ENSEIRB-Matmeca
Year 2024-2025

IF Course
IT234

Tutorial #7 - Types and Proofs

Exercice 1: Phantom types in OCaml

A *phantom type* is a type parameter having no influence at runtime. There are used as indications (tags) for the type system. In this exercise, we use these indications to represent logic properties on the values. Let us define the following types for denoting emptiness and non-emptiness :

```
type empty
type nonempty
```

These types are uninhabited[1] : they stand for an empty set of values. Now it becomes possible to use these tags to make the compiler check particular expressions, depending on the tags. In the following example, the `PList` module constructs lists whose type also encodes the property of its emptiness :

```
type empty
type nonempty

module PList : sig (* The signature / interface of the module *)
  type 'a plist
  val nil  : empty plist
  val cons : int → 'a plist → nonempty plist
  val head : nonempty plist → int
end = struct       (* The implementation of the module *)
  type ilist    = Nil | Cons of int * ilist
  type 'a plist = ilist   (* Add a phantom type parameter *)
  let nil = Nil
  let cons x xs = Cons (x, xs)
  let head l = match l with
    | Nil → assert false (* Never reached *)
    | Cons (x, _) → x
end;;
```

1. Create an empty list, and a list containing at least two elements.
   For each of them, note the corresponding type. Verify that taking the head of an empty list is statically detected as an error.
2. What type would the `tail` function have in this setting ? Write the associated function. What is missing here in order to preserve our property ?

1. Beware of the false friend in English : *inhabited* (habité) and *uninhabited* (inhabité).

In order to overcome this limitation, we propose to encode the length of the list directly into the type parameter. Let us begin with the following phantom types :

```
type zero
type 'n succ
```

3. What family of types does this define ? Are they inhabited ?
4. Propose type for generic lists with a phantom type that contains their length encoded with `zero` and `succ`.
5. Rewrite the previous module so as to preserve the length property in the types.

Exercice 2: Phantom types in Haskell

Consider again the problem of creating a datatype for lists whose type contains enough information to determine the length of the list. Let us switch to the Haskell language, that allows other constructs on types that simplify this encoding. For the moment, let us begin with the following definition for lists :

```
{-# LANGUAGE GADTs, TypeOperators, StandaloneDeriving #-}

data Zero
data Succ n

data BList size a where                          -- Lists with encoded length
  Nil  :: BList Zero a                           -- * empty list
  Cons :: a → BList n a → BList (Succ n) a       -- * cons
deriving instance Show a ⇒ Show (BList size a)   -- Allow lists display
```

Considering the question of finding the length of a list, let us bijectively associate to each of our Peano types a value in the following way :

```
data Nat size where                  -- Singleton type for Zero and Succ
  Z :: Nat Zero                      -- * zero
  S :: Nat n → Nat (Succ n)          -- * successor
deriving instance Show (Nat size)    -- Allow Nat display
```

1. Write a `size` function on the `BList n a`. In which way does the type of this function express that a list of n elements possesses effectively n elements ?

Now, suppose that we want to be able to access an element of the list in a type-safe way, through a `get` function. The `get` function takes two arguments, namely a list and an index, and returns the element existing at that index if the index is less than the length of the list. Here, we need to be able to encode the fact that an integer is smaller than another. Consider the following definition for the type "`n :< m`" :

```haskell
data n :< m where                          -- Type witness for 'n < m'
  LessZ :: Zero :< Succ n                  -- LessZ is a proof that '0 < S x' forall x
  LessS :: (m :< n) → Succ m :< Succ n     -- LessS is a proof that 'n < m' implies 'S n < S m'
deriving instance Show (n :< m)
```

2. If we read these types as logical propositions, what do they represent ?
3. The return of the following `lt` function is called a *witness*.
   What does it witness ?

```haskell
lt :: Nat n → Nat m → Maybe (n :< m)
lt Z     (S n) = Just LessZ
lt (S n) (S m) = case lt n m of
                   Just proof → Just (LessS proof)
                   Nothing    → Nothing
lt _     _     = Nothing
```

4. Write the code of a `get` function that is type-safe.
5. In which way is this kind of programming different from the usual ?

▷ LiquidHaskell (`https://ucsd-progsys.github.io/liquidhaskell`) is an extension of the Haskell language that enhances the type system with *refinement types*, a particular sort of type that may contain a logic predicate. The language is associated to a SMT solver that checks whether the logic constraints given in the types can be satisfied. This code defines a type-safe *head* function on lists :

```haskell
{-@ type NonEmpty A = ((len A) > 0) @-}        -- Type definition

{-@ head   :: {v:[a] | (NonEmpty v)} → a @-} -- Liquid Haskell annotation
head (x:_) = x
head []    = liquidError "Never reached"
```

The source code in LiquidHaskell is written in `.hs` files, and is checked with a `liquid` executable (working in combination with the `z3` solver).
One can use the online version at `https://liquidhaskell.goto.ucsd.edu/index.html`.

Exercice 3: Proofs in LiquidHaskell
In the following example in LiquidHaskell, a list of integers is annotated with a more precise type that alows only even integer values inside :

```haskell
{-@ type Even = { v : Int | v mod 2 = 0 } @-} -- Type definition

{-@ weAreEven :: [Even] @-}                  -- Liquid Haskell annotation
weAreEven :: [Int]                           --  Plain Haskell annotation
weAreEven  = [-10, 4, 0, 2, 666]
```

1. What happens when adding an odd value inside the list and checking the code ?
2. The following functions are meant to manipulate `Even` values. Complete the code with the appropriate annotations to enforce this fact.

```
{-@ isEven :: Nat → Bool @-}
isEven   :: Int → Bool
isEven 0 = True
isEven 1 = False
isEven n = not (isEven (n-1))

shift :: [Int] → Int → [Int]
shift xs k = [x + k | x ← xs]

double :: [Int] → [Int]
double xs = [x + x | x ← xs]

{-@ range :: lo:Int → hi:Int → [{v:Int | (lo <= v && v < hi)}] / [hi-lo] @-}
range :: Int → Int → [Int]
range lo hi
  | lo < hi   = lo : range (lo+1) hi
  | otherwise = []

evens :: Int → [Int]
evens n = [i | i ← range 0 n, isEven i]
```

Consider now the following types for lists encoding their length :

```
{-@ type ListN a N  = { v:[a] | len v == N }        @-}
{-@ type Btwn Lo Hi = { v:Int | Lo <= v && v < Hi } @-}
{-@ type NEList a   = { v:[a] | 0 < len v }         @-}
```

3. Propose a safe type for the average function.

```
{-@ avg   :: [Int] → Int @-}
avg       :: [Int] → Int
avg xs    = divide total n where
  total = sum xs
  n     = length xs
```

4. Propose safe types for the `head` and `tail` functions on lists
5. Consider the source code for the `loop` and `dotProduct` functions (given below), and check their types. Write an example that type checks in Haskell and that does not in LiquidHaskell.

```
{-@ loop :: lo:Nat → hi:{Nat|lo <= hi} → a → (Btwn lo hi → a → a) → a @-}
loop :: Int → Int → a → (Int → a → a) → a
loop lo hi base f = go2 base lo where
  {-@ go2 :: a → {i:Nat | i >= lo } → a / [hi-i] @-}
  go2 acc i  | i < hi    = go2 (f i acc) (i + 1)
             | otherwise = acc

{-@ dotProduct :: x:[Int] → { y:[Int] | len x = len y } → Int @-}
dotProduct :: [Int] → [Int] → Int
dotProduct x y = loop 0 sz 0 body
  where
    sz         = length x
    body i acc = acc + (x !! i) * (y !! i)

nhead :: [a] → a
nhead vec = head vec

ntail :: [a] → [a]
ntail l = tail l
```

▷ Coq is a proof assistant written in OCaml and based on the calculus of constructions, a higher-order typed λ-calculus, initially developed by Thierry Coquand. This calculus contains (among others) the 2nd-order λ-calculus. One cas use the online version at `https://jscoq.github.io/scratchpad.html`

Exercice 4: Proofs in Coq

Coq is a proof assistant, and therefore its main object consists in proving theorems. The general idea for proving theorems consists in defining a series of axioms as rewriting rules, and then deducing more general theorems by applying these rewriting rules in a particular manner.

Caveat : this exercise is just an introduction to Coq, makes a series of oversimplifications, and is certainly *not* sophisticated enough to provide a full understanding of its possibilities. Readings of interest are given at the end of the exercise.

Coq contains in its standard library the following definition for integers :

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

For an introduction, we propose to work with the following addition function :

```
(* A definition of plus for nat values *)
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O ⇒ m
  | S p ⇒ S (plus p m)
  end.
```

1. Execute the computations given in the source file one by one, and examine the results.

A *theorem* in Coq is the expression of a logical formula. A *proof* is a sequence of rewriting steps leading to a trivially true expression. The following is an example of a very simple theorem :

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n.
  simpl.
  reflexivity.
Qed.
```

Each step between `Proof` and `End` is a particular operation on expressions.

2. Execute the steps of the proof one by one, checking the state of the proof at each step.
3. Prove that $\forall$`n m`, `S n + m` = `S (n+m)`.

For more complex proofs, more complex tactics are required. In the following proofs, it is necessary to do an `induction n`.

4. Prove that $\forall$`n`, `n + 0` = `n`.
5. Prove that $\forall$`n m`, `n + S m` = `S (n+m)`.
6. Prove that the addition is commutative.

Coq allows a certain form of programming with proofs, mixing code and propositions. In the following example taken from A. Chlipala, a definition is given for lists with their length encoded into their type :

```
Section list_length.

  Inductive ilist (A : Type) : nat → Type :=
  | Nil : ilist A O
  | Cons : forall n, A → ilist A n → ilist A (S n).

  Arguments Nil [A].
  Arguments Cons [A] [n] _ _.

  Inductive fin : nat → Set :=
  | First : forall n, fin (S n)
  | Next : forall n, fin n → fin (S n).

  Arguments First [n].
  Arguments Next [n] _.

  Fixpoint get A n (ls : ilist A n) : fin n → A :=
    match ls with
      | Nil ⇒ fun idx ⇒
        match idx in fin n' return (match n' with
                                      | O ⇒ A
                                      | S _ ⇒ unit
                                    end) with
          | First ⇒ tt
          | Next _ ⇒ tt
        end
      | Cons x ls' ⇒ fun idx ⇒
        match idx in fin n' return (fin (pred n') → A) → A with
          | First ⇒ fun _ ⇒ x
          | Next idx' ⇒ fun get_ls' ⇒ get_ls' idx'
        end (get _ _ ls')
    end.

  Arguments get [A] [n] _ _.

End list_length.
```

Check that the types and the computations both mix types and integer values.

7. Write an example of a call to `get` that type-checks and one that does not because of an unsafe array access.

For further reading about programming and proving theorems with Coq :

— B. C. Pierce et al. *Logical Foundations*, available at `https://softwarefoundations.cis.upenn.edu/`

— A. Chlipala *Certified Programming with Dependent Types*, available at `http://adam.chlipala.net/cpdt/`