Automatic Deduction — LVA 703522
Introduction to Isabelle

Clemens Ballarin

Universität Innsbruck

Some α | None

Contents

- Intro & motivation, getting started with Isabelle
- Foundations & Principles
  - Lambda Calculus
  - Types & Classes
  - Natural Deduction
  - Term rewriting
- Proof & Specification Techniques
  - Isar: mathematics style proofs
  - Inductively defined sets, rule induction
  - Datatypes, structural induction
  - Recursive functions & code generation

Types

The Three Basic Ways of Introducing Theorems

- Axioms

  axioms refl: "$t = t$"

  Normally only used when defining new object-logics.

- Definitions

  definition "inj $f \equiv \forall x\ y.\ f\ x = f\ y \longrightarrow x = y$"

- Proofs

  lemma "inj $(\lambda x.\ x + 1)$"

  The harder, but safe choice.

The Three Basic Ways of Introducing Types

- By name only

  typedecl name

  Introduces new type name without any further assumptions.

- By abbreviation

  types α rel $=$ "$\alpha \Rightarrow \alpha \Rightarrow bool$"

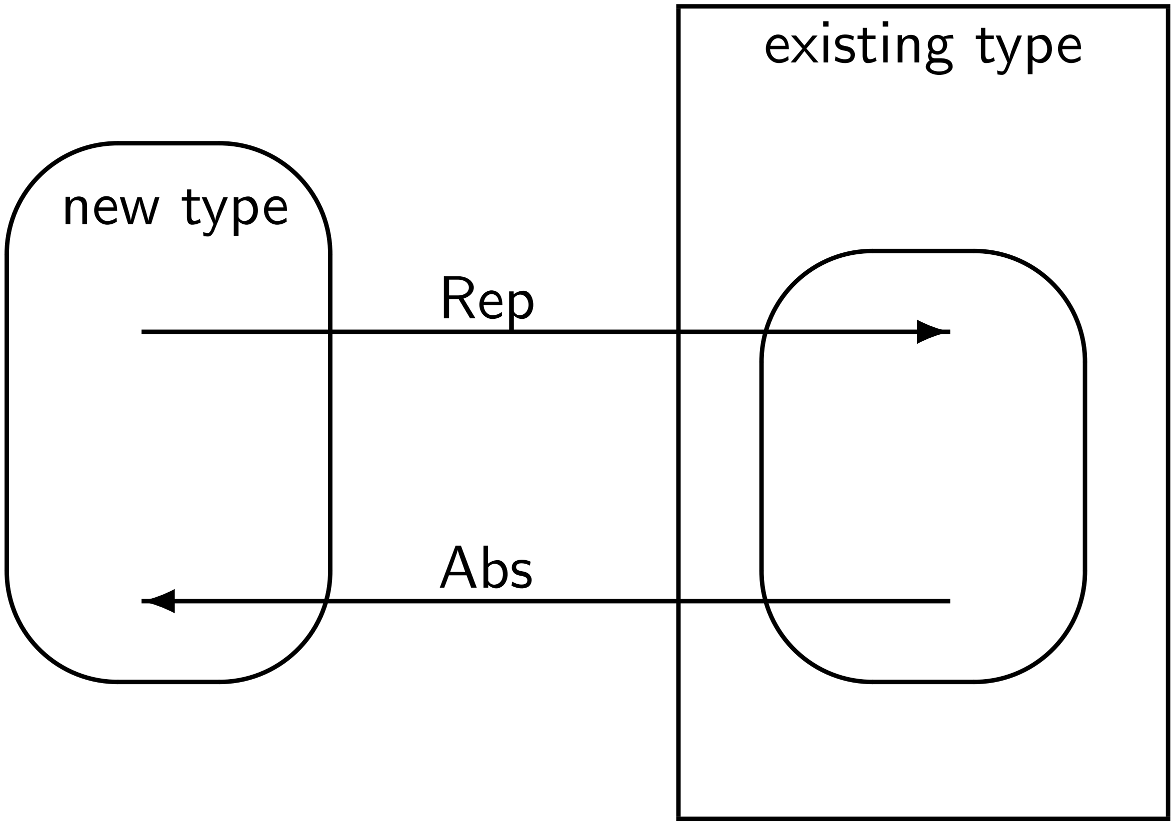
  Introduces abbreviation rel for existing type.
  Type abbreviations are immediatly expanded internally.

- By definiton as a set

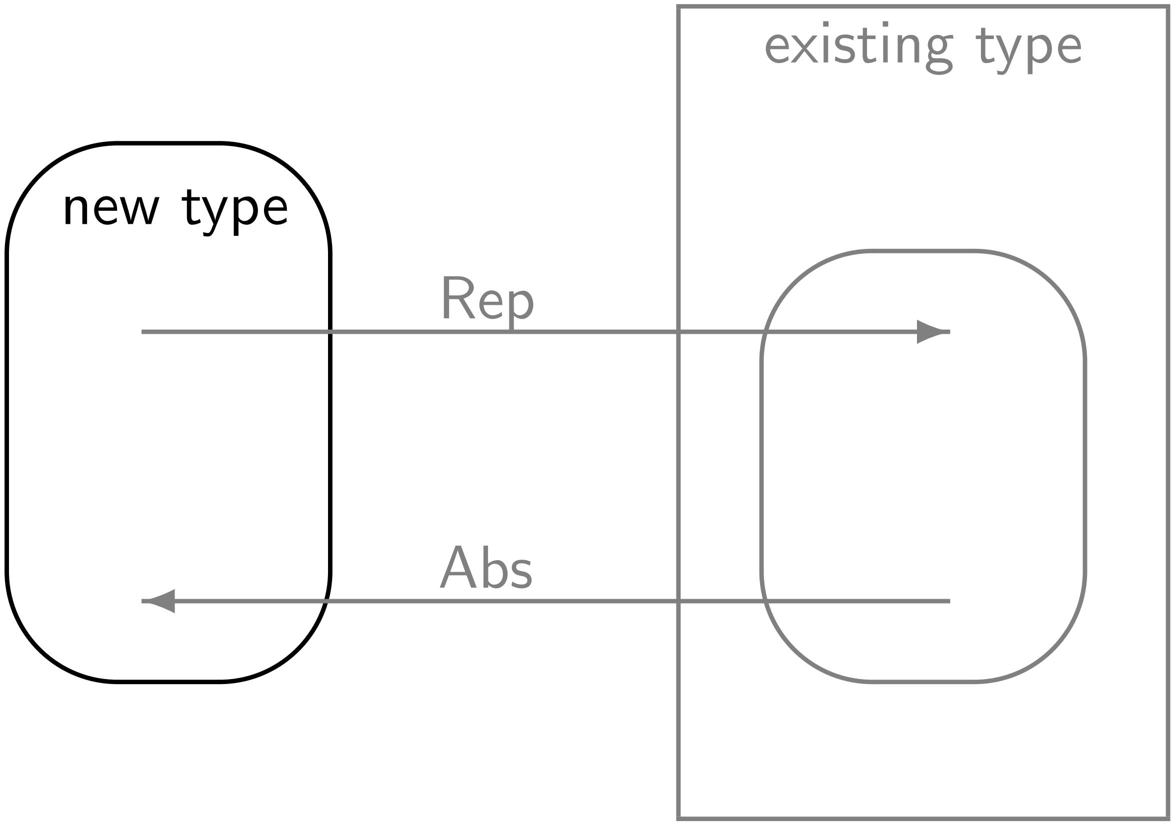
  typedef new_type $=$ "{some set}" $\langle$*proof*$\rangle$

  Introduces a new type as a subset of an existing type.
  The proof shows that the set on the rhs in non-empty.

How Typedef Works



How Typedef Works



Example: Pairs

$$(\alpha, \beta) \text{ Prod}$$

1. Pick existing type: $\alpha \Rightarrow \beta \Rightarrow \text{bool}$
2. Identify subset:
   $(\alpha, \beta) \text{ Prod} = \{f.\ \exists a\ b.\ f = \lambda(x :: \alpha)\ (y :: \beta).\ x = a \wedge y = b\}$
3. We get from Isabelle:
   - functions Abs_Prod, Rep_Prod
   - both injective
   - Abs_Prod (Rep_Prod x) $= x$
4. We now can:
   - define constants Pair, fst, snd in terms of Abs_Prod and Rep_Prod
   - derive all characteristic theorems
   - forget about Rep/Abs, use characteristic theorems instead

Demo: Introducting New Types

Datatypes

Example:

$$\textbf{datatype}\ \text{'a list} = \text{Nil} \mid \text{Cons 'a "'a list"}$$

Properties:

- Constructors:

$$\begin{array}{lll} \text{Nil} & :: & \text{'a list} \\ \text{Cons} & :: & \text{'a} \Rightarrow \text{'a list} \Rightarrow \text{'a list} \end{array}$$

- Distinctness: $\text{Nil} \neq \text{Cons x xs}$
- Injectivity: $(\text{Cons x xs} = \text{Cons y ys}) =$
  $(\text{x} = \text{y} \wedge \text{xs} = \text{ys})$

The General Case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1} \quad | \quad \ldots \quad | \quad \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$$

- Constructors: $\mathsf{C}_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau$
- Distinctness: $\mathsf{C}_i\ \ldots \neq \mathsf{C}_j\ \ldots$ if $i \neq j$
- Injectivity:
  $(\mathsf{C}_i\ x_1 \ldots x_{n_i} = \mathsf{C}_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

Distinctness and injectivity applied automatically.

How is the Type Defined?

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

- Internally defined using typedef.
- Hence: describes a set.
- Set = lists with constructors as nodes.
- Inductive definition to characterize which lists belong to datatype.

More detail: Datatype_Universe.thy

Demo: Defining a Datatype

Datatype Limitations

Must be definable as set.

- Infinitely branching OK.
- Mutually recursive OK.
- Stricly positive (right of function arrow) occurence OK.

Not OK:

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

Because of Cantor's theorem (α set is larger than α)

Case

Every datatype introduces a **case** construct, e.g.

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

In general: one case per constructor

- Same order of cases as in datatype
- No nested patterns (e.g. $x\#y\#zs$)
  (But nested cases allowed)
- Needs () in context

Case Analysis and Induction

cases and induct

- Rule selected according to type:

  (cases "t")   (induct "x")

- Cases identified by constructor names.

Demo: Structural Induction

Recursion

Why Non-Termination 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

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

Old-style command

- Constant must be declared (**consts**).
- No use of **where** and |.

The General Case

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

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

The recursive calls in r_i must be **structurally smaller**
(of the form $f\ y_{i,j}$)

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 a f Nil $\quad = \quad a$

list_rec a f (Cons x xs) $\quad = \quad f\ x\ xs$ (list_rec a f xs)

append ≡ list_rec $(\lambda ys.\ ys)$ $(\lambda x\ xs\ xs'.\ \lambda ys.$ Cons x $(xs'\ ys))$

consts append :: "'a list ⇒ 'a list ⇒ 'a list"

primrec

"append Nil ys = ys"

"append (Cons x xs) ys = Cons x (append xs ys)"

list_rec

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

$$(\mathsf{Nil}, a) \in \mathsf{list\_rel}\ a\ f$$

$$\frac{(xs, xs') \in \mathsf{list\_rel}\ a\ f}{(\mathsf{Cons}\ x\ xs, f\ x\ xs\ xs') \in \mathsf{list\_rel}\ a\ f}$$

$$\mathsf{list\_rec}\ a\ f\ xs \equiv \mathsf{SOME}\ y.\ (xs, y) \in \mathsf{list\_rel}\ a\ f$$

Automatic proof that set definition 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

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

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

consts lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option
primrec

```
  lookup k []       = None
  lookup k (x #xs) = (if fst x = k then Some (snd x)
                               else lookup k xs)
```

Demo: Primitive Recursion

General Recursion

The Choice

- Primitive Recursion (primrec)
  Limited expressiveness, automatic termination
- General Recursion (fun, function)
  High expressiveness, may need to prove termination manually

fun — Examples

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

fun ack :: "nat ⇒ nat ⇒ nat"
where
    "ack 0 n = Suc n"
  | "ack (Suc m) 0 = ack m 1"
  | "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"
```

fun

The Definition

As in functional programming

- Free pattern matching
- Order of rules is important

What it does . . .

- Checks patterns for completeness.
- Inductively defines graph of function.
- Tries to find lexicographic termination order.
- Defines the function.
- Generates induction principle.

fun

Completing Patterns

x # y # zs	$\longrightarrow$	x # y # zs
xs	$\longrightarrow$	[]
	$\longrightarrow$	[x]

Induction Principle

sep.induct:

$$\llbracket\ \bigwedge a\ x\ y\ zs.\ P\ a\ (y\#zs) \Longrightarrow P\ a\ (x\#y\#zs);$$
$$\bigwedge a.\ P\ a\ [];$$
$$\bigwedge a\ w.\ P\ a\ [w];$$
$$\rrbracket \Longrightarrow P\ a\ xs$$

Termination

Isabelle tries to prove termination automatically.

- Works for many functions.
- If not, prove termination manually.

fun = function + termination

fun f :: τ **where** ⟨*rules*⟩ short hand for

function (sequential) f :: τ **where** ⟨*rules*⟩
by pat_completeness auto
termination by lexicographic_order

Proving Termination

- Lexicographic order
  by (lexicographic_order
  add: ⟨*simps*⟩ intro: ⟨*intros*⟩ elim: ⟨*elims*⟩)
- Relation method
  apply (relation R)
  followed by proof that termination relation is well-founded and arguments decrease in each recursive call.

Measure Functions

Definition

A function $m : \alpha \Rightarrow$ nat is a measure function on α.

Lemma

measure $m \equiv \{(x, y).\, m\, x < m\, y\}$ is well-founded.

In Isabelle

wf_measure [intro]: wf (measure ($m :: \alpha \Rightarrow$ nat))

$\longrightarrow$ Wellfounded_Relations.thy

Further reading:
Alexander Krauss. Tutorial on Function Definitions.

We Have Seen so far . . .

- Datatypes
- Primitive recursion
- Case distinction
- Induction
- General recursion