universität innsbruck

Summer Term 2024


# Interactive Theorem Proving using Isabelle/HOL

## Session 9

René Thiemann

Department of Computer Science

# Outline

- Type Definitions in Isabelle
- Lifting and Transfer

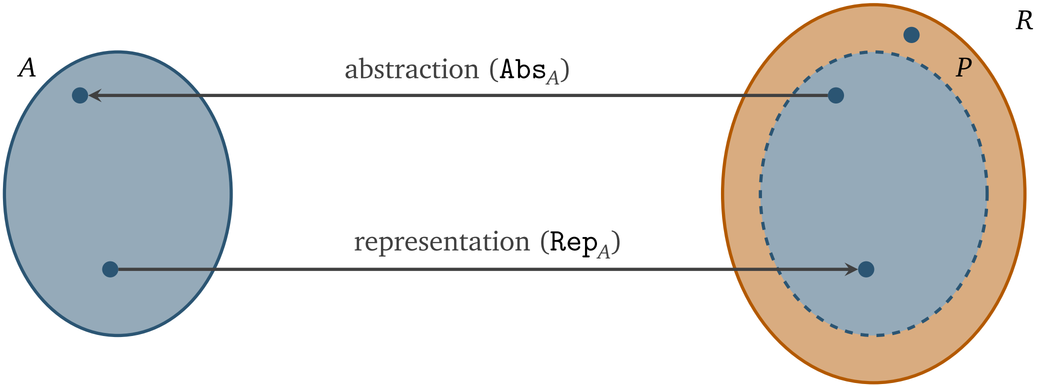
# Type Definitions in Isabelle

## Creation of New Types

- `type_synonym`: just syntactic abbreviation
- `datatype`
  - intuitive high-level command, with several features not mentioned here in this course
  - extensive documentation available (64 pages)

    `isabelle doc datatypes`
  - highly non-trivial construction in the background, based on bounded natural functors (Blanchette et al., citation [2] in the documentation)
- `typedef`
  - core definition principle of types (similar to `definition`)
  - internally used by `datatype`
  - also useful to define types that are not datatypes, e.g., the type of ordered binary trees

# Introducing New Types by `typedef`




- carve out elements satisfying predicate $P$ from existing representation type $R$
- introduce new abstract type $A$ as (non-empty) copy of corresponding subset of $R$
  `typedef 'a`$_1$ `... 'a`$_n$ $A$ `= "{x::`$R$`.` $P$ `x}"` ⟨*proof*⟩
- move between types with abstraction function $\mathtt{Abs}_A$ and representation function $\mathtt{Rep}_A$

## Example: Sets as Functions

```
typedef 'a SET = "{ f :: 'a ⇒ bool. True}" by auto

term "Rep_SET :: 'a SET ⇒ ('a ⇒ bool)"
term "Abs_SET :: ('a ⇒ bool) ⇒ 'a SET"

definition EMPTY :: "'a SET" where
  "EMPTY = Abs_SET (λ _. False)"

definition ELEM :: "'a ⇒ 'a SET ⇒ bool" where
  "ELEM x A = Rep_SET A x"

definition UNION :: "'a SET ⇒ 'a SET ⇒ 'a SET" where
  "UNION A B = Abs_SET (λ x. Rep_SET A x ∨ Rep_SET B x)"

(* properties in the demo theory file *)
```

## Sets as Functions

- since there is no restriction on the functions, one can see that `'a set` and `'a ⇒ bool` are isomorphic
- fact: in earlier version of Isabelle, `'a set` was just a type synonym for `'a ⇒ bool`
- current modeling provides separate views: the set of all even numbers is different from a function that decides whether a number is even

## Live Quiz: What do These Types Represent?

```
typedef 'a ty1 = "{ f :: 'a ⇒ nat . True}"

typedef 'a ty2 = "{ f :: 'a ⇒ nat . finite {x. f x > 0}}"

typedef 'a ty3 = "{ f :: nat ⇒ 'a . True}"

typedef 'a ty4 = "{ (n, f :: nat ⇒ 'a) . (∀ i. i < n ∨ f i = undefined) }"
```

## Example: Ordered Binary Trees

```
datatype 'a tree = Leaf | Node "'a tree" 'a "'a tree"

inductive ordered :: "'a :: linorder tree ⇒ bool"
(* standard definition *)

typedef (overloaded) ('a :: linorder)otree = "{t :: 'a tree. ordered t}"
```

- note: “`(overloaded)`” is required since the type variable `'a` has a type-class constraint
- advantage: when using `'a otree` guards such as “`ordered t` $\Longrightarrow$ …” are no longer required

## Example: Integers

model integer as pair of boolean (sign) and natural number; enforce fixed sign for 0

```
typedef INTEGER = "{ bn. case bn of (b,n :: nat) ⇒ n = 0 ⟶ b}" by auto

definition ZERO :: INTEGER where
  "ZERO = Abs_INTEGER (True, 0)"

(* define addition on representative type *)
fun add :: "bool × nat ⇒ bool × nat ⇒ bool × nat" where
  "add (True,n) (True,m) = (True, n+m)"
| "add (False,n) (False,m) = (False, n+m)"
| "add (True,n) (False,m) = (if m ≤ n then (True, n - m) else (False, m - n))"
| "add (False,n) (True,m) = (if n ≤ m then (True, m - n) else (False, n - m))"

(* and use this for definition of addition on abstract type *)
definition ADD :: "INTEGER ⇒ INTEGER ⇒ INTEGER" where
  "ADD x y = Abs_INTEGER (add (Rep_INTEGER x) (Rep_INTEGER y))"
```

# Properties of Type-Definitions

```
typedef INTEGER = "{bn. case bn of (b,n) ⇒ n = 0 ⟶ b}" by auto
```

besides getting a new type and the two conversion functions, obtain three import properties

- when switching from the abstract type `INTEGER` to the representation type `bool × nat` and then back to the abstract type we get the same abstract element

  ```
  lemma Rep_INTEGER_inverse: "Abs_INTEGER (Rep_INTEGER x) = x"
  ```

- when switching from the abstract type to the representation type, then that representative satisfies the predicate of the type

  ```
  lemma Rep_INTEGER:
    "Rep_INTEGER x ∈ {bn. case bn of (b,n) ⇒ n = 0 ⟶ b}"
  ```

- when switching from the representation type to the abstract type and then back to representation type we get the same representative, provided that the predicate of the type was satisfied

  ```
  lemma Abs_INTEGER_inverse: "y ∈ {bn. case bn of
    (b,n) ⇒ n = 0 ⟶ b} ⟹ Rep_INTEGER (Abs_INTEGER y) = y"
  ```

## Example: Properties of integer implementation

```
lemma "ADD x ZERO = x"

(* proof in the demo theory file *)
```

## Subtypes

consider modeling natural numbers as non-negative integers

```
typedef NAT = "{ n :: int. 0 ≤ n }"
```

- obviously, for addition and multiplication on type `NAT` we can just use addition and multiplication of type `int`
- therefore, properties like associativity and commutativity should directly carry from `int` to the subtype `NAT`
- in Isabelle this is not automatic: all properties have to be manually transferred, i.e., `NAT` is a different type than `int`
- by contrast there are theorem provers that support full subtyping, i.e., there `x :: NAT` implies `x :: int`, and therefore universally quantified properties on type `int` are immediately available for type `NAT`; example: in lemma `x :: int + y = y + x` both `x` and `y` can also be instantiated by numbers of type `NAT`

## Quotient-Types

- recall: `typedef` selects elements by predicate
- alternative: split universe into equivalence classes (quotient-type)
- example
  - model integers as pair of two natural numbers $(n, m)$ which model integer $n - m$
  - several representation are equivalent: $(1,3) \equiv (2,4) \equiv \ldots$
- in Isabelle

```
quotient_type int = "nat × nat" / "(λ(x, y) (u, v). x + v = u + y)"
```

- also for quotient types you will get conversion functions between abstract type and representative type
- further details: `isabelle doc isar-ref` (Chapter 11.9)

# Lifting and Transfer

## Motivation

- problems
  - working with Abs_type and Rep_type manually in definitions is tedious (inserting conversions at correct places is somehow trivial)
  - working with Abs_type and Rep_type in proofs is even more tedious
- solutions
  - the lifting package allows user to directly define functions on abstract type by just giving definition on representative type (automatic insertion of Abs_type and Rep_type)
  - the transfer package converts statements of abstract type into proof obligation that works on representative types (no reasoning on Abs_type and Rep_type required)
  - the predicate, that defined the abstract type, will become visible at certain places (proof obligation or precondition)

## Lifting Package

general workflow

- define type (via predicate $p$) as before
- make type-definition known to lifting package
- create several lifted definitions on abstract types by giving definitions on representative types
- whenever result of function contains abstract type, then a proof is required that resulting values satisfy $p$
  (but one can also assume that each input corresponding to the abstract type satisfies $p$)

## Transfer Package

general workflow

- given property on abstract type
- convert it into property on representative type
- one may assume that each representative element satisfies $p$

# Example: Integer Operations via Lifting Package

```
typedef INTEGER = "{ bn. case bn of (b,n :: nat) ⇒ n = 0 ⟶ b}" by auto

setup_lifting type_definition_INTEGER

lift_definition Zero :: INTEGER is "(True,0)" ⟨proof⟩
(* show that (True,0) satisfies predicate *)

fun add_integer :: "bool × nat ⇒ bool × nat ⇒ bool × nat" where ...

lift_definition Add :: "INTEGER ⇒ INTEGER ⇒ INTEGER" is add_integer ⟨proof⟩
(* show that add_integer bn1 bn2 satisfies predicate,
   whenever bn1 and bn2 satisfy predicate *)

lemma "Add x Zero = x"
proof transfer
(* show that add_integer bn (True,0) = bn,
   whenever bn satisfies predicate *)
```

## Goal-Cases

- `lift_definition` and `transfer` often produce completely new proof obligation (using representative types instead of abstract types)
- typing these manually is tedious (`fix` ... `assume` ... `show` ...)
- structured way to get access is via proof method `goal_cases`
  - `goal_cases` produces one case for each subgoal
  - `case (1 x y z)` starts the first subgoal where `x`, `y`, `z` are user-chosen names for the meta-quantified variables
  - then the label `1` refers to all assumptions and `show ?case` is the current conclusion that has to be shown
  - `next` separates the cases, and a full proof outline is available in output panel

## Demos

- demo of proofs of previous slides
- demo of binary search trees

## Final Remarks on Lifting and Transfer

- lifting- and transfer package are more versatile than the use-case that was illustrated here
- further informations
  - `isabelle doc isar-ref` (Chapter 11.9)
  - Brian Huffman and Ondřej Kunčar: Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL, CPP 2013