Type Systems and Programming

D. Renault

ENSEIRB-Matmeca

Mar. 26th 2025, v.1.5.1

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

Sometimes, a polymorphic type may be **too generic**.
Take the example of an equality function, with the following type :

$$\forall T,\ T \to T \to \mathsf{Bool}$$

Yet **not all** values are comparable, for instance functional values.
It is natural to restrict the possible Ts to a family of types.

$$\forall T \in \textbf{Comparable},\ T \to T \to \mathsf{Bool}$$

This is a form of **constrained polymorphism**, appearing as :

- the `''a` **equality types** in SML, a subset of the types of the language,
- the `Eq a` **type class** in Haskell, defined by a form of overloading,
- the `Comparable<T>` **interface** in Java, with inclusion polymorphism.

Constrained polymorphism example : equality

In OCaml :

```
let rec belongs x l = match l with
| []    → false
| y::ys → (x=y) || (belongs x ys)
```

```
val belongs : 'a→'a list→bool
```

`belongs sin [cos]` compiles, but yields an **exception at runtime**.

```
(* Exception: Invalid_argument
   "equal: functional value". *)
```

In Haskell :

```
belongs x []     = False
belongs x (y:ys) = (x==y) ||
                   (belongs x ys)
```

```
belongs :: Eq t ⇒ t→[t]→Bool
```

The expression `belongs sin [cos]` simply **does not compile**.

```
No instance for (Eq (a0 → a0))
arising from a use of "=="
```

Constrained polymorphism : numeric classes

Definition (Type class)

In Haskell, a **type class** is a set of concrete types sharing a common generic interface. These concrete types are then **instances** of the type class.

Example :

```
class Eq a where
    (==) :: a → a → Bool
    (/=) :: a → a → Bool
```

```
instance Eq Int where
    (==) i j = -- specific code
    (/=) i j = not (i == j)
```

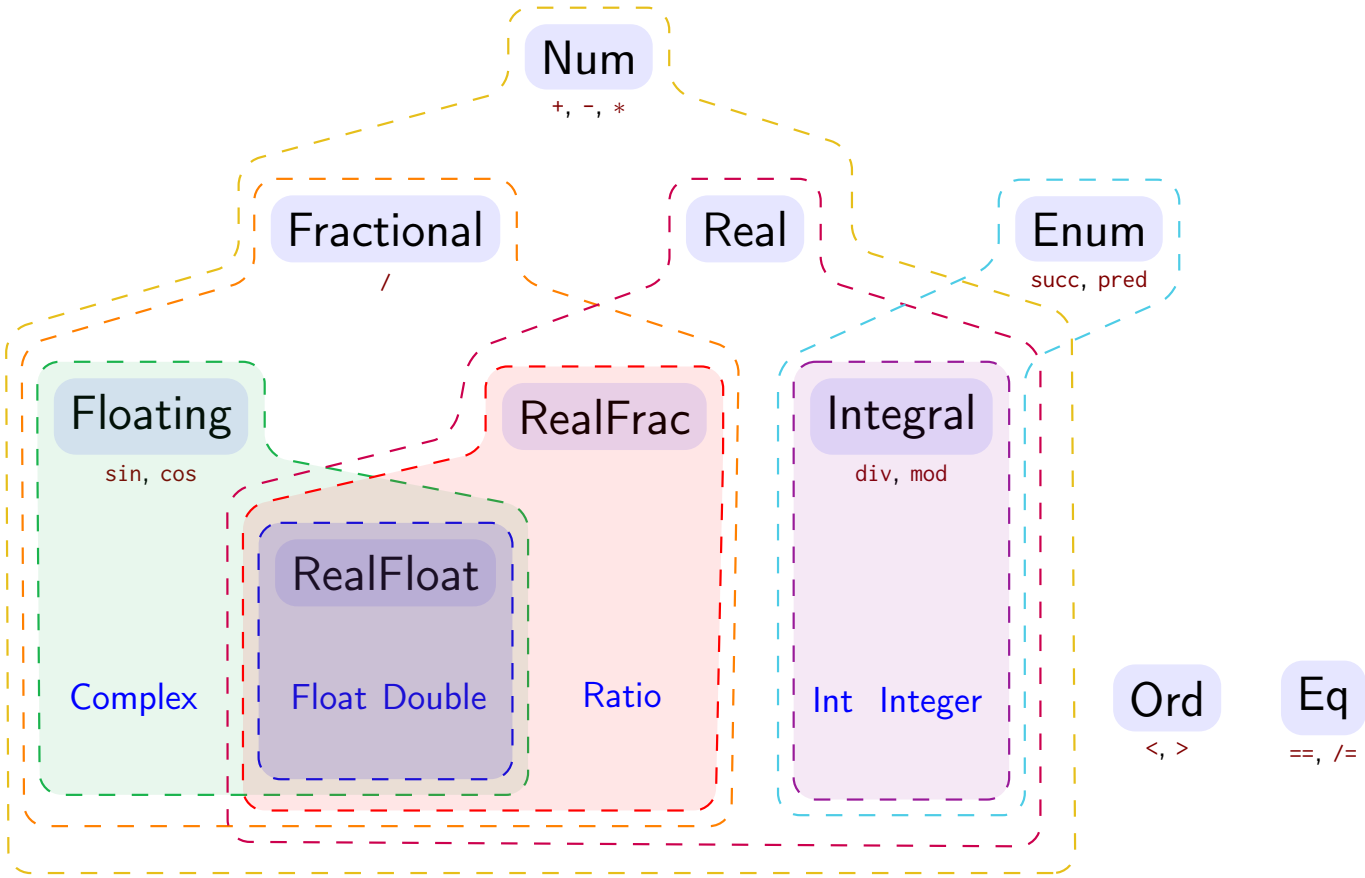
- A type class such as `Eq a` represents the following set of types :

$$\mathsf{Eq}[\mathsf{T}] ::= \{\mathsf{T} \in \mathbf{Typ}, \mathsf{T} \text{ “can be used with” } ==\}$$

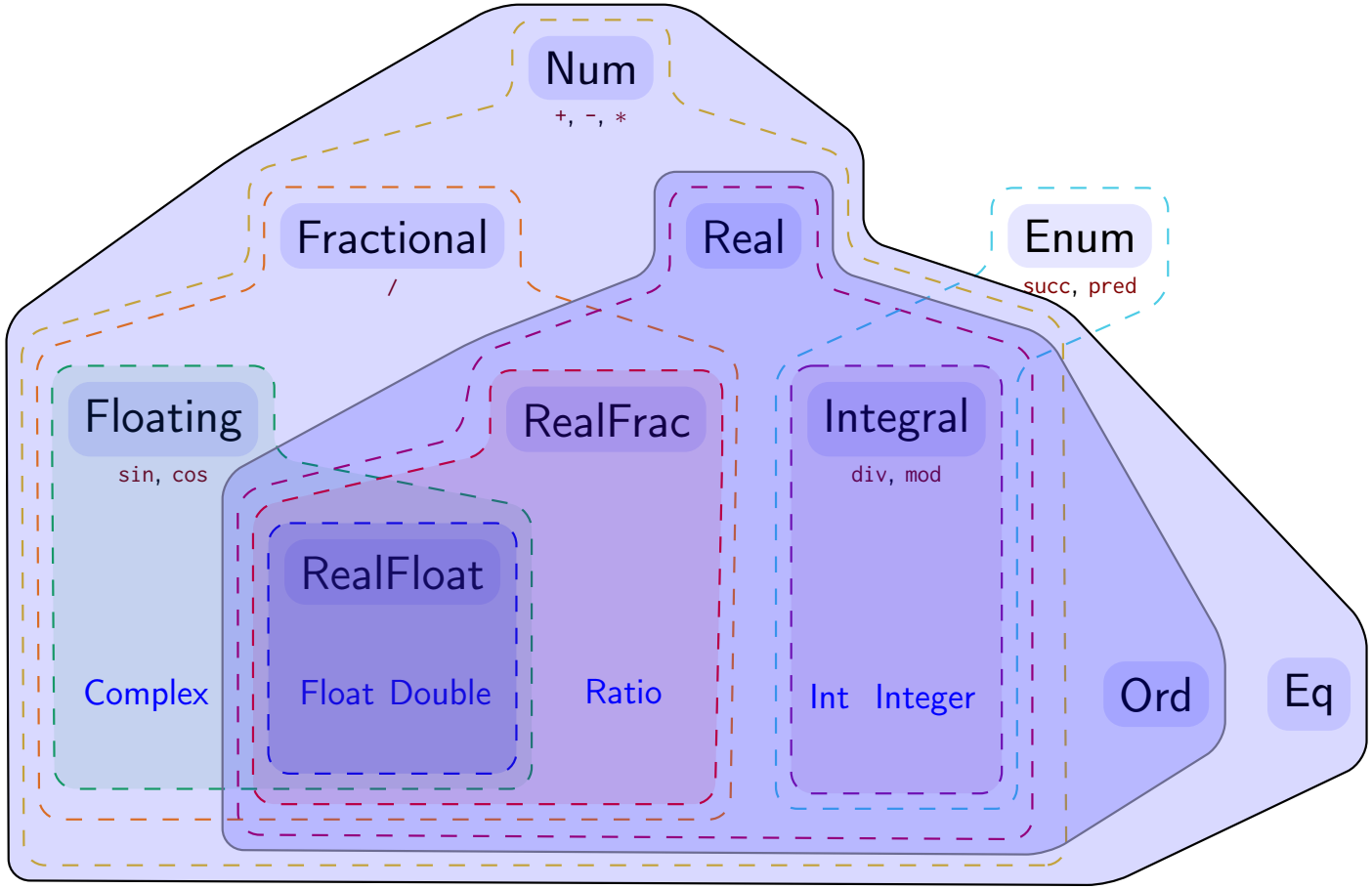
- It is used as a **universal type variable** in the type of `belongs` :

$$\texttt{belongs} : \forall \mathsf{T} \in \mathsf{Eq}[\mathsf{T}],\ \mathsf{T} \to \mathsf{List}[\mathsf{T}] \to \mathsf{Bool}$$

The numeric types in Haskell inherit a structure from the **type classes**.



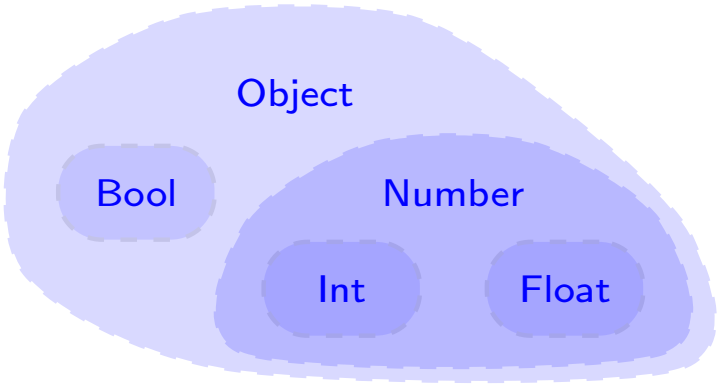
The numeric types in Haskell inherit a structure from the **type classes**.



In the following, we investigate the inclusion relations of sets of values.

Inclusion polymorphism

Inclusion polymorphism is based on the construction of sets of values sharing relations of inclusion.



Whereas parametric polymorphism defines inclusions **bottom-up**, inclusions in this polymorphism are defined **top-down**.

For a better understanding of these relations, we introduce a new type.

Record types

Definition (Record types)

Given a set $\{l_i\}$ of labels and a set $\{\mathtt{t}_i\}$ of expressions of the same size n, a **record value** is defined as the expression $\{l_1 = \mathtt{t_1}, \ldots, l_n = \mathtt{t}_n\}$.

The pairs $(l_i, \mathtt{t}_i)$ are called the **fields** of the record.
The **projection** of $\mathtt{r}$ onto one of its fields $(l_i, \mathtt{t}_i)$, noted $\mathtt{r}\triangleright l_i$, evaluates to $\mathtt{t}_i$.
The **type** of $\mathtt{r}$ is the set of the types of its fields, noted $\{l_1 : \mathsf{T_1}, \ldots, l_n : \mathsf{T}_n\}$.

Examples

- $\{first = \texttt{"Haskell"}, last = \texttt{"Curry"}\} : \{first : \mathsf{String}, last : \mathsf{String}\}$
- $\{hd = 1, tl = \{hd = 2, tl = \{\}\}\} : \{hd : \mathsf{Nat}, tl : \{hd : \mathsf{Nat}, tl : \{\}\}\}$

Syntax

$$
\begin{array}{lll}
t ::= & \ldots & \textit{expressions} \\
& \{l_i = t_i\}_{i \in [1;n]} & \textit{record} \\
& t \triangleright l & \textit{projection} \\
\\
v ::= & \ldots & \textit{values} \\
& \{l_i = v_i\}_{i \in [1;n]} & \textit{record value} \\
\\
T ::= & \ldots & \textit{types} \\
& \{l_i : T_i\}_{i \in [1;n]} & \textit{record type}
\end{array}
$$

Evaluation rules

$$\{l_i = v_i\}_{i \in [1;n]} \triangleright l_j \rightarrow_\beta v_j$$

$$\frac{t \rightarrow_\beta t'}{t \triangleright l \rightarrow_\beta t' \triangleright l}$$

$$\frac{t_j \rightarrow_\beta t'_j}{\{\ldots l_j = t_j \ldots\} \rightarrow_\beta \{\ldots l_j = t'_j \ldots\}}$$

Typing Rules

$$\frac{\text{for each } i,\ \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i\} : \{l_i : T_i\}}$$

$$\frac{\Gamma \vdash \{l_i = t_i\} : \{l_i : T_i\}}{\Gamma \vdash t \triangleright l_i : T_i}$$

All-purpose records

Records are good examples of the saying "he who can do more, can do less".
Consider the following function :

$$\texttt{half\_size} ::= \lambda \texttt{r}.(\texttt{r} \triangleright \textit{size}/2)$$

It can be happily applied to every record possessing a field *size*.

$$\begin{array}{ll}
\texttt{r}_1 ::= \{\textit{size} = 2\} & \texttt{half_size}(\texttt{r}_1) \rightarrow_\beta 1 \\
\texttt{r}_2 ::= \{\textit{size} = 6, \textit{name} = \texttt{"Alonzo"}\} & \texttt{half_size}(\texttt{r}_2) \rightarrow_\beta 3 \\
\texttt{r}_3 ::= \{\textit{size} = 2, \textit{contents} = \texttt{Cons(1,Cons(2,Nil))}\} & \texttt{half_size}(\texttt{r}_3) \rightarrow_\beta 1
\end{array}$$

A natural typing for this function is $\texttt{half\_size} : \{\textit{size} : \mathsf{Nat}\} \rightarrow \mathsf{Nat}$.
Yet it is too restrictive : it only authorizes the first application. So what ?

Inclusion subtyping

Lemma (Record subtyping) :

Let $\mathsf{T} = \{l_i : \mathsf{T}_i\}$ and $\mathsf{T}' = \{m_i : \mathsf{T}'_i\}$ be two record types.

- **Width subtyping** : if $\mathsf{T} \supset \mathsf{T}'$ as sets of pairs labels/types, then $\mathsf{T} <: \mathsf{T}'$.
- **Depth subtyping** : if T and T' share exactly the same labels and $\forall i, \mathsf{T}_i <: \mathsf{T}'_i$, then $\mathsf{T} <: \mathsf{T}'$.

Examples

- **Width** : $\{size : \mathsf{Nat}, name : \mathsf{String}\} <: \{size : \mathsf{Nat}\}$
- **Depth** : if $\mathsf{Int} <: \mathsf{Number}$, then $\{size : \mathsf{Int}\} <: \{size : \mathsf{Number}\}$

With the following definitions :

- $\texttt{half\_size} ::= \lambda \texttt{r}.(\texttt{r} \triangleright \textit{size}/2)$
- $\texttt{person} ::= \{\textit{size} = 7, \textit{name} = \texttt{"Haskell"}\}$

Let $\Gamma ::= \{\texttt{half\_size} : \{\textit{size} : \text{Nat}\} \rightarrow \text{Nat}, \texttt{person} : \{\textit{size} : \text{Nat}, \textit{name} : \text{String}\}\}$

Then the expression $(\texttt{half\_size}\ \texttt{person})$ is well-typed :

$$\dfrac{\dfrac{\texttt{half\_size} \in \Gamma}{\Gamma \vdash \texttt{half\_size} : \{\textit{size} : \text{Nat}\} \rightarrow \text{Nat}} \qquad \dfrac{\dfrac{\texttt{person} \in \Gamma}{\Gamma \vdash \texttt{person} : \{\textit{size} : \text{Nat}, \textit{name} : \text{String}\}}}{\Gamma \vdash \texttt{person} : \{\textit{size} : \text{Nat}\}}}{\Gamma \vdash (\texttt{half\_size}\ \texttt{person}) : \text{Nat}}$$

In this case, the subtyping rules solve the "do-more, do-less" problem.
But what implications does this have on our sets of values ?

Existential types

Proposition

A record type is by nature an existential type.

All records having a field *size* of type Nat can be typed with $\{size : \mathsf{Nat}\}$.

$$\begin{aligned} \{size : \mathsf{Nat}\} &\equiv \bigcup_{\mathsf{T}} \{size : \mathsf{Nat}\} \cup \mathsf{T} \\ &\equiv \exists \mathsf{T}.\{size : \mathsf{Nat}\} \cup \mathsf{T} \end{aligned}$$

Example

$$\texttt{half\_size} : \left(\exists \mathsf{T}.\{size : \mathsf{Nat}\} \cup \mathsf{T}\right) \rightarrow \mathsf{Nat}$$

Aside : upcast, downcast

Definition (Casting)

Casting (or ascription) consists in ascribing a particular type to an expression in an explicit manner. It has no effect on the value.
The expression v as T is a called a cast from v into the type T.

$$v \text{ as } T \to_\beta v$$

$$\frac{\Gamma \vdash t : ?}{\Gamma \vdash t \text{ as } T : T}$$

- A cast can be seen as an operation redefining the type of an expression.
- Casting into a supertype is also called an **upcast**.
  Upcasts are implicit with the ▶ subsumption rule.
- Casting into a subtype is also called an **downcast**.
  Downcasts are usually checked dynamically.

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t \text{ as } T : T}$$

$$\frac{\mathcal{E} \vdash v : T}{v \text{ as } T \to_\beta v}$$

Consider the following function :

$$\texttt{cut\_in\_half} ::= \lambda \texttt{r}.\left\{ \texttt{r}\triangleright\textit{size} \texttt{ := } \left(\texttt{half\_size}\ \left(\texttt{r}\triangleright\textit{size}\right)\right); \texttt{r} \right\}$$

It basically takes a record with a *size* field and returns a record where this field has been modified and the others left untouched.

What is a good type for `cut_in_half` ?

Consider the following function :

$$\texttt{cut\_in\_half} ::= \lambda \texttt{r}.\left\{ \texttt{r}\triangleright\textit{size} \texttt{ := } (\texttt{half\_size } (\texttt{r}\triangleright\textit{size})); \texttt{r} \right\}$$

It basically takes a record with a *size* field and returns a record where this field has been modified and the others left untouched.

What is a good type for `cut_in_half` ?

- `cut_in_half` : $\forall \mathsf{T},\ \mathsf{T} \to \mathsf{T}$?
  Too generic, no way of ensuring the existence of the *size* field.
- `cut_in_half` : $\{\textit{size} : \mathsf{Ref}[\mathsf{Nat}]\} \to \{\textit{size} : \mathsf{Ref}[\mathsf{Nat}]\}$?
  Too restrictive, the return type constrains the result.
  Same behavior as the `clone` method in Java, requiring downcasts.

Consider the following function :

$$\texttt{cut\_in\_half} ::= \lambda \texttt{r}.\left\{ \texttt{r}\triangleright \textit{size} \texttt{ := } \left(\texttt{half\_size } \texttt{r}\triangleright \textit{size}\right); \texttt{r} \right\}$$

It basically takes a record with a *size* field and returns a record where this field has been modified and the others left untouched.

What is a good type for `cut_in_half` ?

What about : $\forall \mathsf{T}, \left(\{\textit{size} : \textit{Nat}\} \cup \mathsf{T}\right) \to \left(\{\textit{size} : \textit{Nat}\} \cup \mathsf{T}\right)$ ✓

Or is it an existential type ?

$\left(\exists \mathsf{T}.\{\textit{size} : \textit{Nat}\} \cup \mathsf{T}\right) \to \left(\exists \mathsf{U}.\{\textit{size} : \textit{Nat}\} \cup \mathsf{U}\right)$ ✗

$\exists \mathsf{T}.\left(\{\textit{size} : \textit{Nat}\} \cup \mathsf{T}\right) \to \left(\{\textit{size} : \textit{Nat}\} \cup \mathsf{T}\right)$ ✗

Aside on existentials and universals

Beware : $\forall \mathsf{T},\ \{size : \mathsf{Nat}\} \cup \mathsf{T} \neq \exists \mathsf{T}.\{size : \mathsf{Nat}\} \cup \mathsf{T}$

A universal type T can substitute for **all** possible types.
An existential type T can substitute for only **one**.

Nevertheless, there are equivalences :

Equivalence theorem

$$\Big(\exists x.P(x)\Big) \Rightarrow Q \qquad \Leftrightarrow \qquad \forall x.\Big(P(x) \Rightarrow Q\Big)$$

Example

$$\Big(\exists \mathsf{T}.\{size : \mathsf{Nat}\} \cup \mathsf{T}\Big) \rightarrow \mathsf{Nat} \equiv \forall \mathsf{T},\ \Big(\{size : \mathsf{Nat}\} \cup \mathsf{T} \rightarrow \mathsf{Nat}\Big)$$

Naming the existential variables prompts for **more precise types** :

- Existential types in PureScript :

```
cut_in_half :: forall b. { size :: Int | b } → { size :: Int | b }
```

- Open types in OCaml :

```
cut_in_half : (<get_size : int; set_size : int → unit; ..> as 'a) → 'a
```

But in general, libraries contain few functions requiring such types.

Existentials as an abstraction means

Definition (Abstract data type)

An **abstract data type** or ADT consists of :

- a type variable T and a set of operation types acting on values of type T.
- a concrete type S and an implementation of these operation types where the variable T is substituted by S.

Akin to interfaces in Java or module signatures in ML.

```
interface Counter {


  int get();
  Counter incr();
}
```

```
class CImpl implements Counter {
  private int val = 0;
  CImpl(int in)  { val = in; }
  int get()      { return val; }
  Counter incr() { return new CImpl(val+1); }
}
```

The existential type corresponds here to the "abstract" part of the ADT.

Existentials as an abstraction means

Definition (Abstract data type)

An abstract data type or ADT consists of :

- a type variable T and a set of operation types acting on values of type T.
- a concrete type S and an implementation of these operation types where the variable T is substituted by S.

Akin to interfaces in Java or module signatures in ML.

```
module type COUNTER = sig
  type counter
  val new_c : unit → counter
  val get   : counter → int
  val inc   : counter → counter
end
```

```
module C : COUNTER = struct
  type counter = int
  let new_c () = 0
  let get c  = c
  let inc c  = c + 1
end
```

The existential type corresponds here to the "abstract" part of the ADT.

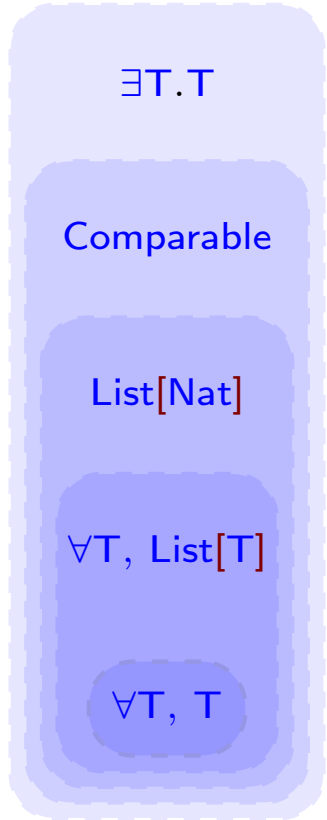
Existentials vs Universals

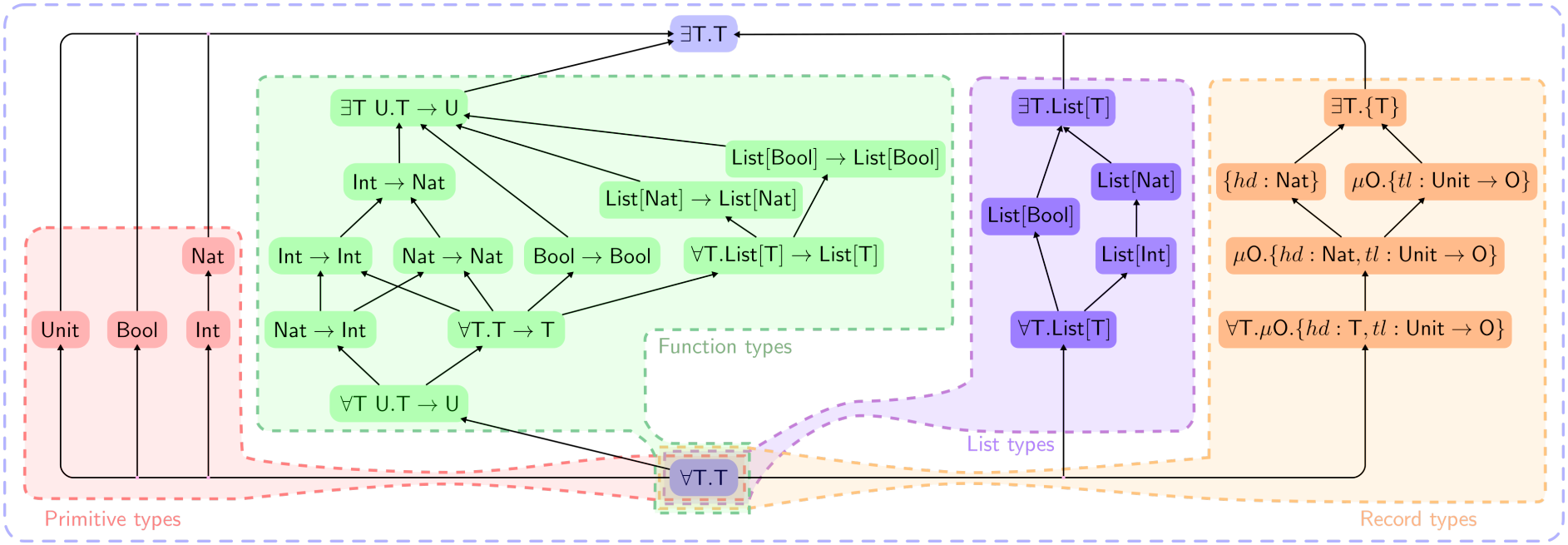
- Existentials are items of **abstraction**.
  Allowing multiple implementations,
  Offering a precise protocol of exchange.
- Universals are items of **genericity**.
  Maximising code reuse,
  With few knowledge on the values they manipulate.

In that, the combination of both polymorphisms is natural.

```
<T extends Comparable<T>> void sort(List<T> list)
```

$$\forall T, \exists U.\mathsf{List}[\mathbf{Comparable}[T] \cup \mathbf{U}] \rightarrow \mathsf{Unit}$$




∃T.T
∃T U.T → U
List[Bool] → List[Bool]
Int → Nat
List[Nat] → List[Nat]
Int → Int
Nat → Nat
Bool → Bool
∀T.List[T] → List[T]
Nat → Int
∀T.T → T
∀T U.T → U
∀T.T
Nat
Unit
Bool
Int
Function types
∃T.List[T]
List[Nat]
List[Bool]
List[Int]
∀T.List[T]
List types
∃T.{T}
{hd : Nat}
μO.{tl : Unit → O}
μO.{hd : Nat, tl : Unit → O}
∀T.μO.{hd : T, tl : Unit → O}
Primitive types
Record types

Object types (1)

Objects can be modelled as records with access to `self` : they are **recursive**.

```
let (c:cpt) = let rec self = {    (* Recursive definition *)
                    v   = 0;
                    get = (fun () → self.v);
                    set = (fun y  → self.v ← y);
                    inc = (fun () → self.set (self.get() + 1)); }
              in self;;
```

In OCaml, classes are indeed identified to their **recursive** constructors :

```
class cpt = fun init → object (self)
                    val mutable v:int = init
                    method get    = v
                    method set d  = v ← d
                    method inc () = self#set (self#get + 1)
            end;;
```

Object types (2)

- A class, representing a set of values, can be identified to a type.
- Accordingly, these types are also recursive :

$$\begin{array}{rcl} \text{Object} & ::= & \textbf{fix}(\lambda \text{T}.\{ \\ & & \quad \textit{equals} : \text{T} \rightarrow \text{Bool}, \\ & & \quad \textit{clone} : \text{Unit} \rightarrow \text{T} \\ & & \}) \end{array}$$

  . . . where **fix** is a fixed-point operator.
- This definition yields an **infinite** type represented by a rewriting rule.
- Usually, this fixed-point is made invisible by nominal types :

$$\begin{array}{rcl} \text{Object} & ::= & \{ \\ & & \quad \textit{equals} : \text{Object} \rightarrow \text{Bool}, \\ & & \quad \textit{clone} : \text{Unit} \rightarrow \text{Object} \\ & & \} \end{array}$$

Summary on the inclusion polymorphism

- The inclusion polymorphism allows the definition of types by refining sets of values into more specific subsets.
- These sets of values can be identified as logic formulas containing **existentially** quantified variables.

$$\exists T.F[T] \equiv \bigcup_{U \in \mathbf{Typ}} F[U]$$

- Subtyping relations with existential types promote **abstraction** by masking concrete types :

$$F[U] <: \exists T.F[T] \text{ where } U \text{ concrete}$$

- Object types are at the same time existential and recursive types.

Aside on Inheritance

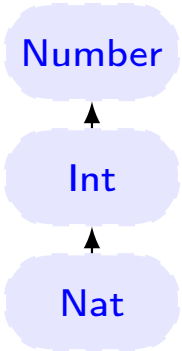
Inheritance

Inheritance is a mechanism to derive new classes from old ones by :

(i) adding implementation for new methods

(ii) and / or **overriding** implementations of old methods.

Consider a method `m` defined in Number and overriden in its subclasses.




```
class Number           { m(..); }
class Int extends Number { m(..); }
class Nat extends Int    { m(..); }
```

$m(x : \text{Number}, \ldots)$

$m(x : \text{Int}, \ldots)$

$m(x : \text{Nat}, \ldots)$

The method `m` can be considered as a function whose definition is **selected** depending on the value of its first parameter.

Overloading

Overloading

Overloading is a mechanism allowing the use of a single identifier for the representation of multiple values, distinguished according to their type.

Example : string concatenation in C++

```
string operator+ (const string& lhs, const string& rhs);
string operator+ (const string& lhs, const char*   rhs);
string operator+ (const char*   lhs, const string& rhs);
string operator+ (const string& lhs, char          rhs);
string operator+ (char          lhs, const string& rhs);
```

Example : default values in Haskell

```
class Default a where def :: a      -- | The default value for this type
instance Default Int where def = 0
instance Default [a] where def = []
instance (Default a, Default b) ⇒ Default (a, b) where def = (def, def)
```

- A manner to represent overloaded values consists in **packing** all the implementations together in a single object.

$\texttt{def} ::= \texttt{0} \oplus \texttt{[]}$	packing `0` and `[]` together
$\texttt{plus} ::= \texttt{plus}_{\mathsf{Int}} \oplus \texttt{plus}_{\mathsf{List}}$	packing addition and concatenation

- For typing purposes, an overloaded value possesses the types of **all** the values it merges :

- A manner to represent overloaded values consists in **packing** all the implementations together in a single object.

$\texttt{def} ::= \texttt{0} \oplus \texttt{[]}$	packing `0` and `[]` together
$\texttt{plus} ::= \texttt{plus}_{\mathsf{Int}} \oplus \texttt{plus}_{\mathsf{List}}$	packing addition and concatenation

- For typing purposes, an overloaded value possesses the types of **all** the values it merges : an **intersection type**.

$$\texttt{def} : \mathsf{Nat} \ \& \ \forall A,\ \mathsf{List}[A]$$

- It is **not** a union type : as a value, `def` has the possibility to be used indifferently as a number **and** as a list (but only one at a time).

Consequence

If an identifier is overloaded, a correct implementation must be **selected** every time the identifier is used (at compile-time or at runtime)

Syntax	Evaluation rules	Typing rules
$t ::= \ldots$ *expressions* $t \oplus t$ *merge* $T ::= \ldots$ *types* $T \,\&\, T$ *intersec. type*	$\dfrac{t_1 \to_\beta t'_1}{t_1 \oplus t_2 \to_\beta t'_1 \oplus t_2}$ $\dfrac{t_2 \to_\beta t'_2}{t_1 \oplus t_2 \to_\beta t_1 \oplus t'_2}$ $v_1 \oplus v_2 \to_\beta v_1$ $v_1 \oplus v_2 \to_\beta v_2$	$\dfrac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 \oplus t_2 : T_1 \,\&\, T_2}$ $\dfrac{\Gamma \vdash t : T_1 \,\&\, T_2}{\Gamma \vdash t : T_1}$ $\dfrac{\Gamma \vdash t : T_1 \,\&\, T_2}{\Gamma \vdash t : T_2}$

- Caution : these rules **break** the type safety of the system.
- At least, a mechanism must be introduced to ensure that the value used at runtime is compatible with the type checked at compile-time.

Lemma (Intersection subtyping) :

If T_1 and T_2 are two different types, $T_1 \,\&\, T_2 <: T_1$ and $T_1 \,\&\, T_2 <: T_2$

Some tactics to select an overloaded method (1)

Let `def` ::= `1` $\oplus$ `true` : Bool `&` Nat be an overloaded value.
How can we evaluate the expression : `if def then 0 else def` ?

Some tactics to select an overloaded method (1)

Let $\texttt{def} ::= 1 \oplus \texttt{true} : \texttt{Bool \& Nat}$ be an overloaded value.
How can we evaluate the expression : $\texttt{if def then 0 else def}$?

- Decide the implementation used at **compile-time** (cf. Haskell),

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{def} : \texttt{Bool \& Nat}}{\Gamma \vdash \texttt{def} : \texttt{Bool}}}{\Gamma \vdash \texttt{def as Bool} : \texttt{Bool}} \qquad \dfrac{}{\Gamma \vdash \texttt{0} : \texttt{Nat}} \qquad \dfrac{\dfrac{\Gamma \vdash \texttt{def} : \texttt{Bool \& Nat}}{\Gamma \vdash \texttt{def} : \texttt{Nat}}}{\Gamma \vdash \texttt{def as Nat} : \texttt{Nat}}}{\Gamma \vdash \texttt{if def as Bool then 0 else def as Nat} : \texttt{Nat}}$$

Some tactics to select an overloaded method (1)

Let $\texttt{def} ::= \texttt{1} \oplus \texttt{true} : \texttt{Bool \& Nat}$ be an overloaded value.
How can we evaluate the expression : $\texttt{if def then 0 else def}$?

- Decide the implementation used at **runtime** (cf. Python),

$$\frac{\dfrac{\mathcal{E} \vdash \texttt{def} : \texttt{Bool \& Nat}}{\texttt{def} \rightarrow_\beta \texttt{def as Bool}}}{\texttt{if def then 0 else def} \rightarrow_\beta \texttt{if def as Bool then 0 else def} \rightarrow_\beta \cdots}$$

Some tactics to select an overloaded method (1)

Let $\texttt{def} ::= \texttt{1} \oplus \texttt{true} : \texttt{Bool \& Nat}$ be an overloaded value.
How can we evaluate the expression : $\texttt{if def then 0 else def}$?

- Decide the implementation used at **compile-time** (cf. Haskell),

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{def} : \texttt{Bool \& Nat}}{\Gamma \vdash \texttt{def} : \texttt{Bool}}}{\Gamma \vdash \texttt{def as Bool} : \texttt{Bool}} \qquad \dfrac{}{\Gamma \vdash \texttt{0} : \texttt{Nat}} \qquad \dfrac{\dfrac{\Gamma \vdash \texttt{def} : \texttt{Bool \& Nat}}{\Gamma \vdash \texttt{def} : \texttt{Nat}}}{\Gamma \vdash \texttt{def as Nat} : \texttt{Nat}}}{\Gamma \vdash \texttt{if def as Bool then 0 else def as Nat} : \texttt{Nat}}$$

- Decide the implementation used at **runtime** (cf. Python),

$$\dfrac{\dfrac{\mathcal{E} \vdash \texttt{def} : \texttt{Bool \& Nat}}{\texttt{def} \rightarrow_\beta \texttt{def as Bool}}}{\texttt{if def then 0 else def} \rightarrow_\beta \texttt{if def as Bool then 0 else def} \rightarrow_\beta \cdots}$$

- . . . or use a combination of both (cf. Java, C++).

Static/ Late binding

The presence of subtyping allows the following technique :

Static / Late binding

To every object value is attached a type called its **concrete type**.
It may differ from the **apparent type** of this same value in an expression.

At the callpoint of an overloaded method, the appropriate code is selected.
In **static binding**, the selection depends on the **apparent type**.
In **late binding**, the selection depends on the **concrete type**.

Example

```
String s = new String("Concrete") // Apparent : String / Concrete : String
Object o = (Object) s;            // Apparent : Object / Concrete : String
o.equals(s);                      // Which equals method is called ?
```

Consider the following examples based on a Counter class :

```
class Counter {
   int v; // count calls to inc


   public Counter(int v) { this.v = v; };
   int get() { return v; }
   void set(int v) { this.v = v; }
   void inc() { this.set(this.get() + 1); }

}
```

```
1  class CounterExt extends Counter {
2
3     int a; // count calls to set
4
5     public CounterExt(int v) { super(v); a = 0;};
6     // inherit get
7     void set(int v) { this.a++; super.set(v);  }
8     // inherit inc
9     int get_a() { return a; }
10 }
```

Consider the following examples based on a Counter class :

```
class Counter {
   int v; // count calls to inc


   public Counter(int v) { this.v = v; };
   int get() { return v; }
   void set(int v) { this.v = v; }
   void inc() { this.set(this.get() + 1); }

}
```

```
1  class CounterExt extends Counter {
2
3     int a; // count calls to set
4
5     public CounterExt(int v) { super(v); a = 0;};
6     // inherit get
7     void set(int v) { this.a++; super.set(v);  }
8     // inherit inc
9     int get_a() { return a; }
10 }
```

Consider the call to `set` inside the `inc` method in Counter :

- In **early binding**, this call is attached to the apparent type.
  For an object of type CounterExt, the `set` method of Counter is used.

Consider the following examples based on a Counter class :

```
class Counter {
   int v; // count calls to inc


   public Counter(int v) { this.v = v; };
   int get() { return v; }
   void set(int v) { this.v = v; }
   void inc() { this.set(this.get() + 1); }

}
```

```
1  class CounterExt extends Counter {
2
3     int a; // count calls to set
4
5     public CounterExt(int v) { super(v); a = 0;};
6     // inherit get
7     void set(int v) { this.a++; super.set(v);  }
8     // inherit inc
9     int get_a() { return a; }
10 }
```

Consider the call to `set` inside the `inc` method in Counter :

- In late binding, this call is attached to the concrete type.
  For an object of type CounterExt, the `set` method of CounterExt is used.

Summary on the overloading polymorphism

- Overloading polymorphism allows the definition of values sharing multiply different implementations :

$$\mathtt{m} ::= \{\mathtt{m}_1 : \mathtt{T}_1, \mathtt{m}_2 : \mathtt{T}_2, \ldots, \mathtt{m}_n : \mathtt{T}_n\}$$

  The types $\mathtt{T}_i$ may not share a common structure.
- Overloaded types may be modelized as finite **intersection types**.

$$\mathtt{T}_1 \ \& \ \mathtt{T}_2 \ \& \ \ldots \ \& \ \mathtt{T}_n = \bigcap_{i=1}^{n} \mathtt{T}_i$$

- In order to use this polymorphism, a **selection** of the correct implementation for $\mathtt{m}$ is necessary, be it static or dynamic.
- The **late binding** is an example of dynamic selection in the case of overloaded types. It appears naturally in object-oriented programming.

Bibliography

- Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.
- Bruce, K. B. *Foundations of Object-oriented Languages : Types and Semantics*. MIT Press, 2002.
- Hindley, J. R. *Basic simple type theory*. Cambridge University Press, 1997.
- Wadler, P. *Propositions as types*. Communications ACM, 2015.