CSc 520

Principles of Programming Languages

16 : Types — Polymorphism

Christian Collberg

collberg+520@gmail.com

Department of Computer Science

University of Arizona

What is polymorphism?

- Polymorphic means "having multiple forms." In programming languages it refers to code and data that can work with values of different types.
- A variable is polymorphic if it can refer to objects of different types.
- A function is polymorphic if you can pass arguments of different types to it.
- We want to define functions that are as reusable as possible.
- Polymorphic functions are reusable because they can be applied to arguments of different types.

Polymorphic Variables

- A variable is polymorphic if it can refer to objects of different types.
- There's a trade-off between having as much static typing as possible (so that the compiler can tell you about errors in your code as early as possible) and allowing as much polymorphism as possible (to make your code flexible).

Polymorphic Variables in Ruby

- In Ruby, a variable can refer to any type of data. No type checking is done until runtime.
- The last expression will fail at runtime since `sub` is expecting a regular expression as its first argument:

```
x = 42; puts x
x = "42"; puts x
x = 42.42; puts x
x = [42,42.0,"42"]; puts x
x = /duck/
puts "duckduckduck".sub(x,"ruby")
x = 42
puts "duckduckduck".sub(x,"ruby")
```

Polymorphic Functions

- A function is polymorphic if you can pass arguments of different types to it.
- You want the `list_length` function to be polymorphic so that you can pass a list-of-integers to it, a list-of-reals, a list-of-lists, etc. It shouldn't be necessary to write one `list_length` function for each list type.
- Similarly, you want to write exactly one `sort` function that can sort an array-of-100-integers, an array-of-1000-reals, etc.

Types of Polymorphism

- There are two major kinds of polymorphism, parametric polymorphism and inclusion polymorphism.
- Explicit parametric polymorphism is also known as genericity.
- Inclusion polymorphism is also known as subtype polymorphism.
- Ada has parametric polymorphism.
- Java and C++ have both parametric polymorphism (templates and generics) and inclusion polymorphism (inheritance, subtyping).

Types of Polymorphism — Parametric

- In a language with parametric polymorphism he code takes a type argument, such as:

```
function sort[T](A : array of T) { ... }
```

  which you then have to instantiate with a particular type:

```
function intSort is new sort[integer];
function stringSort is new sort[String];
```

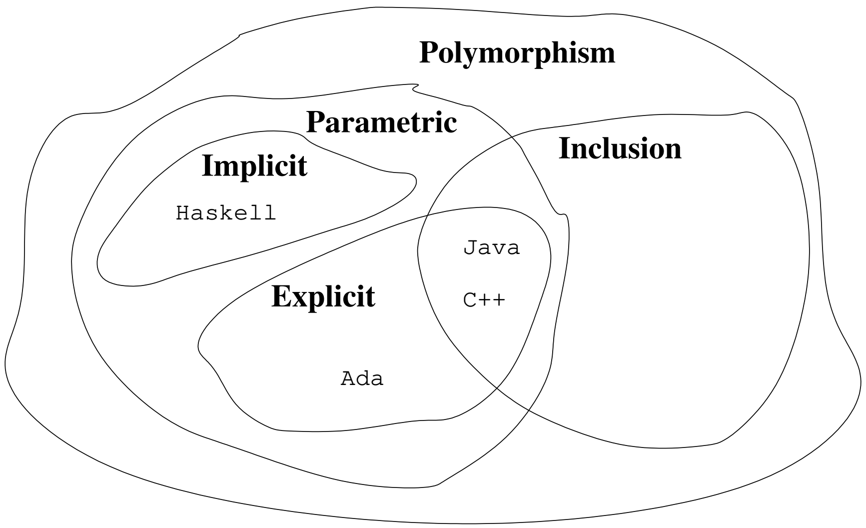
- Parametric polymorphism can be explicit (you have to supply the type, as in the sort function above) or implicit (the compiler figures out the type:

```
function sort(A : array of <some_type>) { ..
...
var x : array [1..100] of real; sort(x);
var y : array [1..100] of integer; sort(y);
```


Types of Polymorphism — Inclusion

- You get inclusion polymorphism in languages that support subtyping, i.e. any object oriented language with inheritance.
- You write your code to handle one particular type T.
- But, you can then create *subtypes* of T, and your code will work on objects of these subtypes also.
- We'll talk a lot about inclusion polymorphism and how it's implemented in the lectures on object-oriented languages.
- Java and C++ support both generics and inclusion polymorphism.

Types of Polymorphism...



Levels of Freedom

- Depending on the language, we get different levels of freedom, and different levels of protection against type-errors.
- In some languages you can only have arrays of one element type:

  ```
  x = [1,2,3,4]
  ```

  but you can call a function with arrays of different types:

  ```
  sort([1,2,3,4])
  sort(["hey","bar","ber","ri","ba"])
  ```


Levels of Freedom...

- In some languages you can mix the array element types:

  ```
  y = [1,2.3,"hello",[4,5]]
  ```

- In languages with inclusion polymorphism you can put "similar" objects in the same array

  ```
  Animals a = [monkey, puffer_fish, amoeba];
  ```

  but the compiler (or runtime system) will complain if you try to put "really different" ones together:

  ```
  Animals b = [monkey, honda_civic];
  ```


Language Examples

- Let's have a look at a couple of languages and how they define polymorphism!
- I'm going to take a roughly historical approach:
  1. We'll start with C and Pascal (where you have to abuse the type system to make polymorphic functions), and then we'll look at
  2. Modula-2 which has a (very) limited form of polymorphic arrays called *conformant arrays*, and on to
  3. Ada which has fullblown generic functions and modules, and then
  4. Java which supports inclusion polymorphism through its subtyping mechanism, and finally
  5. Haskell's implicit parametric polymorphism.

C's Generic Reference Types

C

- C doesn't have any real support to build polymorphic functions.
- But, since it's so easy to bypass any static typing in C (we can cast pointers willy-nilly), we can still build functions that can take multiple types of argument.
- Of course, when we're doing this we give up any type of static type-checking — it's up to the programmer to make sure that he does the right thing — the compiler won't be able to help him at all!

Polymorphic sort in C

- C's standard library has a function `qsort()`:

```
void qsort(void *base,
           size_t n,
           size_t elsize,
           int (*compare)(void *, void *));
```

- It takes four arguments:
  1. `base` is a pointer to the array
  2. `n` is a number of elements
  3. `elsize` is the size of each element
  4. `compare` is a function pointer which returns -1 for less than, 0 for equal, and 1 for greater than.

Polymorphic sort in C — Sorting `ints`

```
int compare_ints(void *a, void *b) {
    int ia = *(int *)a;
    int ib = *(int *)b;
    return (ia<ib)?-1:1;
}
void main() {
    int data[] = { ... };
    qsort(data, num_elmts,
          sizeof(int), compare_ints);
}
```

- Inside `compare_ints` we have to cast the generic void pointers to the element type of the array, `int`s in our case.
- If we cast to the wrong type, disaster ensues. . .

Pascal's Variant Records

Pascal

- Pascal doesn't have any real support to build polymorphic functions.
- If you've written a sorting routines for arrays of integers, that doesn't help you if you want to sort an array of reals!
- In the original Pascal definition, if you had a sorting routine that would sort an array of 100 integers you couldn't even use it to sort an array of 101 integers!
- The only way to bypass Pascal's rigid type system is to use the loop-hole known as *untagged variant records* (similar to C's `union`s).

Tagged Variant Records

```
type name = record
                field name :  type;
                field name :  type;
                case tag name :  type of
                   case :  (field name:type;...)
                   case :  (field name:type;...)
            end;
```


Tagged Variant Records...

```
type rec = record
              a :  integer;
              case tag :  boolean of
                 true :  (x :  integer);
                 false :  (y :  char);
           end;

var r:  rec;
begin
   r.tag := true; r.x := 55;
   r.tag := false; r.y := 'A';
   r.tag := true; r.y := 55; (* Runtime error?
end.
```

Untagged Variant Records

```
type rec = record
              a :  integer;
              case boolean of
                 true :  (x :  integer);
                 false :  (y :  char);
           end;

var r:  rec;
begin
   r.x := 55; r.y := 'A';
end.
```

- This construct is used to bypass Pascal's strong typing.

Generic sort

- Here are two ways to build a generic sort routine in Pascal.
- In the first one we use tagged variant records to, essentially, allow run-time typing of data.
- In the second case we use untagged variant records but pass along a comparison function.
- We have to write one comparison function for every type of data we want to sort.

Generic sort in Pascal 1

```
type R = record
            case tag : boolean of
               true : (x : integer);
               false : (y : real);
         end;
     A = array [1..100] of R;

procedure sort(p:A);
var greater : boolean;
    i,j     : integer;
begin
   i := 1;j := 2;
   if p[i].tag=true then greater := p[i].x > p[
   else                  greater := p[i].y > p[
end;
```


Generic sort in Pascal 1...

```
var x:A;
begin
   x[1].tag := true;
   x[1].x := 42;
   x[2].tag := true;
   x[2].x := 52;
   sort(x);
   x[1].tag := false;
   x[1].y := 42.5;
   x[2].tag := false;
   x[2].y := 52.3;
   sort(x);
end.
```

Generic sort in Pascal 2

```
program p;
type R = record
            case boolean of
               true : (x : integer);
               false : (y : real);
         end;

    A = array [1..100] of R;

function cmpInts(p:A;i:integer;j:integer):boole
begin return p[i].x > p[j].x; end;

function cmpReals(p:A;i:integer;j:integer):bool
begin return p[i].y > p[j].y; end;
```


Generic sort in Pascal 2. . .

```
procedure sort(p:A;
    function cmp(x:A;i:integer;j:integer):boole
var greater : boolean;
    i,j     : integer;
begin
   i := 1; j := 2;
   greater := cmp(p,i,j);
end;
```


Generic sort in Pascal 2. . .

```
var x:A;
begin
   x[1].x := 42;
   x[2].x := 52;
   sort(x, greaterInts);
   x[1].y := 42.5;
   x[2].y := 52.3;
   sort(x, greaterReals);
   x[1].x := 42;
   x[2].x := 52;
   sort(x, greaterReals);
end.
```


Modula-2’s conformant arrays

Pascal array parameters

- Originally, a Pascal function that took an array-of-100-integers as argument wouldn't accept an array-of-101-integers.

```
program p;

type A = array [1..100] of integer;

procedure sort(p:A);
begin end;

var x:A;
begin
   sort(x);
end.
```


Modula-2's arrays

- Modula-2 relaxed this by allowing conformant arrays, array parameters that could be of any length.
- You still couldn't pass an array-of-ints and an array-of-floats to the same function, though. . .

```
PROCEDURE Sum(vector : ARRAY OF REAL) : REAL
VAR i:INTEGER;
    tot:REAL;
BEGIN
   tot := 0;
   FOR i := 0 TO HIGH(vector) DO
      tot := tot + Donkey[vector];
   END;
   RETURN tot;
END Sum;
```


Ada's Generics

Ada's generic modules

- An Ada module specification has two parts, a public part and a private part.
- The private part contains the definitions of all those items that we don't want a user to know about. In this example, the private part reveals that the stack is implemented as an array.

Ada

```
generic
   type ITEM is private;
package GENERIC_STACK is
   type STACK (SIZE : POSITIVE) is limited private;
   procedure PUSH (S : in out STACK; E : in ITEM);
   procedure POP ( S : in out STACK; E : out ITEM);
   pragma INLINE (PUSH, POP);
private
   type VECTOR is array (POSITIVE range < >) of ITEM;
   type STACK (SIZE : POSITIVE) is record
       SPACE : VECTOR (1 .. SIZE);
       INDEX : NATURAL := 0;
     end record;
end GENERIC_STACK;
```


Ada. . .

- Before we can use a stack we have to `instantiate` it with the type of the element and the number of elements we want:

```
package body GENERIC_STACK is
   -- Implementations of ... PUSH and POP ...
end GENERIC_STACK;

with GENERIC_STACK;
procedure MAIN is
   package STACK_INT is new GENERIC_STACK (INTEGER);
   S : STACK_INT.STACK (100);
begin
   STACK_INT.PUSH (S, 314);
end MAIN;
```


Generics as Macros

- Instantiating a generic module is just like making a copy of the source code for the module, and replacing the type with the one we want.
- You could implement Ada style generics using C's macros!

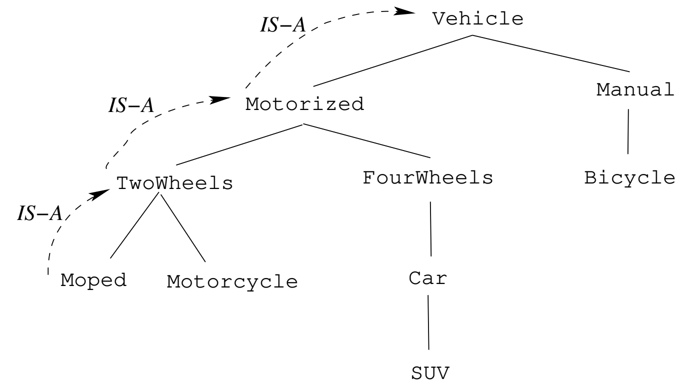
Avoiding code-bloat

- One of the reasons for having generics in the Ada language was to avoid code-bloat: we don't want to have one `int_sort` routine, one `float_sort` routine, etc.
- However, it was up to the compiler writer to decide whether to share code between instantiations! So, there was never any guarantee that you would save on memory.
- ALSO, these days code bloat is less of an issue, given the large memories of modern machines.

Inclusion Polymorphism in Java

Java

- Java is an object oriented language.
- This means we can create hierarchies of types:



- Each of these types is a class.
- An `SUV` *is-a* type of a `Car` which *is-a* type of a `FourWheels` vehicle, which *is-a* kind of `Motorized` vehicle, which *is-a* `Vehicle`

A Java class hierarchy

- This is what it looks like in Java source (indentation is just for clarity):

```
class Vehicle {}
   class Motorized extends Vehicle {}
      class TwoWheels extends Motorized {}
          class Moped extends TwoWheels {}
          class Motorcycle extends TwoWheels {}
      class FourWheels extends Motorized {}
         class Car extends FourWheels {}
            class SUV extends Car {}
    class Manual extends Vehicle {}
       class Bicycle extends Manual {}
```


Inclusion polymorphism

- Object-oriented languages support inclusion polymorphism, also known as subtype polymorphism.
- Unlike C's void pointers which you can assign and cast however you like, the Java type hierarchy restrics what you're allowed to assign.
- In general, you can assign a variable who's type is high up in the type hierarchy a variable that's lower down. The reason is that the lower down you get in the hierarchy, the more *specific* the type gets.
- For example, an `SUV` *is-a* kind of `Car` (it has all the characteristics of a car, plus some extra ones, such as being bad for the environment), so it's OK to assign an object of type `SUV` to a variable of type `Car`.
- Some type checks can be done at compile time, but

A Java class hierarchy...

```
class Main {
   public static void main(String args[]) {
      Car civic = new Car();
      Vehicle v = civic;     /* OK */
      Bicycle b = civic;     /* static error */
      FourWheels f = civic; /* OK */
      SUV s = (SUV)f;        /* runtime error */
   }
}
```

- `f`'s static type is `Vehicle` so it *could* contain an object of type `SUV`, we just don't know (until runtime) when a `ClassCastException` is thrown.

Parametric Polymorphism in Haskell

Haskell Polymorphic Functions

- Haskell is a statically typed functional language.
- Functions of polymorphic type are defined by using type variables in the signature:

```
length ::  [a] -> Int
length s = ...
```

- `length` is a function from lists of elements of some (unspecified) type `a`, to integer. I.e. it doesn't matter if we're taking the length of a list of integers or a list of reals or strings, the algorithm is the same.

```
length [1,2,3]        ⇒ 3 (list of Int)
length ["Hi ", "there", "!"] ⇒ 3 (list of String
length "Hi!"          ⇒ 3 (list of Char)
```


Constrained Parametric Polymorphism

- Not every type can be used in every context. For example, to be able to sort an array the array elements must be "comparable", i.e. they must support the <-operator.
- Haskell uses **context predicates** to restrict polymorphic types:

  ```
  member ::  Eq [a] => [a] -> a -> bool
  ```

  Now, `member` may only be applied to list of elements where the element type has == and \= defined.
- `Eq` is called a **type class.** `Ord` is another useful type class. It is used to restrict the polymorphic type of a function to types for which the relational operators (`<`, `<=`, `>`, `>=`) have been defined.

Context Predicates

- Consider the Haskell `sort` function which has the following type:

```
sort ::  (Ord a) => [a] -> [a]
```

- `sort` can be applied to any type of arrays for which we've defined the relational operators (`<, <=, >, >=`)
- This is called constrained parametric polymorphism. Ada's generics support this also.

Readings and References

- Readings in Scott:
  1. Polymorphism and related concepts (pp. 145-149).
  2. Polymorphism (pp. 309-311).
  3. Overloading and coercion (pp. 330-331)
  4. Generic reference types (pp. 331-332)
  5. Conformant arrays (pp. 355-356)
  6. Generic subroutines and modules (pp. 434-440)
  7. Polymorphism in object oriented languages (505–508)