ConstLog: Constructive Logic

Lecture Notes on Heyting Arithmetic and Recursion

Frank Pfenning André Platzer

Carnegie Mellon University || Karlsruhe Institute of Technology

Lecture 7

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. We reason about data using *induction*.

In this lecture we discuss the data type of natural numbers. They serve as a prototype for a variety of inductively defined data types, such as lists or trees, because they are the easiest case of a data type generated by a constructor taking the data type itself as an argument (successor) from an atomic constructor (0). Together with quantification from Lecture 6, this allow us to reason constructively about natural numbers and extract corresponding functions from constructive proofs. The constructive system for reasoning logically about natural numbers is called *intuitionistic arithmetic* or *Heyting arithmetic* [Hey56]. The *classical* version of the same principles is called *Peano arithmetic* [Pea89]. Both of these are usually introduced *axiomatically* rather than as an extension of natural deduction as we do here.

The lecture notes also include several examples illustrating the correspondence of inductive proofs and recursive functions.

2 Induction

As usual, we think of the type of natural numbers as defined by its introduction form. Note, however, that nat is a *type* rather than a proposition. It is possible to completely unify these concepts to arrive at *type theory*, something we might explore later in this

course. For now, we just specify cases for the typing judgment $t : \tau$, read term t has type τ, that was introduced in Lecture 6, but for which we have seen no specific instances yet. We distinguish this from $M : A$ which has the same syntax, but relates a proof term to a proposition instead of a term to a type. We now fill the typing judgment on terms with life as $t : \mathsf{nat}$ compared to its abstract treatment in the quantification lecture.

There are two introduction rules, one for zero and one for successor.

$$\frac{}{\mathsf{0} : \mathsf{nat}}\ \mathsf{nat}I_0 \qquad \frac{n : \mathsf{nat}}{\mathsf{s}\,n : \mathsf{nat}}\ \mathsf{nat}I_s$$

Intuitively, these rules express that $\mathsf{0}$ is a natural number ($\mathsf{nat}I_0$) and that the successor $\mathsf{s}\,n$ is a natural number if n itself is a natural number. This definition has a different character from previous definitions. We defined the meaning of $A \wedge B$ *true* from the meaning of A *true* and of B *true*, all of which are propositions. It is even different from the proof term assignment rules where, e.g., we defined $\langle M, N\rangle : A \wedge B$ in terms of $M : A$ and $N : B$. In each case, the proposition itself is decomposed into its parts.

The types in the conclusion and premise of the $\mathsf{nat}I_s$ rules stay the same, namely nat. Fortunately, the *term* n in the premise is a part of the term $\mathsf{s}\,n$ in the conclusion, so the definition is not circular, because the judgment in the premise is still smaller than the judgment in the conclusion even if this is not because of the types. In (verificationist) constructive logic, truth is defined by the introduction rules. The resulting implicit principle, that nothing is true unless the introduction rules prove it to be true, is of deep significance here. Nothing else is a natural number, except the objects constructed via $\mathsf{nat}I_s$ from $\mathsf{nat}I_0$. The rational number $\frac{7}{4}$ cannot sneak in claiming to be a natural number (which, by $\mathsf{nat}I_s$ would also make its successor $\frac{11}{4}$ claim to be natural).

But what should the elimination rule be? We cannot decompose the proposition into its parts, as it has no parts, so we decompose the term instead. Natural numbers have two introduction rules just like disjunctions. Their elimination rule, thus, also proceeds by cases, accounting for the possibility that a given n of type nat is either $\mathsf{0}$ or $\mathsf{s}\,x$ for some x. A property $C(n)$ is true if it holds no matter whether the natural number n was introduced by $\mathsf{nat}I_0$ so is zero or was introduced by $\mathsf{nat}I_s$ so is a successor:

$$\frac{n : \mathsf{nat} \quad C(\mathsf{0})\ \mathit{true} \qquad \begin{array}{c}\dfrac{}{x : \mathsf{nat}} \quad \dfrac{}{C(x)\ \mathit{true}}\,u \\ \vdots \\ C(\mathsf{s}\,x)\ \mathit{true}\end{array}}{C(n)\ \mathit{true}}\ \mathsf{nat}E^{x,u}$$

In words: In order to prove property C of a natural number n we have to prove $C(\mathsf{0})$ and also $C(\mathsf{s}\,x)$ under the assumption that $C(x)$ for a new parameter x. The scope of x and u is just the rightmost premise of the rule $\mathsf{nat}E^{x,u}$. This corresponds exactly to proof by induction, where the proof of $C(\mathsf{0})$ is the base case, and the proof of $C(\mathsf{s}\,x)$ from the assumption $C(x)$ is the induction step. That is why $\mathsf{nat}E^{x,u}$ is also called an *induction rule* for nat.

We managed to state this rule without any explicit appeal to universal quantification, using parametric judgments instead. We could, however, write the reasoning principle down with explicit quantification as a proposition, in which case it becomes:

$$\forall n{:}\mathsf{nat}.\ C(0) \supset (\forall x{:}\mathsf{nat}.\ C(x) \supset C(\mathsf{s}\,x)) \supset C(n)$$

for an arbitrary property C of natural numbers. It is an easy exercise to prove this with the induction rule above, since the respective introduction rules lead to a proof that exactly has the shape of $\mathsf{nat}E^{x,u}$.

All natural numbers are zero or successors. To illustrate elimination rule $\mathsf{nat}E^{x,u}$ in action, we start with a very simple property: every natural number is either 0 or has a predecessor. First, a detailed induction proof in the usual mathematical style and then a similar formal proof.

> **Theorem:** $\forall x{:}\mathsf{nat}.\ x = 0 \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y$.
> **Proof:** By induction on x.
>
> **Base:** $x = 0$. Then the left disjunct is true.
>
> **Step:** $x = \mathsf{s}\,x'$. That is, x is the successor of some natural number (x') for which the theorem holds. Then the right disjunct is true: pick $y = x'$ and observe $x = \mathsf{s}\,x' = \mathsf{s}\,y$.

Now we write this in the formal notation of inference rules. It is instructive to construct this proof step-by-step; we show only the final deduction. We assume there is either a primitive or derived rule of inference called refl expressing reflexivity of equality on natural numbers ($n = n$). We use the same names as in the mathematical proof.

$$\dfrac{\dfrac{}{x : \mathsf{nat}} \quad \dfrac{\dfrac{}{0 = 0\ \mathit{true}}\ \mathsf{refl}}{0 = 0 \vee \exists y{:}\mathsf{nat}.\ 0 = \mathsf{s}\,y\ \mathit{true}}\ \vee I_1 \quad \dfrac{\dfrac{\dfrac{}{x' : \mathsf{nat}} \quad \dfrac{}{\mathsf{s}\,x' = \mathsf{s}\,x'\ \mathit{true}}\ \mathsf{refl}}{\exists y{:}\mathsf{nat}.\ \mathsf{s}\,x' = \mathsf{s}\,y\ \mathit{true}}\ \exists I}{\mathsf{s}\,x' = 0 \vee \exists y{:}\mathsf{nat}.\ \mathsf{s}\,x' = \mathsf{s}\,y\ \mathit{true}}\ \vee I_2}{x = 0 \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y\ \mathit{true}}\ \mathsf{nat}E^{x',u}$$
$$\dfrac{x = 0 \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y\ \mathit{true}}{\forall x{:}\mathsf{nat}.\ x = 0 \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y\ \mathit{true}}\ \forall I^x$$

This is a simple proof by cases and, in this particular proof, does not even use the induction hypothesis $x' = 0 \vee \exists y{:}\mathsf{nat}.\ x' = \mathsf{s}\,y\ \mathit{true}$, which would have been labeled u. It is also possible to finish the proof by eliminating from that induction hypothesis, but the proof then ends up being more complicated. At our present level of understanding, the computational counterpart for the above proof might be a zero-check function for natural numbers. It takes any natural number and provides the left disjunct if that number was 0 while providing the right disjunct if it was a successor. Making use of the witness, we will later discover more general computational content once we have a proof term assignment.

In the application of the induction rule $\mathsf{nat}E$ we used the property $C(x)$, which is a proposition with the free variable x of type nat, saying that x is either zero or a successor of some natural number. More explicitly:

$$C(x) = (x = \mathsf{0} \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\, y)$$

While getting familiar with formal induction proofs it may be a good idea to write out the induction formula explicitly.

3 Equality

We already used equality in the previous example, without justification, so we now introduce it properly into our formal system. Equality is certainly a central part of Heyting (and Peano) arithmetic.

There are many ways to define and reason with equality. The one we choose here is the one embedded in arithmetic where we are only concerned with numbers. Thus we are trying to define $x = y$ only for natural numbers x and y. Of course, $x = y$ must be a *proposition*, not a term. As a proposition, we will use the techniques of the course and define its truth by means of introduction and elimination rules!

The introduction rules for equality are straightforward based on the two introduction rules for natural numbers.[1]

$$\frac{}{\mathsf{0} = \mathsf{0}\ \mathit{true}}\ {=}I_{00} \qquad \frac{x = y\ \mathit{true}}{\mathsf{s}\, x = \mathsf{s}\, y\ \mathit{true}}\ {=}I_{ss}$$

If this is our definition of equality on natural numbers, how can we use the knowledge that $n = k$? If n and k are both $\mathsf{0}$, we cannot learn anything, because no knowledge was used for ${=}I_{00}$. If both are successors, we know their argument must be equal. Finally, if one is a successor and the other zero, then this is contradictory and we can derive anything (any proposition C).

$$\text{no rule } E_{00} \qquad \frac{\mathsf{0} = \mathsf{s}\, x\ \mathit{true}}{C\ \mathit{true}}\ {=}E_{0s} \qquad \frac{\mathsf{s}\, x = \mathsf{0}\ \mathit{true}}{C\ \mathit{true}}\ {=}E_{s0} \qquad \frac{\mathsf{s}\, x = \mathsf{s}\, y\ \mathit{true}}{x = y\ \mathit{true}}\ {=}E_{ss}$$

Local soundness is very easy to check, but what about local completeness? It turns out to be a complicated issue so we will not discuss it here.

4 Equality is Reflexive

As a simple inductive theorem we show the reflexivity of equality.

Theorem 1. $\forall x{:}\mathsf{nat}.\ x = x$

[1] As a student observed in lecture, we could also just state $x = x\ \mathit{true}$ as an inference rule with no premise. However, it is difficult to justify the elimination rules we need for Heyting arithmetic from this definition.

Proof: By induction on x.

Base: $x = 0$. Then, indeed, $0 = 0$ by rule $=I_{00}$

Step: Assume the theorem $x = x$ already holds for x and show $\mathsf{s}\,x = \mathsf{s}\,x$ for the successor $\mathsf{s}\,x$, which follows by $=I_{ss}$.

□

This proof is small enough so we can present it in the form of a natural deduction. For induction, we use $C(n) = (n = n)$.

$$\dfrac{\dfrac{}{n : \mathsf{nat}} \quad \dfrac{}{0 = 0\ \mathit{true}}\,{=}I_{00} \quad \dfrac{\dfrac{}{x = x\ \mathit{true}}\,u}{\mathsf{s}\,x = \mathsf{s}\,x\ \mathit{true}}\,{=}I_{ss}}{\dfrac{n = n\ \mathit{true}}{\forall x{:}\mathsf{nat}.\, x = x\ \mathit{true}}\,\forall I^{n}}\,\mathsf{nat}E^{x,u}$$

The hypothesis $x : \mathsf{nat}$ introduced by $\mathsf{nat}E^{x,u}$ is implicitly used to establish that $x = x$ is a well-formed proposition, but is not explicit in the proof.

The above theorem justifies a derived rule of inference:

$$\dfrac{x : \mathsf{nat}}{x = x\ \mathsf{true}}\,\mathsf{refl}$$

by using $\forall E$ with the theorem just proved. We usually suppress the premise $x : \mathsf{nat}$ since we already must know $x : \mathsf{nat}$ for the proposition $x = x$ to be well-formed at all. This is the rule we used in Section 2.

5 Primitive Recursion

Reconsidering the elimination rule for natural numbers, we notice that we exploit the knowledge that $n : \mathsf{nat}$, but we only do so when we are trying to establish the truth of a proposition, $C(n)$. However, we are equally justified in using $n : \mathsf{nat}$ when we are trying to establish not another proposition $C(n)$ but another typing judgment of the form $t : \tau$. The rule, also called *rule of primitive recursion* for nat, then becomes

$$\dfrac{n : \mathsf{nat} \quad t_0 : \tau \qquad \begin{array}{c}\dfrac{}{x : \mathsf{nat}} \quad \dfrac{}{r : \tau} \\ \vdots \\ t_s : \tau\end{array}}{R(n, t_0, x.\, r.\, t_s) : \tau}\,\mathsf{nat}E^{x,r}$$

Here, R is a new term constructor,[2] the term t_0 is the term used for the zero case where $n = \mathsf{0}$, and the term t_s captures the successor case where $n = \mathsf{s}\, n'$. In the latter case x is a new parameter of type nat introduced in the rule that stands for the predecessor n'. And r is a new parameter of type τ that stands for the result of the function R when applied to that predecessor n', which corresponds to an appeal to the induction hypothesis. The term t_s of type τ can use parameter x to refer to the predecessor and use parameter r to refer to the result at x. Following a common notational convention of logic, the dots in the notation $x.\, r.\, t_s$ of the third argument of R indicate that occurrences of x and r in t_s are bound with scope t_s. The fact that both are bound corresponds to the newly introduced assumptions $x : \mathsf{nat}$ and $r : \tau$ that are both introduced by $\mathsf{nat}E^{x,r}$ to prove $t_s : \tau$ in the rightmost premise.

The local reduction rules may help explain this concept. We first write them down just on the terms, where they are computation rules.

$$\begin{array}{lcl} R(\mathsf{0}, t_0, x.\, r.\, t_s) & \Longrightarrow_R & t_0 \\ R(\mathsf{s}\, n', t_0, x.\, r.\, t_s) & \Longrightarrow_R & [R(n', t_0, x.\, r.\, t_s)/r][n'/x]\, t_s \end{array}$$

The first argument of R carries the recursion. The first reduction reduces recursors at $\mathsf{0}$ to their designated result t_0 coming from the second argument to R. The second reduction reduces, at a successor $\mathsf{s}\, n'$, to the term t_s from the third argument to R with parameter x instantiated to the concrete number n' for the inductive hypothesis and with parameter r instantiated to the particular value that the same recursor R had at that predecessor n'. So the argument t_0 of R indicates the output to use for $n = \mathsf{0}$ while t_s indicates the output to use for $n = \mathsf{s}\, x$ as a function of the predecessor x and of r for the recursive outcome of $R(n, t_0, x.\, r.\, t_s)$.

These proof terms are still somewhat unwieldy, so we consider a more readable schematic form, called *primitive recursion schema*. If we define f by cases

$$\begin{array}{lcl} f(\mathsf{0}) & = & t_0 \\ f(s\, x) & = & t_s(x, f(x)) \end{array}$$

where the *only* occurrence of f on the right-hand side is the one shown applied to x, then we could have equivalently defined f explicitly with:

$$f = (\mathsf{fn}\ n \Rightarrow R(n, t_0, x.\, r.\, t_s(x, r)))$$

To verify this, apply f to $\mathsf{0}$ and apply the reduction rules and also apply f to $\mathsf{s}\, n$ for an arbitrary n and once again apply the reduction rules.

$$\begin{array}{lcl} f(\mathsf{0}) & \Longrightarrow_R & R(\mathsf{0}, t_0, x.\, r.\, t_s(x, r)) \\ & \Longrightarrow_R & t_0 \end{array}$$

noting that the x in $x.r.t_s(\dots)$ is not a free occurrence (indicated by the presence of the dot in x.) since it corresponds to the hypothesis $x : \mathsf{nat}$ in $\mathsf{nat}E^{x,r}$. Finally

$$\begin{array}{lcl} f(\mathsf{s}\, n) & \Longrightarrow_R & R(\mathsf{s}\, n, t_0, x.\, r.\, t_s(x, r)) \\ & \Longrightarrow_R & t_s(n, R(n, t_0, x.\, r.\, t_s(x, r))) \\ & = & t_s(n, f(n)) \end{array}$$

[2] The name R starting the proof term for primitive recursion suggests recursion.

The last equality is justified by a (meta-level) induction hypothesis, because we are trying to show that $f(n) = R(n, t_0, x.\, r.\, t_s(x, r))$ and, when showing it for $\mathsf{s}\, n$, can assume to already have established it for the smaller n itself.

So far, our knowledge of typing judgments is limited to natural numbers. Before we use primitive recursion for something exciting, we first need to understand how it works on more general types τ.

6 Function Types

For moving beyond natural number types, we consider the type $\tau \to \sigma$ that a term has that is a function from input of type τ to output of type σ. We also reuse the notion of functional abstraction (already used[3] to describe proof terms of $A \supset B$ and $\forall x{:}\tau.\, A(x)$ to describe functions at the level of data). We write $\tau \to \sigma$ for functions from type τ to type σ and present them here without further justification since they just mirror the kinds of rules we have seen multiple times already.

$$\frac{\begin{array}{c}\overline{x : \tau} \\ \vdots \\ s : \sigma\end{array}}{\mathsf{fn}\, x{:}\tau \Rightarrow s \;:\; \tau \to \sigma}\ {\to}I \qquad \frac{s : \tau \to \sigma \quad t : \tau}{s\, t : \sigma}\ {\to}E$$

The local reduction is

$$(\mathsf{fn}\, x{:}\tau \Rightarrow s)\, t \quad \Longrightarrow_R \quad [t/x]s$$

Now we can define function double via the schema of primitive recursion.

$$\begin{array}{lcl} \mathsf{double}(\mathsf{0}) & = & \mathsf{0} \\ \mathsf{double}(\mathsf{s}\, x) & = & \mathsf{s}\,(\mathsf{s}\,(\mathsf{double}\, x)) \end{array}$$

We can read off the closed-form definition if we wish:

$$\mathsf{double} = (\mathsf{fn}\, n \Rightarrow R(n, \mathsf{0}, x.\, r.\, \mathsf{s}\,(\mathsf{s}\, r)))$$

After having understood this, we will be content with using the schema of primitive recursion. We define addition and multiplication as exercises.

$$\begin{array}{lcl} \mathsf{plus}(\mathsf{0}) & = & \mathsf{fn}\, y \Rightarrow y \\ \mathsf{plus}(\mathsf{s}\, x) & = & \mathsf{fn}\, y \Rightarrow \mathsf{s}\,((\mathsf{plus}\, x)\, y) \end{array}$$

Notice that plus is a function of type $\mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat})$ that is primitive recursive in its (first and only) argument.

$$\begin{array}{lcl} \mathsf{times}(\mathsf{0}) & = & \mathsf{fn}\, y \Rightarrow \mathsf{0} \\ \mathsf{times}(\mathsf{s}\, x) & = & \mathsf{fn}\, y \Rightarrow (\mathsf{plus}\ ((\mathsf{times}\, x)\, y))\, y \end{array}$$

Modulo currying/uncurrying to the appropriate function types, these are the expected definitions of addition/multiplication of natural numbers.

[3] The case for unifying all these notions in type theory looks pretty strong at this point.

7 Proof Terms

With proof terms for primitive recursion in place, we can now revisit and make a consistent proof term assignment for the elimination form also with respect to the truth of propositions. It stands to reason to use the same proof terms as for typing judgments.

$$\dfrac{\begin{array}{c} n : \mathsf{nat} \quad M_0 : C(\mathsf{0}) \qquad \begin{array}{c}\overline{x : \mathsf{nat}} \quad \overline{u : C(x)}\,^{u} \\ \vdots \\ M_s : C(\mathsf{s}\,x)\end{array} \end{array}}{R(n, M_0, x.\, u.\, M_s) : C(n)}\ \mathsf{nat}E^{x,u}$$

Except for the type of judgment (proof terms and propositions versus typing judgments), this elimination rule $\mathsf{nat}E^{x,u}$ is the same as the (primitive) recursion rule $\mathsf{nat}E^{x,r}$, just on propositions instead of data.

The local reductions we discussed before for terms representing data, also work for these proofs terms, because they are both derived from slightly different variants of the elimination rules (one with proof terms, one with data terms).

$$\begin{array}{lcl} R(\mathsf{0}, M_0, x.\, u.\, M_s) & \Longrightarrow_R & M_0 \\ R(\mathsf{s}\,n', M_0, x.\, u.\, M_s) & \Longrightarrow_R & [R(n', M_0, x.\, u.\, M_s)/u][n'/x]\, M_s \end{array}$$

Computationally, we can conclude that proofs by induction correspond to functions defined by primitive recursion, and that they compute in the same way!

Returning to the earlier example, we can write the proof terms.

Theorem: $\forall x{:}\mathsf{nat}.\ x = \mathsf{0} \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y$.
Proof: By induction on x.

Base: $x = \mathsf{0}$. Then the left disjunct is true.

Step: $x = \mathsf{s}\,x'$. Then the right disjunct is true: pick $y = x'$ and observe $x = \mathsf{s}\,x' = \mathsf{s}\,y$.

The extracted function we obtain by marking its natural deduction proof with proof terms is the predecessor function (note how the use of **inl** versus **inr** correspond to the zero check that predecessor functions need to perform on natural numbers as $\mathsf{0}$ has no predecessor):

$$\mathsf{pred} \quad = \quad \mathsf{fn}\, x{:}\mathsf{nat} \Rightarrow R(x, \mathbf{inl}\ \_, x.\, r.\, \mathbf{inr}(x, \_))$$

Here we have suppressed the evidence for equalities, since we have not yet introduced proof terms for them. We just write $\_$ for proofs of equality (whose computational content we do not care about).

$$\dfrac{\dfrac{\overline{x : \mathsf{nat}}\quad \dfrac{\dfrac{}{\_ : 0 = 0}\,\mathsf{refl}}{\mathbf{inl}\ \_ : 0 = 0 \vee \exists y{:}\mathsf{nat}.\ 0 = \mathsf{s}\,y}\vee I_1 \quad \dfrac{\dfrac{\overline{x' : \mathsf{nat}}\quad \dfrac{}{\_ : \mathsf{s}\,x' = \mathsf{s}\,x'}\,\mathsf{refl}}{(x', \_) : \exists y{:}\mathsf{nat}.\ \mathsf{s}\,x' = \mathsf{s}\,y}\exists I}{\mathbf{inr}(x', \_) : \mathsf{s}\,x' = 0 \vee \exists y{:}\mathsf{nat}.\ \mathsf{s}\,x' = \mathsf{s}\,y}\vee I_2}{R(x, \mathbf{inl}\ \_, x'.r'.\mathbf{inr}(x', \_)) : x = 0 \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y}\,\mathsf{nat}E^{x',r'}}{\mathsf{fn}\,x \Rightarrow R(x, \mathbf{inl}\ \_, x'.r'.\mathbf{inr}(x', \_)) : \forall x{:}\mathsf{nat}.\ x = 0 \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y}\,\forall I^x$$

8 Local Proof Reduction

For harmony, we would like to check that the rules for natural numbers are locally sound and complete.

Local Soundness. For soundness, we verify that no matter how we introduce the judgment $n : \mathsf{nat}$, we can find a "more direct" proof of the conclusion. This is easy for $\mathsf{nat}I_0$, because the second premise already establishes our conclusion directly.

$$\dfrac{\dfrac{}{0 : nat}\,\mathsf{nat}I_0 \quad \begin{matrix}\mathcal{E}\\ C(0)\ true\end{matrix} \quad \dfrac{\overline{x : nat}\quad \overline{C(x)\ true}^{\,u} \quad \begin{matrix}\mathcal{F}\\ C(\mathsf{s}\,x)\ true\end{matrix}}{C(0)\ true}\,\mathsf{nat}E^{x,u}}{C(0)\ true} \quad \Longrightarrow_R \quad \begin{matrix}\mathcal{E}\\ C(0)\ true\end{matrix}$$

The case where $n = \mathsf{s}\,n'$ is more difficult and more subtle. Intuitively, we should be using the deduction of the second premise for this case.

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ n' : \mathsf{nat}\end{matrix}}{\mathsf{s}\,n' : \mathsf{nat}}\,\mathsf{nat}I_s \quad \begin{matrix}\mathcal{E}\\ C(0)\ true\end{matrix} \quad \dfrac{\overline{x : \mathsf{nat}}\quad \overline{C(x)\ true}^{\,u} \quad \begin{matrix}\mathcal{F}\\ C(\mathsf{s}\,x)\ true\end{matrix}}{C(\mathsf{s}\,n')\ true}\,\mathsf{nat}E^{x,u}}{C(\mathsf{s}\,n')\ true}$$

$$\Longrightarrow_R \quad \dfrac{\begin{matrix}\mathcal{D}\\ n' : \mathsf{nat}\end{matrix} \quad \dfrac{\begin{matrix}\mathcal{D}\\ n' : \mathsf{nat}\end{matrix} \quad \begin{matrix}\mathcal{E}\\ C(0)\ true\end{matrix} \quad \dfrac{\overline{x : \mathsf{nat}}\quad \overline{C(x)\ true}^{\,u} \quad \begin{matrix}\mathcal{F}\\ C(s\,x)\ true\end{matrix}}{C(n')\ true}\,\mathsf{nat}E^{x,u}}{[n'/x]\mathcal{F}'\\ C(s\,n')\ true}$$

It is difficult to see, however, in which way this is actually a reduction: $\mathcal{D}$ is duplicated, $\mathcal{E}$ persists, and we still have an application of $\mathsf{nat}E$. The key to appreciating this as a reduction, regardless, however, is that the *term* we are eliminating with the application of $\mathsf{nat}E$ becomes smaller: from $\mathsf{s}\,n'$ to n'. In hindsight we should have expected this, because the term is also the only component getting smaller in the second introduction rule for natural numbers. Fortunately, the term that $\mathsf{nat}E$ is applied to can only get smaller finitely often, because it will ultimately just be $\mathsf{0}$, so will be back in the first local reduction case.

The computational content of this local reduction directly explains in what way the natural number gets smaller even if the form of the judgment stays the same, because it corresponds directly to the reductions for primitive recursors $R(n, M_0, x.\, u.\, M_s)$.

The question of local expansion does not make sense solely in the truth setting without the rule for typing judgments. The difficulty is that we need to show that we can apply the elimination rules in such a way that we can reconstitute a proof of the original judgment. However, the elimination rule for the truth judgment cannot really reintroduce $n : \mathsf{nat}$, since the only two introduction rules $\mathsf{nat}I_0$ and $\mathsf{nat}I_s$ do not apply.

9 Local Expansion

Using primitive recursion on typing judgments, we obtain a local expansion for $n : \mathsf{nat}$.

$$\begin{array}{c}\mathcal{D}\\ n : \mathsf{nat}\end{array} \quad \Longrightarrow_E \quad \dfrac{\begin{array}{c}\mathcal{D}\\ n : \mathsf{nat}\end{array} \quad \dfrac{}{\mathsf{0} : \mathsf{nat}}\,\mathsf{nat}I_0 \quad \dfrac{\overline{x : \mathsf{nat}}}{\mathsf{s}\,x : \mathsf{nat}}\,\mathsf{nat}I_s}{R(n, \mathsf{0}, x.\, r.\, \mathsf{s}\,x) : \mathsf{nat}}\,\mathsf{nat}E^{x,r}$$

A surprising observation about the local expansion is that it does not use the recursive result, r, which corresponds to a use of the induction hypothesis. Consequently, a simple proof-by-cases that uses $\mathsf{nat}E_0$ when n is zero and uses $\mathsf{nat}E_s$ when n is a successor would also have been locally sound and complete already.

This is a reflection of the fact that the local completeness property we have does not carry over to a comparable global completeness. The difficulty is the well-known property that in order to prove a proposition A by induction, we may have to first generalize the induction hypothesis to some B, prove B by induction and also prove $B \supset A$. Such proofs do *not* have the subformula property that proofs only use subformulas of the conclusion, which means that our strict verificationist program of explaining the meaning of propositions from the meaning of their parts breaks down in arithmetic. In fact, there is an entire hierarchy of arithmetic theories, depending on which propositions we may use as induction formulas.

10 Example: Integer Square Root

Armed with the formal understanding of primitive recursion in Heyting arithmetic, we now go through several examples informally, presenting proofs in the usual mathemat-

ical style and their computational contents as recursive functions. The explicit discussion of inductive arguments here is helpful when performing induction on structures other than natural numbers.

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 2 (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 it cannot pattern-match against $x + 1$, so we have to rewrite this slightly. We also use 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 13). This illustrates, yet again, that since constructive proofs and functions are in Curry-Howard correspondence, different proofs of the same theorem can have different efficiency in terms of their computational content.

11 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 3 (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 4. $\forall b{:}\mathsf{nat}.\, \forall n{:}\mathsf{nat}.\, \forall c{:}\mathsf{nat}.\, \exists y{:}\mathsf{nat}.\, y = c \times b^n$

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 $\exists y{:}\mathsf{nat}.\, y = c_1 \times b^{n+1}$.

$\forall c$:nat. $\exists y$:nat. $y = c \times b^n$	by IH
$\exists y$: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 $\exists E$
$\exists y$:nat. $y = c_1 \times b^{n+1}$	picking $y = a$ and $\exists I$
$\forall c$:nat. $\exists y$:nat. $y = c \times b^{n+1}$	by $\forall I$

□

Now a second proof for Theorem 3 no longer requires a proof by induction, directly calling on the above lemma instead.

Theorem 5 (Exponential closure)**.** $\forall b$:nat. $\forall n$:nat. $\exists y$: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 $O(\log(n))$ 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 6. $\forall b$:nat. $\forall n$:nat. $\forall c$:nat. $\exists y$: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
```

12 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.[4] 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 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. Here, $\mathsf{path}_V(x, y)$ is considered as a formula in three arguments, x, y and list 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 7. $\forall V{:}N\ \mathsf{list}.\, \forall x{:}N.\, \forall y{:}N.\, \mathsf{path}_V(x, y) \vee \neg\mathsf{path}_V(x, y)$

Proof: By induction on the structure of V.

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

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:

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 the short-circuit evaluation `orelse` and `andalso`, for the sake of brevity and readability.

```
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.[5] 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)
```

13 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

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

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.

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

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

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

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

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

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

[Hey56] Arend Heyting. *Intuitionism: An Introduction*. North-Holland Publishing, Amsterdam, 1956. 3rd edition, 1971.

[Pea89] Giuseppe Peano. *Arithmetices Principia, Nova Methodo Exposita*. Fratres Bocca, 1889.

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