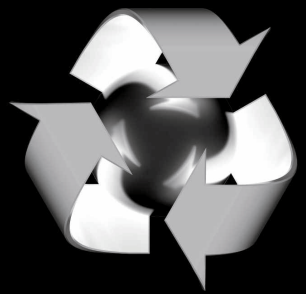
COMP4161: Advanced Topics in Software Verification



Gerwin Klein, Johannes Åman Pohjola, Christine Rizkallah, Miki Tanaka

T3/2020

Content

- ➜ Foundations & Principles
  - Intro, Lambda calculus, natural deduction [1,2]
  - Higher Order Logic, Isar (part 1) [2,3[a]]
  - Term rewriting [3,4]
- ➜ Proof & Specification Techniques
  - Inductively defined sets, rule induction, datatype induction, primitive recursion [4,5]
  - General recursive functions, termination proofs [7[b]]
  - Proof automation, Hoare logic, proofs about programs, invariants [8]
  - C verification [9,10]
  - Practice, questions, examp prep [10[c]]

[a] a1 due; [b] a2 due; [c] a3 due

Datatypes

Example:

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

Properties:

Datatypes

Example:

$$\textbf{datatype}\ \text{'a list} = \text{Nil} \mid \text{Cons 'a "'a list"}$$

Properties:

- ➜ Constructors:

$$\begin{array}{lcl} \text{Nil} & :: & \text{'a list} \\ \text{Cons} & :: & \text{'a} \Rightarrow \text{'a list} \Rightarrow \text{'a list} \end{array}$$

Datatypes

Example:

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

Properties:

- ➜ Constructors:

$$\begin{array}{lll} \text{Nil} & :: & \text{'a list} \\ \text{Cons} & :: & \text{'a} \Rightarrow \text{'a list} \Rightarrow \text{'a list} \end{array}$$

- ➜ Distinctness: $\text{Nil} \neq \text{Cons x xs}$

Datatypes

Example:

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

Properties:

- ➜ Constructors:

$$\begin{array}{lcl} \text{Nil} & :: & \text{'a list} \\ \text{Cons} & :: & \text{'a} \Rightarrow \text{'a list} \Rightarrow \text{'a list} \end{array}$$

- ➜ Distinctness: $\text{Nil} \neq \text{Cons x xs}$
- ➜ Injectivity: $(\text{Cons x xs} = \text{Cons y ys}) = (\text{x} = \text{y} \wedge \text{xs} = \text{ys})$

More Examples

Enumeration:

$$\textbf{datatype}\ \text{answer} = \text{Yes} \mid \text{No} \mid \text{Maybe}$$

More Examples

Enumeration:

datatype answer = Yes | No | Maybe

Polymorphic:

datatype 'a option = None | Some 'a

datatype ('a,'b,'c) triple = Triple 'a 'b 'c

More Examples

Enumeration:

datatype answer = Yes | No | Maybe

Polymorphic:

datatype 'a option = None | Some 'a
datatype ('a,'b,'c) triple = Triple 'a 'b 'c

Recursion:

datatype 'a list =

More Examples

Enumeration:

datatype answer = Yes | No | Maybe

Polymorphic:

datatype 'a option = None | Some 'a
datatype ('a,'b,'c) triple = Triple 'a 'b 'c

Recursion:

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

More Examples

Enumeration:

```
datatype answer = Yes | No | Maybe
```

Polymorphic:

```
datatype 'a option = None | Some 'a
datatype ('a,'b,'c) triple = Triple 'a 'b 'c
```

Recursion:

```
datatype 'a list =   Nil | Cons 'a "'a list"
datatype 'a tree =   Tip | Node 'a "'a tree" "'a tree"
```

More Examples

Enumeration:

```
datatype answer = Yes | No | Maybe
```

Polymorphic:

```
datatype 'a option = None | Some 'a
datatype ('a,'b,'c) triple = Triple 'a 'b 'c
```

Recursion:

```
datatype 'a list =   Nil | Cons 'a "'a list"
datatype 'a tree =   Tip | Node 'a "'a tree" "'a tree"
```

Mutual Recursion:

```
datatype even =
and odd =
```

More Examples

Enumeration:

```
datatype answer = Yes | No | Maybe
```

Polymorphic:

```
datatype 'a option = None | Some 'a
datatype ('a,'b,'c) triple = Triple 'a 'b 'c
```

Recursion:

```
datatype 'a list =   Nil | Cons 'a "'a list"
datatype 'a tree =   Tip | Node 'a "'a tree" "'a tree"
```

Mutual Recursion:

```
datatype even =  EvenZero | EvenSucc odd
and odd =  OddSucc even
```

Nested

Nested recursion:

datatype 'a tree = Tip | Node 'a "'a tree list"

datatype 'a tree = Tip | Node 'a "'a tree option" "'a tree option"

Nested

Nested recursion:

datatype 'a tree = Tip | Node 'a "'a tree list"

datatype 'a tree = Tip | Node 'a "'a tree option" "'a tree option"

➜ Recursive call is under a type constructor.

The General Case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \begin{array}{l} \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1} \\ \mid\ \ldots \\ \mid\ \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k} \end{array}$$

The General Case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \begin{array}{l} \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1} \\ \mid\ \ldots \\ \mid\ \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k} \end{array}$$

➜ Constructors: $\mathsf{C}_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau$

The General Case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \begin{array}{l} \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1} \\ \mid\ \ldots \\ \mid\ \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k} \end{array}$$

- ➜ Constructors: $\mathsf{C}_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau$
- ➜ Distinctness: $\mathsf{C}_i\ \ldots \neq \mathsf{C}_j\ \ldots$ if $i \neq j$

 CC-BY-4.0 License

The General Case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \begin{array}{l} \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1} \\ \mid\ \ldots \\ \mid\ \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k} \end{array}$$

- ➜ Constructors: $\mathsf{C}_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau$
- ➜ Distinctness: $\mathsf{C}_i\ \ldots \neq \mathsf{C}_j\ \ldots$ if $i \neq j$
- ➜ Injectivity: $(\mathsf{C}_i\ x_1 \ldots x_{n_i} = \mathsf{C}_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

 CC-BY-4.0 License

The General Case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \begin{array}{l} \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1} \\ \mid\ \ldots \\ \mid\ \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k} \end{array}$$

- ➜ Constructors: $\mathsf{C}_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau$
- ➜ Distinctness: $\mathsf{C}_i\ \ldots \neq \mathsf{C}_j\ \ldots$ if $i \neq j$
- ➜ Injectivity: $(\mathsf{C}_i\ x_1 \ldots x_{n_i} = \mathsf{C}_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity applied automatically

How is this Type Defined?

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

➜ internally reduced to a single constructor, using product and sum

How is this Type Defined?

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

- ➜ internally reduced to a single constructor, using product and sum
- ➜ constructor defined as an inductive set (like typedef)

How is this Type Defined?

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

- ➜ internally reduced to a single constructor, using product and sum
- ➜ constructor defined as an inductive set (like typedef)
- ➜ recursion: least fixpoint

How is this Type Defined?

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

- ➜ internally reduced to a single constructor, using product and sum
- ➜ constructor defined as an inductive set (like typedef)
- ➜ recursion: least fixpoint

More detail: Tutorial on Datatype in Isabelle documentation

Datatype Limitations

Must be definable as set.

Datatype Limitations

Must be definable as set.

➜ Infinitely branching ok.

Datatype Limitations

Must be definable as set.

- ➜ Infinitely branching ok.
- ➜ Mutually recursive ok.

Datatype Limitations

Must be definable as set.

- ➜ Infinitely branching ok.
- ➜ Mutually recursive ok.
- ➜ Strictly positive (right of function arrow) occurrence ok.

Datatype Limitations

Must be definable as set.

- ➜ Infinitely branching ok.
- ➜ Mutually recursive ok.
- ➜ Strictly positive (right of function arrow) occurrence ok.

Not ok:

$$\textbf{datatype}\ t \quad = \quad C\ (t \Rightarrow \text{bool})$$

Datatype Limitations

Must be definable as set.

- ➜ Infinitely branching ok.
- ➜ Mutually recursive ok.
- ➜ Strictly positive (right of function arrow) occurrence ok.

Not ok:

$$\begin{array}{lcl} \textbf{datatype}\ t & = & C\ (t \Rightarrow \mathsf{bool}) \\ & | & D\ ((\mathsf{bool} \Rightarrow t) \Rightarrow \mathsf{bool}) \end{array}$$

Datatype Limitations

Must be definable as set.

- ➜ Infinitely branching ok.
- ➜ Mutually recursive ok.
- ➜ Strictly positive (right of function arrow) occurrence ok.

Not ok:

$$\begin{array}{lcl} \textbf{datatype}\ \mathsf{t} & = & \mathsf{C}\ (\mathsf{t} \Rightarrow \mathsf{bool}) \\ & | & \mathsf{D}\ ((\mathsf{bool} \Rightarrow \mathsf{t}) \Rightarrow \mathsf{bool}) \\ & | & \mathsf{E}\ ((\mathsf{t} \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}) \end{array}$$

Because: Cantor's theorem (α set is larger than α)

Datatype Limitations

Not ok (nested recursion):

datatype ('a, 'b) fun_copy = Fun "'a ⇒ 'b"

datatype 'a t = F "('a t, 'a) fun_copy"

Datatype Limitations

Not ok (nested recursion):

datatype ('a, 'b) fun_copy = Fun "'a ⇒ 'b"

datatype 'a t = F "('a t, 'a) fun_copy"

➜ recursion in ('a1, ...,'an) t is only allowed on a subset of 'a1 ... 'an

➜ these arguments are called *live* arguments

Datatype Limitations

Not ok (nested recursion):

datatype ('a, 'b) fun_copy = Fun "'a ⇒ 'b"

datatype 'a t = F "('a t, 'a) fun_copy"

- ➜ recursion in ('a1, ...,'an) t is only allowed on a subset of 'a1 ... 'an
- ➜ these arguments are called *live* arguments
- ➜ Mainly: in "'a ⇒ 'b", 'a is dead and 'b is live
- ➜ Thus: in ('a, 'b) fun_copy, 'a is dead and 'b is live

Datatype Limitations

Not ok (nested recursion):

datatype ('a, 'b) fun_copy = Fun "'a ⇒ 'b"

datatype 'a t = F "('a t, 'a) fun_copy"

- ➜ recursion in ('a1, ...,'an) t is only allowed on a subset of 'a1 ... 'an
- ➜ these arguments are called *live* arguments
- ➜ Mainly: in "'a ⇒ 'b", 'a is dead and 'b is live
- ➜ Thus: in ('a, 'b) fun_copy, 'a is dead and 'b is live

- ➜ type constructors must be registered as *BNFs** to have live arguments
- ➜ BNF defines well-behaved type constructors, ie where recursion is allowed

* BNF = Bounded Natural Functors.

Datatype Limitations

Not ok (nested recursion):

datatype ('a, 'b) fun_copy = Fun "'a ⇒ 'b"

datatype 'a t = F "('a t, 'a) fun_copy"

- ➜ recursion in ('a1, ...,'an) t is only allowed on a subset of 'a1 ... 'an
- ➜ these arguments are called *live* arguments
- ➜ Mainly: in "'a ⇒ 'b", 'a is dead and 'b is live
- ➜ Thus: in ('a, 'b) fun_copy, 'a is dead and 'b is live

- ➜ type constructors must be registered as *BNFs** to have live arguments
- ➜ BNF defines well-behaved type constructors, ie where recursion is allowed
- ➜ datatypes automatically are BNFs (that's how they are constructed)

* BNF = Bounded Natural Functors.

Datatype Limitations

Not ok (nested recursion):

datatype ('a, 'b) fun_copy = Fun "'a ⇒ 'b"

datatype 'a t = F "('a t, 'a) fun_copy"

- ➜ recursion in ('a1, ...,'an) t is only allowed on a subset of 'a1 ... 'an
- ➜ these arguments are called *live* arguments
- ➜ Mainly: in "'a ⇒ 'b", 'a is dead and 'b is live
- ➜ Thus: in ('a, 'b) fun_copy, 'a is dead and 'b is live

- ➜ type constructors must be registered as *BNFs** to have live arguments
- ➜ BNF defines well-behaved type constructors, ie where recursion is allowed
- ➜ datatypes automatically are BNFs (that's how they are constructed)
- ➜ can register other type constructors as BNFs — not covered here**

* BNF = Bounded Natural Functors.

** *Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL*

Case

Every datatype introduces a **case** construct, e.g.

$$(\text{case } xs \text{ of } [] \Rightarrow \ldots \mid y \ \#ys \Rightarrow \ldots \ y \ \ldots \ ys \ \ldots)$$

Case

Every datatype introduces a **case** construct, e.g.

$$(\text{case } xs \text{ of } [] \Rightarrow \ldots \mid y \ \#ys \Rightarrow \ldots \ y \ \ldots \ ys \ \ldots)$$

In general: one case per constructor

Case

Every datatype introduces a **case** construct, e.g.

$$(\text{case } xs \text{ of } [] \Rightarrow \ldots \mid y \ \#ys \Rightarrow \ldots \ y \ \ldots \ ys \ \ldots)$$

In general: one case per constructor

➜ Nested patterns allowed: $x\#y\#zs$

Case

Every datatype introduces a **case** construct, e.g.

$$(\text{case } xs \text{ of } [] \Rightarrow \ldots \mid y \ \#ys \Rightarrow \ldots \ y \ \ldots \ ys \ \ldots)$$

In general: one case per constructor

- ➜ Nested patterns allowed: $x\#y\#zs$
- ➜ Dummy and default patterns with _

Case

Every datatype introduces a **case** construct, e.g.

$$(\text{case } xs \text{ of } [] \Rightarrow \ldots \mid y \ \#ys \Rightarrow \ldots \ y \ \ldots \ ys \ \ldots)$$

In general: one case per constructor

- ➜ Nested patterns allowed: $x\#y\#zs$
- ➜ Dummy and default patterns with _
- ➜ Binds weakly, needs () in context

Cases

apply (case_tac t)

Cases

apply (case_tac t)

creates k subgoals

$$[\![t = C_i\ x_1 \ldots x_p; \ldots]\!] \Longrightarrow \ldots$$

one for each constructor C_i

Demo

Recursion

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow$$

$$0 = 1$$

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow$$

$$0 = 1$$

! All functions in HOL must be total !

Primitive Recursion

primrec guarantees termination structurally

Example primrec def:

Primitive Recursion

primrec guarantees termination structurally

Example primrec def:

```
primrec  app :: "'a list ⇒ 'a list ⇒ 'a list"
where
"app Nil ys = ys" |
"app (Cons x xs) ys = Cons x (app xs ys)"
```

The General Case

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then $f :: \tau \Rightarrow \tau'$ can be defined by **primitive recursion**:

$$\begin{array}{lcl} f\ (C_1\ y_{1,1}\ \ldots\ y_{1,n_1}) & = & r_1 \\ \vdots & & \\ f\ (C_k\ y_{k,1}\ \ldots\ y_{k,n_k}) & = & r_k \end{array}$$

The General Case

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then $f :: \tau \Rightarrow \tau'$ can be defined by **primitive recursion**:

$$\begin{array}{lcl} f\ (C_1\ y_{1,1}\ \ldots\ y_{1,n_1}) & = & r_1 \\ \vdots & & \\ f\ (C_k\ y_{k,1}\ \ldots\ y_{k,n_k}) & = & r_k \end{array}$$

The recursive calls in r_i must be **structurally smaller**
(of the form $f\ a_1\ \ldots\ y_{i,j}\ \ldots\ a_p$)

How does this Work?

primrec just fancy syntax for a **recursion operator**

Example:

How does this Work?

primrec just fancy syntax for a **recursion operator**

Example:

$$\begin{aligned}
&\text{list_rec} :: \text{''}'b \Rightarrow ('a \Rightarrow 'a\ \text{list} \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ \text{list} \Rightarrow 'b\text{''} \\
&\text{list_rec}\ f_1\ f_2\ \text{Nil} &&= && f_1 \\
&\text{list_rec}\ f_1\ f_2\ (\text{Cons}\ x\ xs) &&= && f_2\ x\ xs\ (\text{list_rec}\ f_1\ f_2\ xs)
\end{aligned}$$

How does this Work?

primrec just fancy syntax for a **recursion operator**

Example:

list_rec :: "'b ⇒ ('a ⇒ 'a list ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b"

list_rec f_1 f_2 Nil = f_1

list_rec f_1 f_2 (Cons x xs) = f_2 x xs (list_rec f_1 f_2 xs)

app ≡ list_rec ($\lambda ys.\ ys$) ($\lambda x\ xs\ xs'.\ \lambda ys.$ Cons x (xs' ys))

primrec app :: "'a list ⇒ 'a list ⇒ 'a list"
where
"app Nil ys = ys" |
"app (Cons x xs) ys = Cons x (app xs ys)"

list_rec

Defined: automatically, first inductively (set), then by epsilon

list_rec

Defined: automatically, first inductively (set), then by epsilon

$$\frac{}{(\mathsf{Nil}, f_1) \in \mathsf{list\_rel}\ f_1\ f_2} \qquad \frac{(xs, xs') \in \mathsf{list\_rel}\ f_1\ f_2}{(\mathsf{Cons}\ x\ xs, f_2\ x\ xs\ xs') \in \mathsf{list\_rel}\ f_1\ f_2}$$

list_rec

Defined: automatically, first inductively (set), then by epsilon

$$\frac{}{(\mathsf{Nil}, f_1) \in \mathsf{list\_rel}\ f_1\ f_2} \qquad \frac{(xs, xs') \in \mathsf{list\_rel}\ f_1\ f_2}{(\mathsf{Cons}\ x\ xs, f_2\ x\ xs\ xs') \in \mathsf{list\_rel}\ f_1\ f_2}$$

$$\mathsf{list\_rec}\ f_1\ f_2\ xs \equiv \mathsf{THE}\ y.\ (xs, y) \in \mathsf{list\_rel}\ f_1\ f_2$$

Automatic proof that set def indeed is total function
(the equations for list_rec are lemmas!)

Predefined Datatypes

nat is a datatype

$$\textbf{datatype}\ \text{nat} = 0 \mid \text{Suc nat}$$

nat is a datatype

datatype nat $= 0 \mid \text{Suc nat}$

Functions on nat definable by primrec!

primrec

$$
\begin{array}{lcl}
f\ 0 & = & \ldots \\
f\ (\text{Suc}\ n) & = & \ldots\ f\ n\ \ldots
\end{array}
$$

Option

datatype 'a option = None | Some 'a

Important application:

'b $\Rightarrow$ 'a option $\sim$ partial function:

Option

datatype 'a option = None | Some 'a

Important application:

'b ⇒ 'a option	∼	partial function:
None	∼	no result
Some *a*	∼	result *a*

Example:
primrec lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option
where

Option

datatype 'a option = None | Some 'a

Important application:

'b ⇒ 'a option	∼	partial function:
None	∼	no result
Some *a*	∼	result *a*

Example:

primrec lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option
where
lookup k [] = None |
lookup k (x #xs) =

Option

datatype 'a option = None | Some 'a

Important application:

'b ⇒ 'a option	∼	partial function:
None	∼	no result
Some *a*	∼	result *a*

Example:

primrec lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option
where
lookup k [] = None |
lookup k (x #xs) = (if fst x = k then Some (snd x) else lookup k xs)

Demo

primrec

Induction

Structural induction

$P\ xs$ holds for all lists xs if

- ➜ P Nil
- ➜ and for arbitrary x and xs, $P\ xs \Longrightarrow P\ (x\#xs)$

Structural induction

$P\ xs$ holds for all lists xs if

- ➜ P Nil
- ➜ and for arbitrary x and xs, $P\ xs \Longrightarrow P\ (x\#xs)$
  Induction theorem **list.induct:**
  $[\![P\ []; \bigwedge a\ list.\ P\ list \Longrightarrow P\ (a\#list)]\!] \Longrightarrow P\ list$

Structural induction

$P\ xs$ holds for all lists xs if

- ➜ P Nil
- ➜ and for arbitrary x and xs, $P\ xs \Longrightarrow P\ (x\#xs)$
  Induction theorem **list.induct:**
  $[\![P\ []; \bigwedge a\ list.\ P\ list \Longrightarrow P\ (a\#list)]\!] \Longrightarrow P\ list$
- ➜ General proof method for induction: **(induct x)**
  - x must be a free variable in the first subgoal.
  - type of x must be a datatype.

Basic heuristics

Theorems about recursive functions are proved by induction

Induction on argument number i of f
if f is defined by recursion on argument number i

Example

A tail recursive list reverse:

```
primrec itrev :: 'a list ⇒ 'a list ⇒ 'a list
where
itrev []       ys =     |
```

Example

A tail recursive list reverse:

```
primrec itrev :: 'a list ⇒ 'a list ⇒ 'a list
where
itrev []       ys = ys |
itrev (x#xs)   ys =
```

Example

A tail recursive list reverse:

```
primrec itrev :: 'a list ⇒ 'a list ⇒ 'a list
where
itrev []       ys = ys |
itrev (x#xs)   ys = itrev xs (x#ys)
```

Example

A tail recursive list reverse:

primrec itrev :: 'a list ⇒ 'a list ⇒ 'a list
where
itrev [] $ys = ys$ |
itrev $(x\#xs)$ ys = itrev xs $(x\#ys)$

lemma itrev xs [] = rev xs

Demo

Proof Attempt

Generalisation

Replace constants by variables

lemma itrev xs ys = rev xs@ys

Generalisation

Replace constants by variables

lemma itrev xs ys = rev xs@ys

Quantify free variables by $\forall$
(except the induction variable)

Generalisation

Replace constants by variables

lemma itrev *xs ys* = rev *xs*@*ys*

Quantify free variables by $\forall$
(except the induction variable)

lemma $\forall ys$. itrev *xs ys* = rev *xs*@*ys*

Or: **apply (induct xs arbitrary: ys)**

We have seen today ...

➜ Datatypes

We have seen today ...

- ➜ Datatypes
- ➜ Primitive recursion

We have seen today ...

- ➜ Datatypes
- ➜ Primitive recursion
- ➜ Case distinction

We have seen today ...

- ➜ Datatypes
- ➜ Primitive recursion
- ➜ Case distinction
- ➜ Structural Induction

Exercises

- define a primitive recursive function **lsum** :: nat list $\Rightarrow$ nat that returns the sum of the elements in a list.
- show "$2 * \text{lsum}\ [0..<\mathit{Suc}\ n] = n * (n + 1)$"
- show "$\text{lsum}\ (\text{replicate}\ n\ a) = n * a$"
- define a function **lsumT** using a tail recursive version of listsum.
- show that the two functions are equivalent: $\text{lsum}\ xs = \text{lsumT}\ xs$