

Chapter 3

Inductive types

This chapter is concerned with inductive definitions in Coq. A lot of data types can be defined in λ -calculus. This yields however representations that are not always very efficient. More efficient representations can be obtained using inductive types.

3.1 Expressivity

Untyped lambda calculus. In the course *Introduction to theoretical computer science*, we have seen representations of the data types natural numbers, booleans, and pairs with some elementary operations. Moreover, in untyped λ -calculus a fixed point operator can be defined. A *fixpoint* of a function $f : X \rightarrow X$ is an $x \in X$ such that $f(x) = x$. For untyped λ -calculus there is the following important result:

Theorem 3.1.1. *For every λ -term F there is a term M such that*

$$FM =_{\beta} M.$$

The proof can be given by using the fixed point combinator

$$\mathbf{Y} = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$$

The fixed point operator can be used to defined recursive functions, like for instance

$$\begin{aligned} 0! &= 1, \\ (n+1)! &= (n+1) \cdot n!. \end{aligned}$$

in λ -calculus. Here a function is defined in terms of itself. The important thing is that on the right-hand side, it is applied to an argument that is essentially simpler than the argument on the left-hand side.

The class of recursive functions is the smallest class of functions $\mathbb{N} \rightarrow \mathbb{N}$ that contains the *initial functions*

1. *projections*:
 $U_i^m(n_1, \dots, n_m) = n_i$ for all $i \in \{1, \dots, m\}$,
2. *successor*:
 $Suc(n) = n + 1$,
3. *zero*:
 $Zero(n) = 0$,

and that is moreover closed under the following:

1. *composition*:
 if $g : \mathbb{N}^k \rightarrow \mathbb{N}$ and $h_1, \dots, h_k : \mathbb{N}^m \rightarrow \mathbb{N}$ are recursive, then $f : \mathbb{N}^m \rightarrow \mathbb{N}$ defined by

$$f(n_1, \dots, n_m) = g(h_1(n_1, \dots, n_m), \dots, h_k(n_1, \dots, n_m))$$

is also recursive,

2. *primitive recursion*:
 if $g : \mathbb{N}^k \rightarrow \mathbb{N}$ and $h : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ are recursive, then $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ defined by

$$\begin{aligned} f(0, n_1, \dots, n_k) &= g(n_1, \dots, n_k) \\ f(n+1, n_1, \dots, n_k) &= h(f(n, n_1, \dots, n_k), n, n_1, \dots, n_k) \end{aligned}$$

is also recursive,

3. *minimalization*:
 if $g : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ is recursive, and for all n_1, \dots, n_k there exists an n such that $g(n, n_1, \dots, n_k) = 0$, then $f : \mathbb{N}^k \rightarrow \mathbb{N}$ defined by

$$f(n_1, \dots, n_k) = \mu n. g(n, n_1, \dots, n_k)$$

is also recursive.

In the last clause, $\mu n. g(n, n_1, \dots, n_k)$ denotes the smallest natural number m such that $g(m, n_1, \dots, n_k) = 0$.

Kleene has shown the following:

Theorem 3.1.2. *All recursive functions are λ -definable.*

The proof is given using the following lemma:

Lemma 3.1.3.

1. *The initial functions are λ -definable.*
2. *The λ -definable functions are closed under composition, primitive recursion, and minimalization.*

Simply typed lambda calculus. The fixpoint combinator \mathbf{Y} is not typable in the simply typed λ -calculus. As a matter of fact, the simply typed λ -calculus has essentially less expressive power than the untyped λ -calculus. A result by Schwichtenberg, given below, states that the functions that are definable in simply typed λ -calculus are exactly the extended polynomials, which is a smaller class than the one of all recursive functions.

The class of *extended polynomials* is the smallest class of functions $\mathbb{N} \rightarrow \mathbb{N}$ that contains

1. *projections*:
 $U_i^m(n_1, \dots, n_m) = n_i$ for all $i \in \{1, \dots, m\}$,
2. *constant function*:
 $k(n) = k$,
3. *signum function*:
 $sig(0) = 0$,
 $sig(n + 1) = 1$,

and that is moreover closed under the following:

1. *addition*:
 if $f : \mathbb{N}^i \rightarrow \mathbb{N}$ and $g : \mathbb{N}^j \rightarrow \mathbb{N}$ are extended polynomials, then $(f + g) : \mathbb{N}^{i+j} \rightarrow \mathbb{N}$ defined by

$$(f + g)(n_1, \dots, n_i, m_1, \dots, m_j) = f(n_1, \dots, n_i) + g(m_1, \dots, m_j)$$

is also an extended polynomial,

2. *multiplication*:
 if $f : \mathbb{N}^i \rightarrow \mathbb{N}$ and $g : \mathbb{N}^j \rightarrow \mathbb{N}$ are extended polynomials, then $(f \cdot g) : \mathbb{N}^{i+j} \rightarrow \mathbb{N}$ defined by

$$(f \cdot g)(n_1, \dots, n_i, m_1, \dots, m_j) = f(n_1, \dots, n_i) \cdot g(m_1, \dots, m_j)$$

is also an extended polynomial.

Theorem 3.1.4. *The functions definable in simply typed λ -calculus are exactly the extended polynomials.*

An example of a function that is definable in λ -calculus but not in simply typed λ -calculus is *Ackermann's Function*, denoted by \mathbf{A} , that is defined as follows:

$$\begin{aligned} \mathbf{A}(0, n) &= n + 1, \\ \mathbf{A}(m + 1, 0) &= \mathbf{A}(m, 1), \\ \mathbf{A}(m + 1, n + 1) &= \mathbf{A}(m, \mathbf{A}(m + 1, n)). \end{aligned}$$

A representation of the natural numbers. Let A be some type. The natural numbers can be represented as countably infinitely many different closed inhabitants in normal form of type $(A \rightarrow A) \rightarrow (A \rightarrow A)$: represent $k \in \mathbb{N}$ as $\lambda s:A \rightarrow A. \lambda A:n. s^k(n)$ where we use the notation k defined as follows:

$$\begin{aligned} F^0(P) &= P, \\ F^{n+1}(P) &= F(F^n(P)). \end{aligned}$$

A few natural numbers in their encoding as Church numeral:

- The representation of 0 is $\lambda s:A \rightarrow A. \lambda n:A. n$. As a proof:

$$\frac{\frac{[A^n]}{A \rightarrow A} I[n] \rightarrow}{(A \rightarrow A) \rightarrow (A \rightarrow A)} I[s] \rightarrow$$

- The representation of 1 is $\lambda x:A \rightarrow A. \lambda n:A. (s n)$. As a proof:

$$\frac{\frac{[A \rightarrow A^s] \quad [A^n]}{A} E \rightarrow}{\frac{A}{A \rightarrow A} I[n] \rightarrow} I[s] \rightarrow$$

- The representation of 2 is $\lambda x:A \rightarrow A. \lambda n:A. (s (s n))$. As a proof:

$$\frac{[A \rightarrow A^s] \quad \frac{[A \rightarrow A^s] \quad [A^n]}{A} E \rightarrow}{\frac{A}{A \rightarrow A} I[n] \rightarrow} I[s] \rightarrow$$

Below we will see an easier encoding of natural numbers as an inductive type.

3.2 Universes of Coq

Every expression in Coq is typed. The type of a type is also called a sort. The three sorts in Coq are the following:

- **Prop** is the universe of logical propositions. If $A:\text{Prop}$ then A denotes the class of terms representing proofs of the proposition A .
- **Set** is the universe of program types or specifications. Besides programs also well-knowns sets as the booleans and the natural numbers are in **Set**.
- **Type** is the universe of **Prop** and **Set**. In fact there is an infinite hierarchy of sorts $\text{Type}(i)$ with $\text{Type}(i) : \text{Type}(i + 1)$. From the user point of view we have $\text{Prop}:\text{Type}$ and $\text{Set}:\text{Type}$, and further $\text{Type}:\text{Type}$. This can be seen using **Check**.

3.3 Inductive types

Inductive types: introduction. In order to explain the notion of inductive type we consider as an example the inductive type representing the natural numbers. Its definition is as follows:

```
Inductive nat : Set := 0 : nat | S : nat->nat.
```

This definition is already present in Coq, check this with `Check nat..` Let us comment on the definition. A new element of `Set` is declared with name `nat`. The constructors of `nat` are `0` and `S`. Because this is an inductive definition, the set `nat` is the smallest set that contains `0` and is closed under application of `S`, so `0` and `S` determine all elements of `nat`. The constructors of an inductive definition correspond to introduction rules. In this case, the introduction rules for `nat` can be seen as follows:

$$\frac{}{0 : \text{nat}} \quad \frac{p : \text{nat}}{(S p) : \text{nat}}$$

Recursive definitions. In Coq recursive functions can be defined using the `Fixpoint` construction. For non-recursive functions the `Definition` construction is used. In case a recursive function is defined where the input is of an inductive type, in the definition all constructors of the inductive type are considered in turn. This is done using the `match` construction. As an example we consider the definition of the function `plus : nat -> nat -> nat`:

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
| 0 => m
| S p => S (plus p m)
end.
```

Here the recursion is in the first argument of `plus`. In the recursive call, the argument `p` is strictly smaller than the original argument `n` which is `(S p)`.

The first line of the definition reads as follows: a recursive function `plus` is defined. It takes as input one argument of type `nat` which is bound to the parameter `n`. Then the output is a function of type `nat -> nat`. The output is the function that is on the second till the last line: it takes as input an argument of type `nat` which is bound to the argument `m`. Then the behaviour is described distinguishing cases: either `n` is `0` or `n` is of the form `(S p)`.

Inductive types: elimination. Elimination for `nat` corresponds to reasoning by cases: for an element `n` of `nat` there are two possibilities: either `n = 0` or `n = (S p)` for some `p` in `nat`. With the definition of an inductive type, automatically three terms that implement reasoning by cases are defined. In the case of `nat`, we have

```

nat_ind
nat_rec
nat_rect

```

Here we consider only the first one. Using the command `Check`, we find:

```

nat_ind : forall P : nat -> Prop,
          P 0 ->
          (forall n : nat, P n -> P (S n)) ->
          forall n : nat, P n

```

This type can be read as follows. For every property P of the natural numbers, if we have that

- the property P holds for 0, *and*
- the property P holds for n implies that the property P holds for the successor of n ,

then the property P holds for every natural number.

The tactic `elim` tries to apply `nat_ind`. That is, if the current goal G concerns an element n of `nat`, then `elim` tries to find an abstraction P of the current goal such that $Pn = G$. Then it applies `nat_ind P`. See also the use of `elim` in the following example.

Example. This example concerns the proof of an elementary property of the function `plus`. On the left is the ‘normal’ proof and on the right are the corresponding steps in Coq.

<p>Lemma. $\forall n \in \mathbb{N} : n = \text{plus } n \ 0$</p> <p>Let $n \in \mathbb{N}$. <i>To Prove:</i> $n = \text{plus } n \ 0$. induction on n.</p> <p><i>Basisstep</i> <i>To Prove:</i> $0 = \text{plus } 0 \ 0$. Definition of <code>plus</code> yields <code>plus 0 0 = 0</code>. <i>To Prove:</i> $0 = 0$.</p> <p><i>Inductionstep</i> <i>To Prove:</i> $\forall m \in \mathbb{N} :$ $m = \text{plus } m \ 0 \rightarrow$ $S m = \text{plus } (S m) \ 0$.</p> <p>Let $m \in \mathbb{N}$. Suppose that $m = \text{plus } m \ 0$. <i>To Prove:</i> $S m = \text{plus } (S m) \ 0$ Definition of <code>plus</code> yields <code>plus (S m) 0 = S(plus m 0)</code>. <i>To Prove:</i> $S m = S(\text{plus } m \ 0)$. By the induction hypothesis <code>plus m 0 = m</code>. <i>To Prove:</i> $S m = S m$.</p>	<p>Lemma <code>plus_n_0</code> :</p> <code>forall n : nat, n = plus n 0.</code> <p><code>intro n.</code> <i>Goal:</i> $n = \text{plus } n \ 0$. <code>elim n.</code></p> <p><i>Goal 1:</i> $0 = \text{plus } 0 \ 0$. <code>simpl.</code></p> <p><i>Goal 1:</i> $0 = 0$. <code>reflexivity.</code></p> <p><i>Goal:</i> <code>forall m : nat,</code> <code>m = plus m 0 -></code> <code>S m = plus (S m) 0).</code></p> <p><code>intro m.</code> <code>intro IH.</code> <i>Goal:</i> $S m = \text{plus } (S m) \ 0$. <code>simpl.</code></p> <p><i>Goal:</i> $S m = S(\text{plus } m \ 0)$. <code>rewrite <- IH.</code></p> <p><i>Goal:</i> $S m = S m$. <code>reflexivity.</code></p>
--	--

Prop and bool. `Prop` is the universe of the propositions in Coq. We have `True : Prop` and `False : Prop`. The proposition `True` denotes the class of terms representing proofs of `True`. It has one inhabitant. The proposition `False` denotes the class of terms representing proofs of `False`. It has no inhabitants since we do not have a proof of `False`. Both `True` and `False` are defined inductively:

```
Inductive True : Prop := I : True .
Inductive False : Prop := .
```

Further, `bool` is the inductive type of the booleans:

```
Inductive bool : Set := true : bool | false : bool .
```

We have `bool:Set`. Operations on elements of `Prop`, like for instance `not`, cannot be applied to elements of `bool`. If in doubt, use `Check`.

3.4 Coq

- **Inductive**

This is not the complete syntax! The construct `Inductive` is used to give an inductive definition of a set, as in the example of natural numbers. Use `Print nat.` and `Print bool.` to see examples of inductive types.

- `match ... with ... end.`

This construct is used to distinguish cases according to the definition of an inductive type. For instance, if `m` is of type `nat`, we can make a case distinction as follows:

```
match m with
| 0   => t1
| S p => t2
end.
```

Here the case distinction is: `m` is either `0` or of the form `(S p)`. The `|` is used to separate the cases, and `t1` and `t2` are terms.

- **Fixpoint**

This construct is used to give a recursive definition. (For a non-recursive definition, use `Definition`.) To illustrate its use we consider as an example an incomplete definition of `plus: nat->nat->nat`.

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
| 0 => m
| S p => S (plus p m)
end.
```

Here a recursive definition of the function `plus` taking as input two natural numbers is given; the recursion is in the *first* argument (indicated by the parameter `n`). An alternative definition is:

```
Fixpoint plus (n m : nat) {struct m} : nat :=
  match m with
  | 0 => n
  | S p => S (plus n p)
  end.
```

Here the recursion is on the *second* argument (indicated by the parameter `m`).

- `elim`

So far we have seen the use of `elim x` if `x` is the label of a hypothesis of the form $A \wedge B$ or $A \vee B$.

In fact then `elim x` applies either `and_ind` or `or_ind`.

In case that `x` is of some inductive type, say `t`, then `elim x` tries to apply `t_ind`. The result is that induction on `x` is started.

- `induction id` .

This tactic does `intros until id ; elim id`. In the example, instead of the two first steps

```
intro n .
elim .
```

we can use

```
induction n .
```

- `Eval compute in`

This is an abbreviation for `Eval cbv iota beta zeta delta in`. We will see more about this later on; for now: it can be used to compute the normal form of a term.

- `simpl` .

This tactic can be applied to any goal. The goal is simplified by performing $\beta\iota$ -reduction. This means that β -redexes are contracted, and also redexes for definitions like for instance the function `plus` are reduced.

- `rewrite id` . Let `id` be of type `forall (x1:A1) ... (xn:An), t1 = t2`, so some universal quantifications followed by an equality.

An application of `rewrite id` or equivalently `rewrite -> id` yields that in the goal all occurrences of `t1` are replaced by `t2`.

An application of `rewrite <- id` yields that in the goal all occurrences of `t2` are replaced by `t1`.

- `reflexivity` .

This tactic can be applied to a goal of the form `t=u`. It checks whether `t` and `u` are convertible (using $\beta\iota$ -reduction) and if this is the case solves the goal.

3.5 Inversion

The tactic `inversion` is, very roughly speaking, used to put a hypothesis in a form that is more useful. It can be used with argument *label* if *label* is the label of a hypothesis that is built from an inductive predicate. We consider two examples of the use of `inversion`.

In the first example we use the predicate `le` and a declared variable `P`:

```
Inductive le (n:nat) : nat -> Prop :=
  le_n : le n n
| le_S : forall m : nat, le n m -> le n (S m).
Variable P : nat -> nat -> Prop .
```

Suppose we want to prove an implication of the following form:

```
Lemma een : forall n m : nat, le (S n) m -> P n m .
```

We start with the necessary introductions. This yields:

```
1 subgoal

  n : nat
  m : nat
  H : le (S n) m
  =====
  P n m
```

Now `inversion H` generates two new goals, one for each of the ways in which `H` could have been derived from the definition of `le`:

```
2 subgoals

  n : nat
  m : nat
  H : le (S n) m
  H0 : S n = m
  =====
  P n (S n)
```

subgoal 2 is:

```
  n : nat
  m : nat
  H : le (S n) m
  m0 : nat
  H0 : le (S n) m0
  H1 : S m0 = m
  =====
  P n (S m0)
```

In the second example, we again use P and in addition the predicate `leq`:

```
Inductive leq : nat -> nat -> Prop :=
  leq_0 : forall m : nat, leq 0 m |
  leq_s : forall n m : nat, leq n m -> leq (S n) (S m) .
```

Suppose we wish to prove the following implication:

```
Lemma twee : forall n m : nat, leq (S n) m -> P n m .
```

We start with the necessary introductions. This yields:

```
1 subgoal
```

```
  n : nat
  m : nat
  H : leq (S n) m
  =====
  P n m
```

Now `inversion H` yields one new goal, since there was only one possible way in which H could have been concluded from the definition of `leq`:

```
1 subgoal
```

```
  n : nat
  m : nat
  H : leq (S n) m
  n0 : nat
  m0 : nat
  H1 : leq n m0
  H0 : n0 = n
  H2 : S m0 = m
  =====
  P n (S m0)
```