

Type Systems and Programming

D. Renault

ENSEIRB-Matmeca

Apr. 9th 2025, v.1.5.1

<https://www.labri.fr/perso/renault/working/teaching/stp/stp.php>

1 Proofs with types

- Phantom types
- Refinement types
- Dependent types

General idea

Use types to enforce logic properties on the values they represent.


Examples

- having values that can be compared (`Comparable`, `Eq a ...`)
- having numeric-like values (`Number`, `Num a ...`)
- having a list-like representation (`Cons`, `Nil`)

More generally, any form of interface can be seen as a logic property.

Can we generalize and find other sorts of properties represented by types?

Representation of particular sets of values

Consider the following algebraic data type  for representing lists :

$$\text{List}[T] ::= \text{Nil} \quad | \quad \text{Cons}(T, \text{List}[T])$$

Suppose that this type is realized with the following constructors :

- $\text{nil} : \text{List}[T]$ is the empty list
- $\text{cons} : T \rightarrow \text{List}[T] \rightarrow \text{List}[T]$ is a constructor for lists.

The usual accessors `head` and `tail` are provided :

- $\text{head} : \text{List}[T] \rightarrow T$ returns the first element
- $\text{tail} : \text{List}[T] \rightarrow \text{List}[T]$ returns all but the first element.

These accessors are **problematic** : they are not defined for `nil`.
Could these accessors be typed for only non-empty lists ?

Refinement of sets of values

In this very case, it is natural to define two types :

- `EmptyList` containing the `nil` value
- `NonEmptyList[T]` containing the non-empty lists

Both types are naturally subtypes of `List[T]`.

With this refinement, the accessors can be defined as total functions :

- `head` : `NonEmptyList[T] → T` returns the first element
- `tail` : `NonEmptyList[T] → List[T]` returns all but the first element.

`(head_nil)` becomes a **non-typable** expression instead of stuck at runtime.

This idea can be generalized, restricting values to :

- non-zero or positive numeric values,
- open file descriptors in contrast to closed ones,
- non-null pointers in contrast to null ones.

It is desirable to have the possibility to refine the implementation and the logical properties **independently**.

- Otherwise, an implementation of an `AssocList[T] <: List[T]` must provide code for both empty and non-empty lists.
- One must devise a mechanism for attaching logical properties to existing types without hindering the usual inheritance mechanisms.

Phantom types

Definition (Phantom type)

A type T is said to be a **phantom type** if it has no influence at runtime, i.e its values never occur in any computation.

A type variable T in a parameterized type $F[T]$ is said to be a **phantom type** if it is only meant to be applied to phantom types.

Examples

```
interface Phantom {} // In Java
```

```
type phantom (* In OCaml *)
```

A sufficient condition to be a phantom type is to stand for the empty set of values.

Example : write-restricted objects

Consider two phantom types `readonly` and `readwrite`.

Let us create a parameterized type `Readable[T]` such that `T` constrains its capabilities : only `Readable[readwrite]` can be modified.

```
(* Interface *)
module type REF = sig
  type 'a t
  val create : int → readwrite t
  val set    : readwrite t → int → unit
  val get    : 'a t → int
  val freeze : 'a t → readonly t
end
```

```
(* Implementation *)
module Ref : REF = struct
  type 'a t    = int ref
  let create x = ref x
  let set r x  = r := x
  let get r    = !r
  let freeze x = x
end
```

```
let rw = create 4;;
let ro = freeze rw;;
set ro 7;; (* Type error : This expression has type readonly t *)
           (* but an expression was expected of type readwrite t *)
```


Example : write-restricted objects


Consider two phantom types `readonly` and `readwrite`.

Let us create a parameterized type `Readable[T]` such that `T` constrains its capabilities : only `Readable[readwrite]` can be modified.

```
class Readable<A extends Access> {  
  int val;  
  Readable(int t){ val = t; }  
  
  static Readable<ReadWrite> create(int t)           { .. }  
  static void          set(Readable<ReadWrite> c, int t) { .. }  
  static int          get(Readable<?> c)             { .. }  
  static Readable<ReadOnly> freeze(Readable<?> c)   { .. } }
```

```
Readable<ReadWrite> rw = Readable.create(5);  
Readable<ReadOnly>  ro = Readable.freeze(rw);  
Readable.set(ro, 11); // Incompatible types: Readable<ReadOnly> cannot  
                      // be converted to Readable<ReadWrite>
```

Definition (GADT)


A **generalized algebraic datatype** is an algebraic datatype  containing a phantom type, and whose constructors can enforce restrictions on the phantom type.

Example

```
type _ data =  
  | Int  : int          →      int data  
  | Str  : string       →      string data  
  | Pair : 'a data * 'b data → ('a * 'b) data
```

```
let x = Int 1 and y = Str "one" in Pair(x, y);; (* → (int*string) data *)
```

Definition (GADT)

A **generalized algebraic datatype** is an algebraic datatype  containing a phantom type, and whose constructors can enforce restrictions on the phantom type.

Example

```
type _ data =  
  | Int  : int           →      int data  
  | Str  : string        →      string data  
  | Pair : 'a data * 'b data → ('a * 'b) data
```

```
let add (Int u) (Int v) = Int(u+v);; (* int data → int data → int data *)
```

Example : typed evaluator

In this example, a GADT is used to represent typed computations :

```
type _ expr =  
  | Bool : bool → bool expr  
  | Int  : int  → int  expr  
  | If   : bool expr * 'a expr * 'a expr → 'a expr  
  | Eq   : 'a expr * 'a expr → bool expr  
  | Add  : int  expr * int  expr → int  expr
```

The `eval` function returns the value encapsulated inside the expression :

```
let rec eval : type a. a expr → a = function (* with type  $\forall T, (Expr[T] \rightarrow T)$  *)  
  | Bool b      → b  
  | Int i       → i  
  | If (b, l, r) → if eval b then eval l else eval r  
  | Eq (a, b)   → (eval a) = (eval b)  
  | Add (a,b)   → (eval a) + (eval b) ;; (* Addition on integers *)
```

- A GADT value is an existential value, involving runtime checking.
- The compiler checks the constraints for each constructor individually.

Reification of types

- Once phantom types have been attached to other types, it becomes natural to apply computations on these.

<code>plus(int, float) ⇒ float</code>	<code>append(Int, String) ⇒ Vector<Any></code>
<code>plus(int, int) ⇒ int</code>	<code>append(Char, String) ⇒ String</code>

- Not all languages allow computations at the type level, and therefore mimic these computations at the value level.

Definition (Reification)

A set of types $\mathcal{T} ::= \{T_i\}$ is said to be **reified** into a set of values $\mathcal{V} ::= \{v_i\}$ if there exists a bijection between the \mathcal{T} and \mathcal{V} .

Ideally, the set \mathcal{V} is represented as (another) type supporting this bijection. If both sets are of size 1, the type is called a **singleton type**.

Example : reification of naturals

In this example, the phantom types represent the **Peano naturals** :

```
type zero      (* Type for representing zero *)
type 'a succ   (* Type for representing the successor *)

type _ nat =   (* Bridge Nat[T] between values and types *)
| NZ : zero nat      (* Value for representing zero *)
| NS : 'a nat → ('a succ) nat (* Value for representing the successor *)
```

- `NZ` and `NS` are values typed by `Nat[T]` in bijection with the naturals :

$$\underbrace{(\text{NS } \dots \text{ NS } \text{NZ})}_{k \text{ times}} : \underbrace{(\text{succ } \dots \text{ succ } \text{zero})}_{k \text{ times}}$$

- Computations on types can be carried over onto values :

```
let rec nat_to_int : type a. a nat → int = fun x → match x with
| NZ   → 0
| NS n → 1 + nat_to_int n
```

Example : length-encoded lists

Let's extend this example to create lists whose type contains their length :

```
type (_,_) seq = (* 1st parameter = type of elements, 2nd parameter = length *)  
  | Nil : ('a, zero) seq  
  | Cons : 'a * ('a,'n) seq → ('a, 'n succ) seq
```

```
let rec head : type a n. (a, n succ) seq → a = function  
  | Cons(x, _) → x
```

```
let rec tail : type a n. (a, n succ) seq → (a, n) seq = function  
  | Cons(_, s) → s
```

```
let rec map : type a b n. (a → b) → (a,n) seq → (b,n) seq =  
  fun f l → match l with  
  | Nil → Nil  
  | Cons (x, s) → Cons (f x, map f s)
```

- The type of `map` encodes the fact that it preserves the length of lists.

Example : length-encoded lists

- In Haskell, it's even possible to express computations on types :

```
data Zero          -- Phantom types for naturals
data Succ nat

type family nat1 :+: nat2 :: * -- Type family for the “:+” function on naturals

type instance Zero      :+: nat2 = nat2
type instance Succ nat1 :+: nat2 = Succ (nat1 :+: nat2)
```

- This yields the following type for the concatenation on lists :

```
(++) :: List a len1 → List a len2 → List a (len1 :+: len2)
Nil      ++ list = list
Cons el  els ++ list = Cons el (els ++ list)
```

- The type of ++ encodes the fact that the length of the concatenation of two lists is the sum of the lengths of its components.

Composition of static properties

Problem

Annotated types do not compose well in general.

Computations
returning an integer

Computations
applying a division

1) Consider the example of the `mean` function on lists of integers :

```
mean ::= fun l → let n = List.length l in (sum l) / n
```

- The `length` function **must** return a generic non-negative integer.
- The division function **should** take a generic positive integer.

Composition of static properties

Problem

Annotated types do not compose well in general.

Computations re-
turning a list

Computations taking
the **head** of a list

2) Consider a function returning the first even integer in a list :

```
fst_even ::= fun l → let m = filter is_even l in head m
```

Without static knowledge that `m` is non-empty, one must check dynamically.

Composition of static properties

Problem

Annotated types do not compose well in general.

Computations re-
turning a list

Computations taking
the **head** of a list

2) Consider a function returning the first even integer in a list :

```
fst_even ::= fun l → let m = filter is_even l in head m
```

Without static knowledge that `m` is non-empty, one must check dynamically.

This is a consequence of the **undecidability of evaluation** : logic properties that evolve at runtime cannot be decided statically in general.

Strategy for composing properties

In some cases, it is possible to provide a **static** proof of the property.

Consider the problem of accessing the n -th element of a list :

`get : Nat → List[T] → T`

How could we make `get` access only concrete indices of the list ?

- Construct a type `Leq[m, n]` expressing the fact that $m \in [0; n[$:

```
data Leq[m,n] where
  LessZ :: Leq[Zero, Succ n]           -- Proof that :
  LessS :: Leq[m,n] → Leq[Succ m, Succ n] -- if m < n, then m+1 < n+1
```

- A value of type `Leq[m, n]` is computed dynamically when required.

Example : lists with safe access

- The GADT reifying the property $m < n$:

```
data Leq[m,n] where
  LessZ :: Leq[Zero,Succ n]
  LessS :: Leq[m,n] → Leq[Succ m,Succ n]
```

- The `less-than` function computes a proof that $m < n$ (if any) :

```
lt :: Nat m → Nat n → Maybe (Leq[m,n])
lt Zero      (Succ n) = Just LessZ
lt (Succ m) (Succ n) = case lt m n of Some proof → Some (LessS proof)
                                Nothing      → Nothing
lt _         _       = Nothing
```

- The type-safe `get` function can only access safe indices of a list :

```
get :: Leq[m,n] → List[a,n] → a
get LessZ      (Cons x xs) = x
get (LessS k) (Cons x xs) = get k xs
```

Refinement types

Going further along these lines, it is possible to attach a proof-checker to help the compilation phase, as is done in Liquid Haskell or in Dafny.

Consider the problem of defining a type-safe `divide` function on integers :

```
type NonZero = { v : Int | v /= 0 } -- type for non-zero integers

divide :: Int → NonZero → Int
divide _ 0 = die "divide_by_zero" -- can never happen
divide n d = n `div` d
```

- A type attached with a logical property is called a **refinement type**.
- Logical assertions are transferred and checked by a SMT solver.

Example : iterating on vectors (1)

Here, `loop` iterates a function over the integers in the interval `[lo;hi[` :

```
loop :: lo:Nat → hi:{Nat|lo <= hi} → a → (Btwn lo hi → a → a) → a
loop lo hi base f = loop_rec base lo where
  loop_rec acc i | i < hi    = loop_rec (f i acc) (i + 1)
  loop_rec _   _ | otherwise = acc
```

Typically, `loop 0 n x0 f` computes the sequence :

$$\begin{cases} x_0 \text{ given} \\ x_{k+1} = f(k, x_k) \end{cases}$$

The type of the `loop` function is verified by the compiler and ensures that :

- $lo \leq hi$, forming an interval `Btwn lo hi ::= [lo;hi[`;
- `f` accesses only integers in the interval `Btwn lo hi`.

Example : iterating on vectors (2)

The `loop` function can then be used to write a `dotProduct` function :

```
loop :: lo:Nat → hi:{Nat|lo <= hi} → a → (Btwn lo hi → a → a) → a
```

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

- The compiler is able to infer that the indices accessed are always valid.
- This function **only** requires a proof that both vectors have same length.
- It does **not** need to check that all the array accesses are safe.

Termination proofs

Liquid Haskell is able to prove the **termination** of the following function :

```
fib :: i:Int → Int
fib i | i == 0    = 0
      | i == 1    = 1
      | otherwise = fib (i-1) + fib (i-2)
```

- Applying a series of well-chosen heuristics, the compiler finds a well founded metric that decreases at each recursive call.
- More generally, it can automatically prove termination for a particular but **expressive** class of recursive functions (▶ strong normalization).

... which in itself is a pretty amazing feat.

The frontier of automaticity

In some cases, the compiler is not able to infer the proofs **automatically**.

- More complex calculi exist with particularly powerful type systems.
Examples : Martin-Löf's type theory, the calculus of constructions ...
- As type inference became undecidable for λ_2 , it is not surprising that it remains **undecidable** for more powerful calculi.

These proofs may be provided, possibly with the help of a proof-assistant.

- Proofs become another software component, at the same level as code.
Examples : languages with proof assistants such as Coq, Agda, Idris, ...

Definition (Dependent type)

A **dependent type** is a type whose definition is parameterized by a value.

Note : allowing values inside types dramatically complexifies a type system

Example

The type $\text{Vec}[n, A]$ of the vectors of n elements of type A .

It is technically called a dependent product written $\prod_{n \in \mathbb{N}} \text{Vec}_n[A]$.

```
Inductive vec a : nat → Type := (* Dependent type written as a function *)
  | nil : vec a 0
  | cons : forall (h:a) (n:nat), vec a n → vec a (S n).
```

```
Definition hd {a} {n} (v:vec a (S n)) : a
```

```
Definition tl {a} {n} (v:vec a (S n)) : vec a n
```

```
Definition nth {a} {n} {p} (v:vec a n) (H: p < n) : a
```

```
Fixpoint append {a} {n} {p} (v:vec a n) (w:vec a p) : vec a (n+p)
```

Proof example : associativity of concatenation

```
data List a = Nil | a ::: List a deriving (Eq)
-- Definition of a concatenation function '++' on lists
Nil      ++ ys = ys
(x ::: xs) ++ ys = x ::: (xs ++ ys)
```

```
assocThm xs ys zs = (xs ++ ys) ++ zs == xs ++ (ys ++ zs)
```

```
assocPf :: xs:_ → ys:_ → zs:_ → { assocThm xs ys zs }
```

```
assocPf Nil ys zs = (Nil ++ ys) ++ zs
                ==.   ys ++ zs
                ==.   Nil ++ (ys ++ zs)
```

```
assocPf (x ::: xs) ys zs = ((x ::: xs) ++ ys) ++ zs
                ==. (x ::: (xs ++ ys)) ++ zs
                ==. x ::: ((xs ++ ys) ++ zs)
                ==. x ::: (xs ++ (ys ++ zs)) ? assocPf xs ys zs
                ==. (x ::: xs) ++ (ys ++ zs)
```

Conclusion

- Type systems offer a general framework to verify the safety of the composition of programming expressions.
- The association between **types** and **logic properties** is natural in this framework and mechanisms exist to facilitate this association :



- These logic properties constitute another form of programming. Types / proofs become a natural component accompanying the code.
- The mechanisms for the **verification** of these properties **grow in complexity** accordingly with the expressivity of the properties : type annotations, SMT-solvers . . . up to proof assistants.
- **Undecidability** problems occur for the highest levels of complexity, hindering the verification capabilities for programmers.

The present and future of type systems

- The development of more recent type systems and even more recent programming languages displays a high level of activity.
- As an example, regions constitute a mechanism to describe zones of code and memory determined statically.
- Effects systems restrict the kind of operations allowed in certain of these regions, typically reading or writing to memory.

Many of these languages are experimentations derived from **Haskell**.

- The **Rust** programming language is an example of the last generation of general-purpose languages incorporating some of these advances.
- It claims solving the problems of dangling pointers, uses-after-free and even data races for some classes of concurrent programs.