Polymorphism

- **poly** = many, **morph** = shape
- Code is **polymorphic** if it can be used with values from more than one type
- Have already seen **subtype polymorphism** in Iota

```
interface set { ... }
class hashSet implements set { ... }
class arraySet implements set { ... }
intersect(s1, s2: set): set
```

- **intersect** works with **s1, s2** from any class that implements the **set** interface

Outline

- Two other forms of polymorphism
  - ad hoc polymorphism (overloading)
  - parametric polymorphism
- **Language design issues**
- Type checking
- Code generation
- Appel Chap. 16

Overloading

- Overloading: same name can be reused with different types (also: **ad-hoc polymorphism**)
- Ambiguity resolved by static argument types

```
print(int x)
print(string s)
print(float f)
```

- Looks like one polymorphic function **print**
- Reality: three different functions bound to same name—**not** true polymorphism
  - Three separate entries in symbol table
- Overloading relies on knowing argument types; conflicts with templates, type inference

Parametric polymorphism

- Subtype, ad-hoc polymorphism don’t allow abstraction over types
- Example: generic array sort routine
  - **Type T** is a **parameter** to function
  - **Types** used like values (can pass as arguments). Useful to separate different kinds of arguments:
```
sort(T: type, a: array[T])
```

Generic array sort

```
sort[T: type](a: array[T]) = ( 
  i,j:int = 1;
  while (j < n) ( 
    e: T = a[j];
    i = j-1;
    while (i >= 0 && a[i] > e) 
      ( a[i+1] = a[i]; i--; )
    a[i+1] = e
  )
)
```

When do we type check this?
**Templates (C++)**

- Idea: partially evaluate `sort(T,a)` for each distinct `T` used in program; type-check partially-evaluated code (instantiation)

  ```
  sort[T](a: array[T]) = 
  i,j:int = 1; 
  while (j < n) ( 
    e: T = a[j]; 
    i = j-1; 
    while (i >= 0 && a[i] > e) ( a[i+1] = a[i]; i-- ) 
    a[i+1] = e)
  `}

- But: get type-checking errors in library code
- Libraries must be shipped in source form

  ```
  int > int : ok
  int int
  int
  ```

**Parametricity**

- Can write code in way that doesn’t depend on `T` using 1st class function:

  ```
  sort[T: type](a: array[T], gt: function(T,T): bool) = ( 
  i,j:int = 1; 
  while (j < n) ( 
    e: T = a[j]; 
    i = j-1; 
    while (i >= 0 && gt(a[i], e)) ( a[i+1] = a[i]; i-- ) 
    a[i+1] = e)
  `}

- Now: code doesn’t depend on what `T` is: fully parametric w/ respect to `T`
- Can type-check and generate code once for all instantiations (ML)

**Type variables**

- Identifier considered a legal type if it appears in the environment

  ```
  α: type ∈ A
  A ⊢ α type
  ```

  No other typing rules mention types `α` explicitly
  - only rules mentioning `no specific types can apply`
  - ensures static checking works for all actual types used as parameters to code

**Code generation**

- Can generate same code for all instantiations of a parameterized abstraction if all types have the same size

  ```
  sort[T: type][a: array[T], gt: function(T,T): bool] = ( 
  i,j:int = 1; 
  while (j < n) ( 
    e: T = a[j]; 
    i = j-1; 
    while (i >= 0 && gt(a[i], e)) ( a[i+1] = a[i]; i-- ) 
    a[i+1] = e)
  `}

  Option 1: different code for `sort[int/long]`
  Option 2: auto box/unbox to ensure 1-word size

**Boxing/unboxing**

- Idea: some types `T` have two different run-time representations

  1. inline representation
  2. reference to heap
  - To support parametric polymorphism: large types (long, double, closures) boxed when used as parameters
  - To support primitive type `T < Object`, all primitive types boxed when cast to object
First-class vs. second-class polymorphic values

sort\[T: type\] (a: array\[T\], gt: function\(T, T\): bool)

- sort\[int\] has type array\[int\] \(\times\) (int\(\times\)int\(\rightarrow\)bool) \(\rightarrow\) unit
- sort is a polymorphic value, type \(\forall T.\) array\[int\] \(\times\) (T\(\times\)T\(\rightarrow\)bool) \(\rightarrow\) unit
- Most languages (incl. ML): polymorphic values are in environment but can only be instantiated
- First class polymorphic values: powerful, but can’t figure out all instantiations in advance! (must box)

Parameterized types

- sort\[T: type\] : parameterized function that works for any type \(T\)
- array\[T\]: parameterized type that can be constructed for any type \(T\)
  array: type \(\rightarrow\) type
- Can a language allow user-defined parameterized types?
- Useful for data structures: Set\[T\], Map\[K,V\], Stack\[T\], Vector\[T\], Hashtable\[K,V\]

Parameterized types vs. Instantiations

- Parameterized types (Vector) are not types; can’t have a value of type Vector (in PolyJ)
  Vector \(\in\) Vector\[\text{int}\]
- Instantiations (Vector\[\text{int}\]) are types

Using parameterized types

- class Vector\{void add(Object e); Object get(int i); Object set(int i, Object e);\} Vector v; Animal a; v.add(a); // unchecked a = (Animal)v.get();

- class Vector\[T\] {void add(T e); T get(int i); T set(int i, T e);} Vector[Animal] v; Animal a; v.add(a); a = v.get();

Subtyping relations

- What subtyping relations can hold for instantiation types? –depends on parameterized type
- class Vector\[T\] {void add(T e); T get(int i);} Elephant <: Animal
  Vector\[Elephant\] <: Vector\[Animal\]? Vector\[Animal\] <: Vector\[Elephant\]?
Applying subtyping rules

- Vector[Elephant] <: Vector[Animal]
  
  ```java
  Vector[Elephant] <: Vector[Animal]
  { void add(Elephant e); }
  { void add(Animal e); }
  }
  
  Nope: add

- Vector[Animal] <: Vector[Elephant]
  
  ```java
  Vector[Animal] <: Vector[Elephant]
  { void add(Animal e); }
  { void add(Elephant e); }
  }
  
  Nope: add

Rule:
- Subtyping on instantiation is covariant if type parameters appear only as return values
- is contravariant if appear only as arguments

Constrained parametric polymorphism

```java
class set[T] {
  T[] elements;
  boolean contains(e: T) {
    for (int i=0; i < elements.length(); i++) {
      if (e.equals(elements[i]))
        return true;
    }
    return false;
  }
}
```

- Set[T] doesn’t make sense unless T has a notion of equality
- SortedSet[T] : T must have a total ordering relation

Signature constraints

```java
SortedSet[T] where T { int compare(T); } {
  T[] elements;
  boolean contains(e: T) {
    for (int i=0; i < elements.length(); i++) {
      if (0 == e.compare(elements[i]))
        return true;
    }
    return false;
  }
}
```

- Constraint: contract between instantiator and parameterized code
- Can only be instantiated on types w/ compare
- SortedSet code only uses operations guaranteed to exist on any actual type used in instantiation

Subtype constraints?

```java
class SortedSet[T] where T <: Comparable {
  T[] elements;
  boolean contains(e: T) {
    for (int i=0; i < elements.length(); i++) {
      if (0 == e.compare(elements[i]))
        return true;
    }
    return false;
  }
}
```

```java
class Comparable {
  int compare(Comparable x);
}
```

Implementation

```java
SortedSet[T] where T { int compare(T); } {
  ... if (0 == e.compare(elements[i])) ... 
}
```

- How to generate code once for all instantiations?
- Problem: don’t know index of compare in DV of T
- Solution: separate dispatch vector for each instantiation of SortedSet[T]

Summary

- Overloading is not true polymorphism
- Parametric polymorphism helps write correct code conveniently
- Simple approach: templates. Breaks separate compilation, causes code bloat
- Unconstrained parametric polymorphism: simple to implement, may require boxing/unboxing
- Constrained parametric polymorphism: avoids more run-time errors, can be folded into dispatch vector