Course 2D1453, 2006-07

Advanced Formal Methods

Lecture 4: Isabelle – Types and Terms

Mads Dam
KTH/CSC

Some material from Paulson

Types in Isabelle

Types:

$$T ::= A \mid X \mid X :: C \mid T \Rightarrow T \mid (T_1,\ldots,T_n)\ K$$

where:

- $A \in$ {bool, int, ...} base type
- $X \in$ {'α, 'β,...} type variable
- $K \in$ {set, list,...} type constructor
  Used for defining new types
- $C \in$ {order, linorder, type,...} type classes
  Used for associating axioms to types

Examples:

- int list, int set ,...
- nat :: order, int :: field, ...

Introducing New Types

Types in Isabelle are nonempty

Theorem in HOL: $\exists$ x :: T . x = x

So all types must be inhabited

Three basic mechanisms:

- Type declarations
- Type abbreviations
- Recursive type definitions

Type Declarations

Syntax:
  typedecl K

Example:
  typedecl addr

Introduces an abstract type of addresses

Nothing known of an x :: addr

But: Some x :: addr exists

Type Abbreviations

Syntax:
  types ('α_1,...,'α_n) K = T

Examples:
  types number = nat
    tag = string
    'α taglist = ('α $\times$ tag) list

All type abbreviations are expanded in Isabelle
Not visible in internal representation or Isabelle output

Recursive Type Definitions

datatype 'α list = Nil | Cons 'α ('α list)

Defines a recursive datatype with associated constants:
  Nil :: 'α list
  Cons :: 'α $\Rightarrow$ 'α list $\Rightarrow$ 'α list

Plus axioms:
  Distinctness: Nil $\neq$ Cons x xs
  Injectivity: (Cons x xs = Cons y ys) = (x = y $\wedge$ xs = ys)

Also axioms for induction

Datatypes Generally

datatype $('\alpha_1,\ldots,'\alpha_n)\ K =$

$\quad \mathrm{constr}_1\ T_{1,1}\ \ldots\ T_{1,n_1}$

$\quad \ldots$

$\quad \mathrm{constr}_m\ T_{m,1}\ \ldots\ T_{m,n_m}$

Constants and types as previous slide

Note:

- Simplifier automatically extended with distinctness and injectivity
- Induction must be handled explicitly
- Not trivial that $(T_1,\ldots,T_n)\ K$ exists!
- Proof goals automatically added and discharged

This Scheme Does Not Always Work

Consider

datatype lam = mkfun (lam $\Rightarrow$ lam)

Note: Can interpret untyped lambda calculus using lam!

Problematic definition:

- Cardinality of $T \Rightarrow T$ *as set* is strictly greater than that of T, for *any* T
- So need to rule out most functions
- LCF and domain theory: $T \Rightarrow T$ is set of continuous functions on complete lattice or cpo
- LCF embedding in Isabelle exists

Simple Recursion

datatype $('\alpha_1,\ldots,'\alpha_n)\ K =$

$\quad \mathrm{constr}_1\ T_{1,1}\ \ldots\ T_{1,n_1}$

$\quad \ldots$

$\quad \mathrm{constr}_m\ T_{m,1}\ \ldots\ T_{m,n_m}$

Each type $T_{i,j}$ can be either:

- Non-recursive: All type constants K' in $T_{i,j}$ are defined "prior" to the definition of K
- An expression of the form $(T_1',\ldots,T_n')\ K$ where each T_k' is non-recursive

Mutual Recursion

datatype

$('\alpha_1,\ldots,'\alpha_n)\ K =$

$\quad \mathrm{constr}_1\ T_{1,1}\ \ldots\ T_{1,n_1}$

$\quad \ldots$

$\quad \mathrm{constr}_m\ T_{m,1}\ \ldots\ T_{m,n_m}$

and

$('\alpha_1',\ldots,'\alpha_n')\ K'$

$\quad \mathrm{constr}_1'\ T_{1,1}'\ \ldots\ T_{1,n_1'}'$

$\quad \ldots$

$\quad \mathrm{constr}_{m'}'\ T_{m',1}'\ \ldots\ T_{m',n_{m'}'}'$

Each $T_{i,j}$, $T_{i,j}'$ is either non-recursive or of the form ... K or ... K'

Covariance and Contravariance

Introduce relations $X \leq^+ T$ and $X \leq^- T$

- $X \leq^+ T$: T is covariant in X
- $X \leq^- T$: T is contravariant in X

$$\frac{-}{X \leq^+ X} \qquad \frac{X \leq^+ T_1 \quad X \leq^- T_2}{X \leq^- T_1 \Rightarrow T_2} \qquad \frac{X \leq^- T_1 \quad X \leq^+ T_2}{X \leq^+ T_1 \Rightarrow T_2}$$

$$\frac{X \leq^+ T_i \quad 1 \leq i \leq n}{X \leq^+ (T_1,\ldots,T_n)\ K} \qquad \frac{X \leq^- T_i \quad 1 \leq i \leq n}{X \leq^- (T_1,\ldots,T_n)\ K}$$

Covariance = monotonicity: As *sets*, if $X \leq^+ T$ then $A \subseteq B$ implies $T[A/X] \subseteq T[B/X]$

Contravariance = antimonotonicity: If $X \leq^- T$ then $A \subseteq B$ implies $T[B/X] \subseteq T[A/X]$

Nested Recursion

datatype $('\alpha_1,\ldots,'\alpha_n)\ K =$

$\quad \mathrm{constr}_1\ T_{1,1}\ \ldots\ T_{1,n_1}$

$\quad \ldots$

$\quad \mathrm{constr}_m\ T_{m,1}\ \ldots\ T_{m,n_m}$

Each type $T_{i,j}$ is of form

$\quad T[(T_{1,1}',\ldots,T_{1,n}')\ K/X_1,\ldots,[(T_{k,1}',\ldots,T_{k,n}')\ K/X_k]$

such that

- $X_i \leq^+ T$ for all i: $1 \leq i \leq k$
- Any K' occurring in T is defined prior to K

Note: Simple recursion is special case

Mutual, nested recursion possible too

Type Classes

Used to associate axioms with types

Example: Preorders

axclass ordrel < type
consts le :: ('α :: ordrel) $\Rightarrow$ 'α $\Rightarrow$ bool

axclass preorder < ordrel
orderrefl: le x x
ordertrans: (le x y) $\wedge$ (le y z) $\Rightarrow$ le x z

Advanced topic – return to this later

Terms in Isabelle

Terms:

t ::= x | c | ?x | t t | λx. t

where:

- x $\in$ Var – variables
- C $\in$ Con – constants
- ?x – schematic variable
- λx. t - must be typable

Schematic variables:

- Free variables are fixed
- Schematic variables can be instantiated during proof

Schematic Variables

State lemma with free variables
lemma foobar : f(x,y) = g(x,y)
...
done

During proof: x, y must never be instantiated!

After proof is finished, Isabelle converts free var's to schematic var's

f(?x,?y) = g(?x,?y)

Now can use foobar with ?x $\mapsto$ f and ?y $\mapsto$ a, say

Defining Terms

Three basic mechanisms:

- Defining new constants non-recursively
  No problems
  Constructs: **defs**, **constdefs**
- Defining new constants by primitive recursion
  Termination can be proved automatically
  Constructs: **primrec**
- General recursion
  Termination must be proved
  Constructs: **recdef**

Non-Recursive Definitions

Declaration:
consts
sq :: nat $\Rightarrow$ nat

Definition:
defs
sqdef: sq n = n * n

Or combined:
constdefs
sq :: nat $\Rightarrow$ nat
sq n = n * n

Unfolding Definitions

Definitions are not always unfolded automatically by Isabelle

To unfold definition of sq:
apply(unfold sqdef)

Tactics such as simp and auto do unfold constant definitions

Definition by Primitive Recursion

consts
  append :: 'α list ⇒ 'α list ⇒'α list
primrec
  append Nil ys = ys
  append (Cons x xs) ys = Cons x (append xs ys)

Append applied to strict subterm xs of Cons x xs:
  Termination is guaranteed

Primitive Recursion, General Scheme

Assume data type definition of T with constructors $constr_1, \ldots, constr_m$

Let $f :: T_1 \Rightarrow \ldots \Rightarrow T_n \Rightarrow T'$ and $T_i = T$

Primitive recursive definition of f:
  $f\ x_1 \ldots (constr_1\ y_1 \ldots y_{k_1}) \ldots x_n = t_1$
  ...
  $f\ x_1 \ldots (constr_m\ y_1 \ldots y_{k_m}) \ldots x_n = t_m$

Each application of f in $t_1, \ldots, t_m$ of the form $f\ t_1' \ldots y_{k_j} .. t_n'$

Partial Functions

datatype 'α option = None | Some 'α

Important application:
  T ⇒ 'α option ≈ partial function:
    None ≈ no result
    Some t ≈ result t
Example:
consts lookup :: 'α ⇒ ('α × 'β) list ⇒ 'β option
primrec
  lookup k [] = None
  lookup k (x#xs) =
    (if fst x = k then Some(snd x) else lookup k xs)

The Case Construct

Every datatype introduces a *case* construct, e.g.
  (case xs of Nil ⇒ . . . | (Cons y ys) ⇒ ... y ... ys ...)

In general: one case per constructor
- No nested patterns, e.g. Cons y_1 (Cons y_2 ys)
- But cases can be nested

Case distinctions:
  apply(case_tac t)
creates k subgoals
  $t = constr_i\ y_1 \ldots y_{k_i} \Rightarrow \ldots$
one for each constructor $constr_i$

Mutual and Nested Primitive Recursion

Primitive recursion scheme applies also for mutual and nested recursion

Assume data type definition of T^1 and T^2 with constructors $constr_1^1, \ldots, constr_{m_1}^1, constr_1^2, \ldots, constr_{m\{2\}}^2$, respectively

Let:
  $f :: T_1 \Rightarrow \ldots \Rightarrow T_{n_f} \Rightarrow T_f', T_i = T^1$,
  $g :: T_1 \Rightarrow \ldots \Rightarrow T_{n_g} \Rightarrow T_g', T_j = T^2$

Mutual and Nested Recursion, II

Mutual, primitive recursive definition of f and g:
  $f\ x_1 \ldots (constr_1^1\ y_1 \ldots y_{k_{1,1}}) \ldots x_{n_f} = t_{1,f}$
  ...
  $f\ x_1 \ldots (constr_{m_1}^1\ y_1 \ldots y_{k_{m1,1}}) \ldots x_{n_f} = t_{m_1,f}$

  $g\ x_1 \ldots (constr_1^2\ y_1 \ldots y_{k_{1,2}}) \ldots x_{n_g} = t_{1,g}$
  ...
  $g\ x_1 \ldots (constr_m^2\ y_1 \ldots y_{k_{m2,2}}) \ldots x_{n_g} = t_{m_2,g}$

Each application of f or g in $t_{1,f}, \ldots, t_{m_1,f}, t_{1,g}, \ldots, t_{m_2,g}$ of the form $h\ t_1' \ldots y_k \ldots t_n'$, $h \in \{f,g\}$
Slightly more general schemes possible too

General Recursion

In Isabelle, recursive functions must be proved total before they ”exist”

General mechanism for termination proofs: Well-founded induction

Definition: Structure (A,R) is *well-founded*, if for every non-empty subset B of A there is some $b \in B$ such that *not* b' R b for any $b' \in B$.

Well-foundedness ensures that there cannot exist any infinite sequence $a_0, a_1, \ldots, a_n, \ldots$ such that a_{n+1} R a_n for all $n \in \omega$. Why?

Examples: The set of natural numbers under < is well-ordered. The set of reals is not.

Well-founded Induction

Principle of well-founded induction:

Suppose that (A,R) is a well-founded structure.

Let B be a subset of A.

* Suppose $x \in A$ and $y \in B$ whenever y R x implies $x \in B$.

Then A = B

Here: A is the type, B is the property. Goal is $\forall a :: A.\ a \in B$

Proof: For a contradiction suppose $A \neq B$. Then A – B is nonempty. Since (A,R) is well-founded, there is some $a \in A - B$ such that not a' R a for all $a' \in A - B$. But $a \in A$ and whenever y R a then $y \in B$. But then by (*), $a \in A$, a contradiction.

Well-founded Induction in Isabelle

consts

  $f :: T_1 \times \ldots \times T_n \Rightarrow T$

recdef f R

  $f(pattern_{1,1},\ldots,pattern_{1,n}) = t_1$

  ...

  $f(pattern_{m,1},\ldots,pattern_{m,n}) = t_m$

where

1. R well-founded relation on T
2. Defining clauses are exhaustive
3. Definition bodies $t_1,\ldots,t_m$ can use f freely
4. Whenever $f(t_1',\ldots,t_n')$ is a subterm of t_i then $(t_1',\ldots,t_n')$ R $(pattern_{i,1},\ldots,pattern_{i,n})$

Recdef Using Progress Measures

Let $g :: T_1 \times \ldots \times T_n \rightarrow nat$

Define: measure g = $\{(t_1,t_2) \mid g\ t_1 < g\ t_2\}$

Then can use instead:

recdef f (measure g)

  $f(pattern_{1,1},\ldots,pattern_{1,n}) = t_1$

  ...

  $f(pattern_{m,1},\ldots,pattern_{m,n}) = t_m$

and condition 4. becomes:

- Whenever $f(t_1',\ldots,t_n')$ is a subterm of t_i then $g(t_1',\ldots,t_n') < g(pattern_{i,1},\ldots,pattern_{i,n})$

Example: Fibonacci

consts fib :: nat $\Rightarrow$ nat

recdef fib (measure (λn. n))

fib 0 = 0

fib (Suc 0) = 1

fib (Suc(Suc x)) = fib x + fib (Suc x)

Many more examples in tutorial

Exercises

Exercise 1:

Define a little imperative language of booleans b and commands c as follows

  $b ::= b_a \mid \text{not } b \mid b \text{ and } b$

  $c ::= c_a \mid \text{if } b\ c\ c \mid \text{while } b\ c \mid c\ ;\ c \mid \text{done}$

b_a is an atomic boolean, and c_a an atomic command. Represent the languages as a mutually recursive datatype in Isabelle. Define the semantics of booleans as a function

  boolsem :: boolean $\Rightarrow$ state $\Rightarrow$ bool

  cmdsem :: cmd $\Rightarrow$ state $\Rightarrow$ cmd $\Rightarrow$ state $\Rightarrow$ bool

where state is a primitive type. The idea of cmdsem is that cmdsem c1 s1 c2 s2 = true iff one step of evaluation of c1 in state s1 results in state s2 with command c2 left to evaluate. Make suitable assumptions on atomic booleans and commands. In particular, assume that evaluation of atomic commands is deterministic. Represent the languages and semantics in Isabelle, and prove that command evaluation is deterministic.

Exercise 2: Derive (pen and paper) natural number induction from well-founded induction