))))));;
- : [< `L of 'a | `R of 'b ] -> ('a -> 'c) -> ('b -> 'c) -> 'c = <fun>
```

3.4 Case Study: NuPRL

While our examples of term extraction are relatively tame, the utility of this mechanism lies in a more powerful polymorphic programming logic. In particular, the NuPRL system—developed by our very own Cornell University—is based on full intuitionistic type theory and can thus fully leverage first-order logic with equality on any domain. Furthermore, NuPRL includes an interactive proof environment in refinement style such that terms may be extracted into NuPRL's programming language. We will study one particular case in which the computational content of a proof actually encoded a very useful algorithm: fast integer square root [4].

The square root of an integer x is defined as r satisfying $r^2 \leq x \leq (r+1)^2$. It turns out every integer has a square root, and the proof of that claim encodes the algorithm to find it. The proof is quite long, but we will summarize its key aspects: by induction on x, one produces $r = 0$ for $x = 0$, and then for the inductive case, perform case analysis on x to produce r based on the square root of $x/4$. NuPRL includes an induction tactic that extracts recursive terms—induction is dual to recursion, which is what we showed. As a result, the extract term for this proof is given below: a recursive algorithm to find the square root of an integer.

```
letrec sqrt(x) =
  if x = 0 then
    0
  else
    let z := x / 4 in
    let r2 := 2 * (sqrt z) in
    let r3 := r2 + 1 in
    if (x) < (r3 * r3) then
      r2
    else
      r3 in sqrt(x)
```

4 Conclusion and Future Work

To summarize, we first discussed the viability of OCaml as a polymorphic programming logic by first noting that it implements a subset of second-order propositional logic. Then, we developed a singleton encoding of the natural numbers according to Kiselyov. We studied the implementation of Leibnizian equality and type isomorphism by Yallop and Kiselyov but noted its inability to express type constructor injectivity, corresponding to the second part of Leibniz's law. As a result, we were able to encode a variant of first-order logic with equality over the natural numbers (i.e. Peano Arithmetic). Using this information, we implemented a framework for refinement proof assistants with extract terms in OCaml. However, our analysis is not yet complete: how would we work towards a refinement logic for Peano Arithmetic? The main difficulty is finding an appropriate set of inference rules [1]. For example, the realizer for the proposition $\exists x.\, x = 1$ in OCaml is `(S Z, StrongEq.refl) : z s nat * (z s, z s) StrongEq.eq`. If were to prove such a statement in `refined logic`, we would probably have a proof tree like below, which introduces a rule `eqReflR` that allows

one to close off a subgoal upon a reflexive statement of equality.

```
|- exists x. x = S Z  by existsR (S Z)  ev = (S Z, StrongEq.refl)
  |- S Z = S Z        by eqReflR        ev = StrongEq.refl
```

This implies that we would need some inference rules concerning equality. In fact, it is plausible that we would need to only specifically encode reflexivity, substitutivity, and Leibniz's law as rules. The next difficulty, so to speak, is encoding Peano Axioms as rules. Perhaps the most enlightening is induction, because its extract term would be recursive, and thus would have to instantiate the conclusion itself as a subgoal (i.e. the inductive case).

Overall, such a project would be interesting from a purely implementation point-of-view, as we have demonstrated the theory aspect. As such, we hope that after reading this paper, the reader has become aware of the computational content of logic both in the theoretical sense and in our implementations of certain logics. This duality between logic and computer science has proven to be useful as a tool for not only verifying software correctness (as in the case of fast integer square root), but also in aiding mathematicians in proofs (as in the general case of proof assistants).

References

[1] Robert Constable. *First-Order Logic*. URL: http://www.cs.cornell.edu/courses/CS4860/2012fa/lec-13.pdf.

[2] Robert Constable. *Refinement Proofs and Evidence Construction*. URL: http://www.cs.cornell.edu/courses/CS4860/2012fa/lec-04.pdf.

[3] Oleg Kiselyov. *Lightweight Dependent-type Programming*. Jan. 2008. URL: http://okmij.org/ftp/Computation/lightweight-dependent-typing.html.

[4] Cristoph Kreitz. *Derivation of a Fast Integer Square Root Algorithm*. URL: http://www.cs.cornell.edu/courses/CS4860/2016fa/FastInteger.pdf.

[5] *Proof*. URL: https://rosettacode.org/wiki/Proof.

[6] Didier Remy. *Type systems for programming languages*. URL: http://gallium.inria.fr/~remy/mpri/cours2.pdf.

[7] Jeremy Yallop and Oleg Kiselyov. *First-class modules: hidden power and tantalizing promises*. URL: http://okmij.org/ftp/ML/first-class-modules/first-class-modules.pdf.