CS321 Languages and Compiler Design I

Winter 2012

Lecture 15

Polymorphism

So far we've implicitly assumed that every value, identifier, expression, function, etc. has a unique type.

But often it makes sense for program entities to be used at many different types.

Some operations only work on one type of value:

- $(e_1$ `or` $e_2)$ makes sense only if e_1 and e_2 are `boolean`

Some operations work on any type of value:

- e`[42]` makes sense if e has type `array of` T for any type T.

Some operations work on a restricted set of values:

- $(e_1 < e_2)$ makes sense if e_1 and e_2 both have type `int` or both have type `float`

Terminology

A **monomorphic** operator works only one type of argument. (e.g. boolean operators).

A **polymorphic** operator works on multiple types of arguments.

- In **parametric polymorphism**, essentially the same implementation is used for all types of arguments (e.g. array indexing).
- In **ad-hoc polymorphism**, different implementations are used for different types of arguments (e.g. numeric comparison).
- **Subtype polymorphism** is a kind of ad-hoc polymorphism based on the idea that any operation on a given type will also work on its subtypes.

Subtype Polymorphism for Collections

```
abstract class Temp {
  double t;
  abstract boolean isCold();
}

class FTemp extends Temp {
  FTemp (double t) {this.t = t;}
    public String toString() {return "" + t + "F";}
  boolean isCold() {return t < 32.0;}
}

class CTemp extends Temp {
  CTemp (double t) {this.t = t;}
  public String toString() {return "" + t + "C";}
  boolean isCold() {return t < 0;}
}

class TList {
  Temp data;
  TList  next;
  TList(Temp data, TList next) { this.data = data;  this.next = next; }
}

class Test1 {
  public static void main (String argv[]) {
    TList tlist = null;
    tlist = new TList(new CTemp(-2.0),tlist);
    tlist = new TList(new FTemp(31.0),tlist);
    for (TList tl = tlist; tl != null; tl = tl.next)
      if (tl.data.isCold()) System.out.println("" + tl.data + " brr...");
  }
}
```

Limitations of Subtype Polymorphism

Subtype polymorphism works very well when we have a collection of different kinds of objects that can be accessed by the same **interface signature**.

• E.g., in Java, this means extending a single Class or implementing a single Interface.

• The static type of the collection elements is their common super-type.

• Different sub-classes may implement methods in completely different ways.

But there is a problem: when we extract an object from a collection, its static type is the common super-type.

• We lose (static) information about which particular class the object belongs to at runtime.

Example of code that will not typecheck...


```
abstract class Temp {
  double t;
  abstract boolean isCold();
}

class FTemp extends Temp {
  FTemp (double t) {this.t = t;}
  public String toString() {return "" + t + "F";}
  boolean isCold() {return t < 32.0;}
  CTemp toCelsius() {return new CTemp((t-32.0) * 5.0/9.0);}
}

class CTemp extends Temp {
  CTemp (double t) {this.t = t;}
  public String toString() {return "" + t + "C";}
  boolean isCold() {return t < 0;}
  FTemp toFahrenheit() {return new FTemp(9.0/5.0 * t + 32.0);}
}

class TList {
  Temp   data;
  TList  next;
  TList(Temp data, TList next) { this.data = data;  this.next = next; }
}

class Test2 {
  public static void main (String argv[]) {
    TList ftlist = null;
    ftlist = new TList(new FTemp(32.0),ftlist);
    ftlist = new TList(new FTemp(212.0),ftlist);
    for (TList tl = ftlist; tl != null; tl = tl.next)
      System.out.println ("" + tl.data + " = " + tl.data.toCelsius());

    TList ctlist = null;
    ctlist = new TList(new CTemp(0.0),ctlist);
    ctlist = new TList(new CTemp(100.0),ctlist);
    for (TList tl = ctlist; tl != null; tl = tl.next)
      System.out.println ("" + tl.data + " = " + tl.data.toFahrenheit());
  }
}
```


An Unpleasant Solution: Downcasting at Runtime

```
abstract class Temp {
  double t;
  abstract boolean isCold();
}

class FTemp extends Temp {
  FTemp (double t) {this.t = t;}
  public String toString() {return "" + t + "F";}
  boolean isCold() {return t < 32.0;}
  CTemp toCelsius() {return new CTemp((t-32.0) * 5.0/9.0);}
}

class CTemp extends Temp {
  CTemp (double t) {this.t = t;}
  public String toString() {return "" + t + "C";}
  boolean isCold() {return t < 0;}
  FTemp toFahrenheit() {return new FTemp(9.0/5.0 * t + 32.0);}
}

class TList {
  Temp   data;
  TList  next;
  TList(Temp data, TList next) { this.data = data;  this.next = next; }
}

class Test2a {
  public static void main (String argv[]) {
    TList ftlist = null;
    ftlist = new TList(new FTemp(32.0),ftlist);
    ftlist = new TList(new FTemp(212.0),ftlist);
    for (TList tl = ftlist; tl != null; tl = tl.next)
      System.out.println ("" + tl.data + " = " + ((FTemp)tl.data).toCelsius());

    TList ctlist = null;
    ctlist = new TList(new CTemp(0.0),ctlist);
    ctlist = new TList(new CTemp(100.0),ctlist);
```



```
    for (TList tl = ctlist; tl != null; tl = tl.next)
      System.out.println ("" + tl.data + " = " + ((CTemp)tl.data).toFahrenheit());
  }
}
```

Another Unpleasant Solution: Duplicating Code

```
abstract class Temp {
  double t;
  abstract boolean isCold();
}

class FTemp extends Temp {
  FTemp (double t) {this.t = t;}
  public String toString() {return "" + t + "F";}
  boolean isCold() {return t < 32.0;}
  CTemp toCelsius() {return new CTemp((t-32.0) * 5.0/9.0);}
}

class CTemp extends Temp {
  CTemp (double t) {this.t = t;}
  public String toString() {return "" + t + "C";}
  boolean isCold() {return t < 0;}
  FTemp toFahrenheit() {return new FTemp(9.0/5.0 * t + 32.0);}
}

class FTList {
  FTemp   data;
  FTList  next;
  FTList(FTemp data, FTList next) { this.data = data;  this.next = next; }
}

class CTList {
  CTemp   data;
  CTList  next;
  CTList(CTemp data, CTList next) { this.data = data;  this.next = next; }
}

class Test3 {
  public static void main (String argv[]) {
    FTList ftlist = null;
    ftlist = new FTList(new FTemp(32.0),ftlist);
    ftlist = new FTList(new FTemp(212.0),ftlist);
```



```
    for (FTList tl = ftlist; tl != null; tl = tl.next)
      System.out.println ("" + tl.data + " = " + tl.data.toCelsius());

    CTList ctlist = null;
    ctlist = new CTList(new CTemp(0.0),ctlist);
    ctlist = new CTList(new CTemp(100.0),ctlist);
    for (CTList tl = ctlist; tl != null; tl = tl.next)
      System.out.println ("" + tl.data + " = " + tl.data.toFahrenheit());
  }
}
```


Parametric Polymorphism

Downcasting is ugly, carries runtime cost, and may cause runtime failures.

Code duplication is ugly and error-prone.

A better approach is to observe that if every element of the collection has the same type, we should be able to track this fact **statically**.

Idea: can add **type parameters** to data type and function definitions, and **instantiate** them differently at each use.

Example...


```
abstract class Temp {
  double t;
  abstract boolean isCold();
}

class FTemp extends Temp {
  FTemp (double t) {this.t = t;}
  public String toString() {return "" + t + "F";}
  boolean isCold() {return t < 32.0;}
  CTemp toCelsius() {return new CTemp((t-32.0) * 5.0/9.0);}
}

class CTemp extends Temp {
  CTemp (double t) {this.t = t;}
  public String toString() {return "" + t + "C";}
  boolean isCold() {return t < 0;}
  FTemp toFahrenheit() {return new FTemp(9.0/5.0 * t + 32.0);}
}

class List<Elem> {
  Elem data;
  List<Elem>  next;
  List(Elem data, List<Elem> next) { this.data = data;  this.next = next; }
}

class Test4 {
  public static void main (String argv[]) {
    List<FTemp> ftlist = null;
    ftlist = new List<FTemp>(new FTemp(32.0),ftlist);
    ftlist = new List<FTemp>(new FTemp(212.0),ftlist);
    for (List<FTemp> tl = ftlist; tl != null; tl = tl.next)
      System.out.println ("" + tl.data + " = " + tl.data.toCelsius());

    List<CTemp> ctlist = null;
    ctlist = new List<CTemp>(new CTemp(0.0),ctlist);
    ctlist = new List<CTemp>(new CTemp(100.0),ctlist);
    for (List<CTemp> tl = ctlist; tl != null; tl = tl.next)
      System.out.println ("" + tl.data + " = " + tl.data.toFahrenheit());
  }
}
```

Parametric Polymorphism Is Widespread

Appears under many names:

- C++ templates
- Java and C# generics
- Ada generics
- Functional languages (ML, Haskell, etc.)

And with many implementation alternatives:

- Implicit or explicit instantiation.
- Separate copy of the code for each instantiation, or
- One copy of the code for all instantiations (requires a uniform data representation for all instantiable types).
- Reduction to subtype polymorphism + casting.

Need for Abstraction

Suppose we have a facility for defining new type names in our language and we type-check using name equivalence.

E.g., let's implement a **stack** using an array (in pseudo-fab syntax):

```
type stack := @integer;
var s := stack {100 of 0};
var top := 0;
func push(i:integer, s: stack) {
  s[top] := i;
  top := top + 1
}
```

Are named types like this "just as good" as the built-in types? Is a new type name a genuinely new type, equivalent to the built-in types?

No, if user of `stack` can **abuse** stack discipline, e.g.,

```
s[random] := 42;
```

• `stack`, `s`, `t`, `push`, etc. aren't grouped together. • Intended use of stack isn't explicit.

Abstraction for Built-in Types

Contrast this with the situation for built-in types with machine support.

For example, we don't normally write code like

```
if (x & 0x80000000) printf ("x is negative");
```

to inspect an integer. Instead we rely on built-in operators (like $<$) to interface to the underlying representation.

Can we do the same for user-defined types?

Abstract Data Types (ADT's)

Ideally, to mimic the behavior of built-in hardware-based types, user-defined types should have an associated set of **operators**, and it should only be possible to manipulate types via their operators (and maybe a few generic operators such as assignment or equality testing).

In particular, when new types are given a **representation** in terms of existing types, it shouldn't be possible for programs to inspect or change the fields of the representation.

Such a type is called an **abstract** data type (**ADT**), because to clients (users) of the type, its implementation is hidden; only its **interface** is known.

We can implement an ADT by combining a type definition together with a set of function operating on the type into a **module** (or **package**, **cluster**, **class**, etc.) Additional **hiding** features are needed to make the type's representation more-or-less invisible outside the module.

Abstraction

Compare to **procedural** abstraction: procedure can be **called** if its **type** is known, even if its implementation is not.

Benefits of abstraction:

- Implementation and client can be developed **independently**.
- Implementation can be **changed** without affecting client's code.
- Improves clarity, maintainability, etc.

Example: ADT for Environments (pseudo-fab)

```
signature env_sig {
  type env;
  var empty : env;
  func extend (e:env,s:string,i:integer) -> env;
  func lookup (e:env,s:string) -> integer
}

module env_mod : env_sig {
  record env = { id: string, val:integer, next : env };
  var empty : env := nil;
  func extend (e:env;s:string;i:integer) -> env {
    return env {id := s, val := i, next := e }
  };
  func lookup (e:env,s:string) -> integer {
    while e <> nil do
      if e.id = s then
        return e.val
      else e := e.next;
    return -1
  }
}
```


Example from Client's Side

Client code is restricted:

```
[* client *]
var x := env_mod.empty;
x := env_mod.extend(x,"abc",99);
env_mod.lookup(x,"def");

write (x.next.val);  [* NONO! *]
```

Thus, the **implementation** of the operations can be changed without affecting clients.

(The following implementation is not actually as general as the first one, but it still matches the signature.)

Alternative Implementation of Environment ADT

```
module env_mod2 : env_sig {
  record envr = { id: string; val:integer };
  type env = @envr;
  var empty : @envr := env { 100 of nil };
  func extend (e:env,s:string,i:integer) -> env {
    var c := 0;
    while (e[c] <> nil) do c := c + 1;
    e[c] := envr { id = s, val = i: };
    return e
  }
  func lookup (e:env,s:string) -> integer {
    var c := 0;
    var a := -1;
    while (e[c] <> nil) do {
      if (e[c].id = s) a := e[c].val;
       c := c + 1
    };
    return a
  }
}
```

Interface vs. Implementation

Ideally, the client of an ADT is not supposed to know or care about its internal **implementation** details – only about its exported **interface**. Thus, it makes sense to separate the **textual** description of the interface from that of the implementation, e.g., into separate files.

• Specifications give the names of types, and the names and types of functions in the package.

• Bodies give the definitions of the types and functions mentioned in the specification, and possibly additional private definitions.

One advantage of this separation is that clients of module `X` can be **compiled** on the basis of the information in the specification of `X`, without needing access to the the body of `X` (which might not even exist yet!)

But many languages, particularly in the C/C++ tradition, don't make this separation very cleanly.

Is Abstraction Always Desirable?

Although the idea of defining explicitly all the operators for a type makes good logical sense, it can get quite inconvenient.

Programmers are used to **assigning** values or passing them as **arguments** without worrying about their types. They may also expect to be able to **compare them**, at least for equality, without regard to type.

So most languages that support ADT's have built-in support for these basic operations, defined in a uniform way across all types – and sometimes also mechanisms for programmers to customize these.

But it is impossible for clients to generate code for operations that move or compare data without knowing the **size** and **layout** of the data. And these are characteristics of the type's **implementation**, not its interface. So these "universal" operations break the abstraction barrier around type and preventing separate compilation.

A common fix is to treat **all** abstract values as fixed-size pointers to heap-allocated values.

Modules in General

An ADT is one particular kind of **module**, containing:

• a single abstract type, with its representation;

• a collection of operators, with their implementations.

Instances of the ADT are typically created dynamically, and contain space for the components of the representation; all the instances share the same operator code.

More generally, modules might contain:

• multiple type definitions;

• arbitrary collections of functions (not necessarily abstract operators on the type);

• variables; • constants; • exceptions; etc.

Primary purpose is to **divide** large programs into (somewhat) independent sections, offering **separate namespaces** and perhaps **separate compilation**.

Interfaces

Even when a module does not represent a particular abstract data type, it usually represents a kind of **abstraction** over some set of facilities, in which some implementation information will be hidden behind an **interface**.

Clients of a module want to know **what** module does, not **how** it does it. Of course, specifying "what" is a hard problem! A key goal is that it should be possible to change the implementation without rewriting (or ideally, even recompiling) the client code that depends on the interface.

Most languages use **type** information to give a **partial** characterization of what a module does. An interface definition is then a collection of identifiers with their types.

In many languages it is possible to write and compile client code based solely on type interfaces. Of course, there must also be an (at least informal) specification of what the module's facilities **do**, and few languages provide any support for making sure that the implementations adhere to more than a type specification.