COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray

Slide 1

Content

Rough timeline

- ➜ Intro & motivation, getting started [1]
- ➜ Foundations & Principles
  - Lambda Calculus, natural deduction [2,3,4[a]]
  - Higher Order Logic [5,6[b],7]
  - Term rewriting [8,9,10[c]]
- ➜ Proof & Specification Techniques
  - Isar [11,12[d]]
  - Inductively defined sets, rule induction [13[e],15]
  - Datatypes, recursion, induction [16,17[f],18,19]
  - Calculational reasoning, mathematics style proofs [20]
  - Hoare logic, proofs about programs [21[g],22,23]

[a] a1 out; [b] a1 due; [c] a2 out; [d] a2 due; [e] session break; [f] a3 out; [g] a3 due

Slide 2

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)

Slide 3

The General Case

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

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

Distinctness and Injectivity applied automatically

Slide 4

How is this Type Defined?

NICTA

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

Slide 5

Datatype Limitations

NICTA

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 ⇒ bool)
| D ((bool ⇒ t) ⇒ bool)
| E ((t ⇒ bool) ⇒ bool)

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

Slide 6

Case

NICTA

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

(case xs of [] ⇒ $\ldots$ | 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

Slide 7

Cases

NICTA

apply (case_tac t)

creates k subgoals

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

one for each constructor C_i

Slide 8

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

Slide 13

How does this Work?

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)"

Slide 14

list_rec

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

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

list_rec f_1 f_2 $xs \equiv$ SOME $y.\ (xs, y) \in$ list_rel f_1 f_2

Automatic proof that set def indeed is total function
(the equations for list_rec are lemmas!)

Slide 15

Predefined Datatypes

Slide 16

nat is a datatype

datatype nat $=$ 0 | Suc nat

Functions on nat definable by primrec!

primrec

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

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

Slide 17

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)

Slide 18

DEMO: PRIMREC

Slide 19

INDUCTION

Slide 20

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 [] $\quad ys = ys$ |
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 $xs @ ys$

Slide 25

We have seen today ...

- ➜ Datatypes
- ➜ Primitive recursion
- ➜ Case distinction
- ➜ Structural Induction

Slide 26