



COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski



Slide 1



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
 - Isar [5]
 - Inductively defined sets, rule induction [6^b]
 - Datatypes, recursion, induction [7^c, 8]
 - Calculational reasoning, code generation [9]
 - Hoare logic, proofs about programs [10^d,11,12]

^aa1 due; ^ba2 due; ^csession break; ^da3 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

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

→ 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

Slide 4

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`

Slide 5

Datatype Limitations

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

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

(case xs of [] \Rightarrow ... | $y \#ys \Rightarrow$... $y \dots ys \dots$)

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

`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

Slide 8





DEMO

Slide 9



RECURSION

Slide 10

Why nontermination can be harmful

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

Subtract $f\ x$ on both sides.

$$\begin{aligned} & \implies \\ & 0 = 1 \end{aligned}$$

! All functions in HOL must be total !

Slide 11

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)"
```

Slide 12

The General Case



If τ is a datatype (with constructors C_1, \dots, C_k) then $f :: \tau \Rightarrow \tau'$ can be defined by **primitive recursion**:

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

The recursive calls in r_i must be **structurally smaller**
(of the form $f a_1 \dots y_{i,j} \dots a_p$)

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 f1 f2 Nil = f1`
`list_rec f1 f2 (Cons x xs) = f2 x xs (list_rec f1 f2 xs)`

`app ≡ list_rec (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)"`

Slide 14

list_rec



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

$$\frac{}{(\text{Nil}, f_1) \in \text{list_rel } f_1 f_2} \quad \frac{(xs, xs') \in \text{list_rel } f_1 f_2}{(\text{Cons } x xs, f_2 x xs xs') \in \text{list_rel } f_1 f_2}$$

`list_rec f1 f2 xs ≡ SOME y. (xs, y) ∈ list_rel f1 f2`

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

$f\ 0 = \dots$

$f\ (Suc\ n) = \dots f\ n \dots$



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 , $P xs \implies P (x\#xs)$

Induction theorem **list.induct**:

$\llbracket P []; \bigwedge a list. P list \implies P (a\#list) \rrbracket \implies P list$

- General proof method for induction: **(induct x)**
 - x must be a free variable in the first subgoal.
 - type of x must be a datatype.

Slide 21



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

Slide 22



Example

A tail recursive list reverse:

primrec itrev :: 'a list \Rightarrow 'a list \Rightarrow 'a list

where

itrev [] $ys = ys$ |

itrev (x#xs) $ys = \text{itrev } xs (x\#ys)$

lemma itrev xs [] = rev xs

Slide 23



DEMO: PROOF ATTEMPT

Slide 24



Generalisation



Replace constants by variables

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

Quantify free variables by \forall
(except the induction variable)

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

Slide 25

We have seen today ...



- Datatypes
- Primitive recursion
- Case distinction
- Structural Induction

Slide 26