Polymorphic Types

Monomorphic vs Polymorphic types

In general, if a type system does not allow us to express that some types accept some other types as parameters, we call the type system **monomorphic**.

As an example, in a monomorphic type system we cannot define a “generic” function to find the maximum between two elements, regardless of their type, and we would need to define specific functions for each instantiation of a comparable data type, e.g.,

`max_int: int->int->int`, `max_float: float->float->float`, etc.

although in the implementation we could probably abstract from knowing we are handling `integers` and `floats`, as long as we can compare their inhabitants.

Type Polymorphism

Summarising, a polymorphic type system allows developers to specify a set of operations on a given, parametric type that abstracts from some details of the concrete instantiation of the type.

Although the concept of polymorphism in types is generally understood, the way in which programming languages interpret and implement it varies.

Broadly, there are three kinds of polymorphism:

- **ad-hoc polymorphism** (**overloading**), where we overload the definition of a given operation on different specific types;
- **subtype polymorphism** (**subtyping**), where we set abstract-to-specific relations among types and obtain polymorphism with operations over abstract ones;
- **parametric polymorphism** (**universal**), where we have abstract symbols that represent type parameters.

Ad-hoc Polymorphism (Overloading)

As the terms **ad-hoc** and **overload** suggest, this kind of polymorphism uses the capability of a language compiler/runtime to distinguish from the call context among alternative definitions of operations with the same name.

The most common example of ad-hoc polymorphism are the arithmetic operators. E.g., `+`, in most languages is overloaded (on numerals, like `ints` and `floats`, but also on `chars` and `strings`).

Depending on the language and the call-site context, the compiler or the runtime must have enough information from types to know what implementation of a given, overloaded operation the program needs to use at a given call site.

Ad-hoc Polymorphism (Overloading)

Overloading is a sort of syntactic abbreviation that disappears as soon as we resolve the invocation, making overloading a **dispatch** mechanism.

When the dispatch happens **statically** (e.g., at compile time) we replace each overloaded symbol with an unambiguous name that uniquely denotes the specific implementation.

When the dispatch happens at runtime we have **dynamic dispatch**, usually happening through lookup tables.

Notably, overloading and type coercion frequently go hand in hand, e.g., (depending on the language) we can write `1 + 2.0` which, e.g., would coerce `1` into `1.0` and use the definition of `+` for `floats` (which also returns a `float`, e.g., 3.0).

Ad-hoc Polymorphism (Overloading)

From our example on the “max” generic function, the ad-hoc polymorphic solution could be (written in Java — C and Rust do not support overloading)

```
int max( int i, int j ){
 return i > j ? i : j;
}

float max( float i, float j ){
 return i > j ? i : j;
}
```

Subtype Polymorphism (Subtyping)

Subtype polymorphism relies on a binary relation $<:$ on types where $S <: T$ reads "S is a subtype of T". The meaning of the relation is usually that of **specification**, i.e., if $S <: T$ then S is a type more specific than T and we can safely use S in every place where T is needed (as postulated by the Liskov substitution principle [1]).

A typical example of subtype polymorphism is the `Animal` type and its subtypes `Cat` and `Dog`. We can use values of the `Cat` and `Dog` type as long as we needed an `Animal` (e.g., they both can breath and sleep).

[1] Liskov, Barbara H., and Jeannette M. Wing. "**A behavioral notion of subtyping**." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (1994): 1811-1841.

Subtype Polymorphism (Subtyping)

Usually, $<:$ is a **preoder** ($T <: T$ and $S <: T \land R <: S \implies R <: T$) as such, we cannot use `Animal` where we assume to have a `Cat` or a `Dog`, e.g., we cannot assume all `Animals` (have the operation) `bark` as the a `Dog` would.

Completing the picture, $<:$ is usually also antisymmetric, making it a **partial preorder**, i.e., it also holds true that $T <: S \land S <: T \implies T = S$.

Subtype polymorphism is typically found in object-oriented languages, given its closeness to the concept of inheritance (we will see this when talking about OO).

[1] Liskov, Barbara H., and Jeannette M. Wing. "**A behavioral notion of subtyping**." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (1994): 1811-1841.

Subtype Polymorphism, record subtyping

Intuitively, we can view $S <: T$ as a relation between some $Target$ type the context assumes and some $Substitute$ type we want to use instead of T.

Let's consider the record types

```
type Animal:{name:string}
type Dog:{name:string,bark:string}
```

where, for some <:, `Dog <: Animal`.

We can define an AnimalHouse and a DogHouse

```
type AnimalHouse:{tenant: Animal}
type DogHouse:{tenant:Dog}
```

`Animal` and `Dog` are an example of **width subtyping**, where subtype records add more fields than their super types.

`AnimalHouse` and `DogHouse` exemplify **depth subtyping**, where we replace the fields with their subtypes, **excluding writes**.

Subtype Polymorphism, record subtyping

Intuitively, we can view $S <: T$ as a relation between some $Target$ type the context assumes and some $Substitute$ type we want to use instead of T.

Let's consider the record types

```
type Animal:{name:string}
type Dog:{name:string,bark:string}
```

where, for some <:, `Dog <: Animal`.

We can define an AnimalHouse and a DogHouse

```
type AnimalHouse:{tenant: Animal}
type DogHouse:{tenant:Dog}
```

Excluding writes,

`DogHouse <: AnimalHouse`

The fact that the "direction" of the subtyping relation between DogHouse and AnimalHouse and Dog and Animal is the same is called **covariance**.

Subtype Polymorphism, record subtyping

Intuitively, we can view $S <: T$ as a relation between some $Target$ type the context assumes and some $Substitute$ type we want to use instead of T.

Let's consider the record types

```
type Animal:{name:string}
type Dog:{name:string,bark:string}
```

where, for some <:, `Dog <: Animal`.

We can define an AnimalHouse and a DogHouse

```
type AnimalHouse:{tenant: Animal}
type DogHouse:{tenant:Dog}
```

If we **consider writes**,

`DogHouse :> AnimalHouse`

reverses the "direction" of the subtyping relation `Dog <: Animal`, making it **contravariant**. Hence, we can safely replace writes on values of type `DogHouse` with values of type `AnimalHouse`.

Subtype Polymorphism, covariance and contravariance

Intuitively, we can view $S <: T$ as a relation between some $Target$ type the context assumes and some $Substitute$ type we want to use instead of T.

Let's consider the record types

```
type Animal:{name:string}
type Dog:{name:string,bark:string}
```

where, for some <:, `Dog <: Animal`.

We can define two functions A2B and D2B

```
type A2B: Animal -> Bool
type D2B: Dog -> Bool
```

A similar phenomenon happens with function types. `A2B` and `D2B` have the same output (`Bool`) but, although `Dog <: Animal`, `A2B <: D2B`, i.e., they have a **contravariant** relation.

The reason is that we can always replace `D2B`, to which the context expects to provide Dogs, with `A2B`, which uses fewer information.

Subtype Polymorphism, covariance and contravariance

Intuitively, we can view $S <: T$ as a relation between some $Target$ type the context assumes and some $Substitute$ type we want to use instead of T.

Let's consider the record types

```
type Animal:{name:string}
type Dog:{name:string,bark:string}
```

where, for some <:, `Dog <: Animal`.

We can define two functions U2A and U2D

```
type U2A: Unit -> Animal
type U2D: Unit -> Dog
```

In this case, `U2D <: U2A` is **covariant**.

The two functions expect the same input (Unit) from the context and the context expect `Animals` as output, which can be safely replaced by `Dogs`.

Subtype Polymorphism, covariance and contravariance

Intuitively, we can view $S <: T$ as a relation between some $Target$ type the context assumes and some $Substitute$ type we want to use instead of T.

A bit more elaborate example includes the record type

```
type EliteDog: { name: string, bark: string, pedigree: string }
```

where, for some <:, `EliteDog <: Dog <: Animal`.

Then, we can define two functions D2D and A2E

```
type D2D: Dog -> Dog
type A2E: Animal -> EliteDog
```

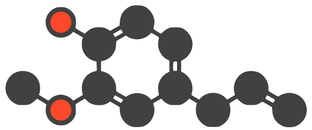
`A2E <: D2D`, which exemplifies a general rule of functional types: **argument types are contravariant** (Dog <: Animal) and **return types are covariant** (EliteDog <: Dog).

Subtype Polymorphism, covariance and contravariance

Contravariance

Product

Fuel <: Wood <: Bamboo



Consumers

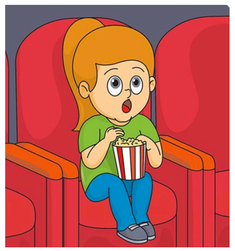


Consumer of fuel <: Consumer of wood <: Consumer of bamboo

Covariance

Product

Entertainment <: Music <: Metal



Producers



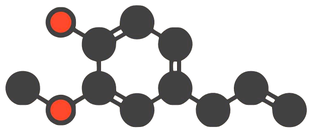
Producer of Entertainment <: Producer of Music <: Producer of Metal

Subtype Polymorphism, covariance and contravariance

Contravariance

Product

Fuel <: Wood <: Bamboo



Consumers

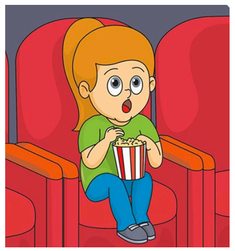


Fuel -> Unit Wood -> Unit Bamboo -> Unit

Covariance

Product

Entertainment <: Music <: Metal



Producers



Unit -> Ent. Unit -> Music Unit -> Metal

Subtype Polymorphism, subsumption

The act of deciding whether $S <: T$ is called **subsumption**.

There are mainly two strategies to define subsumption (dependent on the two ways in which we can define type membership): **extensionally** or **intensionally**.

Extensionally, we have that $S <: T$ if $\forall s \in S, [\![s]\!] \in T$.

Intensionally, if $S <: T$, then the predicate that defines the membership for T 1) must be part of the predicate for S and 2) must apply on the same domain as the one for S.

In many cases, type systems define specific subtyping relations for basic types, e.g., `int <: float` or `char <: string`

Subtype Polymorphism

From our example on the “max” generic function, the subtype polymorphic solution could be written

```
Integer <: Float

max( Float i, Float j ) -> Float { … }
```

Notice that, using this solution, we lose information on the values of the subtypes that we have as input and, indeed, we can only safely return a value of the **supertype** (`Float`) and not the specific one (which is only known at call site of `max`).

This can cause problems, e.g., due to the need of forcing a casting of the value returned from `max` to be usable in a given context (e.g., we find the `maximum` between two `integers` and we feed the former into a function that accepts `integers`).

Subtype Polymorphism

From our example on the “max” generic function, the subtype polymorphic solution could be written

```
Integer <: Comparable

Float <: Comparable

max( Comparable i, Comparable j ) -> Comparable { … }
```

Notice that, using this solution, we lose information on the values of the “comparables” that we have as input and, indeed, we can only safely return a value of the **supertype** (comparable) and not the specific one (which is only known at call site of max).

This can cause problems, e.g., due to the need of forcing a casting of the value returned from max to be usable in a given context (e.g., we find the maximum between two integers and we feed the former into a function that accepts integers).

Parametric Types

Some data structures have invariants that let us provide type safety although we do not fully know their full shape.

An example we already saw of these data structures is the type `Set`, for which we said there are usually defined some typical operations such as union, intersection, subtraction, and inclusion testing.

An observation we can make is that, for an implementor of the `Set` data structure, the operations on sets remain the same whether a given set instance handles `integers`, `chars`, or `arrays`. More specifically, the operations of the `Set` are **parametric** to the elements in the set, e.g., (simplistically) if we need to test for inclusion, we would apply the definition of `integer` equivalence if we have a `Set` of `integers` and the same goes for `chars`, `arrays`, etc.

In this case, we say that `Set` is a **parametric type** and, when instantiated with, e.g., `integers`, that the latter is its (actual) **type parameter**.

Parametric Polymorphism (Universal)

Extending a type system with parametric types introduces the possibility of **parametric polymorphism**.

In general, when using parametric polymorphism, we cannot make any assumptions on the shape of the type parameters, which essentially forces us (and the type system) to consider any possible types in implementations.

This is the reason why parametric polymorphism is also called **universal** polymorphism, i.e., because we read the type parameter declaration `Set(T)` as `∀ T . Set(T)`.

Theorems for free, example

As an example, consider

```
r( T ): List( T ) -> List( T ),
f: A -> B,
map( T, S ): List( T ) -> (T -> S) -> List( S )
```

then, `let l: List( A )`

```
map( r( l ), f ) = r( map( l, f ) )
```

Intuitively, since `r` ignores the shape of `T` (which can literally be any type ∀T) it must only work on the non-parametric type constructor `List` and all `r` can do is rearrange the list (e.g., remove items based on their index, duplicate them, etc.) independent of their values. Thus, we can safely (*as far as types can "see"*) swap applying `f` to each element of the rearranged list (left) and rearrange the list after the application of `f` to each of its elements (right).

[1] Wadler, Philip. "**Theorems for free!**." Proceedings of the fourth international conference on Functional programming languages and computer architecture. 1989.

Hybrid and Bounded Parametric Polymorphism

While parametric polymorphism assumes no knowledge on type parameters (and could provide useful properties on this abstraction [1]), some languages with parametric polymorphism provide a **type introspection** operator (e.g., **instanceof** in Java) that lets the user check whether a value belongs in a given type, at the detriment of the properties of abstraction.

Parametric polymorphism captures the universality of type expressions, but sometimes it is useful to express **boundaries** over the universal quantifier, e.g., to limit the quantification only on types with certain properties.

A common way to express these constraints is by mixing subtyping and parametric polymorphism, as done in Java and Rust.

For example, a polymorphic version of our `max` function can have the type

```
∀ T, T <: Comparable, max: T -> T -> T
```

[1] Wadler, Philip. "**Theorems for free!**." Proceedings of the fourth international conference on Functional programming languages and computer architecture. 1989.

Parametric Polymorphism

To be able to express parameters in types, we need to introduce a new notation that makes it explicit when types accept parameters.

For example, Java and Rust can support parametric types via **generics** (and traits). We can write a parametric version of Set as `Set<T>`, where Set is a polymorphic type that accepts one (formal) **type parameter**, here captured with the **type variable** `T`. Similarly, we can define `max` as (Java and Rust)

```
<T extends Comparable> T max (T x, T y){…}
```

```
fn max<T: Comparable>(x:T, y:T)->T{…}
```

which takes in and **returns a T**.

Parametric Polymorphism (Universal)

Parametric Polymorphism introduces the notion of universal types — as a shorthand for **universally quantified types** — where definitions like

```
<T extends Comparable> T max (T x, T y){…}
```

and

```
fn max<T: Comparable>(x:T, y:T)->T{…}
```

have `max` with type $\forall T.T <: \text{Comparable}.T \times T -> T$ in which the universal $\forall$ indicates that the definition is valid for (and parametric to) any type T (as long as it is a subtype of Comparable).

Parametric Polymorphism and subtyping

Mixing parametric polymorphism and subtyping introduces the notion of subtyping of parametric types, e.g.,

if `Dog <: Animal`, does `Set<Dog> <: Set<Animal>`?

The matter is similar to that of depth subtyping for records, where it is safe to consider `Set<Dog> <: Set<Animal>` as long as we do not perform writes on the `Set`.

The notion above plays with width subtyping of the parametric types, e.g.,

```
type Set<T>: {add:T->(),remove:T->(),includes:T->Bool}

type List<T>:{add:T->(),remove:T->(),includes:T->Bool,get:int->T}
```

where `List<S> <: Set<T>` depending on the usage of the parametric type and the relation of the type parameters S and T.

Parametric Polymorphism and subtyping

While in the general case it is safe to consider `Set<Dog> <: Set<Animal>` as long as we do not perform writes on the `Set`, languages with (some form of) parametric polymorphism **annotate explicitly the intended "direction"** the user expects to use parametric types

```
Vector< Dog > dv = new Vector< Dog >();
dv.add( new Dog() );
// covariant (* :> Dog), read only
Vector< ? extends Dog > cov = dv;
Dog d = cov.get( 0 ); // Dog <: Dog
Animal a = cov.get( 0 ); // Dog <: Animal
// contravariant (* <: Dog), write only
Vector< ? super Dog > con = dv;
con.add( new EliteDog() ); // EliteDog <: Dog
con.add( new Dog() ); // Dog <: Dog
// below, since we do not specify its use in the type
// we assume "invariance" and report a typing error
Vector< Animal > inv = dv;  ❌
```

The (strange case) of C++/Java's arrays

In Java and C++ the type **array** is a parametric polymorphic type... in disguise. The languages added support for polymorphic types in later releases, while much of the preexisting code used non-polymorphic arrays. To make arrays useful, without polymorphism, the implementors decided to make both writes and reads **covariant**. Hence, the type system does not reject

```
Animal[] a = new Dog[1];
a[ 0 ] = new Dog();
```

```
// ← which we should not accept in general, without
// knowing the "direction" of the usage of a
// i.e., if we are going to use it in a covariant (read)
// contravariant (write) way
```

reads are fine for both the type checker and the runtime

```
Animal a0 = a[ 0 ];
```

and since we "forced" covariance, writing is OK for the type checker ...

```
a[0] = new Animal();
```

Expensive

... but it breaks runtime checks (for contravariance of array update — e.g., it correctly accepts EliteDog, but not Animal), and it produces an `ArrayStoreException`.

Monadic types: Option and Result Types

Monads, proposed by Eugenio Moggi to represent computations in programming languages, are an abstraction used mainly in functional languages to simplify the composition and resolution (handling the control flow, side effects, etc.) of chains of functions.

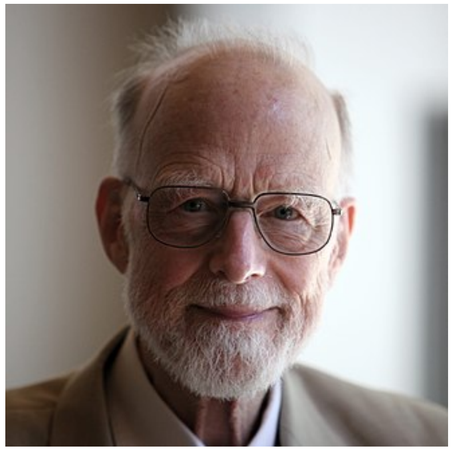
Here, we look at monads simplistically.

We consider them as “containers” that encapsulate some functionality and we focus on **monadic types** that have become more and more common in mainstream programming languages: **`Option`**s and **`Result`**s.

Option/Maybe Type

The Option (also called Maybe) type is useful to gracefully handle (and dispense a language from) `null` pointers.

Recalling pointer types, we remarked the existence, in some type systems, of a special inhabitant of the type, called `null`, which indicates that the pointer does not refer to a valid (initialised/usable) memory location. The inventor of null references, Turing-awardee **Tony Hoare**, famously called it his “billion-dollar mistake”.



Also known for the invention of **quicksort**, **Hoare logic**, the **dining philosophers** problem, and communicating sequential processes (**CSP**) language.

> [...] in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to **innumerable errors**, **vulnerabilities**, and **system crashes**, which have probably caused a billion dollars of pain and damage in the last forty years.

Option/Maybe Type

Option/Maybe types mitigate (if not remove) the problem of null pointers by presenting, at the type level, the duality of valid vs invalid pointers, mixing parametric and sum types.

There are different, equivalent interpretations of Option/Maybe types, e.g.,

```
type Maybe<T> : Some<T> + None
```

where a value of type Maybe<T> is either a wrapper around some value of type T (Some<T>) or the singleton value of type None (equivalent to Unit).

Java introduced the Optional type in version 8. Optional provides a dedicated, functional-inspired interface, e.g.,

```
Optional< Integer > opt = Option.of( 42 );
var d = 2 * opt.orElseGet( () -> 0 );
```

Rust uses enums to encode Option types and mainly relies on pattern matching to manage its cases

```
let opt: Option< i32 > = Some( 42 );
let d = match opt {
 Some( x ) => 2*x,
 None => 0        }
```

Result Type

We can think of Result types as a refinement of Maybe/Option types, where we use polymorphic and sum types to distinguish between the successful result of some computation and some faulty execution, signalled/informed by some error.

A possible implementation of this idea is

```
type Result< T, E >  : Ok< T > + Err< E >
```

In essence, Result types represent an alternative to exception handling (and reasoning/dealing with the dynamic behaviour associated to it) that forces a more linear handling of computation states, since the programmer must (as long as the type system forces them to) either totally discard the result (e.g., because they do not care about its execution) or consider all its possible states.

```
let res: [ Result< i32, &str >; 3 ] =
 [Ok(40), Err("Error"), Ok(2)];
let mut acc: i32 = 0;
for r in &res {
 acc += match r {
  Ok( _r ) => _r,
  Err( _ ) => 0
 }
}
```