Natural Deduction Rules for Inductive Types

Reductions in Natural Deduction Proofs

Inductive Types

Induction from finite constructability:

If one proves $P(0)$, and $\forall x{:}\mathrm{Nat}\ P(x) \to P(\mathrm{succ}(x))$, then $P(x)$ is true of all natural numbers.

The intuitive reason for correctness is that every natural number is reachable from 0 through a finite number of steps.

Unfortunately, this explanation is useless from the formal point of view, because 'finite' means 'in a number of steps that can be counted by a natural number'.

Induction from $\Rightarrow$-smallest predicate(1):

Another way of justifying the induction priciple is through the notion of $\Rightarrow$ smallest predicate: Define $P \Rightarrow Q$ iff $\forall x{:}X\ P(x) \to Q(x)$.

$\Rightarrow$ can be interpreted as 'smaller than' in some sense.

then Nat is the $\Rightarrow$-smallest predicate for which $\mathrm{Nat}(0)$ and $\forall x\ \mathrm{Nat}(x) \to \mathrm{Nat}(\mathrm{succ}(x))$.

Then, if for some predicate S, one can prove $S(0)$ and $\forall x\ S(x) \to S(\mathrm{succ}(x))$, by $\Rightarrow$-minimality of Nat, we have $\mathrm{Nat} \Rightarrow S$, which means $\mathrm{Nat} \to S$, which implies $\forall x\ \ x \in \mathrm{Nat} \to x \in S$.

Induction from $\to$-smallest predicate(2):

Not all is said about the integers yet: $\subseteq$-minimality does not exclude the possibility that $\mathrm{succ}^i(x) = 0$, for some i.

This causes problems if one wants to define recursive subsets of Nat, (for example the even numbers), and to prove that certain numbers are not even.

Therefore, one needs to add distinction axioms.

Later, with the 'proofs as types paradigm', we will see that $\Rightarrow$ can be seen as the possibility to construct functions. $P \Rightarrow$ iff there exists (in intuitionistic sense) a function from P to Q.

This guarantees distinction.

Inductions Rule for Nat:

induction-Nat:

$$
\begin{array}{|l}
P[x := 0] \\
\ldots \\
\begin{array}{|l}
P[x := y] \\
\hline
\ldots \\
P[x := \mathrm{succ}(y)]
\end{array} \\
\forall x{:}\,\mathrm{Nat}\ P(x)
\end{array}
$$

$\forall$-elimination: (unchanged)

$$
\begin{array}{|l}
\forall x{:}\,\mathrm{Nat}\ P(x) \\
\ldots \\
P[x := t] \qquad (\ t \text{ must be of type Nat }\ )
\end{array}
$$

Difference Axioms for Nat:

$$
\begin{array}{|l}
\\
0 \not\approx \mathrm{succ}(t)
\end{array}
$$

$$
\begin{array}{|l}
t_1 \not\approx t_2 \\
\\
\mathrm{succ}(t_1) \not\approx \mathrm{succ}(t_2)
\end{array}
$$

Induction Rule for List:

list induction:

$$
\begin{array}{|l}
P[x := \mathrm{nil}] \\
\ldots \\
\begin{array}{|l}
P[x := y] \\
\hline
P[x := \mathrm{cons}(z, y)]
\end{array} \\
\forall x{:}\,\mathrm{List}\ P(x)
\end{array}
$$

y and z must be variables that are not used in assumptions.

$\forall$-elimination for List (unchanged)

$$
\begin{array}{|l}
\forall x{:}\,\mathrm{List}\ P(x) \\
\ldots \\
P[x := t] \qquad (\ t \text{ must be of type List})
\end{array}
$$

Difference Rules for List

$\text{nil} \not\approx \text{cons}(t_1, u_1)$

$t_1 \not\approx u_1$

$\text{cons}(t_1, u_1) \not\approx \text{cons}(t_2, u_2)$

$t_2 \not\approx u_2$

$\text{cons}(t_1, u_1) \not\approx \text{cons}(t_2, u_2)$

t_1, u_1, t_2, u_2 are arbitrary terms.

Induction Rule for Bool:

Bool is a finite type, consisting of **false** and **true**. In order to prove something for all Bool, it is enough to prove it for **false** and **true** :

$P[x := \text{false}\ ]$

$\ldots$

$P[x := \text{true}\ ]$

$\ldots$

$\forall x{:}\,\text{Bool}\ P(x)$

Difference Rule for Bool:

$\text{false} \not\approx \text{true}$

Induction Rules for $\leq$:

$\leq$-induction

$P[x := n]$

$\ldots$

$n \leq y$

$P[x := y]$

$\ldots$

$P[x := \text{succ}(y)]$

$\forall x{:}\,\text{Nat}\ (n \leq x) \rightarrow P(x)$

Introduction rules for $\leq$:

$$\begin{array}{|l} \ldots \\ n \leq n \end{array}$$

$$\begin{array}{|l} \ldots \\ n \leq m \\ \ldots \\ n \leq \mathrm{succ}(m) \end{array}$$

Elimination rule for $\leq$:

$$\begin{array}{|l} \ldots \\ n \leq t \\ \ldots \\ \forall x{:}\,\mathrm{Nat}\ (n \leq x) \to P(x) \\ \ldots \\ P(t) \end{array}$$

Induction Rules for Even:

$\forall$-even introduction:

$$\begin{array}{|l} P[x := 0] \\ \ldots \\ \begin{array}{|l} \mathrm{even}(x) \\ P[x := y] \\ \hline \ldots \\ P[x := \mathrm{succ}(\mathrm{succ}(y))] \end{array} \\ \forall x{:}\,\mathrm{Nat}\ \mathrm{even}(x) \to P(x) \end{array}$$

Introduction Rules for Even:

$$\begin{array}{|l} \ldots \\ \mathrm{even}(0) \end{array}$$

$$\begin{array}{|l} \ldots \\ \mathrm{even}(t) \\ \ldots \\ \mathrm{even}(\mathrm{succ}(\mathrm{succ}(t))) \end{array}$$

Elimination rule for Even:

$\ldots$

$\mathrm{even}(t)$

$\ldots$

$\forall x{:}\,\mathrm{Nat}\ \mathrm{even}(x) \to P(x)$

$\ldots$

$P(t)$

From the previous examples, we see that there are two types of inductive types.

1. Those defining a datatype (nat, list).
2. Those defining a predicate over an already existing type. (even, $\leq$).

For (1), one usually wants the set to be freely generated. This means that elements of the type with different derivations are different. In order to obtain this, difference rules for the datatypes were added.

For (2), there are explicit introduction rules. These rules correspond to the typing rules for datatypes. We did not give them for the datatypes (1), because we assumed they are implicit in the system.

General Pattern of Inductive Types

An inductive type is defined by a set of functions, which are called constructors.

(For Nat, the constructors are 0 and succ. For List, the constructors are nil and cons.

The induction rules correspond to the constructors. It is not to hard to understand this intuitively. However, it is very hard to give the general pattern (as an algorithm) to obtain the induction principle from the constructors. It is implemented in Coq.

The $\exists$-quantifier

We did not give elimination rules for $\exists$. In Coq, the existential quantifier is defined. It is defined as an inductive type. It is the smallest predicate that can be proven from witnesses. Actually, nearly all operators are defined as inductive types in Coq.

Before one can define $\exists$ like this, one first needs to assume the Formulas as Types-equivalence. We come to this later. For the moment, we continue without $\exists$-elimination rules.

If one uses induction on the simple datatypes, one will notice that it is extremely hard to prove simple properties.

For example: $\forall x, y{:}\,\mathrm{Nat}\ x + y \approx y + x$ from the axioms $\forall x{:}\,\mathrm{Nat}\ x + 0 \approx x$ and $\forall x, y{:}\,\mathrm{Nat}\ x + \mathrm{succ}(y) \approx \mathrm{succ}(x + y)$.

$\forall x, y, z{:}\,\mathrm{Nat}\ x \leq y \wedge y \leq z \rightarrow x \leq z$.
$\forall x, y{:}\,\mathrm{Nat}\ x \leq y \wedge y \leq x \rightarrow x \approx y$
$\forall x, y{:}\,\mathrm{Nat}\ x \leq y \vee y \leq x$.

Constructivity of Intuitionistic Logic: Reductions in Proofs

Intuitionistic Logic is interesting for computer science, because proofs have a computational meaning. The computational meaning of proofs is reflected by the reduction properties of natural deduction.

If one first introduces and a formula, and later eliminates the formula, then the resulting proof can be reduced.

Reduction rules for $\wedge$:

π (this is an abbreviation for the proof of A)
A
$\ldots$
B
$\ldots$
$A \wedge B$
$\ldots$
A

reduces into

π
A

reduction rules for $\vee$:

π
A
$\ldots$
$A \vee B$
| A
| ρ
| C

| B
| $\ldots$
| C

C

reduces into

π
A
ρ
C

Reduction rule for $\to$:

$$\begin{array}{|l}
\pi \\
A \\
\ldots \\
\begin{array}{|l}
A \\ \hline
\rho \\
B
\end{array} \\
A \to B \\
B
\end{array}
\qquad \text{reduces into:} \qquad
\begin{array}{|l}
\pi \\
A \\
\rho \\
B
\end{array}$$

Reduction rule for $\forall$:

$$\begin{array}{|l}
\pi(y) \quad \text{(abbreviation of the proof of } P[x := y] \text{)} \\
P[x := y] \\
\forall x{:}X \;\, P(x) \\
P[x := t]
\end{array}$$

reduces into

$$\begin{array}{|l}
\pi(t) \quad \text{(replay of } \pi(y) \text{ with } y \text{ replaced by } t\text{)} \\
P[x := t]
\end{array}$$

Reduction rule for $\exists$:

$$\begin{array}{|l}
\pi \\
P[x := t] \\
\ldots \\
\begin{array}{|l}
P[x := y] \\ \hline
\rho(y) \\
C
\end{array} \\
C
\end{array}
\qquad \text{reduces into} \qquad
\begin{array}{|l}
\pi \\
P[x := t] \\
\rho(t) \\
C
\end{array}$$

Reduction rule for induction over Nat (0):

$$\begin{array}{|l}
\pi \\
P[x := 0] \\
\ldots \\
\begin{array}{|l}
P[x := y] \\ \hline
\rho(y) \\
P[x := \mathrm{succ}(y)]
\end{array} \\
\forall x{:}\mathrm{Nat} \;\, P(x) \\
P[x := 0]
\end{array}
\qquad \Rightarrow \qquad
\begin{array}{|l}
\pi \\
P[x := 0]
\end{array}$$

Reduction rules for induction over Nat (succ):

$$\left|\begin{array}{l}\pi\\ P[x:=0]\\ \cdots\\ \left|\begin{array}{l} P[x:=y]\\ \hline \rho(y)\\ P[x:=\mathrm{succ}(y)]\end{array}\right.\\ \forall x{:}\,\mathrm{Nat}\ P(x)\\ P[x:=\mathrm{succ}(t)]\end{array}\right. \quad\Rightarrow\quad \left|\begin{array}{l}\pi\\ P[x:=0]\\ \cdots\\ \left|\begin{array}{l} P[x:=y]\\ \hline \rho(y)\\ P[x:=\mathrm{succ}(y)]\end{array}\right.\\ \forall x{:}\,\mathrm{Nat}\ P(x)\\ P[x:=t]\\ \rho(t)\\ P[x:=\mathrm{succ}(t)]\end{array}\right.$$

In case that t is an explicit term, i.e. has form $\mathrm{succ}^i(0)$, the proof will be completely expanded into a proof of form

$$\left|\begin{array}{l}\pi\\ P[x:=0]\\ \rho(0)\\ P[x:=\mathrm{succ}(0)]\\ \rho(1)\\ P[x:=\mathrm{succ}^2(0)]\\ \rho(2)\\ \cdots\\ P[x:=\mathrm{succ}^{i-1}(0)]\\ \rho(i-1)\\ P[x:=\mathrm{succ}^i(0)]\end{array}\right.$$

In this proof, the induction application has been recursively expanded until it disappeared.

Reduction rule for induction over List (nil)

$$\left|\begin{array}{l}\pi\\ P[x:=\mathrm{nil}]\\ \left|\begin{array}{l} P[x:=y]\\ \hline \rho(z,y)\\ P[x:=\mathrm{cons}(z,y)]\end{array}\right.\\ \forall x{:}\,\mathrm{List}\ P(x)\\ P[x:=\mathrm{nil}]\end{array}\right. \quad \text{reduces into} \quad \left|\begin{array}{l}\pi\\ P[x:=\mathrm{nil}]\end{array}\right.$$

Reduction rule for induction over List (cons)

$$\left|\begin{array}{l} \pi \\ P[x := \mathrm{nil}] \\ \left|\begin{array}{l} P[x := y] \\ \hline \rho(z, y) \\ P[x := \mathrm{cons}(z, y)] \end{array}\right. \\ \forall x\colon \mathrm{List}\ P(x) \\ P[x := \mathrm{cons}(t, u)] \end{array}\right. \quad \text{reduces into} \quad \left|\begin{array}{l} \pi \\ P[x := \mathrm{nil}] \\ \left|\begin{array}{l} P[x := y] \\ \hline \rho(z, y) \\ P[x := \mathrm{cons}(z, y)] \end{array}\right. \\ \forall x\colon \mathrm{List}\ P(x) \\ P[x := u] \\ \rho(t, u) \\ P[x := \mathrm{cons}(t, u)] \end{array}\right.$$

Reduction rule for induction over Bool

$$\left|\begin{array}{l} \pi \\ P[x := \mathrm{false}\ ] \\ \rho \\ P[x := \mathrm{true}\ ] \\ \ldots \\ \forall x\colon \mathrm{Bool}\ P(x) \\ P[x := \mathrm{false} \end{array}\right. \quad \text{reduces into} \quad \left|\begin{array}{l} \pi \\ P[x := \mathrm{false}] \end{array}\right.$$

The rule for true is analogeous.

Reduction rule for induction over $\leq$ (base):

$$\left|\begin{array}{l} \pi \\ P[x := n] \\ \ldots \\ \left|\begin{array}{l} n \leq y \\ P[x := y] \\ \hline \rho(y) \\ P[x := \mathrm{succ}(y)] \end{array}\right. \\ \forall x\colon \mathrm{Nat}\ (n \leq x) \to P(x) \\ n \leq n \\ P[x := n]. \end{array}\right. \quad \text{reduces into} \quad \left|\begin{array}{l} \pi \\ P[x := n] \end{array}\right.$$

Reduction rule for induction over $\leq$ (succ):

$$\left|\begin{array}{l} \pi \\ P[x := n] \\ \ldots \\ \left|\begin{array}{l} n \leq y \\ P[x := y] \\ \hline \rho(y) \\ P[x := \mathrm{succ}(y)] \end{array}\right. \\ \forall x\colon \mathrm{Nat}\ (n \leq x) \to P(x) \\ n \leq m \\ n \leq \mathrm{succ}(m) \\ P[x := \mathrm{succ}(m)]. \end{array}\right. \quad \text{reduces into} \quad \left|\begin{array}{l} \pi \\ P[x := n] \\ \ldots \\ \left|\begin{array}{l} n \leq y \\ P[x := y] \\ \hline \rho(y) \\ P[x := \mathrm{succ}(y)] \end{array}\right. \\ \forall x\colon \mathrm{Nat}\ (n \leq x) \to P(x) \\ n \leq m \\ P[x := m]. \\ \rho(m) \\ P[x := \mathrm{succ}(m)] \end{array}\right.$$

Constructiveness

Intuitively, constructiveness means that all existential statements can be justified by an example.

Classical logic does not have this property, as is demonstrated by:

$$\vdash A \lor \neg A, \text{ and}$$

$$p(0), \neg p(s(s(0))) \vdash \exists x{:}\, X\ p(x) \land \neg\, p(s(x)).$$

We will now explain this property in more detail:

Strong Normalization of Natural Deduction (Prawitz)

The reduction rules allow it to construct reduction sequences: $\pi_0 \Rightarrow \pi_1 \Rightarrow \cdots \Rightarrow \pi_n$. Write $\pi_0 \Rightarrow^* \pi_n$ if π_n can be obtained from π_0 through a finite number (possibly zero) of reductions.

The following properties can be shown:
THEOREM:

1. Every reduction sequence $\pi_0 \Rightarrow \pi_1 \Rightarrow \pi_2 \Rightarrow \cdots$ is finite. (When reducing proofs, one will eventually reach a proof that cannot be reduced anymore)
2. If $\pi \Rightarrow^* \rho_0$ and $\pi \Rightarrow^* \rho_1$, both ρ_0 and ρ_1 are in normal form, then $\rho_0 = \rho_1$ (ignoring inessential differences in the order)

The hardest part of this theorem consists of proving (strong) termination: No proof has an infinite reduction sequence. This proof is hard, because the proof may actually get more complicated during a reduction when an induction application is reduced against an instantiation.

Uniqueness of the normal form can be shown by proving the Church-Rosser property: If $A \Rightarrow B_1$ and $A \Rightarrow B_2$, then there is a C, for which $B_1 \Rightarrow^* C$ and $B_2 \Rightarrow^* C$.

The Church-Rossor property is shown by checking that all reductions are independent, i.e. take place at different positions in the proof.

THEOREM: Let π be a natural deduction proof in which none of the premisses contains a positively occurring $\exists$ or $\lor$.

If the conclusion of the proof is of form $\exists x{:}\, X\ p(x)$, then there is a term t, for which $p(t)$ can be proven from the same premisses.

If the conclusion of the proof has form $A \lor B$, then either A or B can be proven from the same premisses.

(Note that the theorem can be repeated: If one has a proof of $\exists x{:}\, X\ p(x) \lor q(x)$, then there is a term t for which either $p(t)$ or $q(t)$ is provable.

A consequence of the theorem is that proofs are algorithms: Suppose that we have a proof of $\forall x{:}X\ \exists y{:}Y\ p(x,y)$ in which none of the premisses contains a $\vee$ or $\exists$. If t has type X, then the proof can be extended into a proof of $y{:}Y\ p(t,y)$ through $\forall$-elimination. Because of the theorem, there is a u, such that $p(t,u)$ is provable.

In order to prove the theorem, first assume that the proof is in normal form. Then the following sequence induction hypotheses needs to be proven by induction on the length of the (sub)proof.

1. Every formula of form $A \rightarrow B$, that does not positively occur in the premisses was obtained by $\rightarrow$-introduction, equality replacement or $\perp$-elimination.
2. Every formula of form $\forall x{:}X\ p(x)$ that does not positively occur in the premisses, was obtained through $\forall$-introduction, equality replacement or $\perp$-elimination.
3. Every formula of form $A \wedge B$ that does not positively occur in one of the premisses, was obtained through $\wedge$-introduction, equality replacement or $\perp$-elimination.
4. Every formula of form $A \vee B$ that does not positively occur in one of the premisses was obtained through $\vee$-introduction, equality replacement or $\perp$-elimination.
5. Every formula of form $\exists x{:}X\ p(x)$ that does not positively occur in one of the premisses, was obtained through $\exists$-introduction, equality replacement or $\perp$-elimination.

The induction hypotheses can be equivalently formulated as follows:

1. Every formula of form $A \rightarrow B$, that does not positively occur in the premisses, was obtained, possibly through a sequence of equality replacements, by $\rightarrow$-introduction, or $\perp$-elimination.
2. Every formula of form $\forall x{:}X\ p(x)$ that does not positively occur in the premisses, was obtained by $\forall$-introduction or $\perp$-elimination, possibly through a sequence of equality replacements.
3. Every formula of form $A \wedge B$ that ..., was obtained by $\wedge$-introduction or ...
4. Every formula of form $A \vee B$ that ..., was obtained by $\vee$-introduction or ...
5. Every formula of form $\exists x{:}X\ p(x)$ that ... , was obtained through $\exists$-introduction or ...

Proof (for 3):

We check the possible rules through wich $A \wedge B$ was obtained. If $A\wedge$ was obtained through $\wedge$-introduction, $\approx$-replacement, or $\perp$-introduction, then everything is fine. Otherwise, the following possibilities remain:

$A \wedge B$ is obtained by $\wedge$-elimination as follows:

$(A \wedge B) \wedge C$

$\ldots$

$(A \wedge B)$

Because $A \wedge B$ does not occur in a premisse, neither does $(A \wedge B) \wedge C$. Since it occurs earlier in the proof, the induction hypotheses can be applied on it. Therefore, $(A \wedge B) \wedge C$ can be traced back through a sequence of equality replacements to a formula $(A_1 \wedge B_1) \wedge C_1$, which is introduced by either by $\wedge$-introduction, or $\perp$-elimination.

If $(A_1 \wedge B_1) \wedge C_1$ is obtained by $\wedge$-introduction, then one can derive $A_1 \wedge B_1$, and apply the relevant equality replacements on it, to obtain $A \wedge B$. The resulting proof can be further normalized.

If $(A_1 \wedge B_1) \wedge C_1$ is obtained by $\perp$-elimination, one can directly derive $A \wedge B$.

$A \wedge B$ is obtained by $\rightarrow$-elimination as follows:

$C \rightarrow (A \wedge B)$

$\ldots$

$(A \wedge B)$

Because $A \wedge B$ does not occur positively in a premise, neither does $C \rightarrow (A \wedge B)$. Since it occurs earlier in the proof, the induction hypothesis can be applied on it. Therefore, $C \rightarrow (A \wedge B)$ can be traced back through a sequence of equality replacements to a formula $C_1 \rightarrow (A_1 \wedge B_1)$, which is introduced either by $\rightarrow$-introduction, or $\perp$-elimination. If $C_1 \rightarrow (A_1 \wedge B_1)$ is obtained by $\rightarrow$-