WIKIPEDIA

Polymorphism (computer science)

In programming languages and type theory, **polymorphism** is the provision of a single interface to entities of different types[1] or the use of a single symbol to represent multiple different types.[2]

The most commonly recognised major classes of polymorphism are:

- *Ad hoc polymorphism*: defines a common interface for an arbitrary set of individually specified types.
- *Parametric polymorphism*: when one or more types are not specified by name but by abstract symbols that can represent any type.
- *Subtyping* (also called *subtype polymorphism* or *inclusion polymorphism*): when a name denotes instances of many different classes related by some common superclass.[3]

Contents

History

Interest in polymorphic type systems developed significantly in the 1960s, with practical implementations beginning to appear by the end of the decade. *Ad hoc polymorphism* and *parametric polymorphism* were originally described in Christopher Strachey's *Fundamental Concepts in Programming Languages*[4], where they are listed as "the two main classes" of polymorphism. Ad hoc polymorphism was a feature of Algol 68, while parametric polymorphism was the core feature of ML's type system.

In a 1985 paper, Peter Wegner and Luca Cardelli introduced the term *inclusion polymorphism* to model subtypes and inheritance,[2] citing Simula as the first programming language to implement it.

Types

Ad hoc polymorphism

Christopher Strachey[5] chose the term *ad hoc polymorphism* to refer to polymorphic functions that can be applied to arguments of different types, but that behave differently depending on the type of the argument to which they are applied (also known as function overloading or operator overloading). The term "ad hoc" in this context is not intended to be pejorative; it refers simply to the fact that this type of polymorphism is not a fundamental feature of the type system. In the Pascal / Delphi example below, the `Add` functions seem to work generically over various types when looking at the invocations, but are considered to be two entirely distinct functions by the compiler for all intents and purposes:

```
program Adhoc;

function Add(x, y : Integer) : Integer;
```

```
begin
    Add := x + y
end;

function Add(s, t : String) : String;
begin
    Add := Concat(s, t)
end;

begin
    Writeln(Add(1, 2));                    (* Prints "3"             *)
    Writeln(Add('Hello, ', 'World!'));    (* Prints "Hello, World!" *)
end.
```

In dynamically typed languages the situation can be more complex as the correct function that needs to be invoked might only be determinable at run time.

Implicit type conversion has also been defined as a form of polymorphism, referred to as "coercion polymorphism".[2][6]

Parametric polymorphism

Parametric polymorphism allows a function or a data type to be written generically, so that it can handle values *uniformly* without depending on their type.[7] Parametric polymorphism is a way to make a language more expressive while still maintaining full static type-safety.

The concept of parametric polymorphism applies to both data types and functions. A function that can evaluate to or be applied to values of different types is known as a *polymorphic function*. A data type that can appear to be of a generalized type (e.g. a list with elements of arbitrary type) is designated *polymorphic data type* like the generalized type from which such specializations are made.

Parametric polymorphism is ubiquitous in functional programming, where it is often simply referred to as "polymorphism". The following example in Haskell shows a parameterized list data type and two parametrically polymorphic functions on them:

```
data List a = Nil | Cons a (List a)

length :: List a -> Integer
length Nil         = 0
length (Cons x xs) = 1 + length xs

map :: (a -> b) -> List a -> List b
map f Nil         = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

Parametric polymorphism is also available in several object-oriented languages. For instance, templates in C++ and D, or under the name generics in C# and Java:

```
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}

List<B> map(Func<A, B> f, List<A> xs) {
    ...
}
```

John C. Reynolds (and later Jean-Yves Girard) formally developed this notion of polymorphism as an extension to lambda calculus (called the polymorphic lambda calculus or System F). Any parametrically polymorphic function is necessarily restricted in what it can do, working on the shape of the data instead of its value, leading to the concept of parametricity.

Subtyping

Some languages employ the idea of *subtyping* (also called *subtype polymorphism* or *inclusion polymorphism*) to restrict the range of types that can be used in a particular case of polymorphism. In these languages, subtyping allows a function to be written to take an object of a certain type *T*, but also work correctly, if passed an object that belongs to a type *S* that is a subtype of *T* (according to the Liskov substitution principle). This type relation is sometimes written *S* <: *T*. Conversely, *T* is said to be a *supertype* of *S*—written *T* :> *S*. Subtype polymorphism is usually resolved dynamically (see below).

In the following example we make cats and dogs subtypes of animals. The procedure `letsHear()` accepts an animal, but will also work correctly if a subtype is passed to it:

```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}

static void letsHear(final Animal a) {
    println(a.talk());
}

static void main(String[] args) {
    letsHear(new Cat());
    letsHear(new Dog());
}
```

In another example, if `Number`, `Rational`, and `Integer` are types such that `Number :> Rational` and `Number :> Integer`, a function written to take a `Number` will work equally well when passed an `Integer` or `Rational` as when passed a `Number`. The actual type of the object can be hidden from clients into a black box, and accessed via object identity. In fact, if the `Number` type is *abstract*, it may not even be possible to get your hands on an object whose *most-derived* type is `Number` (see abstract data type, abstract class). This particular kind of type hierarchy is known—especially in the context of the Scheme programming language—as a *numerical tower*, and usually contains many more types.

Object-oriented programming languages offer subtype polymorphism using *subclassing* (also known as *inheritance*). In typical implementations, each class contains what is called a *virtual table*—a table of functions that implement the polymorphic part of the class interface—and each object contains a pointer to the "vtable" of its class, which is then consulted whenever a polymorphic method is called. This mechanism is an example of:

- *late binding*, because virtual function calls are not bound until the time of invocation;
- *single dispatch* (i.e. single-argument polymorphism), because virtual function calls are bound simply by looking through the vtable provided by the first argument (the `this` object), so the runtime types of the other arguments are completely irrelevant.

The same goes for most other popular object systems. Some, however, such as Common Lisp Object System, provide *multiple dispatch*, under which method calls are polymorphic in *all* arguments.

The interaction between parametric polymorphism and subtyping leads to the concepts of variance and bounded quantification.

Row polymorphism

Row polymorphism is a similar, but distinct concept from subtyping. It deals with structural types. It allows the usage of all values whose types have certain properties, without losing the remaining type information.

Polytypism

A related concept is *polytypism* (or *data type genericity*). A polytypic function is more general than polymorphic, and in such a function, "though one can provide fixed ad hoc cases for specific data types, an ad hoc combinator is absent".[8]

Implementation aspects

Static and dynamic polymorphism

Polymorphism can be distinguished by when the implementation is selected: statically (at compile time) or dynamically (at run time, typically via a virtual function). This is known respectively as *static dispatch* and *dynamic dispatch,* and the corresponding forms of polymorphism are accordingly called *static polymorphism* and *dynamic polymorphism*.

Static polymorphism executes faster, because there is no dynamic dispatch overhead, but requires additional compiler support. Further, static polymorphism allows greater static analysis by compilers (notably for optimization), source code analysis tools, and human readers (programmers). Dynamic polymorphism is more flexible but slower—for example, dynamic polymorphism allows duck typing, and a dynamically linked library may operate on objects without knowing their full type.

Static polymorphism typically occurs in ad hoc polymorphism and parametric polymorphism, whereas dynamic polymorphism is usual for subtype polymorphism. However, it is possible to achieve static polymorphism with subtyping through more sophisticated use of template metaprogramming, namely the curiously recurring template pattern.

See also

- Duck typing for polymorphism without (static) types
- Polymorphic code (Computer virus terminology)
- System F for a lambda calculus with parametric polymorphism.
- Type class
- Type theory
- Virtual inheritance

References

1. Bjarne Stroustrup (February 19, 2007). "Bjarne Stroustrup's C++ Glossary" (http://www.stroustrup.com/glossary.html#Gpolymorphism). "polymorphism – providing a single interface to entities of different types."
2. Cardelli, Luca; Wegner, Peter (December 1985). "On understanding types, data abstraction, and polymorphism" (http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf) (PDF). *ACM Computing Surveys*. New York, NY, USA: ACM. **17** (4): 471–523. doi:10.1145/6041.6042 (https://doi.org/10.1145%2F6041.6042). ISSN 0360-0300 (https://www.worldcat.org/issn/0360-0300).: "Polymorphic types are types whose operations are applicable to values of more than one type."
3. Booch, et al 2007 *Object-Oriented Analysis and Design with Applications.* Addison-Wesley.
4. Strachey, Christopher (2000). "Fundamental Concepts in Programming Languages" (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.332.3161&rep=rep1&type=pdf). *Higher-Order and Symbolic Computation*. Kluwer Academic Publishers. **13** (1/2): 11–49. doi:10.1023/A:1010000313106 (https://doi.org/10.1023%2FA%3A1010000313106). ISSN 1573-0557 (https://www.worldcat.org/issn/1573-0557). Retrieved 11 January 2019.
5. Christopher Strachey. *Fundamental Concepts in Programming Languages* (http://www.itu.dk/courses/BPRD/E2009/fundamental-1967.pdf) (PDF). *www.itu.dk*. Kluwer Academic Publishers.
6. Allen B. Tucker (28 June 2004). *Computer Science Handbook, Second Edition* (https://books.google.com/books?id=9IFMCsQJyscC&pg=SA91-PA5). Taylor & Francis. pp. 91–. ISBN 978-1-58488-360-9.
7. Pierce, B. C. 2002 *Types and Programming Languages.* MIT Press.
8. Ralf Lammel and Joost Visser, "Typed Combinators for Generic Traversal", in *Practical Aspects of Declarative Languages: 4th International Symposium* (2002), p. 153.

External links

- C++ examples of polymorphism (http://www.cplusplus.com/doc/tutorial/polymorphism/)
- Objects and Polymorphism (Visual Prolog) (http://wiki.visual-prolog.com/index.php?title=Objects_and_Polymorphism)