COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein
Formal Methods
Intro & motivation, getting started with Isabelle

Foundations & Principles
- Lambda Calculus
- Higher Order Logic, natural deduction
- Term rewriting

Proof & Specification Techniques
- Inductively defined sets, rule induction
- Datatypes, recursion, induction
- Well founded recursion, Calculational reasoning
- Hoare logic, proofs about programs
- Locales, Presentation
Example:

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

Properties:

→ Constructors:

```plaintext
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** $\left( \alpha_1, \ldots, \alpha_n \right) \tau = C_1 \tau_{1,1} \ldots \tau_{1,n_1} \\
\downarrow \quad \ldots \\
\downarrow C_k \tau_{k,1} \ldots \tau_{k,n_k}$

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

$\rightarrow$ Distinctness: $C_i \ldots \neq C_j \ldots$ if $i \neq j$

$\rightarrow$ 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 characterize which trees belong to datatype

More detail: Datatype_Universe.thy
**Datatype Limitations**

Must be definable as set.

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

**Not ok:**

```plaintext
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.

```
(case xs of [] ⇒ ... | y # ys ⇒ ... y ... ys ...)
```

**In general:** one case per constructor

- Same order of cases as in datatype
- Nested patterns allowed: $x#y#zs$
- Binds weakly, needs () in context
apply \ (\text{case\_tac} \ t)\\

creates \( k \) subgoals\\

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

one for each constructor \( C_i \)
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 !
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)"
```

If $\tau$ is a datatype (with constructors $C_1, \ldots, C_k$) then $f :: \tau \Rightarrow \tau'$ can be defined by \textbf{primitive recursion}:

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

The recursive calls in $r_i$ must be \textbf{structurally smaller} (of the form $f \ a_1 \ldots y_{i,j} \ldots a_p$)
primrec just fancy syntax for a recursion operator

**Example:**

\[
\begin{align*}
\text{list}_\text{rec} & : \"b \Rightarrow ('a \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b\" \\
\text{list}_\text{rec} \ f_1 \ f_2 \ \text{Nil} & = \ f_1 \\
\text{list}_\text{rec} \ f_1 \ f_2 \ \text{(Cons } \ x \ \text{xs)} & = \ f_2 \ x \ \text{xs} \ (\text{list}_\text{rec} \ f_1 \ f_2 \ \text{xs})
\end{align*}
\]

\[
\text{app} \equiv \text{list}_\text{rec} \ (\lambda ys. \ ys) \ (\lambda x \ xs \ xs'. \ \lambda ys. \ \text{Cons } x \ (xs' \ ys))
\]

**primrec** \ app :: "'a list ⇒ 'a list ⇒ 'a list"

where

"app Nil ys = ys" |
"app (Cons x xs) ys = Cons x (app xs ys)"
**Defined:** automatically, first inductively (set), then by epsilon

\[(\text{Nil}, f_1) \in \text{list}_\text{rel} f_1 f_2\]

\[(\text{Cons } x \ xs, f_2 x xs xs') \in \text{list}_\text{rel} f_1 f_2\]

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

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 ...
```
**OPTION**

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

**Important application:**

\[ 'b \Rightarrow 'a \text{ option} \sim \text{ partial function:} \]

- None \sim \text{no result}
- Some \ a \sim \text{result} \ a

**Example:**

```haskell
primrec lookup :: 'k \Rightarrow ('k \times 'v) \text{ list} \Rightarrow 'v \text{ option}
where
lookup k [] = None |
lookup k (x #xs) = (if fst x = k then Some (snd x) else lookup k xs)
```
DEMO: PRIMREC
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 list.induct:
\[ [P \; []; \\land a \; \text{list.} \; P \; \text{list} \implies P \; (a\#\text{list})] \implies P \; \text{list} \]

$\rightarrow$ General proof method for induction: (induct $x$)

- $x$ must be a free variable in the first subgoal.
- type of $x$ must be a datatype.
Theorems about recursive functions are proved by induction

Induction on argument number $i$ of $f$
if $f$ is defined by recursion on argument number $i$
A tail recursive list reverse:

\textbf{primrec} \texttt{itrev} :: 'a list \Rightarrow 'a list \Rightarrow 'a list

\textbf{where}

\texttt{itrev} [] \quad ys = ys |

\texttt{itrev} (x#xs) \quad ys = \texttt{itrev} xs (x#ys)

\textbf{lemma} \texttt{itrev} x s [] = \texttt{rev} x s
DEMO: PROOF ATTEMPT
Generalisation

Replace constants by variables

\[
\text{lemma } \text{itrev } x s \ y s = \text{rev } x s @ y s
\]

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

\[
\text{lemma } \forall y s. \ \text{itrev } x s \ y s = \text{rev } x s @ y s
\]
We have seen today ...

- Rule induction in Isar
- Datatypes
- Primitive recursion
- Case distinction
- Induction
→ look at http://isabelle.in.tum.de/library/HOL/Datatype_Universe.html

→ define a primitive recursive function \textbf{lsum} :: nat list ⇒ nat that returns the sum of the elements in a list.

→ show 
\[ 2 \times \text{lsum} [0.. < Suc n] = n \times (n + 1) \]

→ show 
\[ \text{lsum} (\text{replicate } n \ a) = n \times a \]

→ define a function \textbf{lsumT} using a tail recursive version of \text{listsum}.

→ show that the two functions are equivalent: \text{lsum } xs = \text{lsumT } xs