Lecture Notes on Induction and Recursion

15-317: Constructive Logic
Frank Pfenning*

Lecture 8
February 6, 2020

1 Introduction

At this point in the course we have developed a good formal understanding how *propositional intuitionistic logic* is connected to computation: propositions are types, proofs are programs, and proof reduction is computation. We have also introduced the universal and existential quantifiers, but, of course, without some data types such as natural numbers, integers, lists, trees, etc. we cannot reason about or compute with such data.

Accompanying the formal development in the Heyting arithmetic lecture, we devote today's lecture to develop some *informal* understanding of how to reason constructively with such types, and what the programs look like that correspond to such proofs.

We reason about data using *induction*. As will be explicit in the primitive recursion studied in the Heyting arithmetic lecture, the computational content of such proofs will be functions defined by *recursion*. The goal of today's lecture is to see in action the principle that induction is recursion.

We now go through several examples, presenting proofs in the usual mathematical style and their computational contents as recursive functions. The explicit discussion of inductive arguments here is helpful for later lectures that perform induction on structures other than natural numbers.

*Edits by André Platzer

2 Example: Integer Square Root

At first, we might think of specifying the square root with the theorem

$$\forall x{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y^2 = x$$

This is a great place to start: if we could prove that, the function extracted would take an integer x as an argument and return a witness $y = \sqrt{x}$ and a proof that indeed $y^2 = x$. Unfortunately, this is not a theorem because not every natural number is a square. So we have to allow for rounding down or up. The following specification

$$\forall x{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y^2 \leq x \wedge x < (y+1)^2$$

does in fact round down. For example, for $x = 10$ the witness $y = 3$ will have the property $3^2 = 9 \leq 10 \wedge 10 < 16 = (3+1)^2$.

Now how do we prove this? The natural attempt is to prove it by *mathematical induction* on x. This means to prove it for $x = 0$, then assume the theorem for x and prove it for $x + 1$. This form of induction is also called *weak induction* because the induction hypothesis only assumes the statement for x. We will encounter *complete induction*, alias *strong induction* later in this lecture, where the induction hypothesis assumes it for $0, 1, 2, \ldots, x$.

Theorem 1 (Integer Square Root) $\forall x{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y^2 \leq x \wedge x < (y+1)^2$

Proof: By mathematical induction on x.

Base: $x = 0$. Then we pick the witness $y = 0$ because $0^2 = 0 \leq 0 \wedge 0 < 1 = (0+1)^2$.

IH: Assume $\exists y{:}\mathsf{nat}.\, y^2 \leq x \wedge x < (y+1)^2$.

Step: We have to prove $\exists y{:}\mathsf{nat}.\, y^2 \leq x+1 \wedge x+1 < (y+1)^2$.

$\exists y{:}\mathsf{nat}.\, y^2 \leq x \wedge x < (y+1)^2$ by IH
$a^2 \leq x \wedge x < (a+1)^2$ for some a, by $\exists E$

Now we distinguish two cases: $x + 1$ is still less than $(a+1)^2$ or it is greater than or equal to $(a+1)^2$. When given x and a, we can decide this inequality so the case distinction is constructively permissible.

Case: $x + 1 < (a+1)^2$. Then we pick the witness $y = a$ because

$a^2 \leq x + 1$	since $a^2 \leq x$
$x + 1 < (a + 1)^2$	this case

Case: $x + 1 \geq (a + 1)^2$. Then we pick the witness $y = a + 1$ because

$x + 1 = (a + 1)^2$	by case since $x < (a + 1)^2$
$(a + 1)^2 \leq x + 1$	from previous line
$x + 1 < (a + 2)^2$	since $x + 1 = (a + 1)^2 < (a + 2)^2$

□

For clarity, we call out the induction hypothesis (IH) explicitly in the above proof, even if this is not necessary in such simple cases.

What is the computational content of this proof? It is a recursive function, where an appeal to the induction hypothesis corresponds to a recursive call. When a witness for an existential is exhibited in the proof, we return this witness. We ignore here the attendant proof that the returned witness is in fact correct, so the function below will have only a portion of all of the information of the proof.

```
fun isqrt 0 = 0
  | isqrt (x+1) =
    let val a = isqrt x
     in if x+1 < (a+1)*(a+1)
        then a
        else a+1
    end
```

This does not literally work in Standard ML because we cannot pattern-match against $x + 1$, so we have to rewrite this slightly. Also, we are using the built-in type `int` instead of nat.

```
fun isqrt 0 = 0
  | isqrt x = (* x > 0 *)
    let val a = isqrt (x-1)
     in if x < (a+1)*(a+1)
        then a
        else a+1
    end
```

This algorithm is not what one would think of as an implementation of a square root. To compute the integer square root of x it runs through all the numbers up to x, essentially adding 1 every time we hit the next square (the

`else` case of the conditional). Computationally this is expensive in time. It is also expensive in space because the function is not tail recursive.

A different proof of the same theorem corresponds to a more efficient function (see Section 5).

3 Example: Exponentiation

We define the mathematical function of exponentiation on natural numbers by $b^0 = 1$ and $b^{n+1} = b \times b^n$ for $n > 0$. We can prove a theorem that natural numbers are closed under exponentiation, so there is an implementation.

Theorem 2 (Exponential closure) $\forall b{:}\mathsf{nat}.\, \forall n{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y = b^n$

Proof: By mathematical induction on n.

Base: $n = 0$. Then pick $y = 1$ because $1 = b^0$.

IH: Assume $\exists y{:}\mathsf{nat}.\, y = b^n$.

Step: We have to show that $\exists y{:}\mathsf{nat}.\, y = b^{n+1}$.

$\exists y{:}\mathsf{nat}.\, y = b^n$	by IH
$a = b^n$	for some a, by $\exists E$
Pick $y = b \times a = b \times b^n = b^{n+1}$	by def
$\exists y{:}\mathsf{nat}.\, y = b^{n+1}$	by $\exists I$

□

The extracted function corresponding to this proof is entirely straightforward. We write it directly in Standard ML form.

```
fun exp b 0 = 1
  | exp b n = (* n > 0 *)
    let val a = exp b (n-1)
    in b * a end
```

Again, this function is not tail-recursive since we take the result a returned by the recursive call and still multiply it by b instead of returning directly.

To obtain a tail-recursive version, we need to find a different proof of the same specification! From our experience in functional programming we know that we need to carry along an accumulator for the result in an auxiliary function. Such an auxiliary function corresponds to a *lemma* on

the mathematical side. The accumulator c is an additional argument, so the lemma has one additional quantifier.

$$\forall b{:}\mathsf{nat}.\, \forall n{:}\mathsf{nat}.\, \forall c{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, ???$$

The tricky question is what does the lemma express? Because we *multiply* the accumulator by the base b at every recursive call, the generalization is also stated multiplicatively. In general, though, coming up with an appropriate generalization of the theorem is a creative and difficult task.

Lemma 3 $\forall b{:}\mathsf{nat}.\, \forall n{:}\mathsf{nat}.\, \forall c{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y = c \times b^n$

Proof: By mathematical induction on n.

Base: $n = 0$. Then pick $y = c$ because $y = c = c \times b^0$.

IH: Assume $\forall c{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y = c \times b^n$.

Step: We have to show $\forall c{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y = c \times b^{n+1}$, that is for the sake of using $\forall I$, for an arbitrary c_1 we have to show $\exists y{:}\mathsf{nat}.\, y = c_1 \times b^{n+1}$.

$\forall c{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y = c \times b^n$	by IH
$\exists y{:}\mathsf{nat}.\, y = (c_1 \times b) \times b^n$	using $c = c_1 \times b$, by $\forall E$
$a = (c_1 \times b) \times b^n = c_1 \times b^{n+1}$	for some a, by $\exists E$
$\exists y{:}\mathsf{nat}.\, y = c_1 \times b^{n+1}$	picking $y = a$ and $\exists I$
$\forall c{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y = c \times b^{n+1}$	by $\forall I$

□

Now a second proof for Theorem 2 no longer requires a proof by induction, directly calling on the above lemma instead.

Theorem 4 (Exponential closure) $\forall b{:}\mathsf{nat}.\, \forall n{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y = b^n$

Proof: From the preceding lemma by using $c = 1$ since $1 \times b^n = b^n$ □

The computational content is now two functions, `exp2_aux` corresponding to the lemma, and `exp2` for the theorem.

```
fun exp2_aux b 0 c = c
  | exp2_aux b n c = (* n > 0 *)
    let val a = exp2_aux b (n-1) (c*b)
    in a end

fun exp2 b n = exp2_aux b n 1
```

The auxiliary function is now tail recursive because the witness a for y in the proof is just the witness from the appeal to the induction hypothesis. We can shorten the program slightly to make this more immediate:

```
fun exp2_aux b 0 c = c
  | exp2_aux b n c = (* n > 0 *)
    exp2_aux b (n-1) (c*b)

fun exp2 b n = exp2_aux b n 1
```

There is still a disadvantage to this implementation in that it carries out n multiplications. There is a yet more efficient implementation which carries out only $O(\mathsf{log}(n))$ multiplications by taking advantage of the observation that $b^{2n} = (b^2)^n$. That is, we can calculate b^{2n} by instead calculating b_2^n for a different base b_2. The corresponding inductive proof has a somewhat different structure from the proofs so far, because the step foreshadowed above reduces computing b^{2n} to computing b_2^n, which means given an (even) $n > 0$, we have to apply the induction hypothesis to $n/2$. A similar reasoning will apply for odd numbers. Fortunately, $n/2 < n$ for $n > 0$, so the principle of *complete induction* allows this pattern of reasoning.

The statement of the lemma itself remains unchanged, only its proof.

Lemma 5 $\forall b{:}\mathsf{nat}.\, \forall n{:}\mathsf{nat}.\, \forall c{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y = c \times b^n$

Proof: By *complete* induction on n.

Base: $n = 0$. Then, as before, pick $y = c$.

IH: $\forall b{:}\mathsf{nat}.\, \forall k{:}\mathsf{nat}.\, (k < n \supset \forall c{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y = c \times b^k)$

Step: $n > 0$. Then we distinguish two cases: n is even or n is odd. Presumably this can be decided in our theory of natural numbers.

Subcase: $n = 2k$ for some $k < n$. We have to prove $\exists y{:}\mathsf{nat}.\, y = c_1 \times b_1^n$ for some arbitrary c_1 and b_1 (by $\forall I$).

$\exists y{:}\mathsf{nat}.\, y = c_1 \times (b_1 \times b_1)^k$
by IH for $b = b_1 \times b_1$, $k < n$, and $c = c_1$
$a = c_1 \times (b_1 \times b_1)^k$ for some a, by $\exists E$
Pick $y = a = c_1 \times (b_1^2)^k = c_1 \times b_1^{2k} = c_1 \times b_1^n$
$\exists y{:}\mathsf{nat}.\, y = c_1 \times b_1^n$ by $\exists I$

Subcase: $n = 2k + 1$ for some $k < n$. We have to prove $\exists y{:}\mathsf{nat}.\, y = c_1 \times b_1^n$ for some arbitrary c_1 and b_1 (by $\forall I$).

$\exists y{:}\mathsf{nat}.\, y = (c_1 \times b_1) \times (b_1 \times b_1)^k$
by IH for $b = b_1 \times b_1$, $k < n$, and $c = c_1 \times b$
$a = (c_1 \times b_1) \times (b_1 \times b_1)^k$ for some a, by $\exists E$
Pick $y = a = (c_1 \times b_1) \times (b_1^2)^k = c_1 \times b_1^{2k+1} = c_1 \times b_1^n$
$\exists y{:}\mathsf{nat}.\, y = c_1 \times b_1^n$ by $\exists I$

□

This proof uses complete induction, since while proving the case $n > 0$ its induction hypothesis assumes the theorem already for all $k < 0$. Note how useful it is to explicitly clarify IH (although a different quantifier ordering $\forall k(k < n \supset \forall b \forall c \exists y \ldots)$ would also have worked). During the induction step to prove it for $n > 0$, it is crucial that IH is only used for k that are indeed strictly smaller than n, as explicitly indicated in the above proof.

The proof of the theorem does not change, but the extracted function now calls upon a different version of the auxiliary function because we have given a different proof. Note that it is still tail recursive, and we were able to put the accumulator to good use in the case of an odd number.

```
fun exp3_aux b 0 c = c
  | exp3_aux b n c = (* n > 0 *)
    if n mod 2 = 0
    then exp3_aux (b*b) (n div 2) c
    else exp3_aux (b*b) ((n-1) div 2) (c*b)

fun exp3 b n = exp3_aux b n 1
```

4 Example: Warshall's Algorithm for Graph Reachability

This example is even less formal and sketchier than the previous section. It concerns induction about structures other than natural numbers, particularly lists.

To start with, how do we even specify graph reachability? We assume we are given a graph G via a type of nodes N and a collection of edges E connecting nodes. We consider the graph G fixed, so we won't repeatedly mention it in every proposition in the rest of this section.

We also have a notion of a *path* through a graph, following the set of edges. We write $\mathsf{path}(x, y)$ when there is a path p in the graph G connecting

x and y.[1] We can then specify graph reachability as

$$\forall x{:}N.\, \forall y{:}N.\, \mathsf{path}(x, y) \vee \neg\mathsf{path}(x, y)$$

Classically, this is a triviality, because it has the form $\forall x.\, \forall y.\, A \vee \neg A$ which is classically obviously true by the law of excluded middle. Constructively, a proof will have to show whether, given an x and y, there is a path connecting them or not. In addition, the proof of $\mathsf{path}(x, y)$ should exhibit such a path, while a proof of $\neg\mathsf{path}(x, y)$ should derive a contradiction from the assumption that there is one. As in the previous examples, we will ignore some of the computational of the content of the proof, focusing on returning the boolean true if there is a path and false if there is none. In a later lecture we may see how we can systematically and formally hide some of the computational contents of proofs while keeping other information.

Now the statement above could be proved in a number of ways. For example, we might proceed by induction over the length of the potential path, or by induction over the number of unvisited nodes in the graph, each giving rise to different implementations. Here, will use a different idea: consider a fixed enumeration of the vertices in the graph (a list of vertices) and proceed by induction over the structure of this list. Given some list V of vertices, we write $\mathsf{path}_V(x, y)$ if there is a path p connecting x and y using only vertices from V as *interior* nodes. That is, the path p must start with x, finish with y, and all other vertices on p must be in V.

Now we mildly generalize our statement so we can prove it inductively:

$$\forall V{:}N\ \mathsf{list}.\, \forall x{:}N.\, \forall y{:}N.\, \mathsf{path}_V(x, y) \vee \neg\mathsf{path}_V(x, y)$$

Our original theorem follows easily by picking $V = N$, because then the path is allowed to contain all vertices.

Theorem 6 $\forall V{:}N\ \mathsf{list}.\, \forall x{:}N.\, \forall y{:}N.\, \mathsf{path}_V(x, y) \vee \neg\mathsf{path}_V(x, y)$

Proof: By induction on the structure of V.

Base: $V = \mathsf{nil}$, the empty list. Then $\mathsf{path}_{\mathsf{nil}}(x, y)$ exactly if there is a direct edge from x to y because no interior nodes are allowed.

IH: Assume $\forall x{:}N.\, \forall y{:}N.\, \mathsf{path}_W(x, y) \vee \neg\mathsf{path}_W(x, y)$.

Step: Consider $V = z :: W$ for some vertex z with remaining list W. To show $\mathsf{path}_V(x, y) \vee \neg\mathsf{path}_V(x, y)$ we use IH and distinguish two cases:

[1] Other representations are possible that make the path explicit, but that is not necessary to understand the basic idea.

Case: $\mathsf{path}_W(x, y)$. Then also $\mathsf{path}_{z::W}(x, y)$ since we do not even need to use the additional vertex z to find a path from x to y in V.

Case: $\neg\mathsf{path}_W(x, y)$. Now we use IH again on W, but this time on x and z, so $\mathsf{path}_W(x, z) \vee \neg\mathsf{path}_W(x, z)$. We once again distinguish two cases:

Subcase: $\mathsf{path}_W(x, z)$. Now we use IH a third time to see if there is a path from z to y using only W: $\mathsf{path}_W(z, y) \vee \neg\mathsf{path}_W(z, y)$. Again, we distinguish these cases:

Subsubcase: $\mathsf{path}_W(z, y)$. Since also $\mathsf{path}_W(x, z)$ we can concatenate these two paths to obtain $\mathsf{path}_{z::W}(x, y)$. Now z has to be added, because it is on the interior of the path that goes from x to y, but that's fine since $V = z :: W$.

Subsubcase: $\neg\mathsf{path}_W(z, y)$. Then $\neg\mathsf{path}_{z::W}(x, y)$: if there is no path from x to y entirely over W, allowing z does not help when there is no path from z to y over W.

Subcase: $\neg\mathsf{path}_W(x, z)$. Then also $\neg\mathsf{path}_{z::W}(x, y)$ because, similar to the previous subsubcase, adding z does not help when it has no path from x.

□

In writing out the computational content we replace the if-then-else constructs with corresponding uses of `orelse` and `andalso`, for the sake of brevity and readability. So we expand as follows:

```
b orelse c  == if b then true else c
b andalso c == if b then c else false
```

The code then turns out to be exceedingly compact.

```
fun warshall edge (nil) x y = edge x y
  | warshall edge (z::W) x y =
    warshall edge W x y orelse
    (warshall edge W x z andalso warshall edge W z y)
```

The compiler tells us

```
val warshall = fn : ('a -> 'a -> bool) -> 'a list -> 'a -> 'a -> bool
```

which means that this works for any type of vertices `'a` as long as we have a function representing the edge relation.

This does not quite capture Warshall's algorithm yet: to get the right complexity we need to represent the *function* `warshall edge V` as a two-dimensional *boolean array* indexed by x and y, and for each z run through all pairs x and y to fill it with booleans.[2] This transformation can be carried out informally, or rigorously as shown in [Pfe90]. Of course, for small graphs the SML source code including a small example works just fine.

If now look back at the proof we can see that it actually contains enough information to also extract the path when there is one. If a path is represented by a list of vertices, the result of of an extracted function which return the path if there is one will have type

```
val warshall2 : ('a -> 'a -> bool) -> 'a list
                -> 'a -> 'a -> 'a list option
```

which is implemented by the following function

```
fun warshall2 edge (nil) x y =
    if edge x y then SOME [x,y] else NONE
  | warshall2 edge (z::W) x y =
    case warshall2 edge W x y
     of SOME p => SOME p
      | NONE => (case warshall2 edge W x z
                  of SOME q => (case warshall2 edge W z y
                                 of SOME r => SOME (q @ tl r)
                                  | NONE => NONE)
                   | NONE => NONE)
```

5 Bonus Example: Tail-Recursive Integer Square Root

We did not have time to discuss this in lecture, but we may consider how we can make the integer square root example more efficient. In particular, we should see if we can make it tail recursive. The key idea is the same that might occur to anyone when ask the implement integer square root: we add a counter c which we increment until c^2 exceeds x. The problem now becomes how to state the theorem and how to find a corresponding proof.

The counter needs to become a new argument of an auxiliary function, so in the lemma there will be additional quantifier. All the quantifiers range over natural numbers, so we omit the type. We try

$$\forall x.\, \forall c.\, c^2 \leq x \supset \exists y.\, y^2 \leq x \wedge x < (y+1)^2$$

[2] See, for example, the Wikipedia page on then Floyd-Warshall algorithm

At first sight this might look wrong since c does not occur in the scope of the quantifier on y, but the information about c may help us to construct such a y anyway.

In the proof, what becomes smaller as we count c upward? Clearly, it is the distance between c and x or, more precisely, the distance between c^2 and x. When this distance becomes 0, we terminate the recursion. This leads to the following lemma, proof, and theorem:

Lemma 7 $\forall x.\, \forall c.\, c^2 \leq x \supset \exists y.\, y^2 \leq x \wedge x \leq (y+1)^2$

Proof: By complete induction on $x - c^2$.

We have to prove, for an arbitrary x and c with $c^2 \leq x$ that $\exists y.\, y^2 \leq x \wedge x < (y+1)^2$. We distinguish two cases:

Case: $x < (c+1)^2$. Then we can pick $y = c$ since $c^2 \leq x$ (by assumption) and $x < (c+1)^2$ (this case).

Case: $(c+1)^2 \leq x$. Then we can apply the induction hypothesis, precisely because $(c+1)^2 \leq x$ and $x - (c+1)^2 = x - c^2 - 2c - 1 < x - c^2$.

$\exists y.\, y^2 \leq x \wedge x < (y+1)^2$ by ind. hyp.

But this is exactly what we needed to prove.

□

Theorem 8 $\forall x.\, \exists y.\, y^2 \leq x \wedge x \leq (y+1)^2$

Proof: Use the lemma for $c = 0$, which satisfies the requirement because $c^2 = 0 \leq x$ for any x. □

The extracted function then looks as follows:

```
fun isqrt2_aux x c = (* c*c <= x *)
    if x < (c+1)*(c+1)
    then c
    else (* (c+1)*(c+1) <= x *)
        isqrt2_aux x (c+1)

fun isqrt2 x = isqrt2_aux x 0
```

We can take the analysis a bit further and try to ask: what does an induction over $x - c^2$ actually mean? One possible interpretation is to add another variable d and constrain it to be *equal* to $x - c^2$ so we apply complete induction on this variable.

Lemma 9 $\forall x.\, \forall d.\, \forall c.\, c^2 \leq x \wedge d = x - c^2 \supset \exists y.\, y^2 \leq x \wedge x \leq (y+1)^2$

Proof: Proof is by complete induction on d. Assume we have an x, d, and c such that $c^2 \leq x$ and $d = x - c^2$.

Previously, we distinguished cases based on whether $x < (c+1)^2$ or not. But we can rephrase test in terms of d: $x < c^2 + 2c + 1$ iff $x - c^2 < 2c + 1$ iff $d < 2c + 1$.

Case: $d < 2c + 1$. Then $y = c$ satisfies the theorem because also $c^2 \leq x$ by assumption and $x < (c+1)^2$ in this case.

Case: $d \geq 2c + 1$. Then $(c+1)^2 \leq x$ and $0 \leq x - (c+1)^2 = x - c^2 - 2c - 1 = d - 2c - 1 < d$ so we can apply the induction hypothesis on $d - 2c - 1$ and $c + 1$ to obtain some y such that $y^2 \leq x \wedge x \leq (y+1)^2$.

□

Theorem 10 $\forall x.\, \exists y.\, y^2 \leq x \wedge x \leq (y+1)^2$

Proof: Use the lemma for $d = x$ and $c = 0$, which satisfy the requirements because $c^2 = 0 \leq x$ for any x and $x - c^2 = x$ □

The code, leaving out any extraneous proof information:

```
fun isqrt3_aux x d c =
    if d < 2*c+1
    then c
    else isqrt3_aux x (d-2*c-1) (c+1)

fun isqrt3 x = isqrt3_aux x x 0
```

Note that this remaind tail recursive and avoids the potentially "costly" multiplication $(c+1) \times (c+1)$ on every recursive call from the previous version.

References

[Pfe90] Frank Pfenning. Program development through proof transformation. *Contemporary Mathematics*, 106:251–262, 1990.