CS3110 Spring 2017 Lecture 9
Inductive proofs of specifications

Robert Constable

1 Lecture Plan

1. Repeating schedule of remaining five problem sets and prelim.
2. Comments on tautologies and the Coq logic.
3. OCaml specifications using types, limitation and comparisons.

2 Schedule of problem sets and in-class prelim

	Date for	Due Date
PS2	Out on Feb 23	March 2
Prelim	Tue. March 14, in class	
PS3	Out on Thur. March 2	March 16
PS4	Out on March 16	March 30
PS5	Out on April 10	April 24
PS6	Out on April 24	May 8 (day of last lecture)

3 Comments on logic in OCaml and Coq

In Lecture 7 we observed this important fact:

If a Boolean assignment shows that an OCaml polymorphic propositional logical specification is not a tautology (true in all assignments of truth values to the atomic types), then it is not programmable.

However, it is not the case that all Boolean tautologies are programmable. A simple example is $L\alpha|R(\alpha \rightarrow void)$ where *void* is the empty type. Another example of this phenomenon shows up in trying to program *Pierce's Law* using polymorphic OCaml types in its specification, e.g. $((\alpha \to \beta) \to \alpha) \to \alpha$. We will look deeper into this situation once we have discussed ways for thinking about the *logical meaning of types*.

These examples raise the question about whether there is a precise characterization of which propositional logical specifications can be seen as polymorphic program specifications. There is such a characterization, and we will gradually explore it as we go deeper into type theory. Meanwhile, we list the *truth tables* for the basic logical operators. When a truth table shows that an OCaml polymorphic propositional type is not a tautology, i.e. true in all possible assignments of truth values to the propositional variables, then we know that the specification is not programmable. We will see deeper into this phenomenon as we examine further the connection between types and logic. We will use the logic of the Coq proof assistant to help us in this task.

A	**B**	$\mathbf{A \to B}$	$\mathbf{\neg A \lor B}$	$\mathbf{A \land B}$	$\mathbf{A \lor B}$
T	T	T	T	T	T
T	F	F	F	F	T
F	T	T	T	F	T
F	F	T	T	F	F

As a simple exercise, show that *Pierce's law*, $((\alpha \to \beta) \to \alpha) \to \alpha$ is a tautology, and give an informal argument that it is *not programmable*. Do the same for the law of excluded middle, $A \lor \sim A$ expressed in polymorphic logic as $(L\alpha \mid R(\alpha \to void))$.

4 Proving that code meets a specification

We have been examining the problem of finding the integer square root of a natural number. We examined an inefficient program that does this, repeated below, but we have not proved that it meets the specification used to discuss it in the article by Professor Kreitz. We want to look at two ways of doing that. Prof. Kreitz uses a very interesting fact that some proofs that a programming problem is solvable give as a by-product a program to solve the problem. Another way to say this is: some proofs that a *specification is realizable* or that a *programming problem is solvable* implicitly construct a program. If the proofs are formal, say given in an *implemented logic* such as a proof assistant, then the logical system can build the program. We say that the program is *extracted* from the formal proof. This leads to the idea that “proofs are programs” [1].

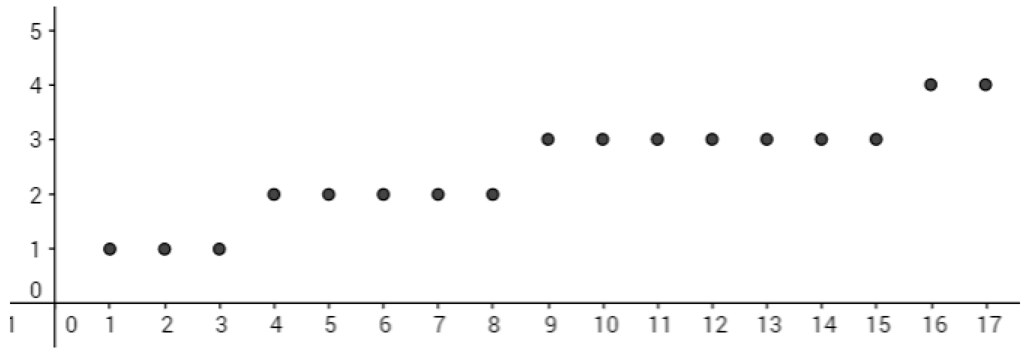


Figure 1: Graph of the integer square root

Here is OCaml code to solve the task of finding the integer square root.

```
# let rec sqrt n = if n = 0 then 0
          else let r = sqrt (n-1) in
          if ((r+1) * (r+1)) <= n then (r+1) else r ;;
val sqrt : int -> int = <fun>
```

The code calculates the “defining” graph.

```
# sqrt 17 ;;
- : int = 4

# sqrt 27878 ;;
- : int = 166
```

4.1 Logical specification of integer square root problem and proof of solvability

This section repeats the corresponding section of Lecture 8 without changes so that the material can be integrated into this lecture.

Here is the *logical specification* that Prof. Kreitz uses for the integer square root.

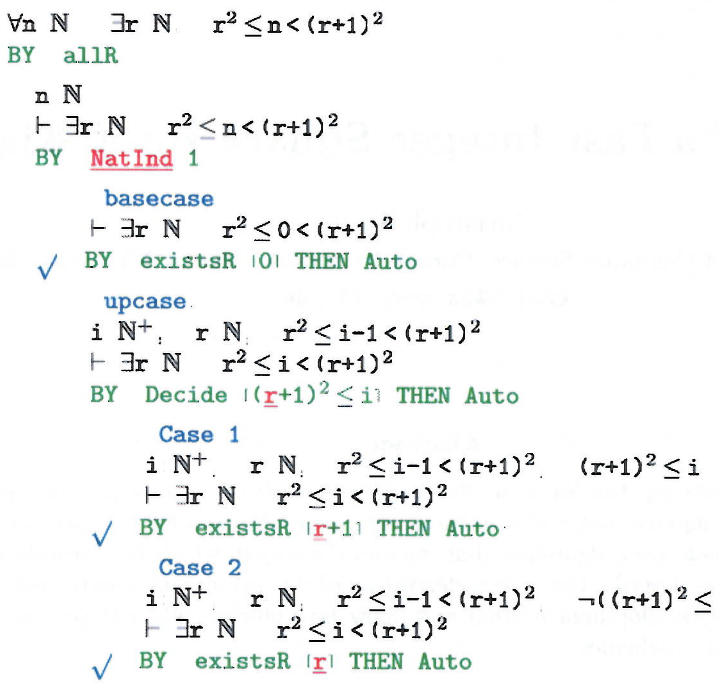
$$\forall n : \mathbb{N}.\exists r : \mathbb{N}.(r^2 \leq n \ \& \ (n < (r+1)^2)).$$

It would be a good personal exercise for you to write out a clear informal proof of this statement, one that will serve to remind you of the main ideas required to know this mathematical truth. Also notice that one reason we believe that the code is correct is that we can "experience instances of the algorithm" it implicitly defines and from that come to understand what it means.

Discovering or learning a previously known proof is another experience that reveals the generality of the idea. As we learn this proof or discover such a proof, we are grasping the generality of the algorithm embodied in the proof itself. That is an idea that a programming language can make very precise, the algorithmic content of an explanation. Explanations that have *algorithmic content* can be explored at the concrete level of *executing the idea on data.*

Here is OCaml code for the faster version of integer square root that Prof. Kreitz develops below.

```
# let rec sqrt2 n = if n=0 then 0
          else let r = sqrt2 (n/4)
          in if n < (2*r +1)*(2*r + 1) then 2*r
             else 2*r + 1 ;;
val sqrt2 : int -> int = <fun>
# sqrt2 278 ;;
- : int = 16
# sqrt2 278784 ;;
- : int = 528
# sqrt2 2787840 ;;
- : int = 1669
# sqrt2 2787845 ;;
- : int = 1669
# sqrt2 278784577;;
- : int = 16696
# sqrt2 27878457793;;
Characters 6-17:
  sqrt2 27878457793;;
        ^^^^^^^^^^^
Error: Integer literal exceeds the range of representable integers of type int
```



```
∀n ℕ   ∃r ℕ   r² ≤ n < (r+1)²
BY  allR

  n ℕ
  ⊢ ∃r ℕ   r² ≤ n < (r+1)²
  BY  NatInd 1

      basecase
      ⊢ ∃r ℕ   r² ≤ 0 < (r+1)²
  √   BY  existsR ⌈0⌉ THEN Auto

      upcase
      i ℕ+,  r ℕ,  r² ≤ i-1 < (r+1)²
      ⊢ ∃r ℕ   r² ≤ i < (r+1)²
      BY  Decide ⌈(r+1)² ≤ i⌉ THEN Auto

          Case 1
          i ℕ+,  r ℕ,  r² ≤ i-1 < (r+1)²,   (r+1)² ≤ i
          ⊢ ∃r ℕ   r² ≤ i < (r+1)²
      √   BY  existsR ⌈r+1⌉ THEN Auto

          Case 2
          i ℕ+,  r ℕ,  r² ≤ i-1 < (r+1)²,   ¬((r+1)² ≤ i)
          ⊢ ∃r ℕ   r² ≤ i < (r+1)²
      √   BY  existsR ⌈r⌉ THEN Auto
```

Figure 1: Proof of the Specification Theorem using Standard Induction.

From the proof of this specification, Kreitz creates an SML program which also executes as the OCaml program given above. An alternative approach is to prove the following theorem by induction on n. We show below his Nuprl proof of this theorem. It is easy to read this formal proof even without knowing the details of the Nuprl logic. A good exercise for interested students is to write an informal proof in a "classroom style" using intuitive logic that also has computational meaning as is the case for the OCaml and Coq logical statements. We will explore in more detail as the course progresses how a logic with computational meaning keeps track of our natural mathematical intuitions.

$$\forall n:\mathbb{N}.(sqrt(n)^2 \leq n \ \& \ (n < (sqrt(n)+1)^2).$$

To do this we need to unfold the function definition and use induction on n.

4.2 A Much Faster Program

Here is the derivation of a much faster integer square root program also by proof. It is not expected that you follow this. Below is the algorithm, and you can test its superiority of performance. The cost is much greater logical complexity.

```
∀n ℕ   ∃r ℕ   r^2≤n<(r+1)^2
BY  allR  THEN  Assert ⌈∀j ℕ   (n-j)^2≤n  ⇒  ∃r≥n-j  r^2≤n<(r+1)^2⌉

     Assertion
   n ℕ;  j ℕ;  (n-j)^2≤n
   ⊢ ∃r≥n-j  r^2≤n<(r+1)^2
   BY  NatInd 2
        basecase
      n ℕ;  (n-0)^2≤n
      ⊢ ∃r≥n-0  r^2≤n<(r+1)^2
   ✓  BY  existsR ⌈n⌉ THEN Auto
        upcase
      n ℕ;  j ℕ+;  (n-(j-1))^2≤n ⇒ ∃r≥n-(j-1)  r^2≤n<(r+1)^2;  (n-j)^2≤n
      ⊢ ∃r≥n-j  r^2≤n<(r+1)^2
      BY  Decide ⌈n<(n-j+1)^2⌉ THEN Auto
           Case 1
         n ℕ;  j ℕ+;  (n-(j-1))^2≤n ⇒ ∃r≥n-(j-1)  r^2≤n<(r+1)^2;  (n-j)^2≤n;
         n<(n-j+1)^2
         ⊢ ∃r≥n-j  r^2≤n<(r+1)^2
      ✓  BY  existsR ⌈n-j⌉ THEN Auto
           Case 2
         n ℕ;  j ℕ+;  (n-(j-1))^2≤n ⇒ ∃r≥n-(j-1)  r^2≤n<(r+1)^2;  (n-j)^2≤n
         ¬(n<(n-j+1)^2)
         ⊢ ∃r≥n-j  r^2≤n<(r+1)^2
         BY  impL 3 THEN Auto

           n ℕ;  j ℕ+;  (n-(j-1))^2≤n ⇒ ∃r≥n-(j-1)  r^2≤n<(r+1)^2;  (n-j)^2≤n
           ¬(n<(n-j+1)^2)
           ⊢ ∃r≥n-j  r^2≤n<(r+1)^2
        ✓  BY  existsR ⌈r⌉ THEN Auto
     Main
   n ℕ;  ∀j ℕ   (n-j)^2≤n  ⇒  ∃r≥n-j  r^2≤n<(r+1)^2
   ⊢ ∃r ℕ  r^2≤n<(r+1)^2
   BY  allL 2 ⌈n⌉ THEN Auto

      n ℕ;  r ℕ;  r≥n-n;  r^2≤n<(r+1)^2
      ⊢ ∃r ℕ  r^2≤n<(r+1)^2
✓     BY  existsR ⌈r⌉ THEN Auto
```

4.3 The Computational Content of Mathematical Induction

How does "extraction of code from a proof" work? Consider the meaning of induction:

$$(P(0) \ \& \ \forall n : \mathbb{N}.(P(n) \Rightarrow P(n+1)) \Rightarrow \forall n : \mathbb{N}.P(n).$$

What is the computational meaning of this statement of mathematical induction?

$$\begin{array}{l} p_0 \in P(0) \\ f : \quad P(n) \rightarrow P(n+1) \end{array}$$

Hence,

$$\begin{array}{lll} f(p_0) \in P(1), & f(f(p_0) \in P(2)), & \ldots \\ f^1(p_0) \in P(1), & f^2(f(p_0) \in P(2)), & \boxed{f^n(p_0) \in P(n)} \end{array}$$

We think of p_0 as evidence for the base case, $P(0)$. The function f maps this evidence for the proposition $P(0)$ into the proposition $P(1)$. We can think of $P(0)$ and $P(1)$ as a specific OCaml types or specific Coq types. We see by the typing of the function f that it maps elements of $P(0)$ into $P(1)$, so $f(p_0)$ is in the type $P(1)$. Then by applying f again, we have $f(f(p_0))$ in the type $P(2)$. By iterating n times the function f, we have evidence in type $P(n)$, and we can do this for any natural number n. This is the computational meaning of this basic form of mathematical induction. We get this meaning from the interpretation of the Coq dependent type $x : nat \rightarrow P(x)$, written in Coq as

```
forall (x:nat), P(x).
```

5 The Coq type system for logic.

The Coq proof assistant uses a rich type theory. We could call it the *Coq type theory*. It is described in the on-line textbook *Software Foundations* [4]. Since I mention Coq frequently and even discuss its programming language, CoqPL, it is worthwhile to tell you a bit more about that type theory. It illustrates the future of types in programming languages. All we need to cover to make this a very useful part of the course are the types needed to define First-Order logic. These are very easy to understand given what we have already said about OCaml types and logic. Here are some examples directly from a short introduction to Coq.

There are full textbooks on Coq [2, 3, 4] and a web cite devoted to it. We will look at only a small subset of the logical rules of Coq. There is a nice short unpublished paper by Bart Jacobs on the web, entitled *The Essence of Coq as a Formal System.*

In Coq *dependent function type* is primitive. Its elements are functions, and the typing has this property.

```
forall (x:T), P(x) defines functions taking inputs t from
type T into vales p(x) in type P(x).
T -> P defines functions taking inputs of type T into outputs of type P,
no dependence.
```

The non-dependent case is the same as the OCaml function type $ty1 \rightarrow ty2$.

The logical operators are defined *inductively.* OCaml would call such types recursive if it had them. The logical operators are defined on these inductive types rather than using polymorphic types. Here is the definition of And and Or. We can see the similarity to the OCaml "or" by the use of the vertical bar to separate the cases.

```
Inductive And(P Q: Type) := and_(p: P)(q: Q).
Inductive Or(P Q: Type) := or_left(p: P) | or_right(q: Q).
```

The definition of And tells us that it is an ordered pair, similar to the OCaml type (ty1 * ty2). The elements of the type are members p and q of the types P and Q respectively.

The definition of Or is essentially an OCaml *variant* with the tags *left* and *right.* The elements of the types are denoted p and q respectively.

The next type is an example of a defined *dependent type.* This kind of type is what gives Coq the power to express logic.

```
Inductive Exists(T: Type)(P: T -> Type) := exists_(t: T)(p: P t).
```

The definition tells us that T is any Type, and P is an element of the space of functions from elements t of type T into into ordered pairs (t:T) (p: P t). This captures the essence of a *dependent type* and defines the existential quantifier, as used in the theorem about square roots. Note, P is the function from type T into Type, and t is an element of the type T. So (p:P t) says that p is an element of the type that depends on the value of the function P at element t in type T. This type construct is not available in OCaml, but Coq shows that we can compute with it and reason about it using more sophisticated methods than are available in OCaml.

```
Inductive Equals(A: Type)(a: A): A -> Type := equals_: Equals A a a.
Inductive False :=.
Inductive True := true_.
```

These definitions give of the empty type False, which can be defined in OCaml as well, and the type True which has just one element in it, true. Both of these types are available in OCaml. We used them to define the negation operator, say $\alpha \ \rightarrow \ void$ for the polymorphic negation.

```
Definition Not(P: Type) := P -> False.

Definition modus_ponens: forall (P Q: Type), P -> (P -> Q) -> Q :=
fun (P Q: Type) => fun (p: P) => fun (pq: P -> Q) => pq p.



Definition distr: forall (P Q R: Type), And P (Or Q R) ->
Or (And P Q) (And P R) :=
fun (P Q R: Type) =>
fun (pqr: And P (Or Q R)) =>
match pqr with
 and_ p qr =>
 match qr with
 or_left q =>
 or_left (And P Q) (And P R) (and_ P Q p q)
 | or_right r =>
 or_right (And P Q) (And P R) (and_ P R p r)
 end
end.
```

```
=======
Inductive Le: nat -> nat -> Type :=
le_refl: forall (n: nat), Le n n
| le_succ: forall (m n: nat), Le m n -> Le m (succ n).
The free variables and the premises of a rule correspond
to the parameters.

Fixpoint Le_succ_succ(m n: nat)(H: Le m n) {struct H}: Le (succ m) (succ n) :=
match H in Le M N return Le (succ M) (succ N) with
 le_refl n0 =>
 (* Goal: Le (succ n0) (succ n0) *)
 le_refl (succ n0)
 | le_succ m0 n0 H0 =>
 (* Goal: Le (succ m0) (succ (succ n0)) *)
le_succ (succ m0) (succ n0) (Le_succ_succ m0 n0 H0)
end.
```

References

[1] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions of Programming Language Systems*, 7(1):53–71, 1985.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[3] Adam Chlipala. An introduction to programming and proving with dependent types in Coq. *Journal of Formalized Reasoning (JFR)*, 3(2):1–93, 2010.

[4] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjberg, and Brent Yorgey. *Software Foundations*. Electronic, 2013.