Australian National University

COMP4011/8011
Advanced Topics in
Formal Methods and Programming Languages

– Software Verification with Isabelle/HOL –

Peter Höfner

August 26, 2024

Section 11

Datatypes

Datatypes

Example:

datatype 'a list = Nil | Cons 'a "'a list"

Properties:

- Constructors:

Nil	::	'a list
Cons	::	'a $\Rightarrow$ 'a list $\Rightarrow$ 'a list

- Distinctness: Nil $\neq$ Cons x xs
- Injectivity: (Cons x xs $=$ Cons y ys) $=$ (x $=$ y $\wedge$ 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
and odd = OddSucc even

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

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

- 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

How is this Type Defined?

datatype 'a list = Nil | Cons 'a "'a list"

- internally reduced to a single constructor, using product and sum
- constructor defined as an inductive set (like typedef)
- recursion: least fixpoint

More detail: Tutorial on (Co-)datatypes Definitions at isabelle.in.tum.de

Datatype Limitations

Must be definable as a (non-empty) set.

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

Not ok:

$$\begin{array}{lcl} \textbf{datatype}\ t & = & C\ (t \Rightarrow bool) \\ & | & D\ ((bool \Rightarrow t) \Rightarrow bool) \\ & | & E\ ((t \Rightarrow bool) \Rightarrow bool) \end{array}$$

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 in ('a1, ...,'an) t is only allowed on a subset of 'a1 ... 'an
- these arguments are called *live* arguments
- Mainly: in "'a ⇒ 'b", 'a is dead and 'b is live
- Thus: in ('a, 'b) fun_copy, 'a is dead and 'b is live
- type constructors must be registered as *BNFs** to have live arguments
- BNF defines well-behaved type constructors, ie where recursion is allowed
- datatypes automatically are BNFs (that's how they are constructed)
- 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.

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

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

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

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.

$$\Longrightarrow$$
$$0 = 1$$

! All functions in HOL must be total !

Primitive Recursion

primrec guarantees termination structurally

Example primrec:

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)"

Australian National University

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: rec_list :: "'a ⇒ ('b ⇒ 'b list ⇒ 'a ⇒ 'a) ⇒ 'b list ⇒ 'a"

$$\begin{array}{lcl} \text{rec\_list}\ f_1\ f_2\ \text{Nil} & = & f_1 \\ \text{rec\_list}\ f_1\ f_2\ (\text{Cons}\ x\ xs) & = & f_2\ x\ xs\ (\text{rec\_list}\ f_1\ f_2\ xs) \end{array}$$

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

primrec app :: "'a list ⇒ 'a list ⇒ 'a list"
where
"app Nil ys = ys" |
"app (Cons x xs) ys = Cons x (app xs ys)"

rec_list

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

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

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

Automatic proof that set def indeed is total function

(the equations for rec_list are lemmas!)

Predefined Datatypes

nat is a datatype

datatype nat $=$ 0 $|$ Suc nat

Functions on nat definable by primrec!

primrec

$$
\begin{aligned}
f\ 0 &= \ \ldots \\
f\ (\mathrm{Suc}\ n) &= \ \ldots\ f\ n\ \ldots
\end{aligned}
$$

Option

$$\textbf{datatype } \text{'a option} = \text{None} \mid \text{Some 'a}$$

Important application:

$$
\begin{array}{rcl}
\text{'b} \Rightarrow \text{'a option} & \sim & \text{partial function:} \\
\text{None} & \sim & \text{no result} \\
\text{Some } a & \sim & \text{result } a
\end{array}
$$

Example:

primrec lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option
where
lookup k [] = None |
lookup k (x #xs) = (if fst x = k then Some (snd x) else lookup k xs)

Demo