Proving Properties of Even Numbers

Induction leads to two cases:

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

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

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

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

  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`:

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

An elimination rule

Rule Induction in General

Set S is defined inductively. To prove

$$x \in S \Longrightarrow P\ x$$

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

$$[\![a_1 \in S;\ \ldots;\ a_n \in S]\!] \Longrightarrow a \in S$$

that P is preserved:

$$[\![P\ a_1;\ \ldots;\ P\ a_n]\!] \Longrightarrow P\ a$$

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

$\texttt{inductive R :: ``}\tau \longrightarrow \texttt{bool''\ where}$

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

$\ldots\ |$

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

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

Format for Inductive Relations Definitions

$$\texttt{inductive R :: ``}\tau \longrightarrow \texttt{bool''\ where}$$
$$\quad [\![\texttt{R}(\texttt{a}_{1,1}); \ldots; \texttt{R}(\texttt{a}_{1,\texttt{n}}); \texttt{A}_{1,1}; \ldots; \texttt{A}_{1,\texttt{k}}]\!] \Longrightarrow \texttt{R}(\texttt{a}_1) \mid$$
$$\quad \ldots \mid$$
$$\quad [\![\texttt{R}(\texttt{a}_{\texttt{m},1}); \ldots; \texttt{R}(\texttt{a}_{\texttt{m},\texttt{l}}); \texttt{A}_{\texttt{m},1}; \ldots; \texttt{A}_{\texttt{m},\texttt{j}}]\!] \Longrightarrow \texttt{R}(\texttt{a}_\texttt{m})$$

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

Format for Mutual Inductive Relations Definitions

$$
\begin{array}{l}
\texttt{inductive} \\
\mathtt{R_1} :: \text{“}\tau_1 \longrightarrow \texttt{bool}\text{”}\ \texttt{and} \\
\ldots \\
\mathtt{R_n} :: \text{“}\tau_n \longrightarrow \texttt{bool}\text{”}\ \texttt{where} \\
\quad [\![\mathtt{R_i}(\mathtt{a_{1,1}}); \ldots;\ \mathtt{R_j}(\mathtt{a_{1,n}}); \mathtt{A_{1,1}};\ \ldots;\ \mathtt{A_{1,k}}]\!] \Longrightarrow \mathtt{R_k}(\mathtt{a_1})\ \mid \\
\qquad \ldots\ \mid \\
\quad [\![\mathtt{R_m}(\mathtt{a_{m,1}}); \ldots;\ \mathtt{R_n}(\mathtt{a_{m,l}}); \mathtt{A_{m,1}};\ \ldots;\ \mathtt{A_{m,j}}]\!] \Longrightarrow \mathtt{R_p}(\mathtt{a_m})
\end{array}
$$

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)"
```