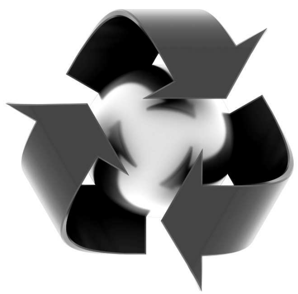
COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Toby Murray, June Andronick, Gerwin Klein



Content

- Intro & motivation, getting started [1]
- Foundations & Principles
  - Lambda Calculus, natural deduction [1,2]
  - Higher Order Logic [3[a]]
  - Term rewriting [4]
- Proof & Specification Techniques
  - Inductively defined sets, rule induction [5]
  - Datatypes, recursion, induction [6, 7]
  - Hoare logic, proofs about programs, C verification [8[b],9]
  - (mid-semester break)
  - Writing Automated Proof Methods [10]
  - Isar, codegen, typeclasses, locales [11[c],12]

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

Datatypes

Example:

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

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

The General Case

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

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

More detail: HOL/Datatype.thy

Datatype Limitations

Must be definable as set.

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

Not ok:

$$\textbf{datatype}\ t = C\ (t \Rightarrow bool)$$
$$\quad\quad\mid D\ ((bool \Rightarrow t) \Rightarrow bool)$$
$$\quad\quad\mid E\ ((t \Rightarrow bool) \Rightarrow bool)$$

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

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

Cases

apply (case_tac t)

creates k subgoals

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

one for each constructor C_i

NICTA

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**:

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

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?

NICTA

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

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

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

Automatic proof that set def indeed is total function

(the equations for list_rec are lemmas!)

PREDEFINED DATATYPES

nat is a datatype

datatype nat $=$ 0 $|$ Suc nat

Functions on nat definable by primrec!

primrec

$$f\ 0 \quad = \quad \ldots$$

$$f\ (\mathsf{Suc}\ n) \quad = \quad \ldots\ f\ n\ \ldots$$

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

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 $\Rightarrow$ 'a list $\Rightarrow$ 'a list

where

itrev $[]$ $\quad ys = ys \mid$

itrev $(x\#xs)$ $\quad ys =$ 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

NICTA

- ➜ define a primitive recursive function **lsum** :: nat list ⇒ 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