Type Systems and Programming

D. Renault
ENSEIRB-Matmeca
Feb. 19th 2025, v.1.5.1

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

Polymorphism

Polymorphism

An expression in a programming language is said to be **polymorphic** whenever it may be typed with multiply different types.

Examples

$$\texttt{fst} : \text{Nat} \rightarrow \text{Bool} \rightarrow \text{Nat} \;\textit{or}\; \text{Bool} \rightarrow \text{Nat} \rightarrow \text{Bool} \;\textit{or}\; \ldots$$

$$\texttt{plus} : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \;\textit{or}\; \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \;\textit{or}\; \ldots$$

- Applies to functions, but also to non-functional values.
- Polymorphism is a natural property aimed at genericity / code reuse

"Write code once, use it anywhere[1]."

1. Type conditions may apply.

Considerations on types

General idea (Types as sets)

A type represents a set of values.

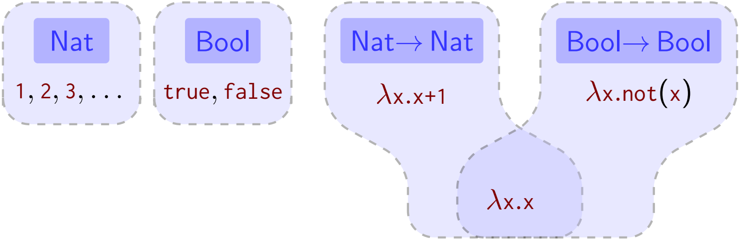
Definition (Set of values)

The **set of values** associated to a type T, noted $\mathrm{vals}(T)$, is the set of values of the language that can be typed by T.
Equivalently, $e \in \mathrm{vals}(T) \Leftrightarrow e : T$.

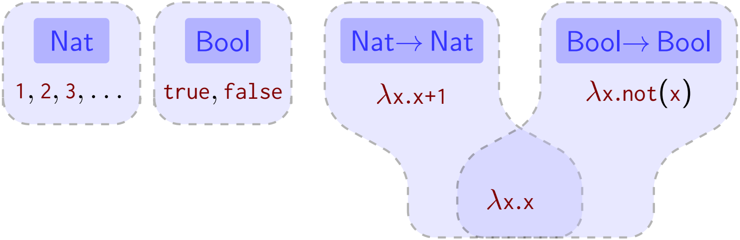
Definition (Subtype)

The type T is said to a **subtype** of the type U, noted $T <: U$, if and only if $\mathrm{vals}(T) \subset \mathrm{vals}(U)$.

- The "types as sets" proposition leads to the following kind of picture :



- The "types as sets" proposition leads to the following kind of picture :



- The identity function `id` $::=$ λ`x.x` does not have a satisfactory type in $\lambda_\rightarrow$.
- An extension of $\lambda_\rightarrow$ is the addition of a new type T_{id} for `id` such that :

$$\mathsf{vals}\,(\mathsf{T}_{\mathsf{id}}) = \mathsf{vals}\,(\mathsf{Nat} \rightarrow \mathsf{Nat}) \bigcap \mathsf{vals}\,(\mathsf{Bool} \rightarrow \mathsf{Bool})$$

- In this type system, `id` : T_{id} may be applied either to Nat or Bool values.

Properties of subtyping

Definition (Subsumption rule)

Whenever $S <: T$, every expression typable by S is also typable by T.

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}\ [\textsc{sub}]$$

▸ Extended Substitution Lemma

Consider an expression $t : T$ containing a free variable $x : S$.
Then x can be substituted to any expression s of type $S' <: S$ without affecting the type of t.

$$\frac{\Gamma, x : S \vdash t : T \qquad \Gamma \vdash s : S' \qquad S' <: S}{\Gamma \vdash [x \mapsto s]t : T}$$

General idea

Create type systems expressing more sophisticated families of types.

Some of these families are well-identified [CW85] :

- **Parametric** : define sets of values with the help of universally quantified type parameters (ex : $\forall X, X \to X \to X$)
- **Inclusion** : define sets of values that are related by inclusion or refinement (ex : Object $\rightsquigarrow$ Number $\rightsquigarrow$ Integer)
- **Overloading** : combine sets of values in an adhoc manner, without a particular structure (ex : Nat $\oplus$ Bool)

... while other families do not fit well in that classification
(cf. for example the Haskell type classes or OCaml open object types)

Parametric polymorphism (1)

Consider the following extension to our **Typ** family :

Definition (Parametric types)

Given a type T containing the variable X, then $\forall X, T$ is also a type, called a **parametric** or **universal** type.
The variable X in $\forall X, T$ is said to be **bound**. Unbound variables are **free**.
A **type scheme** is a parametric type without free variables.

Example

A type for the first projection `fst` $::= \lambda x.\lambda y.x$ is $\forall X, \forall Y, X \rightarrow Y \rightarrow X$

Parametric polymorphism (2)

Parametric types may be considered as **functions** and be applied :

Definition (Parametric expression)

Given an expression $\mathtt{t} : T$, then $\lambda X.\mathtt{t}$ is also an expression, called a **parametric expression** with type $\forall X,\ T$.
A parametric expression $\mathtt{e} : \forall X,\ T$ can be applied to a type U, noted $\mathtt{e}[U]$, which consists in substituting X by U inside T.

This extension introduces a form of computation at the type level :

$$\left(\left[\lambda X.\lambda Y.\left(\lambda x : X.(\lambda y : Y.x)\right)\right][\mathrm{Int}][\mathrm{Bool}]\right)\ \_\ 1\ \_\ \mathtt{true}$$

Syntax and types			Typing rules
$t ::=$	$\ldots$	*expressions*	$\dfrac{t \rightarrow_\beta t'}{t[T] \rightarrow_\beta t'[T]}$
	$\lambda X.t$	*type abstraction*	
	$t[T]$	*type application*	$(\lambda X.t)[U] \rightarrow_\beta [X \mapsto U]t$
$v ::=$	$\ldots$	*values*	$\dfrac{\Gamma \vdash t : T}{\Gamma \vdash \lambda X.t : \forall X,\ T}$
	$\lambda X.t$	*type abstr. value*	
$T ::=$	$\ldots$	*types*	$\dfrac{\Gamma \vdash t : \forall X,\ T}{\Gamma \vdash t[U] : [X \mapsto U]T}$
	$\forall X,\ T$	*universal type*	

Notice the ressemblance with the untyped lambda-calculus ▶.

This defines λ_2 the **polymorphic** or **2^{nd}-order calculus**, or also **System F**.

- It was introduced independently by Girard (1972) and Reynolds (1974).
- It possesses the following properties :

Theorem (Strong normalization) :

In λ_2, every expression reduces to a value in a finite number of steps.

- It is also **incomplete**, and cannot express all computable functions.
  (restricted to the Church naturals, it can only compute the functions definable in 2^{nd}-order Peano arithmetic, among which the Ackermann function)

Theorem (Impossibility of type inference) :

Type inference in λ_2 (without annotations) is undecidable.

Example : polymorphic lists

Syntax	Evaluation rules	Typing rules
`t ::= ...` *exprs* `nil` *empty list* `cons t t` *cons list* `head t` *list head* `tail t` *list tail* `v ::= ...` *values* `nil` *empty list* `cons v v` *cons list* T ::= ... *types* $\mathsf{List}[T]$ *list type*	$\dfrac{\mathtt{t} \rightarrow_\beta \mathtt{t'}}{\mathtt{cons\ t\ u} \rightarrow_\beta \mathtt{cons\ t'\ u}}$ $\dfrac{\mathtt{t} \rightarrow_\beta \mathtt{t'}}{\mathtt{cons\ v\ t} \rightarrow_\beta \mathtt{cons\ v\ t'}}$ $\dfrac{\mathtt{t} \rightarrow_\beta \mathtt{t'}}{\mathtt{head\ t} \rightarrow_\beta \mathtt{head\ t'}}$ $\dfrac{\mathtt{t} \rightarrow_\beta \mathtt{t'}}{\mathtt{tail\ t} \rightarrow_\beta \mathtt{tail\ t'}}$ $\mathtt{head}(\mathtt{cons}\ v_1\ v_2) \rightarrow_\beta v_1$ $\mathtt{tail}(\mathtt{cons}\ v_1\ v_2) \rightarrow_\beta v_2$	$\dfrac{}{\Gamma \vdash \mathtt{nil} : \mathsf{List}[T]}$ $\dfrac{\Gamma \vdash \mathtt{x} : T \qquad \Gamma \vdash \mathtt{l} : \mathsf{List}[T]}{\Gamma \vdash \mathtt{cons\ x\ l} : \mathsf{List}[T]}$ $\dfrac{\Gamma \vdash \mathtt{l} : \mathsf{List}[T]}{\Gamma \vdash \mathtt{head\ l} : T}$ $\dfrac{\Gamma \vdash \mathtt{l} : \mathsf{List}[T]}{\Gamma \vdash \mathtt{tail\ l} : \mathsf{List}[T]}$

Compare the syntax of polymorphic lists in different languages :

- in Scala :

```scala
abstract class List[+A] {
  def map[B](f: (A) ⇒ B): List[B] }
```

- in Java :

```java
interface List<E> { E set(int index, E element); }
```

- in C# :

```csharp
public interface IEnumerable<out T> {
  IEnumerable<R> Select<S, R>(this IEnumerable<S> src, Func<S, R> f)
```

- in C++ via iterators :

```cpp
template<class InputIt, class Function>
  Function for_each(InputIt first, InputIt last, Function fn);
```

- in OCaml (`'a list`) and in Haskell (`[a]`)

```ocaml
val map : ('a → 'b) → 'a list → 'b list
```

Boehm-Berarducci encoding of lists

As a matter of fact, there exists a general technique to encode algebraic types such as the lists in λ_2, called the Boehm-Berarducci encoding :

$$\mathsf{List}[\mathsf{T}] ::= \forall \mathsf{X}, (\underbrace{\mathsf{T} \to \mathsf{X} \to \mathsf{X}}_{\mathtt{cons}}) \to \underbrace{\mathsf{X}}_{\mathtt{nil}} \to \mathsf{X}$$

- The empty list is the following value :

$$\mathtt{nil} ::= \left| \lambda \mathsf{X}. \lambda \mathtt{c} : (\mathsf{T} \to \mathsf{X} \to \mathsf{X}). \lambda \mathtt{n} : \mathsf{X} \right| . \mathtt{n}$$

- Consider $\mathtt{x} : \mathsf{T}$ and $\mathtt{xs} : \mathsf{List}[\mathsf{T}]$. The list beginning with $\mathtt{x}$ and ending with $\mathtt{xs}$ is the following value :

$$\mathtt{cons\ x\ xs} ::= \left| \lambda \mathsf{X}. \lambda \mathtt{c} : (\mathsf{T} \to \mathsf{X} \to \mathsf{X}). \lambda \mathtt{n} : \mathsf{X} \right| . \mathtt{c\ x\ (xs}[\mathsf{X}]\ \mathtt{c\ n)}$$

Type substitution

Definition (Type substitution)

A **type substitution** σ is a finite mapping from type variables to types. We write $[X_i \mapsto T_i]$ for the substitution mapping X_i to T_i.
The **application** of σ to a type $U \equiv \forall X_1, \ldots \forall X_n, T$, noted σU consists in substituting the free occurrences of X_i inside T by T_i simultaneously, and then generalizing the type. Variables bound inside T are left invariant.

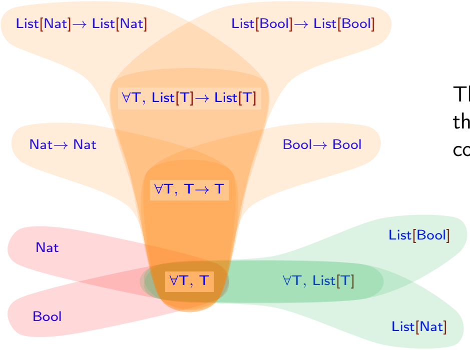
Examples

- $[X \mapsto \mathrm{Nat}, Y \mapsto \mathrm{Bool}]\big(\forall X\ Y,\ X \to Y \to X\big) = \mathrm{Nat} \to \mathrm{Bool} \to \mathrm{Nat}$
- $[X \mapsto Z \to Z]\big(\forall X,\ X \to X\big) = (Z \to Z) \to Z \to Z$

Parametric subtyping

Lemma (Parametric subtyping) :

For all substitutions σ and all types T, T $<:$ σT.



The set of values typed by Bool→ Bool contains the set of values typed by ∀T, T→ T, which also contains those typed by ∀T, T.

Principal types

What is the "best type" for a given expression ?

Definition (Principal type)

Given a typable expression $\mathtt{e}$, the **principal type** of $\mathtt{e}$ is the maximum type T for the subtype relation (when it exists) such that $\mathtt{e} : T$.

- Our inference algorithm infers types that are principal for the Hindley-Milner type system.
- The System F type system does **not** have principal types.

$$\mathtt{t} ::= \lambda\mathtt{f}.\mathtt{if}\ \mathtt{f}(\mathtt{true})\ \mathtt{then}\ \mathtt{f}(1)\ \mathtt{else}\ \mathtt{f}(0) \qquad \begin{cases} \mathtt{t}_1 : (\forall X,\ X \rightarrow X) \rightarrow \mathrm{Nat} \\ \mathtt{t}_2 : (\forall X,\ X \rightarrow \mathrm{Bool}) \rightarrow \mathrm{Bool} \end{cases}$$

Implementations of the parametric polymorphism

To compile (parametric) polymorphic code . . .

```
class HashTbl<Key, Val extends Object> {
  HashTbl() { .. }
  Val get(Key k) { .. }
  Val put(Key k, Val v) { .. }
}
```

Implementations of the parametric polymorphism

To compile (parametric) polymorphic code, two main strategies coexist :

- **Homogeneous translation** : generate code where the generic data has a uniform representation, independent of its real type.

```java
class HashTbl<Key, Val extends Object> {
  HashTbl() { .. }
  Val get(Key k) { .. }
  Val put(Key k, Val v) { .. }
}
```

→

```java
class HashTbl {
  HashTbl() { .. }
  Object get(Object k) { .. }
  Object put(Object k, Object v) { }
}
```

Example : Java with **type erasure**.

Implementations of the parametric polymorphism

To compile (parametric) polymorphic code, two main strategies coexist :

- **Heterogeneous translation** : duplicate and tailor the generated code for each possible concrete type effectively used.

```
class HashTbl<Key, Val extends Object> {
  HashTbl() { .. }
  Val get(Key k) { .. }
  Val put(Key k, Val v) { .. }
}
```

```
class HashTbl_Int_String {
  HashTbl_Int_String() { .. }
  string get(int k) { .. }
  string put(int k, string v) { }}
```

```
class HashTbl_Char_File {
  HashTbl_Char_File() { .. }
  File get(char k) { .. }
  File put(char k, File v) { }}
```

Example : C++ via the preprocessor, Rust monomorphism.

The most general approach is a combination of both styles.

Summary on the parametric polymorphism

- The parametric polymorphism allows the definition of types as logic formulas containing **universally** quantified variables.

$$\forall T,\ F[T] \equiv \bigcap_{U \in \mathbf{Typ}} F[U]$$

- Type substitution on these variables induces subtyping relations.

$$\forall T,\ F[T] <: F[U] \text{ where } U \text{ concrete}$$

- For a polymorphic expression, the maximal type wrt subtyping is called the **principal** type. Not all type systems possess maximal types.
- The **inference** of parametric types is possible in the Hindley-Milner type system, but undecidable in System F.

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.