UNSW SYDNEY

COMP4161
Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Miki Tanaka, Johannes Åman Pohjola

T3/2022

Content

➜ Foundations & Principles

- Intro, Lambda [illegible] atural deduction [1,2]
- Higher Order [illegible] part 1) [2,3[a]]
- Term rewriting [3,4]

➜ Proof & Specification Techniques

- Inductively defined sets, rule induction [4,5]
- Datatype induction, primitive recursion [5,7]
- General recursive functions, termination proofs [7[b]]
- Proof automation, Isar (part 2) [8]
- Hoare logic, proofs about programs, invariants [8,9]
- C verification [9,10]
- Practice, questions, exam prep [10[c]]

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

Datatypes

Example:

datat 

= Nil | Cons 'a "'a list"

Properties:

Datatypes

Example:

datatype 'a "'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 ≠ Cons x xs
- ➜ Injectivity: (Cons x xs = Cons y ys) = (x = y ∧ xs = ys)

More Examples

Enumeration:

datatyp 

= Yes | No | Maybe

More Examples

Enumeration:

datatype = Yes | No | Maybe

Polymorphic:

datatype 'a option = None | Some 'a

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

More Examples

Enumeration:

datatype = 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"

More Examples

Enumeration:

datatype = 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 = 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 tre Node 'a "'a tree list"

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

Nested

Nested recursion:

datatype 'a tre 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

datatype $(\alpha_1, \ldots, \alpha_n)\ \tau = \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}$

$\ldots$

$\mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$

The General Case

datatype $(\alpha_1, \ldots, \alpha_n)\ \tau = \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}$

$\ldots$

$\mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$

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

The General Case

datatype $(\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}$

$\ldots$

$\mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$

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

The General Case

datatype $(\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}$

$\ldots$

$\mid\ \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$

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

The General Case

datatype $(\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}$

$\ldots$

$\mid\ \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$

- ➜ Constructors: $\mathsf{C}_i \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau$
- ➜ Distinctness: $\mathsf{C}_i \ \ldots \neq \mathsf{C}_j \ \ldots \quad$ 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 = Nil | Cons 'a "'a list"

➜ internally reduced to constructor, using product and sum

How is this Type Defined?

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

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

How is this Type Defined?

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

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

More detail: Tutorial on (Co-)datatypes Definitions at isabelle.in.tum.de

Datatype Limitations

Must be definable as a (non-empty) set.

Datatype Limitations

Must be definable as a (non-empty) set.

➜ Infinitely branching

Datatype Limitations

Must be definable as a (non-empty) set.

- ➜ Infinitely branching
- ➜ Mutually recursive

Datatype Limitations

Must be definable as a (non-empty) set.

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

Datatype Limitations

Must be definable as a (non-empty) set.

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

Not ok:

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

Datatype Limitations

Must be definable as a (non-empty) set.

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

Not ok:

datatype t = C (t ⇒ bool)
| D ((bool ⇒ t) ⇒ bool)

Datatype Limitations

Must be definable as a (non-empty) set.

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

Not ok:

datatype t = C (t ⇒ bool)
| D ((bool ⇒ t) ⇒ bool)
| E ((t ⇒ bool) ⇒ bool)

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

Datatype Limitations

Not ok (nested recursion):

datatype ('a, '


y = Fun "'a ⇒ 'b"

datatype 'a t =
'a) fun_copy"

Datatype Limitations

Not ok (nested recursion):

datatype ('a, '
y = Fun "'a ⇒ 'b"

datatype '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, '


y = Fun "'a ⇒ 'b"

datatype '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, 'y = Fun "'a ⇒ 'b"

datatype '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, 'y = Fun "'a ⇒ 'b"

datatype '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, 'y = Fun "'a ⇒ 'b"

datatype '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.

(case *xs*  | *y* #*ys* ⇒ ... *y* ... *ys* ...)

Case

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

(case *xs*  | *y* #*ys* ⇒ ... *y* ... *ys* ...)

In general: one case per constructor

Case

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

(case *xs*  | *y* #*ys* ⇒ ... *y* ... *ys* ...)

In general: one case per constructor

➜ Nested patterns allowed: *x*#*y*#*zs*

Case

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

(case *xs*  | *y* #*ys* ⇒ ... *y* ... *ys* ...)

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.

(case *xs*  | *y* #*ys* ⇒ ... *y* ... *ys* ...)

In general: one case per constructor

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

- ➜ Binds weakly, needs () in context

Cases

Cases

(case_tac t)

creates k subgoals

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

one for each constructor C_i

Demo

Recursion

Why nontermination can be harmful

Why nontermination can be harmful

t $f\ x = f\ x + 1$?

x on both sides.

Why nontermination can be harmful

t $f\ x = f\ x + 1$?

x on both sides.

$0 = 1$

Why nontermination can be harmful

t $f\ x = f\ x + 1$?

x on both sides.

$\Longrightarrow$

$0 = 1$

! All functions in HOL must be total !

Primitive Recursion

primrec g termination structurally

Example primrec def

Primitive Recursion

primrec guarantees termination structurally

Example primrec definition:

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 r**

$$f\ (C_1\ y_{1,1}\ \ldots\ y_{1,n_1}) \quad = \quad r_1$$

$$\vdots$$

$$f\ (C_k\ y_{k,1}\ \ldots\ y_{k,n_k}) \quad = \quad r_k$$

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 r**

$$f\ (C_1\ y_{1,1}\ \ldots\ y_{1,n_1}) \quad = \quad r_1$$

$$\vdots$$

$$f\ (C_k\ y_{k,1}\ \ldots\ y_{k,n_k}) \quad = \quad r_k$$

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  tax for a **recursion operator**

Example:

How does this Work?

primrec just syntax for a **recursion operator**

Example:

$$\text{rec\_list} :: \text{'b} \Rightarrow \text{'b list} \Rightarrow \text{'a} \Rightarrow \text{'a}) \Rightarrow \text{'b list} \Rightarrow \text{'a''}$$

$$\text{rec\_list } f_1 \; f_2 \; \text{Nil} = f_1$$

$$\text{rec\_list } f_1 \; f_2 \; (\text{Cons } x \; xs) = f_2 \; x \; xs \; (\text{rec\_list } f_1 \; f_2 \; xs)$$

How does this Work?

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

Example:

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

rec_list f_1 f_2 Nil = f_1

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

app ≡ rec_list (λys. ys) (λx xs xs'. λys. 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)"

rec_list

Defined: automaticall ctively (set), then by epsilon

rec_list

Defined: automaticall[illegible]ctively (set), then by epsilon

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

rec_list

Defined: automaticall[illegible]ctively (set), then by epsilon

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

$$\text{rec\_list } f_1\ f_2\ xs \equiv \text{THE } y.\ (xs, y) \in \text{list\_rel } f_1\ f_2$$

Automatic proof that set def indeed is total function

(the equations for rec_list are lemmas!)

Predefined Datatypes

nat is a datatype

$\ldots$at = 0 | Suc nat

nat is a datatype

data nat = 0 | Suc nat

Functions definable by primrec!

primrec
$f\ 0 = \ldots$
$f\ (\mathrm{Suc}\ n) = \ldots\ f\ n\ \ldots$

Option

datatype 'a option = None | Some 'a

Important applicatio

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

Option

datatype 'a option = None | Some 'a

Important applicatio

'b ⇒ ~ 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 applicatio

'b ⇒ ~ 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 applicatio

'b ⇒ ~ 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 [illegible] $s \Longrightarrow P\ (x\#xs)$

Induction theorem [illegible]

$[\![P\ []; \bigwedge a\ list.\ P\ list \Longrightarrow P\ (a\#list)]\!] \Longrightarrow P\ list$

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:

  $[\![P\ []; \bigwedge a\ list.\ P\ list \Longrightarrow P\ (a\#list)]\!] \Longrightarrow P\ list$
- ➜ 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 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 r

primre a list ⇒ 'a list ⇒ 'a list
where
itrev [] *ys* =

Example

A tail recursive list r

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

Example

A tail recursive list r

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

Example

A tail recursive list r

primre '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

Re  stants by variables

le $xs\ ys = \mathsf{rev}\ xs @ ys$

Generalisation

Re  **stants by variables**

le $xs\ ys = \text{rev}\ xs @ ys$

Quantify free variables by $\forall$

(except the induction variable)

Generalisation

Re  **stants by variables**

le $xs\ ys = \text{rev}\ xs @ ys$

Quantify free variables by $\forall$

(except the induction variable)

lemma $\forall ys.\ \text{itrev}\ xs\ ys = \text{rev}\ xs @ ys$

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

We have seen today ...

- Datatypes
- Primitive recursion
- Case distinction
- Structural Induction

Exercises

- define a primitive re[illegible]ction **lsum** :: nat list ⇒ nat that returns the sur[illegible]ments in a list.
- show "2 ∗ lsum [0..[illegible] $n * (n + 1)$"
- show "lsum (replicate [illegible] $n * a$"
- define a function **lsumT** using a tail recursive version of listsum.
- show that the two functions are equivalent: lsum *xs* = lsumT *xs*