Proving Properties of Even Numbers

Induction leads to two cases:

- **rule:** $0 \in \mathtt{Ev}$

  $\mathtt{1.}\ 0 + 0 \in \mathtt{Ev}$ case $\mathtt{m} = 0$

- **rule:** $\mathtt{n} \in \mathtt{Ev} \Longrightarrow \mathtt{n} + 2 \in \mathtt{Ev}$

  z $\mathtt{2.}\ \Lambda\mathtt{n}.[\![\mathtt{n} \in \mathtt{Ev};\ \mathtt{n} + \mathtt{n} \in \mathtt{Ev}]\!] \Longrightarrow \mathtt{Suc(Sucn)} + \mathtt{Suc(Sucn)} \in \mathtt{Ev}$

  case $\mathtt{m} = \mathtt{n} + 2$

Rule Induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ \ n$$

by *rule induction* on $\mathtt{n} \in \mathtt{Ev}$ we must prove

- $\mathtt{P}\ \ 0$
- $P\ \ n \Longrightarrow P(n+2)$

Uses rule `Ev.induct`:

$$[\![\mathtt{n} \in \mathtt{Ev};\ \ \mathtt{P}\ \ 0;\ \ \Lambda\mathtt{n}.\ \ \mathtt{P}\ \ \mathtt{n} \Longrightarrow \mathtt{P}(\mathtt{n}+2)]\!] \Longrightarrow \mathtt{P}\ \ \mathtt{n}$$

An elimination rule

Rule Induction in General

Set $\mathtt{S}$ is defined inductively. To prove

$$\mathtt{x} \in \mathtt{S} \Longrightarrow \mathtt{P}\ \mathtt{x}$$

by *rule induction* on $x \in S$ we must prove for every rule

$$[\![\mathtt{a}_1 \in \mathtt{S};\ \ldots;\ \mathtt{a}_n \in \mathtt{S}]\!] \Longrightarrow \mathtt{a} \in \mathtt{S}$$

that $\mathtt{P}$ is preserved:

$$[\![\mathtt{P}\ \mathtt{a}_1;\ \ldots;\ \mathtt{P}\ \mathtt{a}_n]\!] \Longrightarrow \mathtt{P}\ \mathtt{a}$$

In Isabelle/HOL:

`proof(rule S.induct)`

or

`apply(erule S.induct)`

Demo: Inductive Set Definition

Demo: Evens are infinite

Format for Inductive Relations Definitions

inductive R :: "$\tau \longrightarrow$ bool" **where**

$[\![\mathtt{R}(\mathtt{a}_{1,1});\ldots;\ \mathtt{R}(\mathtt{a}_{1,n});\mathtt{A}_{1,1};\ \ldots;\ \mathtt{A}_{1,k}]\!] \Longrightarrow \mathtt{R}(\mathtt{a}_1)\ \ |$

$\ldots\ \ |$

$[\![\mathtt{R}(\mathtt{a}_{m,1});\ldots;\ \mathtt{R}(\mathtt{a}_{m,1});\mathtt{A}_{m,1};\ \ldots;\ \mathtt{A}_{m,j}]\!] \Longrightarrow \mathtt{R}(\mathtt{a}_m)$

where $\mathtt{A}_{i,j}$ are side conditions not involving $\mathtt{R}$.

Format for Inductive Relations Definitions

inductive $\mathtt{R} :: \text{“}\tau \longrightarrow \mathtt{bool}\text{''}$ where

$\llbracket \mathtt{R}(\mathtt{a}_{1,1}); \ldots; \mathtt{R}(\mathtt{a}_{1,n}); \mathtt{A}_{1,1}; \ldots; \mathtt{A}_{1,k} \rrbracket \Longrightarrow \mathtt{R}(\mathtt{a}_1) \mid$

$\ldots \mid$

$\llbracket \mathtt{R}(\mathtt{a}_{m,1}); \ldots; \mathtt{R}(\mathtt{a}_{m,1}); \mathtt{A}_{m,1}; \ldots; \mathtt{A}_{m,j} \rrbracket \Longrightarrow \mathtt{R}(\mathtt{a}_m)$

where $A_{i,j}$ are side conditions not involving R.

Format for Mutual Inductive Relations Definitions

inductive

$\mathtt{R}_1 :: \text{“}\tau_1 \longrightarrow \mathtt{bool}\text{''}$ and

...

$\mathtt{R}_n :: \text{“}\tau_n \longrightarrow \mathtt{bool}\text{''}$ where

$\llbracket \mathtt{R}_i(\mathtt{a}_{1,1}); \ldots; \mathtt{R}_j(\mathtt{a}_{1,n}); \mathtt{A}_{1,1}; \ldots; \mathtt{A}_{1,k} \rrbracket \Longrightarrow \mathtt{R}_k(\mathtt{a}_1) \mid$

$\ldots \mid$

$\llbracket \mathtt{R}_m(\mathtt{a}_{m,1}); \ldots; \mathtt{R}_n(\mathtt{a}_{m,1}); \mathtt{A}_{m,1}; \ldots; \mathtt{A}_{m,j} \rrbracket \Longrightarrow \mathtt{R}_p(\mathtt{a}_m)$

where $\mathtt{A}_{i,j}$ are side conditions not involving any $\mathtt{R}_k$.

Example with Mutual Recursion

```
inductive
Even ::  "nat ⇒ bool" and
Odd :: "nat ⇒ bool" where
ZeroEven [intro!]:  "Even 0" |
OddOne [intro!]:  "Odd (Suc 0)" |
OddSucEven [intro]:  "Odd n ⟹ Even (Suc n)" |
EvenSucOdd [intro]:  "Even n ⟹ Odd (Suc n)"
```


General Recursive Functions: fun

Example:

```
fun fib :: "nat ⇒ nat" where
   "fib 0 = 0" |
   "fib 1 = 1" |
   "fib (Suc(Suc x)) = (fib x + fib (Suc x))"
```

Not primitive recursive because of `fib(Suc(Suc x))` on left, and because of `fib(Suc x)` on right.

fun: Rules of Use

Compared to `primrec`, very few restrictions:

- Can be used to define functions over any type
- Clauses in `fun` must be equations
- Left-hand side is function being defined applied to terms built from data constructors, distinct variables and wildcards
- Right-hand side is a expression made from the function being defined, the variables in the argument on the left, and previously defined terms
- If clauses overlap, first takes precedence.
- Calculates a measure from lexicographic ordering of some collection of arguments

Example: sep

Define a function for putting a separator between all adjacent elements in a list:

```
fun sep ::  "'a * 'a list => 'a list" where
 "sep(a, []) = []" |
 "sep(a, [x]) = [x]" |
 "sep(a, x#y#zs) = x # a # sep(a,y#zs)"
```