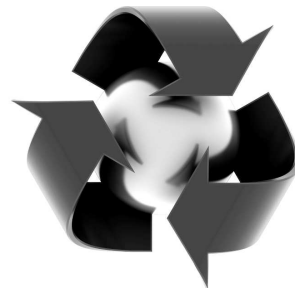


COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski



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]

^a a1 due; ^b a2 due; ^c session break; ^d 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)

The General Case

$$\text{datatype } (\alpha_1, \dots, \alpha_n) \tau \quad = \quad \begin{array}{l} C_1 \tau_{1,1} \dots \tau_{1,n_1} \\ | \dots \\ C_k \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

- 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 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:

```
datatype t = C (t ⇒ bool)
           | D ((bool ⇒ t) ⇒ bool)
           | E ((t ⇒ bool) ⇒ bool)
```

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

Case

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

$$(\text{case } xs \text{ of } [] \Rightarrow \dots \mid y \#ys \Rightarrow \dots y \dots ys \dots)$$

In general: one case per constructor

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

Cases



apply (case_tac t)

creates k subgoals

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

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.

$$\begin{array}{c} \implies \\ 0 = 1 \end{array}$$

! All functions in HOL must be total !

Primitive Recursion

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

How does this Work?

primrec just fancy syntax for a **recursion operator**

Example: $\text{list_rec} :: \text{'b} \Rightarrow (\text{'a} \Rightarrow \text{'a list} \Rightarrow \text{'b} \Rightarrow \text{'b}) \Rightarrow \text{'a list} \Rightarrow \text{'b}$
 $\text{list_rec } f_1 f_2 \text{ Nil} = f_1$
 $\text{list_rec } f_1 f_2 (\text{Cons } x \text{ } xs) = f_2 x \text{ } xs (\text{list_rec } f_1 f_2 xs)$

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

primrec $\text{app} :: \text{'a list} \Rightarrow \text{'a list} \Rightarrow \text{'a list}$

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

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

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

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

$f\ 0 = \dots$

$f\ (\text{Suc } n) = \dots f\ n \dots$

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

- $P \text{ Nil}$
- and for arbitrary x and xs , $P xs \implies P (x\#xs)$

Induction theorem **list.induct**:

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

- 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

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

DEMO: PROOF ATTEMPT

Replace constants by variables

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

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

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

We have seen today ...

- Datatypes
- Primitive recursion
- Case distinction
- Structural Induction