COMP4161
Advanced Topics in Software Verification

Gerwin Klein, Miki Tanaka, Johannes Åman Pohjola, Rob Sison

T3/2023

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 [4,5]
- Datatype induction, primitive recursion [5,7]
- General recursive functions, termination proofs [7]
- Proof automation, Isar (part 2) [8[b]]
- 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:

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:

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

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

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

- ➜ 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
- ➜ 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.

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

Not ok:

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

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"

- ➜ 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

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)

creates k subgoals

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

one for each constructor C_i

Demo

Recursion

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:

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

$$\begin{array}{lcl} \text{rec\_list } f_1\ f_2\ \text{Nil} & = & f_1 \\ \text{rec\_list } f_1\ f_2\ (\text{Cons } x\ xs) & = & f_2\ x\ xs\ (\text{rec\_list } f_1\ f_2\ xs) \end{array}$$

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

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

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

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

Automatic proof that set def indeed is total function

(the equations for rec_list are lemmas!)

Predefined Datatypes

nat is a datatype

datatype nat = 0 | Suc nat

Functions on nat definable by primrec!

primrec

$f\ 0 \quad = \quad ...$

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

Option

$$\textbf{datatype}\ \text{'a option} = \text{None} \mid \text{Some 'a}$$

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, $P\ xs \Longrightarrow P\ (x\#xs)$
  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 [] $ys = ys$ |
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

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

