

## Principles of Programming Languages Handout

Simple typing is too inflexible for a real programming language. For instance, you would need to write a separate sorting function for each type of element. Hence more flexible type systems are important. Two important extensions of simple typing are subtyping and polymorphism.

### 1 Subtyping

Subtyping is what you have in OO languages. Here we have an order  $\leq$  (given by `extends` and `implements`), and we can infer

$$\frac{\vdash M : A}{\vdash M : A'} A \leq A'$$

An example of this is that any object can be cast to `Object`, since  $A \leq \text{Object}$  for all object types  $A$ . A slightly difficult point about subtyping is that it is contravariant in the operand position and covariant in the result position. That is, on functions we have

$$\frac{A' \leq A \quad B \leq B'}{(A \rightarrow B) \leq (A' \rightarrow B')}$$

As an example, consider the following code in Java. Here we have a class `S` with an identity operation, and a class `T` inheriting this operation from `S`. One might naively expect that this yields an identity operation on `T` as well.

```
class S
{
    public static S identity(S x) { return x; }
}

class T extends S { }

class Test
{
    T y = T.identity(new T()); // does not typecheck
    T z = (T)(T.identity(new T())); // this cast is needed
}
```

Class `T` inherits the identity from class `S`, but that does not give us an identity on `T`. We are trying to use a function with type  $S \rightarrow S$  as if it were of type  $T \rightarrow T$ . For the argument that works, since  $T \leq S$ , for the result it does not work, since we do not have  $S \leq T$ . Hence we have to cast the result, which is of type `S`, to `T`.

This need for casting is the reason that collection classes in Java are fundamentally broken: inserting an element into a collection is fine, but what you read back out of a collection needs to be downcast (tedious and unsafe).

### 2 Parametric polymorphism

Parametric polymorphism is based on type variables. For instance,

$$\vdash (\lambda x.x) : (\alpha \rightarrow \alpha)$$

The type variable  $\alpha$  can be instantiated to any type. Functional languages like ML and Haskell have polymorphic type systems in which the most general polymorphic type is inferred automatically. Here is an example in ML (`fn` means  $\lambda$ , the second line is the type inferred by the interactive ML compiler, where `'a` is a type variable):

```
- fn f => fn x => f(x + 1);  
val it = fn : (int -> 'a) -> int -> 'a
```

For example, the function `map` works with lists of any type. If you apply it to a list of integers, you get back a list of integers.

```
- map;  
val it = fn : ('a -> 'b) -> 'a list -> 'b list  
- map (fn x => x + 1) [1,2,3];  
val it = [2,3,4] : int list
```

ML is available in the department under Unix and Solaris: type `setup SML` and start the interaction by `sml`.

C++ has templates, which do a similar job to polymorphism in a messy way. Future versions of Java will contain generics (see current work on Generic Java).