
An introduction to recursion and induction

A recursive datatype: toy lists

datatype *'a list* = *Nil* | *Cons* *'a* *"'a list"*

Nil: empty list

Cons x xs: head *x*, tail *xs*

A recursive datatype: toy lists

`datatype 'a list = Nil | Cons 'a "'a list"`

Nil: empty list

Cons x xs: head *x*, tail *xs*

A toy list: *Cons False (Cons True Nil)*

A recursive datatype: toy lists

`datatype 'a list = Nil | Cons 'a "'a list"`

Nil: empty list

Cons x xs: head x, tail xs

A toy list: `Cons False (Cons True Nil)`

Predefined lists: `[False, True]`

Structural induction on lists

P xs holds for all lists xs if

Structural induction on lists

$P\ xs$ holds for all lists xs if

- $P\ Nil$

Structural induction on lists

$P\ xs$ holds for all lists xs if

- $P\ Nil$
- and for arbitrary x and xs , $P\ xs$ implies $P\ (Cons\ x\ xs)$

A recursive function: append

Declaration

```
consts app :: "'a list ⇒ 'a list ⇒ 'a list"
```

A recursive function: append

Declaration

`consts app :: "'a list ⇒ 'a list ⇒ 'a list"`

and definition by *primitive recursion*:

primrec

`"app Nil ys = ?"`

`"app (Cons x xs) ys = ??"`

A recursive function: append

Declaration

`consts app :: "'a list ⇒ 'a list ⇒ 'a list"`

and definition by *primitive recursion*:

primrec

`"app Nil ys = ?"`

`"app (Cons x xs) ys = ??"`

1 rule per constructor

Recursive calls must drop the constructor \implies Termination

Demo: append and reverse

Proofs

General schema:

```
lemma name : " . . . "  
apply ( . . . )  
apply ( . . . )  
⋮  
done
```

If the lemma is suitable as a simplification rule:

```
lemma name[simp] : " . . . "
```

Proof methods

- **Structural induction**
 - Format: *(induct x)*
x must be a free variable in the first subgoal.
The type of x must be a datatype.
 - Effect: generates 1 new subgoal per constructor
- **Simplification and a bit of logic**
 - Format: *auto*
 - Effect: tries to solve as many subgoals as possible using simplification and basic logical reasoning.