



COMP4161: Advanced Topics in Software Verification



Gerwin Klein, June Andronick, Ramana Kumar
S2/2016

data61.csiro.au



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]

^aa1 due; ^ba2 due; ^ca3 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)

More Examples



Enumeration:

datatype answer = Yes | No | Maybe

Polymorphic:

datatype 'a option = None | Some 'a
datatype ('a,'b,'c) triple = Triple 'a 'b 'c

Recursion:

datatype 'a list = Nil | Cons 'a "'a list"
datatype 'a tree = Tip | Node 'a "'a tree" "'a tree"

Mutual Recursion:

datatype even = EvenZero | EvenSucc odd

Nested



Nested recursion:

```
datatype 'a tree = Tip | Node 'a "'a tree list"
```

```
datatype 'a tree = Tip | Node 'a "'a tree option" "'a tree  
option"
```

→ Recursive call is under a **type constructor**.

The General Case



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

- 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

Datatype Limitations



Must be definable as set.

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

Not ok:

$$\begin{array}{l} \text{datatype } t = C (t \Rightarrow \text{bool}) \\ \quad | D ((\text{bool} \Rightarrow t) \Rightarrow \text{bool}) \\ \quad | E ((t \Rightarrow \text{bool}) \Rightarrow \text{bool}) \end{array}$$

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

Datatype Limitations



Not ok (nested recursion):

```
datatype ('a, 'b) fun_copy = Fun "'a ⇒ 'b"
```

```
datatype 'a t = F "'a t, 'a) fun_copy"
```

- recursion only allowed on *live* arguments
- in "'a ⇒ 'b", 'a is dead and 'b is live
- in ('a, 'b) fun_copy, 'a is dead and 'b is live
- type constructors must be registered as *BNFs** to have live arguments
- datatypes are automatically registered as BNF
- can register other type constructors as BNFs — not covered here**

* BNF = Bounded Natural Functors.

** *Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL*

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

apply (case_tac t)

creates k subgoals

$\llbracket t = C_i x_1 \dots x_{p_i} \dots \rrbracket \implies \dots$

one for each constructor C_i



DATA
61



Demo

Recursion

Why nontermination can be harmful



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

Subtract $f\ x$ on both sides.

$$\implies$$
$$0 = 1$$

! 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: $\text{list_rec} :: "'a \Rightarrow ('a \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b"$

$\text{list_rec } f_1 f_2 \text{ Nil} = f_1$

$\text{list_rec } f_1 f_2 (\text{Cons } x \text{ } xs) = f_2 x \text{ } xs (\text{list_rec } f_1 f_2 \text{ } xs)$

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

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

where

$"\text{app Nil } ys = ys"$ |

$"\text{app } (\text{Cons } x \text{ } xs) \text{ } ys = \text{Cons } x \text{ } (\text{app } xs \text{ } ys)"$

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

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

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

The background of the slide is a repeating pattern of white hexagons on a teal background. The hexagons are arranged in a staggered grid, with each hexagon having a small vertical line extending downwards from its bottom vertex.

DATA
61



Demo

primrec



DATA
61



Induction

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**:

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

→ 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 = itrev xs (x#ys)

lemma itrev xs [] = rev xs



DATA
61



Demo

Proof Attempt

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



- define a primitive recursive function **lsum** :: nat list \Rightarrow nat that returns the sum of the elements in a list.
- show " $2 * \text{lsum } [0.. < \text{Suc } n] = n * (n + 1)$ "
- 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: $\text{lsum } xs = \text{lsumT } xs$