



COMP4161: Advanced Topics in Software Verification



June Andronick, Christine Rizkallah, Miki Tanaka, Johannes Åman Pohjola  
T3/2019

[data61.csiro.au](http://data61.csiro.au)



# Content



- Intro & motivation, getting started [1]
- Foundations & Principles
  - Lambda Calculus, natural deduction [1,2]
  - Higher Order Logic, Isar (part 1) [3<sup>a</sup>]
  - Term rewriting [4]
- Proof & Specification Techniques
  - Inductively defined sets, rule induction [5]
  - Datatypes, recursion, induction, Isar (part 2) [6, 7<sup>b</sup>]
  - Hoare logic, proofs about programs, invariants [8]
  - C verification [9]
  - Practice, questions, exam prep [10<sup>c</sup>]

---

<sup>a</sup>a1 due; <sup>b</sup>a2 due; <sup>c</sup>a3 due

# Datatypes



**Example:**

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

**Properties:**

# Datatypes



## Example:

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

## Properties:

→ Constructors:

```
Nil      :: 'a list  
Cons     :: 'a ⇒ 'a list ⇒ 'a list
```

# Datatypes



## Example:

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

## Properties:

→ Constructors:

```
Nil      :: 'a list  
Cons     :: 'a ⇒ 'a list ⇒ 'a list
```

→ Distinctness:    Nil  $\neq$  Cons x xs

# Datatypes



## Example:

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

## Properties:

### → Constructors:

Nil        ::    'a list  
Cons     ::    'a  $\Rightarrow$  'a list  $\Rightarrow$  'a list

→ Distinctness:    Nil  $\neq$  Cons x xs

→ Injectivity:     (Cons x xs = Cons y ys) = (x = y  $\wedge$  xs = ys)

# More Examples



**Enumeration:**

**datatype** answer = Yes | No | 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



$$\text{datatype } (\alpha_1, \dots, \alpha_n) \tau = \begin{array}{c} C_1 \tau_{1,1} \dots \tau_{1,n_1} \\ \vdots \\ C_k \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$



# The General Case



$$\text{datatype } (\alpha_1, \dots, \alpha_n) \tau = \begin{array}{c} C_1 \tau_{1,1} \dots \tau_{1,n_1} \\ \vdots \\ C_k \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

→ Constructors:  $C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_n) \tau$

# The General Case



$$\text{datatype } (\alpha_1, \dots, \alpha_n) \tau = \begin{array}{c} C_1 \tau_{1,1} \dots \tau_{1,n_1} \\ \vdots \\ C_k \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

- Constructors:  $C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_n) \tau$
- Distinctness:  $C_i \dots \neq C_j \dots$  if  $i \neq j$

# The General Case



$$\text{datatype } (\alpha_1, \dots, \alpha_n) \tau = \begin{array}{c} C_1 \tau_{1,1} \dots \tau_{1,n_1} \\ \vdots \\ C_k \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

- Constructors:  $C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_n) \tau$
- Distinctness:  $C_i \dots \neq C_j \dots$  if  $i \neq j$
- Injectivity:  $(C_i x_1 \dots x_{n_i} = C_i y_1 \dots y_{n_i}) = (x_1 = y_1 \wedge \dots \wedge x_{n_i} = y_{n_i})$

# The General Case



$$\text{datatype } (\alpha_1, \dots, \alpha_n) \tau = \begin{array}{c} C_1 \tau_{1,1} \dots \tau_{1,n_1} \\ \vdots \\ C_k \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

- Constructors:  $C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_n) \tau$
- Distinctness:  $C_i \dots \neq C_j \dots$  if  $i \neq j$
- Injectivity:  $(C_i x_1 \dots x_{n_i} = C_i y_1 \dots y_{n_i}) = (x_1 = y_1 \wedge \dots \wedge x_{n_i} = y_{n_i})$

**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:

**datatype** **t** = C (**t**  $\Rightarrow$  bool)

# Datatype Limitations



Must be definable as set.

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

Not ok:

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

# Datatype Limitations



Must be definable as set.

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

Not ok:

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

**Because:** Cantor's theorem ( $\alpha$  set is larger than  $\alpha$ )

# Datatype Limitations



**Not ok (nested recursion):**

```
datatype ('a, 'b) fun_copy = Fun "'a  $\Rightarrow$  'b"
```

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



# Datatype Limitations



**Not ok (nested recursion):**

```
datatype ('a, 'b) fun_copy = Fun "'a  $\Rightarrow$  '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  $\Rightarrow$  '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  $\Rightarrow$  '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  $\Rightarrow$  '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  $\Rightarrow$  '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  $\Rightarrow$  '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  $\Rightarrow$  '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  $\Rightarrow$  '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  $\Rightarrow$  '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.

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

# Case



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

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

**In general:** one case per constructor

# Case



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

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

**In general:** one case per constructor

→ Nested patterns allowed:  $x\#y\#zs$



# Case



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

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

**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.

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

**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)

**apply** (case\_tac  $t$ )

creates  $k$  subgoals

$\llbracket t = C_i \ x_1 \dots x_p; \dots \rrbracket \implies \dots$

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.

$$\Rightarrow$$
$$0 = 1$$

# Why nontermination can be harmful



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

Subtract  $f\ x$  on both sides.

$$\begin{array}{c} \Rightarrow \\ 0 = 1 \end{array}$$

**! 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  $\Rightarrow$  'a list  $\Rightarrow$  'a list"  
where  
  "app Nil ys = ys" |  
  "app (Cons x xs) ys = Cons x (app xs ys)"
```

# The General Case



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

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

# The General Case



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

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

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

# 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:**  $\text{list\_rec} :: "'b \Rightarrow ('a \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b'$   
 $\text{list\_rec } f_1 \ f_2 \ \text{Nil} \quad = \quad f_1$   
 $\text{list\_rec } f_1 \ f_2 \ (\text{Cons } x \ xs) \quad = \quad f_2 \ x \ xs \ (\text{list\_rec } f_1 \ f_2 \ xs)$



# How does this Work?



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

**Example:**  $\text{list\_rec} :: "'b \Rightarrow ('a \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b"$   
 $\text{list\_rec } f_1 \ f_2 \ \text{Nil} \quad = \quad f_1$   
 $\text{list\_rec } f_1 \ f_2 \ (\text{Cons } x \ xs) \quad = \quad f_2 \ x \ xs \ (\text{list\_rec } f_1 \ f_2 \ xs)$

$\text{app} \equiv \text{list\_rec } (\lambda ys. \ ys) \ (\lambda x \ xs \ xs'. \ \lambda ys. \ \text{Cons } x \ (xs' \ ys))$

**primrec**  $\text{app} :: "'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}"$

**where**

$\text{"app Nil } ys = ys"$  |

$\text{"app (Cons } x \ xs) \ ys = \text{Cons } x \ (\text{app } xs \ ys)"$

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

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

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

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

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

$\text{list\_rec } f_1 f_2 \ xs \equiv \text{THE } y. (xs, y) \in \text{list\_rel } f_1 f_2$   
Automatic proof that set def indeed is total function  
(the equations for list\_rec are lemmas!)

# Predefined Datatypes

# nat is a datatype



**datatype** nat = 0 | Suc nat

# nat is a datatype



**datatype** nat = 0 | Suc nat

Functions on nat definable by primrec!

**primrec**

$f\ 0 \quad = \quad \dots$

$f\ (\text{Suc } n) \quad = \quad \dots\ f\ n\ \dots$

# 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 $\Rightarrow$ 'a option	$\sim$	partial function:
None	$\sim$	no result
Some <i>a</i>	$\sim$	result <i>a</i>

## Example:

**primrec** lookup :: 'k  $\Rightarrow$  ('k  $\times$  'v) list  $\Rightarrow$  'v option  
**where**

# Option



**datatype** 'a option = None | Some 'a

## Important application:

'b  $\Rightarrow$  'a option  $\sim$  partial function:  
None  $\sim$  no result  
Some *a*  $\sim$  result *a*

## Example:

**primrec** lookup :: 'k  $\Rightarrow$  ('k  $\times$  'v) list  $\Rightarrow$  'v option

**where**

lookup *k* [] = None |

lookup *k* (x #xs) =

# Option



**datatype** 'a option = None | Some 'a

## Important application:

'b $\Rightarrow$ 'a option	$\sim$	partial function:
None	$\sim$	no result
Some <i>a</i>	$\sim$	result <i>a</i>

## Example:

**primrec** lookup :: 'k  $\Rightarrow$  ('k  $\times$  'v) list  $\Rightarrow$  '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  $xs$ ,  $P\ xs \implies P\ (x\#xs)$

# Structural induction



$P\ xs$  holds for all lists  $xs$  if

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

Induction theorem **list.induct**:

$$\llbracket P\ []; \bigwedge a\ list. P\ list \implies P\ (a\#list) \rrbracket \implies P\ list$$

# Structural induction



$P\ xs$  holds for all lists  $xs$  if

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

Induction theorem **list.induct**:

$\llbracket P\ []; \bigwedge a\ list. P\ list \implies P\ (a\#list) \rrbracket \implies P\ list$

- General proof method for induction: (**induct x**)
  - $x$  must be a free variable in the first subgoal.
  - type of  $x$  must be a datatype.



**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  $\Rightarrow$  'a list  $\Rightarrow$  'a list
where
itrev []      ys =      |
```

# Example



A tail recursive list reverse:

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

# Example



**A tail recursive list reverse:**

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

# Example



**A tail recursive list reverse:**

**primrec** itrev :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list

**where**

itrev []             $ys = ys \mid$

itrev (x#xs)     $ys = \text{itrev } xs \ (x\#ys)$

**lemma** itrev xs [] = rev xs

# Demo

## Proof Attempt

**Replace constants by variables**

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

**Replace constants by variables**

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

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



**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 " $2 * \text{lsum } [0.. < \text{Suc } n] = n * (n + 1)$ "
- 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:  $\text{lsum } xs = \text{lsumT } xs$