COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein

Formal Methods

Content

NICTA

- ➜ Intro & motivation, getting started with Isabelle
- ➜ Foundations & Principles
  - Lambda Calculus
  - Higher Order Logic, natural deduction
  - Term rewriting
- ➜ **Proof & Specification Techniques**
  - Inductively defined sets, rule induction
  - **Datatypes, recursion, induction**
  - Well founded recursion, Calculational reasoning
  - Hoare logic, proofs about programs
  - Locales, Presentation

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)

THE GENERAL CASE

$$\textbf{datatype}\ (\alpha_1, \dots, \alpha_n)\ \tau \quad = \quad C_1\ \tau_{1,1}\ \dots\ \tau_{1,n_1} \\ \quad | \quad \dots \\ \quad | \quad C_k\ \tau_{k,1}\ \dots\ \tau_{k,n_k}$$

- 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 defined using typedef
- ➜ hence: describes a set
- ➜ set = trees with constructors as nodes
- ➜ inductive definition to characterize which trees belong to datatype

More detail: Datatype_Universe.thy

Datatype Limitations

Must be definable as set.

- ➜ Infinitely branching ok.
- ➜ Mutually recursive ok.
- ➜ Stricly positive (right of function arrow) occurence 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)

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

- ➜ Same order of cases as in datatype
- ➜ Nested patterns allowed: $x\#y\#zs$
- ➜ Binds weakly, needs $()$ in context

Cases

apply (case_tac t)

creates k subgoals

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

one for each constructor C_i

DEMO

Recursion

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

Subtract $f\ x$ on both sides.

$$\Longrightarrow$$
$$0 = 1$$

! **All functions in HOL must be total** **!**

NICTA

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 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 $f\ a_1\ \ldots\ y_{i,j}\ \ldots\ a_p$)

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 $\equiv$ 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)"

NICTA

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

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

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

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

Functions on nat definable by primrec!

primrec

$$
\begin{aligned}
f\ 0 &= \ldots \\
f\ (\mathsf{Suc}\ n) &= \ldots\ f\ n\ \ldots
\end{aligned}
$$

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) = (if fst x = k then Some (snd x) else lookup k xs)

DEMO: PRIMREC

Induction

NICTA

$P\ xs$ 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.

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

A tail recursive list reverse:

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

lemma itrev xs [] $=$ rev xs

Demo: Proof Attempt

NICTA

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 $xs@ys$

We have seen today ...

- ➜ Rule induction in Isar
- ➜ Datatypes
- ➜ Primitive recursion
- ➜ Case distinction
- ➜ Induction

EXERCISES

- look at `http://isabelle.in.tum.de/library/HOL/Datatype_Universe.html`
- define a primitive recursive function **lsum** :: nat list $\Rightarrow$ nat that returns the sum of the elements in a list.
- show "$2 *$ lsum $[0.. < Suc\, n] = n * (n + 1)$"
- show "lsum (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