Usage of Higher-Order Logic

Summary

I give examples of how higher-order logic can be used for formalizing basic facts about inductive datastructures.

I will define some axioms that are frequently used, and give examples of their use.

I give some of the basic proofs. Proofs of basic properties can be surprisingly hard.

Induction for Natural Numbers

In standard mathematics, the principle of complete induction is defined as follows:

Let E be property, s.t. E holds for 0, and whenever E holds for n, then E also holds for $n + 1$.

Then E is true for all natural numbers.

Induction for Natural Numbers (2)

The natural numbers are usually defined as those objects that can be constructed from a constant $0\colon \mathrm{Nat}$ and a function $\mathrm{succ}\colon \mathrm{Nat} \to \mathrm{Nat}$.

The number n has representation $\mathrm{succ}^n(0)$.

This is the most elementary representation possible. (and logicians like minimalism)

Using this representation, the induction principle for natural numbers is the following formula:

$$\forall P\colon \mathrm{Nat} \to \mathrm{Nat} \;\; P(0) \to (\; \forall n\colon \mathrm{Nat}\; P(n) \to (P\; (\mathrm{succ}\; n)\;) \to$$

$$\forall n\colon \mathrm{Nat}\; (P\;\; n).$$

Excursion to Set Theory

In set theory, one would define: 'The set of natural numbers Nat is the smallest set, which contains 0, and which is closed under the succ function'.

$$\mathrm{Nat} = \bigcap \{S \mid 0 \in S \wedge \forall n\ n \in S \rightarrow \mathrm{succ}(n) \in S\}.$$

(Following Von Neumann, one usually takes $0 = \{\ \}$, and $\mathrm{succ}(n) = \{n\} \cup n$)

Since $\bigcap M = \{z \mid \forall m\ m \in M \rightarrow z \in m\}$, one can write

$\mathrm{Nat} = \{z \mid \forall m\ m \in \{S \mid 0 \in S \wedge \forall n\ n \in S \rightarrow \mathrm{succ}(n) \in S\} \rightarrow z \in m\}$.

$m \in \{S \mid P(S)\}$ can be replaced by $P(m)$. (Note that this is a form of β-reduction) The result is:

$$\mathrm{Nat} = \{z \mid \forall m\ (0 \in m \wedge \forall n\ n \in m \rightarrow \mathrm{succ}(n) \in m) \rightarrow z \in m\}.$$

From the previous formula, we see that

$$t \in \mathrm{Nat} \leftrightarrow \forall m\ (0 \in m \wedge \forall n\ n \in m \rightarrow \mathrm{succ}(n) \in m) \rightarrow t \in m\}.$$

We see that the induction principle can be taken as definition of Nat.

In order to show that $t \in \mathrm{Nat}$ has some property P, form the set of elements S_P with this property. Show that $0 \in S_p$, and $\forall n\ n \in S_P \rightarrow \mathrm{succ}(n) \in S_P$. Then $t \in S_P$, using the equivalence above.

Induction is possible because of Minimality

Induction is a consequence of the fact that some type is defined as the smallest set having some closure properties. In set theory, this can be expressed in two possible ways, which are equivalent:

$$I = \bigcap \{ S \mid S \text{ has the desired closure properties } \},$$

or

$$I = \{ i \mid (\forall S \; S \text{ has the desired closure properties }) \rightarrow i \in S \}.$$

Defining Inductive Sets in HOL (1)

In HOL (the system), one must distinguish between the definition of a data type and the definition of an inductive set. Defining inductive sets in HOL is not much different from set theory. For recursive data types, HOL has special constructions.

The set of even numbers is the smallest set containing 0 and closed under two times taking succ :

$$\Phi_E := \lambda P{:}\mathrm{Nat} \to \mathrm{Bool}\, (P\, 0) \wedge (\, \forall n{:}\mathrm{Nat}\, (P\, n) \to (P\, (\mathrm{succ}\, (\mathrm{succ}\, n)))\,).$$

$$E := \lambda n{:}\mathrm{Nat}\ \forall P{:}\mathrm{Nat} \to \mathrm{Bool}\ (\Phi_E\ P) \to (P\ n).$$

For each m, the set of elements greater than m is the smallest set containing m, and closed under succ :

$$\Phi_{\leq} := \lambda n{:}\mathrm{Nat}\ \lambda P{:}\mathrm{Nat} \to \mathrm{Bool}\ \ (P\ n) \wedge \forall m{:}\mathrm{Nat}\ (P\ m) \to (P\ (\mathrm{succ}\, m)).$$

Then $\leq := \lambda m, n{:}\mathrm{Nat}\ \forall P{:}\mathrm{Nat} \to \mathrm{Bool}\ (\Phi_{\leq}\ m\ P) \to (P\ n).$

Inductive Sets in HOL (2)

Proving that something is in an inductive set is usually easy. One only needs the closure properties of the inductive set, and not its minimality.

For example, in order to prove that $\mathrm{succ}^4(0)$ is even, one proves

$$\forall P{:}\,\mathrm{Nat} \rightarrow \mathrm{Bool}\ (\Phi_E\ P) \rightarrow (P\ (\mathrm{succ}^4\ 0)).$$

Since $(\Phi_E\ P)$ means $(P\ 0) \wedge \forall n{:}\,\mathrm{Nat}\ (P\ n) \rightarrow (P\ (\mathrm{succ}\ (\mathrm{succ}\ n)))$, it is easy to prove $(P\ (\mathrm{succ}^4\ 0))$.

Inductive Sets in HOL (3), Free Generation

Proving that something is not in an inductive set can be a real challenge. At this point, the minimality of the inductive set is really essential.

Let S be some inductive set. Then t is in $S \Leftrightarrow$ t is in all sets having the required closure property. In order to show that t is not in S, it is enough to find one set that has the closure property, and that does not contain t.

Before we can prove that there exist non-even numbers, we introduce another property of Nat, namely the fact that it is freely generated by 0 and succ. This means that:

$$s^i(0) = s^j(0) \Rightarrow i = j.$$

The fact that Nat is freely generated, is as fundamental as its minimality. Minimality and free generation together characterize the natural numbers.

More about Free Generation

Suppose that we don't know if Nat is free generated by 0 and succ.

Then we will not be able to prove that $\mathrm{succ}^3(0)$ is not even. It cannot be excluded that $\mathrm{succ}^3(0) = \mathrm{succ}^2(0)$, and then $\mathrm{succ}^3(0)$ is even.

Similarly, we will unable to prove that $\neg\ \mathrm{succ}^4(0) \leq 0$.

Free generation is also essential for the possibility to define functions.

Suppose that one want to define a function $f{:}\,\mathrm{Nat} \longrightarrow \mathrm{Bool}$, s.t. $(f\ 0) = \mathbf{t}$ and $\forall\ n{:}\,\mathrm{Nat}\ (f\ (\mathrm{succ}\ n)) = \mathbf{f}$. If $0 = \mathrm{succ}(0)$, such a function does not exist. (unless also $\mathbf{f} = \mathbf{t}$)

Axioms for Free Generation

There are two ways of specifying that Nat is freely generated:

1. By difference axioms:

$$\forall n{:}\,\mathrm{Nat}\ (\mathrm{succ}\ n) \neq 0,$$

$$\forall m, n{:}\,\mathrm{Nat}\ (\mathrm{succ}\ m) = (\mathrm{succ}\ n) \to m = n.$$

   These two axioms, together with the induction axiom, are called the Peano axioms.

2. Ensuring that sufficiently many functions can be defined. This is usually done by a recursion operator. This approach is taken in the HOL system.

Finally: Non-evenness of $\mathrm{succ}^3(0)$

One needs to find a set which contains 0, which is closed under succ^2, and which does not contain $\mathrm{succ}^3(0)$.

A first guess could be

$$S := \lambda n\colon \mathrm{Nat}\ n \neq \mathrm{succ}^3(0),$$

but we have $(S\ \mathrm{succ}(0))$, and not $(S\ \mathrm{succ}^3(0))$.

The problem can be solved by adding $\mathrm{succ}(0)$ to S.

$$S := \lambda n\colon \mathrm{Nat}\ n \neq \mathrm{succ}(0) \wedge n \neq \mathrm{succ}^3(0).$$

After expanding the definitions, one has to show (using the difference axioms) that

$$0 \neq \mathrm{succ}(0) \wedge 0 \neq \mathrm{succ}^3(0), \text{ and}$$

$$n \neq \mathrm{succ}(0) \wedge n \neq \mathrm{succ}^3(0) \rightarrow \mathrm{succ}^2(n) \neq \mathrm{succ}(0) \wedge \mathrm{succ}^2(n) \neq \mathrm{succ}^3(0).$$

Definition of Addition

Let $+$ be declared as $+\colon \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}$.

Add the axioms:

$$\forall n{:}\,\mathrm{Nat}\ (+\ n\ 0) = n,$$

$$\forall m, n{:}\,\mathrm{Nat}\ (+\ m\ (\mathrm{succ}\ n)) = (\mathrm{succ}\ (+\ m\ n)).$$

Then one can prove (using induction):

1. $\forall n{:}\,\mathrm{Nat}\ (+\ 0\ n) = n$. The induction hypothesis is $\lambda n{:}\,\mathrm{Nat}\ (+\ 0\ n) = n$.

2. $\forall m, n{:}\,\mathrm{Nat}\ (+\ (\mathrm{succ}\ m)\ n) = (\mathrm{succ}\ (+\ m\ n))$. Fix m as arbitrary object. Then the induction hypothesis is: $\lambda n{:}\,\mathrm{Nat}\ (+\ (\mathrm{succ}\ m)\ n) = (\mathrm{succ}\ (+\ m\ n))$.

3. $\forall m, n{:}\,\mathrm{Nat}\ (+\ m\ n) = (+\ n\ m)$. Fix m as arbitrary object. Then the induction hypothesis is: $\lambda n{:}\,\mathrm{Nat}\ (+\ m\ n) = (+\ n\ m)$.

More Functions:

Let $\times$ be declared as $\times\colon \text{Nat} \to \text{Nat} \to \text{Nat}$.

Add the axioms:

$$\forall n\colon\text{Nat}\ (\times\ n\ 0) = 0,$$

$$\forall m, n\colon\text{Nat}\ (\times\ m\ (\text{succ}\ n)) = (+\ (\times\ m\ n)\ m).$$

One can prove for example:

$$\forall m, n\colon\text{Nat}\ (\times\ m\ n) = (\times\ n\ m),$$

$$\forall k, m, n\colon\text{Nat}\ (\times\ k\ (+\ m\ n)) = (+\ (\times\ k\ m)\ (\times\ k\ n)).$$

$$\forall k, m, n\colon\text{Nat}\ (\times\ (+\ m\ n)\ k) = (+\ (\times\ m\ k)\ (\times\ n\ k)).$$

More Inductive Types (1)

Bool is the smallest set containing $\mathbf{f}$ and $\mathbf{t}$, so we have

$$\mathrm{Bool} \colon \mathrm{Type}, \quad \mathbf{f} \colon \mathrm{Bool} \;\text{ and }\; \mathbf{t} \colon \mathrm{Bool}.$$

Bool induction:

$$\forall P \colon \mathrm{Bool} \to \mathrm{Bool}\ (P\ \mathbf{f}) \to (P\ \mathbf{t}) \to \forall b \colon \mathrm{Bool}\ (P\ b).$$

Difference axiom $\mathbf{f} \neq \mathbf{t}$.

(Note that Bool is not defined as inductive set in HOL system)

Some More Inductive Types (2)

For **List**, we have

$$\mathrm{List}\colon \mathrm{Type} \to \mathrm{Type}.$$

$$\mathbf{nil}\colon \Pi T\colon \mathrm{Type}\ (\mathrm{List}\ T),$$

$$\mathbf{cons}\colon \Pi T\colon \mathrm{Type}\ T \to (\mathrm{List}\ T) \to (\mathrm{List}\ T).$$

List induction:

$$\forall T\colon \mathrm{Type}\ \forall P\colon (\mathrm{List}\ T) \to \mathrm{Bool}$$

$$(P\ (\mathbf{nil}\ T)) \to$$

$$(\forall t\colon T\ \forall x\colon (\mathrm{List}\ t)\ (P\ x) \to (P\ (\mathbf{cons}\ T\ t\ x)))$$

$$\to \forall x\colon (\mathbf{List}\ T)\ (P\ x).$$

Difference axioms for List:

$$\forall T\colon \mathrm{Type}\ \forall t\colon T\ \forall x\colon (\mathbf{List}\ T)\ (\mathbf{nil}\ T) \neq (\mathbf{cons}\ T\ t\ x).$$

$$\forall T\colon \mathrm{Type}\ \forall t_1, t_2\colon T\ \forall x_1, x_2\colon (\mathbf{List}\ T)$$

$$(\mathbf{cons}\ T\ x_1\ t_1) = (\mathbf{cons}\ T\ x_2\ t_2) \to x_1 = x_2 \wedge t_1 = t_2.$$

Examples of List Induction

First declare some functions. (Types are omitted)

Declaration of **append**:

$$\forall l_1\ (\mathbf{append\ nil}\ l_1) = l_1,$$

$$\forall e\ l_1\ l_2\ (\mathbf{append}\ (\mathbf{cons}\ e\ l_1)\ l_2) = (\mathbf{cons}\ e\ (\mathbf{append}\ l_1\ l_2)).$$

Declaration of **reverse**:

$$(\mathbf{reverse\ nil}) = \mathbf{nil},$$

$$\forall e\ l_1\ (\mathbf{reverse}\ (\mathbf{cons}\ e\ l_1)) = (\mathbf{append}\ (\mathbf{reverse}\ l_1)\ (\mathbf{cons}\ e\ \mathbf{nil})).$$

1. $\forall l_1\ l_2\ l_3$
   $(\textbf{append}\ l_1\ (\textbf{append}\ l_2\ l_3)) = (\textbf{append}\ (\textbf{append}\ l_1\ l_2)\ l_3)$.
   The induction hypothesis is $\lambda l_1 \forall l_2\ l_3$
   $(\textbf{append}\ l_1\ (\textbf{append}\ l_2\ l_3)) = (\textbf{append}\ (\textbf{append}\ l_1\ l_2)\ l_3)$.

2. $\forall l_1\ (\textbf{append}\ \textbf{nil}) = \textbf{nil}$. The induction hypothesis is
   $\lambda l_1\ (\textbf{append}\ \textbf{nil}) = \textbf{nil}$.

1. $\forall l_1\ l_2$ (**reverse** (**append** $l_1\ l_2$)) $=$ (**append** (**reverse** l_1) (**reverse** l_2)). The induction hypothesis is $\lambda l_1\ \forall l_2$ (**reverse** (**append** $l_1\ l_2$)) $=$ (**append** (**reverse** l_1) (**reverse** l_2)).

2. (How do you prove $\forall l_2\ l_1$ (**reverse** (**append** $l_1\ l_2$)) $=$ (**append** (**reverse** l_1) (**reverse** l_2))?

   First use $\forall$-intro (with eigenvariable l_2), then induction hypothesis λl_1 (**reverse** (**append** $l_1\ l_2$)) $=$ (**append** (**reverse** l_1) (**reverse** l_2)).
   Of course, you can also exchange the quantifiers in (1))

3. $\forall l_1$ (**reverse** (**reverse** l_1)) $= l_1$. The induction hypothesis is λl_1 (**reverse** (**reverse** l_1)) $= l_1$.

A Systematic Approach to Function Definition

Until now, we have taken an ad hoc approach to function introducton: We declare the name of the function, and added some axioms that characterize its behaviour.

It would be better if one could use definitions instead of assumptions, because definitions are conservative. Since definitions can be eliminated from proofs, there are no things that can be proven with definitions, that cannot be proven without definitions.

There are two ways to make functions definable:

1. Assume a recursion operator for Nat.
2. Assume a general function introduction operator.

In HOL (system), the first approach is chosen.

Recursion Operator

The following function is an example of a recursively defined function in C^{++}.

```
unsigned int fact( unsigned int x )
{
   if( x == 0 )
      return 1;
   else
      return x * fact( x - 1 );
}
```

This program works because `unsigned int` is an inductively defined set, and we pretend that `unsigned int` is freely generated by `x = x + 1;` (And this is not true because MAXUNSIGNED $+ 1 = 0$)

Recursion Operator (2)

The recursion operator for Nat has the following type declaration:

$$\mathrm{rec}_{\mathrm{Nat}}\colon \Pi S\colon \mathrm{Type}\ S \to (S \to \mathrm{Nat} \to S) \to \mathrm{Nat} \to S.$$

(In the HOL system, it is slightly different. We come to that later)

The recursion operator introduces the following equivalences:

$(\mathrm{rec}_{\mathrm{Nat}}\ S\ f_0\ f_1\ 0) \equiv f_0,$
$(\mathrm{rec}_{\mathrm{Nat}}\ S\ f_0\ f_1\ (\mathrm{succ}\ n)) \equiv (f_1\ (\mathrm{rec}_{\mathrm{Nat}}\ S\ f_0\ f_1\ n)\ n).$

In HOL, standard equality is used.

Recursion Operator (3)

$$+ := (\lambda n{:}\,\mathrm{Nat}\ (\mathrm{rec}_{\mathrm{Nat}}\ n\ (\lambda p, q{:}\,\mathrm{Nat}\ (\mathrm{succ}\ q)))){:}\,\mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}.$$

$$\times := (\lambda n{:}\,\mathrm{Nat}\ (\mathrm{rec}_{\mathrm{Nat}}\ 0\ (\lambda p, q{:}\,\mathrm{Nat}\ (+\ q\ n)))){:}\,\mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}.$$

$$\mathrm{fact} := (\lambda n{:}\,\mathrm{Nat}\ (\mathrm{rec}_{\mathrm{Nat}}\ (\mathrm{succ}\ 0)\ (\lambda p, q{:}\,\mathrm{Nat}\ (\times\ p\ (\mathrm{succ}\ q)))) \\ {:}\,\mathrm{Nat} \to \mathrm{Nat}.$$

- As a general rule, writing down recursive definitions is difficult and unpleasant. All primitive recursive functions have a recursive definition.
- Does existence of $\mathrm{rec}_{\mathrm{Nat}}$ imply free generatedness? That depends on the goal type.

Some more Recursion Operators

Recursion operator for Bool:

$$\mathbf{rec}_{\mathrm{Bool}} : \Pi S : \mathrm{Type}\ S \to S \to \mathrm{Bool} \to S.$$

$$(\mathbf{rec}_{\mathrm{Bool}}\ f_0\ f_1\ \mathbf{f}) \equiv_\iota f_0,$$

$$(\mathbf{rec}_{\mathrm{Bool}}\ f_0\ f_1\ \mathbf{t}) \equiv_\iota f_1.$$

(It is the if-operator)

Recursion operator for List :

$$\mathbf{rec}_{\mathrm{List}} : \Pi T : \mathrm{Type}\ \ \Pi S : \mathrm{Type}$$

$$S \to (T \to (\mathrm{List}\ T) \to S \to S) \to (\mathrm{List}\ T) \to S.$$

$$(\mathbf{rec}_{\mathrm{List}}\ T\ S\ f_0\ f_1\ (\mathbf{nil}\ T)) \equiv_\iota f_0,$$

$$(\mathbf{rec}_{\mathrm{List}}\ T\ S\ f_0\ f_1\ (\mathbf{cons}\ T\ t\ x)) \equiv_\iota (f_1\ t\ x\ (\mathbf{rec}_{\mathrm{List}}\ T\ S\ f_0\ f_1\ x)).$$

Definitions of append and reverse

$$\textbf{append} = (\textbf{rec}_{\text{List}}\, T\, ((\text{List}\ T) \to (\text{List}\ T))\ (\lambda x{:}\, (\text{List}\ T)\ x)\ (\lambda t{:}\, T\ x{:}\, (\text{List}\ T)$$

$$f{:}\, (\text{List}\ T) \to (\text{List}\ T)\ (\lambda x_1{:}\, (\text{List}\ T)\ (\textbf{cons}\ t\ (f\ x))).$$

$$\textbf{reverse} =$$

An Operator for Function Introduction

Functions can be viewed as a special kind of relations R that satisfy the axioms

$$\forall x \; \exists y \; R(x, y),$$

and

$$\forall x \; \forall y_1, y_2 \; R(x, y_1) \wedge R(x, y_2) \rightarrow y_1 = y_2.$$

If one has a mechanism for obtaining functions from such relations, then functions can be defined like this.

The most radical solution is to introduce the epsilon operator, or global choice function. It has the following type:

$$\epsilon : \Pi T : \text{Type} \; \Pi P : T \rightarrow \text{Bool} \; T.$$

and satisfies the following axiom:

$$\forall T : \text{Type} \; \forall t : T \; \forall P : T \rightarrow \text{Bool} \; (P \; t) \rightarrow (P \; (\epsilon \; T \; P)).$$

The ϵ-operator

ϵ takes a type T and a predicate P over T. If there is a $t{:}T$, such that $(P\ t)$ is true, then $(\epsilon\ T\ P)$ returns an element of T for which $(P\ t)$ is true. If there is no $t{:}T$, s.t. $(P\ t)$ is true, then $(\epsilon\ T\ P)$ returns an arbitrary element of T.

The functions constructed by ϵ are very similar to Skolem functions. (and if one wishes, $\exists$ can be defined in terms of ϵ)

Strange behaviour of the ϵ-operator

Let $<$ be the smaller than relation:

$$(\epsilon \text{ Nat } (\lambda n{:}\text{Nat } n < 0))$$

constructs an unknown Nat, because there is no Nat < 0.

$$(\epsilon \text{ Nat } (\lambda n{:}\text{Nat } n > 0))$$

still constructs an unknown Nat, but at least not 0

$$(\epsilon \text{ Nat } (\lambda n{:}\text{Nat } (n > 3 \wedge n < 6))$$

equals either 4 or 5.

$$(\epsilon \text{ Nat } (\lambda n{:}\text{Nat } (n < 6 \wedge n > 3))$$

also equals 4 or 5 but it need not be the same number.

Introducing functions with the ϵ-operator

Suppose that T_1, T_2 are types and we want to define some function of type $T_1 \rightarrow T_2$.

1. Find a relation $P{:}T_1 \rightarrow T_2 \rightarrow \text{Bool}$ that characterizes the behaviour of the desired function.

2. Prove
$$\forall t_1{:}T_1\ \exists t_2{:}T_2\ (P\ t_1\ t_2).$$

3. Define
$$f := \lambda t_1{:}T_1\ (\epsilon\ T_2\ (P\ t_1\ t_2)){:}T_1 \rightarrow T_2.$$

4. It follows that
$$\forall t_1{:}T_1\ (P\ t_1\ (f\ t_1)).$$

Avoiding introduction of Global Choice

The ϵ-operator has some strange features. For example, it need not choose the same object on different but equivalent predicates.

One could define a weaker version of ϵ, which requires that the existing element is unique:

$$\phi{:}\,\Pi T{:}\,\mathrm{Type}\ \Pi P{:}\,T \to \mathrm{Bool}\ T,$$

with the axiom:

$$\forall T{:}\,\mathrm{Type}\ \forall t{:}\,T\ \forall P{:}\,(T \to \mathrm{Bool})\ (P\ t) \to (\forall t'{:}\,\mathrm{Bool}\ (P\ t') \to t' = t) \to$$

$$(P\ (\phi\ T\ P)).$$

Definition of the recursion operator using ϕ-operator

We will show how the recursion operator can be defined using the ϵ- or ϕ-function. After that, $\mathrm{rec}_{\mathrm{Nat}}$ can be used for defining other functions.

Actually, for some functions, it may be more convenient to give a direct definition using ϵ instead of using ϵ. (for example $\lambda n{:}\mathrm{Nat}\ n - 1$))

Let S be a type. Let $f_0{:}S$, and $f_1{:}S \rightarrow \mathrm{Nat} \rightarrow S$.

We define a relation R that models $\mathrm{rec}_{\mathrm{Nat}}$. After that, we use ϵ to define $\mathrm{rec}_{\mathrm{Nat}}$. The relation R is (of course) inductive, so we define it in two steps: First the closure conditions, then the inductive property.

Defining $\mathrm{rec}_{\mathrm{Nat}}$ (2)

The closure property is:

$$\Phi := \lambda S{:}\,\mathrm{Type}\ \lambda f_0{:}\,S\ \lambda f_1{:}\,S \to \mathrm{Nat} \to S\ \lambda P{:}\,\mathrm{Nat} \to S \to \mathrm{Bool}$$

$$(P\ 0\ f_0) \wedge \forall m{:}\,\mathrm{Nat}\ s{:}\,S\ (P\ m\ s) \to (P\ (\mathrm{succ}\ m)\ (f_1\ s\ m)),$$

and R is defined by:

$$R := \lambda S{:}\,\mathrm{Type}\ \lambda f_0{:}\,S\ \lambda f_1{:}\,S \to \mathrm{Nat} \to S\ \ \lambda m{:}\,\mathrm{Nat}\ \lambda s{:}\,S$$

$$\forall P{:}\,\mathrm{Nat} \to S \to \mathrm{Bool}\ (\Phi\ S\ f_0\ f_1\ P) \to (P\ m\ s).$$

Defining $\mathrm{rec}_{\mathrm{Nat}}$ (3)

1. First prove (using Nat-induction on m)

   $$\forall S{:}\,\mathrm{Type}\,\forall f_0{:}\,S\,\forall f_1{:}\,S \to \mathrm{Nat} \to S\,\forall m{:}\,\mathrm{Nat}\,\exists s{:}\,S\,(R\ S\ f_0\ f_1\ m\ s).$$

2. Then prove by R-induction:

   $$\forall S{:}\,\mathrm{Type}\,\forall f_0{:}\,S\,\forall f_1{:}\,S \to \mathrm{Nat} \to S\,\forall m{:}\,\mathrm{Nat}\,\forall s{:}\,S\,(R\ S\ f_0\ f_1\ m\ s) \to$$
   $$(m = 0 \wedge s = f_0 \vee$$
   $$\exists m'{:}\,\mathrm{Nat}\,\exists s'{:}\,S\ m = \mathrm{succ}(m') \wedge s = (f_1\ s'\ m') \wedge (R\ S\ f_0\ f_1\ m'\ s')).$$

3. After that, prove functionality

   $$\forall S{:}\,\mathrm{Type}\,\forall f_0{:}\,S\,\forall f_1{:}\,S \to \mathrm{Nat} \to S\,\forall m{:}\,\mathrm{Nat}\,\forall s_1, s_2{:}\,S$$
   $$(R\ S\ f_0\ f_1\ m\ s_1) \to (R\ S\ f_0\ f_1\ m\ s_2) \to s_1 = s_2,$$

   using (2) and Nat-induction on m.

Defining $\mathrm{rec}_{\mathrm{Nat}}$ (4)

Define $\mathrm{rec}_{\mathrm{Nat}} :=$

$$\lambda S{:}\,\mathrm{Type}\ \lambda f_0{:}\,S\ \lambda f_1{:}\,S \to \mathrm{Nat} \to S\ \lambda m{:}\,\mathrm{Nat}\ (\epsilon\ S\ (R\ S\ f_0\ f_1\ m\ s)).$$

From (1) follows that

$$\forall S{:}\,\mathrm{Type}\ \forall f_0{:}\,S\ \forall f_1{:}\,S \to \mathrm{Nat} \to S\ \forall m{:}\,\mathrm{Nat}$$

$$(R\ S\ f_0\ f_1\ m\ (\epsilon\ S\ (R\ S\ f_0\ f_1\ m\ s))),$$

and hence

$$(R\ S\ f_0\ f_1\ m\ (\mathrm{rec}_{\mathrm{Nat}}\ S\ f_0\ f_1\ m\ s)).$$

Then the equivalences can be easily proven using (2) and (3):

$$(\mathrm{rec}_{\mathrm{Nat}}\ S\ f_0\ f_1\ 0) = f_0,$$

$$(\mathrm{rec}_{\mathrm{Nat}}\ S\ f_0\ f_1\ (\mathrm{succ}\ n)) = (f_1\ (\mathrm{rec}_{\mathrm{Nat}}\ S\ f_0\ f_1\ n)\ n),$$

(but they became ordinary equalities instead of equivalences)