COMP4161: Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Ramana Kumar, Miki Tanaka
S2/2017

data61.csiro.au
Content

→ Intro & motivation, getting started

→ 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]

^a1 due; ^b2 due; ^c3 due
Datatypes

Example:

\[
\text{datatype} \ 'a \ \text{list} = \text{Nil} \mid \text{Cons} \ 'a \ 'a \ \text{list}
\]

Properties:

→ Constructors:

\[
\begin{align*}
\text{Nil} & : \ 'a \ \text{list} \\
\text{Cons} & : \ 'a \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{list}
\end{align*}
\]

→ Distinctness: \( \text{Nil} \neq \text{Cons} \ x \ \text{xs} \)

→ Injectivity: \( (\text{Cons} \ x \ \text{xs} = \text{Cons} \ y \ \text{ys}) = (x = y \land \text{xs} = \text{ys}) \)
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```
```odd = OddSucc even```
Nested recursion:

```ocaml
datatype 'a tree = Tip | Node 'a ('a tree list)
```

→ Recursive call is under a type constructor.
The General Case

datatype \((\alpha_1, \ldots, \alpha_n) \tau\) = \(C_1 \tau_{1,1} \ldots \tau_{1,n_1} \upharpoonright \ldots \upharpoonright C_k \tau_{k,1} \ldots \tau_{k,n_k}\)

- Constructors: \(C_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n) \tau\)
- Distinctness: \(C_i \ldots \not= C_j \ldots\) if \(i \not= j\)
- Injectivity: \((C_i \; x_1 \ldots x_{n_i} = C_i \; y_1 \ldots y_{n_i}) = (x_1 = y_1 \land \ldots \land x_{n_i} = y_{n_i})\)

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:

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

Because: Cantor’s theorem ($\alpha$ set is larger than $\alpha$)
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.

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

In general: one case per constructor

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

\[ \text{apply (case_tac } t) \]

creates \( k \) subgoals

\[ [t = C_i \ x_1 \ldots \ x_p; \ldots] \xrightarrow{} \ldots \]

one for each constructor \( C_i \)
Demo
Recursion
Why nontermination can be harmful

How about \( f \times = f \times + 1 \)?

Subtract \( f \times \) on both sides.

\[ \implies 0 = 1 \]

All functions in HOL must be total!
Primitive Recursion

primrec guarantees termination structurally

Example primrec def:

```plaintext
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 $\tau$ is a datatype (with constructors $C_1, \ldots, C_k$) then $f : \tau \rightarrow \tau'$ can be defined by primitive recursion:

$$f(C_1 y_{1,1} \ldots y_{1,n_1}) = r_1$$
$$\vdots$$
$$f(C_k y_{k,1} \ldots y_{k,n_k}) = r_k$$

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

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

\[
\begin{align*}
(Nil, f_1) & \in \text{list}_\text{rel} f_1 f_2 \\
(xs, xs') & \in \text{list}_\text{rel} f_1 f_2
\end{align*}
\]

Automatic proof that set def indeed is total function

(\text{the equations for list}_\text{rec} \text{ are lemmas!})
Predefined Datatypes
nat is a datatype

```
datatype nat = 0 | Suc nat
```

Functions on nat definable by primrec!

```
primrec
  f 0       = ...  
  f (Suc n) = ... f n ...
```
Option

**datatype** 'a option = None | Some 'a

**Important application:**

'b ⇒ 'a option ∼ partial function:

None ∼ no result
Some a ∼ result a

**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

primrec
Induction
Structural induction

$P\;xs$ holds for all lists $xs$ if

$\Rightarrow$ $P\;\text{Nil}$

$\Rightarrow$ and for arbitrary $x$ and $xs$, $P\;xs \Rightarrow P\; (x\#xs)$

Induction theorem list.induct:

$[P\;[]; \land \;a\;\text{list}.\;P\;\text{list} \Rightarrow P\;(a\#\text{list})] \Rightarrow P\;\text{list}$

$\Rightarrow$ 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 [] ys = ys |
itrev (x#xs) ys = itrev xs (x#ys)
```

**Lemma** itrev xs [] = rev xs
Demo

Proof Attempt
Generalisation

Replace constants by variables

\textbf{lemma} \ \texttt{itrev} \ \texttt{xs} \ \texttt{ys} = \ \texttt{rev} \ \texttt{xs@ys}

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

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

Or: apply (\texttt{induct} \ \texttt{xs} \ \texttt{arbitrary:} \ \texttt{ys})
We have seen today ...

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

→ define a primitive recursive function \( \text{lsum} :: \text{nat list} \Rightarrow \text{nat} \) that returns the sum of the elements in a list.
→ show "\( 2 \times \text{lsum} \, [0.. < \text{Suc} \, n] = n \times (n + 1) \)"
→ show "\( \text{lsum} \, (\text{replicate} \, n \, a) = n \times a \)"
→ define a function \( \text{lsumT} \) using a tail recursive version of listsum.
→ show that the two functions are equivalent: \( \text{lsum} \, \text{xs} = \text{lsumT} \, \text{xs} \)