**Script** generated by TTT

**Title:** FDS (05.05.2017)

**Date:** Fri May 05 08:30:40 CEST 2017

**Duration:** 95:57 min

**Pages:** 95

---

1. Overview of Isabelle/HOL

2. Type and function definitions

3. Induction Heuristics

4. Simplification

---

**datatype** — the general case

\[
\text{datatype } (\alpha_1, \ldots, \alpha_n)t = \begin{array}{l}
C_1 \tau_{1,1} \ldots \tau_{1,n_1} \\
\vdots \\
C_k \tau_{k,1} \ldots \tau_{k,n_k}
\end{array}
\]

- Types: \( C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)t \)
**datatype — the general case**

\[
\text{datatype } (\alpha_1, \ldots, \alpha_n)t = C_1 \tau_{1,1} \cdots \tau_{1,n_1} \\
| \cdots \\
| C_k \tau_{k,1} \cdots \tau_{k,n_k}
\]

- **Types:** \(C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)t\)
- **Distinctness:** \(C_i \ldots \neq C_j \ldots \) if \(i \neq j\)

**Case expressions**

Like in functional languages:

\[
(case \ t \ of \ pat_1 \Rightarrow t_1 \mid \cdots \mid pat_n \Rightarrow t_n)
\]

Distinctness and injectivity are applied automatically

Induction must be applied explicitly
Case expressions

Like in functional languages:

\[(\text{case } t \text{ of } \text{pat}_1 \Rightarrow t_1 \mid \ldots \mid \text{pat}_n \Rightarrow t_n)\]

Complicated patterns mean complicated proofs!

Tree_Demo.thy

Case expressions

Like in functional languages:

\[(\text{case } t \text{ of } \text{pat}_1 \Rightarrow t_1 \mid \ldots \mid \text{pat}_n \Rightarrow t_n)\]

Complicated patterns mean complicated proofs!

Need ( ) in context

Tree_Demo.thy
The option type

datatype 'a option = None | Some 'a

If 'a has values a₁, a₂, ...
then 'a option has values None, Some a₁, Some a₂, ...

Typical application:

fun lookup :: ('a × 'b) list ⇒ 'a ⇒ 'b option where
lookup [] x = None |
lookup ((a,b) # ps) x =
  (if a = x then Some b else lookup ps x)
Type and function definitions

Type definitions

Function definitions

Non-recursive definitions

Example

\textbf{definition} \quad \textit{sq} :: \textit{nat} \rightarrow \textit{nat} \quad \textbf{where} \quad \textit{sq} \ n = \ n \cdot n

Non-recursive definitions

Example

\textbf{definition} \quad \textit{sq} :: \textit{nat} \rightarrow \textit{nat} \quad \textbf{where} \quad \textit{sq} \ n = \ n \cdot n

No pattern matching, just \( f \ x_1 \ldots \ x_n = \ldots \)
The danger of nontermination

How about \( f \, x = f \, x + 1 \)?

\[
\text{Subtract } f \, x \text{ on both sides.} \\
\implies 0 = 1
\]

Key features of \textbf{fun}

- Pattern-matching over datatype constructors
- Order of equations matters
- Termination must be provable automatically by size measures
Key features of **fun**

- Pattern-matching over datatype constructors
- Order of equations matters
- Termination must be provable automatically by size measures
- Proves customized induction schema

Example: separation

```haskell
fun sep :: 'a ⇒ 'a list ⇒ 'a list where
  sep a (x#y#zs) = x # a # sep a (y#zs)  
  sep a xs = xs
```

**primrec**

- A restrictive version of **fun**
- Means *primitive recursive*
- Most functions are primitive recursive

**primrec**

- A restrictive version of **fun**
- Means *primitive recursive*
- Most functions are primitive recursive
- Frequently found in Isabelle theories
A restrictive version of fun
- Means primitive recursive
- Most functions are primitive recursive
- Frequently found in Isabelle theories

The essence of primitive recursion:
\[
\begin{align*}
f(0) & = \ldots \text{ no recursion} \\
f(Suc\ n) & = \ldots f(n) \ldots \\
g([]) & = \ldots \text{ no recursion} \\
g(x\#xs) & = \ldots g(xs) \ldots 
\end{align*}
\]
Basic induction heuristics

Theorems about recursive functions are proved by induction

Basic induction 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 $i$

A tail recursive reverse

Our initial reverse:

fun rev :: 'a list ⇒ 'a list where
  rev [] = []
  rev (x#xs) = rev xs @ [x]

A tail recursive reverse

Our initial reverse:

fun rev :: 'a list ⇒ 'a list where
  rev [] = []
  rev (x#xs) = rev xs @ [x]

A tail recursive version:

fun itrev :: 'a list ⇒ 'a list ⇒ 'a list where
A tail recursive reverse

Our initial reverse:

\[
\text{fun } \text{rev} :: 'a \text{ list } \Rightarrow 'a \text{ list where} \\
\text{rev } [] &= [] | \\
\text{rev } (x \# xs) &= \text{rev } xs \ @ [x]
\]

A tail recursive version:

\[
\text{fun } \text{itrev} :: 'a \text{ list } \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list where} \\
\text{itrev } [] &= ys = ys | \\
\text{itrev } (x \# xs) &= ys = \text{itrev } xs \ (x \# ys)
\]

A tail recursive reverse

Our initial reverse:

\[
\text{fun } \text{rev} :: 'a \text{ list } \Rightarrow 'a \text{ list where} \\
\text{rev } [] &= [] | \\
\text{rev } (x \# xs) &= \text{rev } xs \ @ [x]
\]

A tail recursive version:

\[
\text{fun } \text{itrev} :: 'a \text{ list } \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list where} \\
\text{itrev } [] &= ys = ys | \\
\text{itrev } (x \# xs) &= ys = \text{itrev } xs \ (x \# ys)
\]

\text{lemma } \text{itrev } xs \ [] &= \text{rev } xs
Induction_Demo.thy

So far, all proofs were by **structural induction**

So far, all proofs were by **structural induction** because all functions were **primitive recursive**. In each induction step, 1 constructor is added. In each recursive call, 1 constructor is removed.
So far, all proofs were by structural induction because all functions were primitive recursive.

In each induction step, 1 constructor is added.
In each recursive call, 1 constructor is removed.
Now: induction for complex recursion patterns.

Example
\[
\text{fun } \text{div2} :: \text{nat} \Rightarrow \text{nat where}
\]
\[
\text{div2 } 0 = 0 \quad | \\
\text{div2 } (\text{Suc } 0) = 0 \quad | \\
\text{div2 } (\text{Suc}(\text{Suc } n)) = \text{Suc(div2 } n)
\]

\[\rightsquigarrow \text{ induction rule div2.induct:}\]
\[
\frac{P(0) \quad P(\text{Suc } 0) \quad P(n) \Rightarrow P(\text{Suc}(\text{Suc } n))}{P(\text{Suc}(\text{Suc } n))}
\]

\[\rightsquigarrow \text{ induction rule div2.induct:}\]
\[
\frac{P(0) \quad P(\text{Suc } 0) \quad \land n. \ P(n) \Rightarrow P(\text{Suc}(\text{Suc } n))}{P(\text{Suc}(\text{Suc } n))}
\]
Computation Induction

If \( \mathsf{f} : \tau \Rightarrow \tau' \) is defined by \( \mathsf{fun} \), a special induction schema is provided to prove \( P(x) \) for all \( x :: \tau \):

for each defining equation

\[
f(e) = \ldots f(r_1) \ldots f(r_k) \ldots
\]

prove \( P(e) \) assuming \( P(r_1), \ldots, P(r_k) \).

Induction follows course of (terminating!) computation

Motto: properties of \( f \) are best proved by rule \( f.induct \)
How to apply \( f.\text{induct} \)

If \( f :: \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau' \):

\[ (\text{induction } a_1 \ldots a_n \text{ rule: } f.\text{induct}) \]

Heuristic:
- there should be a call \( f a_1 \ldots a_n \) in your goal
- ideally the \( a_i \) should be variables.
**Computation Induction**

If \( f :: \tau \Rightarrow \tau' \) is defined by \texttt{fun}, a special induction schema is provided to prove \( P(x) \) for all \( x :: \tau \):

\[
\text{for each defining equation}
\]

Induction follows course of (terminating!) computation

Motto: properties of \( f \) are best proved by rule \texttt{f.induct}

---

**How to apply \texttt{f.induct}**

If \( f :: \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau' \):

---

**Simplification means ...**

Using equations \( l = r \) from left to right
Simplification means...

Using equations \( l = r \) from left to right
As long as possible

Terminology: equation \( \rightsquigarrow \) simplification rule

Simplification means...

Using equations \( l = r \) from left to right
As long as possible

An example

Equations:
\[
\begin{align*}
0 + n &= n & (1) \\
(Suc\ m) + n &= Suc\ (m + n) & (2) \\
(Suc\ m \leq Suc\ n) &= (m \leq n) & (3) \\
(0 \leq m) &= True & (4)
\end{align*}
\]
An example

Equations:
1. $0 + n = n$
2. $(\text{Succ } m) + n = \text{Succ } (m + n)$
3. $(\text{Succ } m \leq \text{Succ } n) = (m \leq n)$
4. $(0 \leq m) = \text{True}$

$0 + \text{Succ } 0 \leq \text{Succ } 0 + x$

Rewriting:

$0 + \text{Succ } 0 \leq \text{Succ } 0 + x \quad (1)$
$\text{Succ } 0 \leq \text{Succ } 0 + x \quad (2)$

An example

Equations:
1. $0 + n = n$
2. $(\text{Succ } m) + n = \text{Succ } (m + n)$
3. $(\text{Succ } m \leq \text{Succ } n) = (m \leq n)$
4. $(0 \leq m) = \text{True}$

$0 + \text{Succ } 0 \leq \text{Succ } 0 + x \quad (1)$
$\text{Succ } 0 \leq \text{Succ } 0 + x \quad (2)$

Rewriting:

$0 \leq 0 + x$
An example

Equations:

\begin{align*}
0 + n &= n \\ (\text{Suc } m) + n &= \text{Suc } (m + n) \\ (\text{Suc } m \leq \text{Suc } n) &= (m \leq n) \\ (0 \leq m) &= \text{True}
\end{align*}

Rewriting:

\begin{align*}
0 + \text{Suc } 0 &\leq \text{Suc } 0 + x \\ \text{Suc } 0 &\leq \text{Suc } 0 + x \\ \text{Suc } 0 &\leq \text{Suc } (0 + x) \\ 0 &\leq 0 + x \\ \text{True}
\end{align*}

Conditional rewriting

Simplification rules can be conditional:

\[ [ P_1; \ldots; P_k ] \implies l = r \]

is applicable only if all \( P_i \) can be proved first, again by simplification.

Example

\[ p(0) = \text{True} \]
\[ p(x) \implies f(x) = g(x) \]
Conditional rewriting

Simplification rules can be conditional:

\[ [P_1; \ldots; P_k] \implies l = r \]

is applicable only if all \( P_i \) can be proved first, again by simplification.

Example

\[
\begin{align*}
p(0) &= True \\
p(x) &\implies f(x) = g(x)
\end{align*}
\]

We can simplify \( f(0) \) to \( g(0) \)

Termination

Simplification may not terminate. Isabelle uses \textit{simp}-rules (almost) blindly from left to right.

Example: \( f(x) = g(x), \ g(x) = f(x) \)

\[ [P_1; \ldots; P_k] \implies l = r \]

is suitable as a \textit{simp}-rule only if \( l \) is “bigger” than \( r \) and each \( P_i \)
Termination

Simplification may not terminate.
Isabelle uses simp-rules (almost) blindly from left to right.

Example: \( f(x) = g(x), \ g(x) = f(x) \)

\[ \begin{array}{c}
    P_1; \ldots; P_k \Rightarrow l = r \\
\end{array} \]

is suitable as a simp-rule only
if \( l \) is “bigger” than \( r \) and each \( P_i \)

\[ \begin{array}{c}
    n < m \Rightarrow (n < Suc m) = True \\
    Suc n < m \Rightarrow (n < m) = True \\
\end{array} \]

Proof method simp

Goal: \( 1. \ [ P_1; \ldots; P_m ] \Rightarrow C \)

apply(simp add: eq_1 \ldots eq_n)

Termination

Simplification may not terminate.
Isabelle uses simp-rules (almost) blindly from left to right.

Example: \( f(x) = g(x), \ g(x) = f(x) \)

\[ \begin{array}{c}
    P_1; \ldots; P_k \Rightarrow l = r \\
\end{array} \]

is suitable as a simp-rule only
if \( l \) is “bigger” than \( r \) and each \( P_i \)

\[ \begin{array}{c}
    n < m \Rightarrow (n < Suc m) = True \quad YES \\
    Suc n < m \Rightarrow (n < m) = True \quad NO \\
\end{array} \]

Proof method simp

Goal: \( 1. \ [ P_1; \ldots; P_m ] \Rightarrow C \)

apply(simp add: eq_1 \ldots eq_n)

Simplify \( P_1 \ldots P_m \) and \( C \) using
- lemmas with attribute simp
Proof method \textit{simp}

Goal: \hspace{0.5cm} 1. \( [P_1; \ldots; P_m] \implies C \)

\texttt{apply}(\textit{simp add: }eq_1 \ldots eq_n)

Simplify \(P_1 \ldots P_m\) and \(C\) using

- lemmas with attribute \textit{simp}
- rules from \texttt{fun} and \texttt{datatype}

Variations:

- \((\textit{simp \ldots del: \ldots})\) removes \textit{simp}-lemmas
- \texttt{add} and \texttt{del} are optional

Proof method \textit{simp}

Goal: \hspace{0.5cm} 1. \( [P_1; \ldots; P_m] \implies C \)

\texttt{apply}(\textit{simp add: }eq_1 \ldots eq_n)

Simplify \(P_1 \ldots P_m\) and \(C\) using

- lemmas with attribute \textit{simp}
- rules from \texttt{fun} and \texttt{datatype}
- additional lemmas \(eq_1 \ldots eq_n\)
- assumptions \(P_1 \ldots P_m\)

auto versus \textit{simp}

- \texttt{auto} acts on all subgoals
- \textit{simp} acts only on subgoal 1
auto versus simp

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more
- *auto* can also be modified:
  \[(\text{auto simp add: } \ldots \text{ simp del: } \ldots)\]

Rewriting with definitions

Definitions (definition) must be used explicitly:

\[(\text{simp add: f_def }\ldots)\]

Case splitting with simp/auto

Automatic:

\[P(\text{if } A \text{ then } s \text{ else } t) = (A \rightarrow P(s)) \land (\neg A \rightarrow P(t))\]

By hand:

\[P(\text{case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. e = \text{Suc } n \rightarrow P(b))\]
Case splitting with \textit{simp/auto}

Automatic:

\[ P(\text{if } A \text{ then } s \text{ else } t) = (A \rightarrow P(s)) \land (\neg A \rightarrow P(t)) \]

By hand:

\[ P(\text{case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. \, e = \text{Suc } n \rightarrow P(b)) \]

Proof method: (\textit{simp split: nat.split})

\begin{align*}
\text{Case splitting with } \textit{simp/auto} \\
\text{Automatic:} \\
\quad P(\text{if } A \text{ then } s \text{ else } t) = (A \rightarrow P(s)) \land (\neg A \rightarrow P(t)) \\
\text{By hand:} \\
\quad P(\text{case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. \, e = \text{Suc } n \rightarrow P(b)) \\
\text{Proof method: (}\textit{simp split: nat.split}) \text{ Or } auto. \text{ Similar for any datatype } t: \textit{t.split}\end{align*}

Case splitting with \textit{simp/auto}

Automatic:

\[ P(\text{if } A \text{ then } s \text{ else } t) = (A \rightarrow P(s)) \land (\neg A \rightarrow P(t)) \]

By hand:

\[ P(\text{case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. \, e = \text{Suc } n \rightarrow P(b)) \]

Proof method: (\textit{simp split: nat.split})

Or \textit{auto}. Similar for any datatype \( t: \textit{t.split} \)

Splitting pairs with \textit{simp/auto}

How to replace

\[ P \left( \text{let } (x, y) = t \text{ in } u \, x \, y \right) \]
Splitting pairs with \texttt{simp/auto}

How to replace

\[
P\left(\text{let } (x, y) = t \text{ in } u \ x \ y\right)
\]

or

\[
P\left(\text{case } t \text{ of } (x, y) \Rightarrow u \ x \ y\right)
\]

\[
\forall x \ y. t = (x, y) \quad \rightarrow \quad P\left(u \ x \ y\right)
\]

Proof method: \texttt{(simp split: prod.split)}

\[
\]