COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray
Content

Intro & motivation, getting started

Foundations & Principles
- Lambda Calculus, natural deduction [2,3,4]
- Higher Order Logic [5,6,7]
- Term rewriting [8,9,10]

Proof & Specification Techniques
- Isar [11,12]
- Inductively defined sets, rule induction [13,15]
- Datatypes, recursion, induction [16,17,18,19]
- Calculational reasoning, mathematics style proofs [20]
- Hoare logic, proofs about programs [21,22,23]

a1 out; a1 due; a2 out; a2 due; session break; a3 out; 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

**datatype** \((\alpha_1, \ldots, \alpha_n) \tau\)  
\(=\)  
\(\begin{array}{c}
C_1 \tau_{1,1} \ldots \tau_{1,n_1} \\
\vdots \\
C_k \tau_{k,1} \ldots \tau_{k,n_k}
\end{array}\)

- **Constructors:**  
  \(C_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n) \tau\)

- **Distinctness:**  
  \(C_i \ldots \neq C_j \ldots\) if \(i \neq j\)

- **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

**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 ($\alpha$ set is larger than $\alpha$)
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
apply (case_tac \( t \))

creates \( k \) subgoals

\[
[t = C_i \, x_1 \ldots x_p; \ldots] \Rightarrow \ldots
\]

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.

\[
\begin{align*}
\Rightarrow \\
0 = 1
\end{align*}
\]

! All functions in HOL must be total !
Primitive Recursion

primrec guarantees termination structurally

Example primrec def:

\textbf{primrec} \ app :: "'a list ⇒ 'a list ⇒ 'a list"

\textbf{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 \textit{primitive recursion}:

\[
\begin{align*}
f (C_1 y_{1,1} \ldots y_{1,n_1}) &= r_1 \\
& \vdots \\
f (C_k y_{k,1} \ldots y_{k,n_k}) &= r_k
\end{align*}
\]

The recursive calls in \( r_i \) must be \textbf{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 \textit{recursion operator}

\textbf{Example:} \quad list\_rec :: "\'b ⇒ ('a ⇒ 'a list ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b"

list\_rec \(f_1\ f_2\ \text{Nil} \quad = \quad f_1\)

list\_rec \(f_1\ f_2\ (\text{Cons} \ x \ xs) \quad = \quad f_2 \ x \ xs \ (\text{list\_rec} \ f_1\ f_2\ xs)\)

\text{app} \equiv \text{list\_rec} (\lambda ys. \ ys) (\lambda x \ xs \ xs'. \ \lambda ys. \ \text{Cons} \ x \ (xs' \ ys))

\textit{primrec} \quad \text{app} :: "\'a list ⇒ 'a list ⇒ 'a list"

\textit{where}

"\text{app} \ \text{Nil} \ ys = ys" | "\text{app} \ (\text{Cons} \ x \ xs) \ ys = \text{Cons} \ x \ (\text{app} \ xs \ ys)"
**list_rec**

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

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

\[
\text{list}_\text{rec } f_1 f_2 xs \equiv \text{SOME } y. (xs, y) \in \text{list}_\text{rel } f_1 f_2
\]

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 = ...
  f (Suc n) = ... f n ...
```
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 $xs$, $P\;xs \implies P\;(x\#xs)$

Induction theorem \texttt{list.induct}:

\[ [P \;\top; \land a\;\text{list.} \;P\;\text{list} \implies P\;(a\#\text{list})] \implies P\;\text{list} \]

$\rightarrow$ General proof method for induction: (\texttt{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:

```haskell
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} \ \textit{itrev} \ xs \ ys = \ \textit{rev} \ xs@ys

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

\textbf{lemma} \ \forall ys. \ \textit{itrev} \ xs \ ys = \ \textit{rev} \ xs@ys
We have seen today ...

- Datatypes
- Primitive recursion
- Case distinction
- Structural Induction