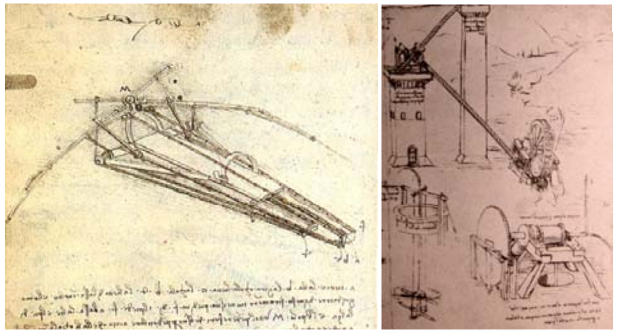
HY540 – Advanced Topics in Programming Language Development



Chapter 5 (three lectures)

Methods

- A method is a function associated to an object that is syntactically visible in the main block of the function through some kind of reserved symbol like *self* or *this*
- Methods are defined as part of the class model with a typed signature
- Methods may be overridden by subclasses in which case dispatching is dynamic
- In classless languages due to lack of compile-time information every invocation is late bound

Roadmap

- *Reification*: first-class concepts
- *Lambda*: anonymous methods
- *Closures*: persistent local state
- Typed world
  - Invariance, covariance and contravariance
  - Single, double and multiple dispatching
- Untyped world
  - Summarizing dispatch policies
  - Reifying the dispatch mechanism: editable vtables

Reification (1/2)

- Reification means to support the concrete handling of previously abstract, hidden or intrinsic items
- Informally it can be seen as the transformation of a second-class element to a first-class element
- Linguistically it can be translated as thing-making, that is turning something abstract into a concrete thing

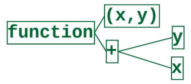
Reification (2/2)

- Pointer - C
  - reifies the low-level details of memory addresses
- Function - C / C++
  - reifies the low-level details of code addresses
- RTTI - Java, C#
  - reifies types
- *eval* function – Lisp, JavaScript, ActionScript
  - reifies the language interpreter
- *sizeof* - C/C++
  - reifies the details of memory size requirements

Lambda functions and methods (1/6)

- Supporting anonymous functions starts from the need to generate unique hidden names
  - only if functions must be named in the language
  - in some interpreted languages all function definitions are universally lambda
    - essentially carrying an AST of function type
    - so referencing requires assignment to a variable

```
f = lambda x y . x + y
f = (function(x,y) { return x + y; });
```



Lambda functions and methods (2/6)

- *A little of lambda calculus (1/2)*
  - $\lambda x.x \equiv$ **`function(x) { return x; }`**
    - lambda definitions accept only one argument
  - multiple arguments are provided by nested lambda definitions a technique known as *currying* (Curry)
  - $\lambda x.\lambda y.x+y \equiv$ **`function(x,y) { return x+y; }`**
  - the notion of invocation is named *application* in lambda calculus and requires parenthesis wherever ambiguity may arise
    - $(\lambda x.\ x * x)\ 9 \equiv$ **`(function(x){ return x*x; })(9)`**
    - same as $(\lambda x.\ x * x)\ (9)$

Lambda functions and methods (3/6)

- *A little of lambda calculus (2/2)*
  - Variables that are found in the argument list of a lambda expression are called *bound* and said to be in the scope of the lambda expression
    - x is a bound variable of $\lambda x.x$ and inside its scope
  - Variables not bound are called *free*
  - All variables bind to their nearest lambda
    - x is a free variable of $\lambda y.x+y$ inside $\lambda x.\underline{\lambda y.x+y}$
  - Lambda expressions with no free variables are said to be *closed*, e.g. $\lambda x.x$ is closed
    - they are analogous to functions referring only to their formal arguments
  - Clearly, *unclosed* lambda expressions will refer to free variables of outer lambda expressions

Lambda functions and methods (4/6)

- We need support recursive lambda functions
  - consider the lambda factorial function below
    `function(n) { return n * ` *what_we_put_here*`(n-1); })`
  - in Delta the keyword *lambda* is reserved for this purpose, so the previous code becomes now
    `function(n) { return n * lambda(n-1); }`
  - `lambda` is also a first-class value so the following code is legal (it implements application of unary functions)
    ```
    f = (function(g,x){g(x); return lambda; });
    f(print,1)(print,2)(print,3);
    ```
  - Interpreted languages will bind *lambda* to the root of the function AST
  - Compiled languages will bind *lambda* to the address of the anonymous function

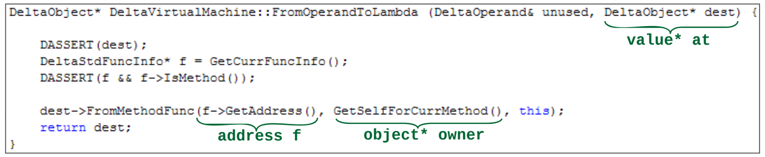
Lambda functions and methods (5/6)

- To support *lambda* methods more work is needed
  - only if methods can be called in the context of their owner without requiring syntactically the owner itself (as in in Python and Delta)
    - Given f method of a object and $ff = a.f \Rightarrow ff() \equiv a.f()$
  - then lambda should be a method value, not merely a function address, which can be handled as follows
    `void make_method (value* at, node* f, object* owner)`
    - *interpreted languages*

    `void make_method (value* at, address f, object* owner)`
    - *complied languages*
  - Thus *lambda* is evaluated by calling *make_method* using the currently called function (node or address) and the ***self*** object passed as the owner

Lambda functions and methods (6/6)

- In Delta all methods are anonymous
  - `method move(dx,dy){…}` is syntactic sugar for
  `{ “move” : (method(dx,dy){…})}`
- Although one may refer to *move* inside its body via `self.move` use of `lambda` is safer and generic
  - it is not affected by renaming
  - it allows a method to be copied to another object with a different index (key)



```
DeltaObject* DeltaVirtualMachine::FromOperandToLambda (DeltaOperand& unused, DeltaObject* dest) {

    DASSERT(dest);
    DeltaStdFuncInfo* f = GetCurrFuncInfo();
    DASSERT(f && f->IsMethod());

    dest->FromMethodFunc(f->GetAddress(), GetSelfForCurrMethod(), this);
    return dest;
}
```

Closures (1/6)

- Lets consider an unclosed function *f* whose address can be gained somehow to a global variable *a*
- In stack-based language implementations, calling *f* via *a* may result in undefined behavior
  - since the activation records of outer functions, whose bound variables are referred by *f*, are not on the stack at the time of the call

```
function f(x) {
        function g() { return x; }
        return g;
}
f10 = f(10);
```
What is the value of `f10()`*?*

$(\lambda x. (\lambda.x)) (10)( ) \equiv 10$

Lazy evaluation

Closures (2/6)

- To have safe execution of unclosed functions the environment, that is the actual arguments and the local data, of every outer function must be copied at the time an unclosed function is used
  - it should be attached to the unclosed function
  - this environment copy actually closes the function, thus the term closure

```
function bind1(f,x) {
       function g() { return f(x); }
       return g;
}
p10 = bind1(print,10);
```

$(\lambda f.\ \lambda x.(\lambda.\ f\ x))\ (print\ 10\ )(\ ) \equiv print(10)$

With closures, `p10()` *is* `print(10)` *Lazy evaluation for binders*

Closures (3/6)

- For closures to be supported
  - we treat function values as pairs of a function address and a snapshot of their environment (including their closure too)
  - the activation records (environments) must be now kept on the heap, not on the stack, and be reference counted (so that they can be collected automatically)
  - internally functions accept their environment as an explicit hidden parameter (no stack pointers for local access)
  - access to locals is redirected to the environment, not the stack pointers
  - environment management this way is slower to stack implementations due to the extra indirections and allocations
  - a reference to the environment of the enclosing function (closure, if any) is also included in the current environment to support access to non-locals

Closures (4/6)

Example program

```
function f(a,b) {
   function g(x,y) {
      print(a,b);
      function h() { print(a); }
      return h;
   }
   return g;
}
```

Closures (5/6)

•In the environment, the element with index 0 is a pointer to the environment of the enclosing function (can be null for global functions).
•Initially the environment of an invoked function is created by the caller to initially include its own environment and the actual arguments, while it is expanded inside the callee to include the local data.

```
function f (env* _ef, a, b) {
  expand_ef with the size of f local data (initially contains only f actual arguments)
  function g (env* _eg, x, y) {
     expand _eg with the size of g local data (initially contains only g actual arguments)
     print(
      _eg[0][1], // env::f then f::a (offset 1)
      _eg[0][2]  // env::f then f::b (offset 2)
     );
     function h (env* _eh) {
       expand _eh with the size of h local data (initially contains only h actual arguments)
       print(_eh[0][0][1]));    // env::g then env::f then f::a
     }
     return _func_(h, _eg);     // function value with enclosing env
   }
   return _func_(g, _ef);       // function value enclosing env
}
```

Closures (6/6)

•With closures we can easily implement objects (closure is object state and methods are inner functions accessing the saved and shared outer environment).
•Interestingly, compared to field dictionaries, the object state via closures is entirely opaque to clients!
•Popular languages with closures are Python, JavaScript, ActionScript and Lua.

```
function Circle(x,y,radius) {
   return {
      move : function(dx,dy) {x += dx; y += dy; },
      area : function() { return pi()*sqr(radius); }
   };
}
c1 = Circle(0,0,10);
c2 = Circle(100,-100, 20);
c1.move(1,1);
print(c2.area());
```

Invariance, covariance and contravariance (1/10)

- Lets consider a superclass *S* with an overridable method *f* having the signature *A f (B b, C c)* where *A*, *B* and *C* are classes in a language with reference semantics for objects
  - thus type *A* is a reference to an *A* object (for languages with stack objects reference or pointer qualifier is assumed)
- Assume any number of subclasses of *S* refining method *f*. Then we need to define the rules regarding:
  - The allowed return types when overriding *f*
  - The allowed argument types when overriding *f*
- We will try to induce the rules and then set what they formally imply

Invariance, covariance and contravariance (2/10)

- Lets focus on the return type. One may assume that the type system should be rigid and force an identical return type upon overriding
  - thus, in any *R* subclass of *S*, the refined *f* may only return a value of type *A*
  - that definitely guarantees type safety and substitutability
- Lets recall the substitutability principle
  - *Let **S** subtype of **T** and a function accepting arguments of type **T**. Then we may replace them with arguments of type **S** without having to alter any of the desirable properties of this function*

☞ Lets study what happens if the refined *f* in *R* is allowed to return a value of type *A′* being a subtype of *A*

Invariance, covariance and contravariance (3/10)

```
class S {
  overridable A f (B, C);
}
```

```
subclass A' of A {
  ...
}
```

```
subclass R of S {
  override A' f (B, C);
}
```

```
function g (x:S) {
  var a:A = s.f(...);
}
```

```
x:S = new S;
g(x); // type matching
```

```
y:R = new R;
var a:A = y.f(); // Safe, upcasting on A
g(y); // Call allowed, upcasting on S
```

- When the refined method returns a subtype of the return type of the base method, substitutability is preserved
  - since the refined method returns a value safely convertible to the return type of the base method

Invariance, covariance and contravariance (4/10)

```
subclass A'' of A' {
  ...
}
```

```
subclass Q of R {
  override A'' f (B, C);
}
```

```
Refinement order
A   S::f (B, C);
A'  R::f (B, C);
A'' Q::f (B, C);
```

```
Subclassing order
Q  <: R  <:  S
A'' <: A' <: A
```

- We observe that the return type of a refined method in a subclass can vary only in a way that it is a subclass of the return type of the base method
  - thus the return type of methods can vary with subtyping in a way similar to the subclasses hosting the refined methods
  - more formally, *the return types of refined methods should preserve the subclass ordering of the classes hosting them*
  - thus, they must vary with the same ordering, i.e. vary together, that is why this support of variant return types for overridable methods is coined the term *covariance on return types*

Invariance, covariance and contravariance (5/10)

- As expected, if the return type cannot change at all, that is cannot vary across subclasses, we have *invariance on return types*
  - clearly, the latter is always type safe
- Now, what if you try to refine a method so that it returns something “smaller”, in particular a supertype of the base method’s return type?
- Lets try formulate such a rule: *we allow refined methods to vary their return type only when it is a supertype of the base methods’ return type*

Invariance, covariance and contravariance (6/10)

```
class S {
  overridable A f (B, C);
}
```

```
subclass R of S {
  override A' f (B, C);
}
```

```
subclass Q of R {
  override A'' f (B, C);
}
```

```
Refinement order
A   S::f (B, C);
A'  R::f (B, C);
A'' Q::f (B, C);
```

```
class A'' {
  ...
}
```

```
subclass A' of A'' {
  ...
}
```

```
subclass A of A' {
  ...
}
```

```
Subclassing order
Q  <: R  <:  S
A'' :> A' :> A
```

```
y:R = new R;
var a:A = y.f(); // Unsafe, returns A'  supertype of A
g(y); // Call allowed, but internally unsafe
```

- When the return types vary with subtyping on the opposite direction of the method refinement ordering we call this *contravariance on return types*
- Since *it is unsafe*, this sort of contravariance is, as expected, avoided in programming languages

Invariance, covariance and contravariance (7/10)

- Now lets study if, and how, we may vary on the argument types of overridable methods upon refinement
- Again, *invariance on the argument types* is type safe and could constitute the default rule in a language
- So, lets firstly study whether we *covariance on argument types* is type safe as it proved to be for the return types

Invariance, covariance and contravariance (8/10)

```
class S {
  overridable A f (B, C);
}
```

```
subclass B' of B {
  ...
}
```

```
subclass R of S {
  override A f (B' , C);
}
```

```
function g (x:S) {
  var b:B = new B;
  x.f(b,...);
}
```

```
y:R = new R;
g(y);
x.f(b,...) ≡y.R::f(b,...)  Here, R::f expects b:B' but b:B with B' <: B inside g, so
the call is unsafe since no viable conversion is possible
```

- As shown in the example, covariance on argument types of overridable methods is unsafe, thus avoided
- Intuitively, although one is allowed to invoke an overridable method with a supertype reference and supply arguments of type implied in the method signature appearing in the supertype
  - the actually called refined method, due to late binding, may assume (expect) “larger” objects are supplied, which would lead to type clashes

Invariance, covariance and contravariance (9/10)

- Lets study the only remaining choice, if *contravariance on argument types* is type safe
- Intuitively, it means a refined method expects less from its arguments compared to what is declared in the base method signature
- Formally, it means the refined method’s argument types can be subtypes of the respective base method’s argument types
- Thus a value of a refined argument type is substitutable by a value of the respective base method argument type since the former is a subtype of the latter

Invariance, covariance and contravariance (10/10)

```
class S {
  overridable A f (B, C);
}
```

```
subclass B of B' {
  ...
}
```

```
subclass R of S {
  override A f (B' , C);
}
```

```
function g (x:S) {
  var b:B = new B;
  s.f(b,...);
}
```

```
y:R = new R;
g(y);
s.f(b,...) ≡y.R::f(b,...)  Here, R::f expects b:B' but b:B and B <: B' so the call is
safe since the conversion is automatic upcasting
```

- *Summary*
  - *covariance on return types* is safe
  - *contravariance on argument types* is safe
  - *invariance on both* is safe, but too rigid

Single, double and multiple dispatch (1/10)

- Single dispatch is when an invoked method is determined using *the actual type* of a single argument
  - the argument is usually intrinsic, not part of the signature, and is explicitly supplied upon invocation with a reserved syntax
  - `class A { virtual f (); };`
    - no *A* argument explicitly provided
  - `A a = new A; a.f();`
    - the *A* argument is supplied via `a.`
  - alternative method versions usually vary with respect the a subclass argument type
    - then they are called refined methods and are defined in the syntactic scope of the respective subclass
    - `class B : A { virtual f (); };`
      - an *f* version when the argument is *B*

Single, double and multiple dispatch (2/10)

- Single dispatch is not overloading as the latter dispatches by *the designated type* of an argument
  - lets say we use two keywords: ***overload*** and ***dispatch*** to distinguish among function overloading and dynamic dispatching (late binding)
  - the example below clarifies the difference and shows that we could have refined methods with an alternative syntax than subclass methods (assume *B subclass of A*)

```
var a = new A; var b = new B;
overload void f (a:A){} void f(b:B){}
f(a); f(b); a=b; f(a); all cases invoke the first version

dispatch void f (a:A){} void f(b:B){}
f(a); f(b); a=b; f(a); last case invokes the second version (late binding)
```

Single, double and multiple dispatch (3/10)

- And why is single dispatch rendered problematic? we will study an example and spot a potential software engineering issue

```
class Bonus {…};
class Weapon  : public Bonus {…};
class Health  : public Bonus {…};
class Key     : public Bonus {…};
class Creature {
 virtual void Claim (const Bonus&);
};
```

- Now, lets assume that different kinds of game creatures have different ways of claiming bonus subclasses such as:
  - *attackers* for *weapons*, *players* for *health*, *saboteurs* for *keys*

Single, double and multiple dispatch (4/10)

```
class Saboteur : public Creature {
 We should introduce argument-type specific versions
 void Claim_Health (const Health* b);
 void Claim_Weapon (const Weapon* b);
 void Claim_Key (const Key* b);

 virtual void Claim (const Bonus& b) {
   if (const Health* h = dynamic_cast<const Health*>(&b))
       Claim_Health(h);
   else
   if (const Weapon* w = dynamic_cast<const Weapon*>(&b))
       Claim_Weapon(w);
   else
   if (const Key* k = dynamic_cast<const Key*>(&b))
       Claim_Key(k);
   else
      throw CantClaimThisKindOfBonusException(b);
 }
};
```

The implementation requires resolve manually the actual type of the second argument and dispatch internally as necessary.

Single, double and multiple dispatch (5/10)

```
class Saboteur : public Creature {
 What about dispatching on multiple arguments directly ?
 void Claim (const Health& b);
 void Claim (const Weapon& b);
 void Claim (const Key& b);
};

..or with an alternative syntax one may have the following (global cope)
dispatch void Claim (Saboteur& b, const Health& b);
dispatch void Claim (Saboteur& b, const Weapon& b);
dispatch void Claim (Saboteur& b, const Key& b);

Creature c = …;
Bonus b = …;
Claim(*c, *b); is dynamically dispatched  on both arguments
```

This technique is called multiple dispatch and the methods dispatching on multiple arguments are called multi-methods

To solve the software dependencies of the previous style the language should allow decouple classes and require awareness of subclasses only inside methods. The latter can be done only if the methods are defined independently of classes.

Single, double and multiple dispatch (6/10)

Lets view a way to avoid the chain of if…else and the heavy-load dynamic_cast operator.

```
class Bonus  {
  virtual void ClaimedBy (Attacker&)   const = 0;
  virtual void ClaimedBy (GateKeeper&) const = 0;
  virtual void ClaimedBy (Saboteur&)   const = 0;
};
class Health : public Bonus {
  void ClaimedBy (Saboteur* s) const
    { similar code to previous Saboteur::Claim_Health implementation }
};
class Saboteur : public Creature {
 virtual void Claim (const Bonus& b)
   { b.ClaimedBy(*this); }
};

Bonus* b = new Health; Creature* c = new Saboteur;
c->Claim(*b);       is firstly dispatched as  c->Saboteur::Claim(b); calling
b.ClaimedBy(*this) which is dispatched as  b.Health::ClaimedBy(b);
```

☞*Because of the two dispatch steps involved this design pattern is called double dispatching*

Single, double and multiple dispatch (7/10)

- In summary, multi-methods are more general to typical overridable methods since the latter are dispatched on the caller argument only
- They are related to typed languages while they introduce extra non-trivial typing problems. For example:

class A;
class B : A;
class C;
class D : C;

***dispatch** f(A, D);*
***dispatch** f(B, C);*

B b = new B;
D d = new D;
f(b,d) → f(b:A (by upcasting),d:D) but also
f(b,d) → f(b:B,d:C (byupcasting))
Which one should be chosen by the compiler?

Single, double and multiple dispatch (8/10)

- The typing problem regarding multi-methods is that the compiler must provide an ordering defining the dispatch priority for the various versions
- As multiple arguments with distinct types are allowed, multiple class hierarchies are involved, thus so no global ordering criterion is possible
  - as shown in the previous example where we miss a rationale criterion to provide higher priority to either *A* or *C* hierarchies
- The problem is solved by defining a resolution order according to the argument position
  - usually a left-to-right ordering is adopted (CLOS, Dylan)
  - *a multi-method signature **B** is more specific to **A** when for an argument at position **j**: (a) the type in **B** is a subtype of the type in **A**; and (ii) left to **j** all argument types of **A** and **B** are identical*

Single, double and multiple dispatch (9/10)

class A;
class AA : A;
class AAA : AA;

class B;
class BB : B;
class BBB : BB;

class C;
class CC : C;
class CCC : CC;

$$\begin{tikzcd} \boxed{A} & \boxed{AA} \arrow[l] & \boxed{AAA} \arrow[l] & \boxed{B} & \boxed{BB} \arrow[l] & \boxed{BBB} \arrow[l] & \boxed{C} & \boxed{CC} \arrow[l] & \boxed{CCC} \arrow[l] \end{tikzcd}$$

***dispatch** f(A,*	*B,*	*C);*
***dispatch** f(AAA,*	*BB,*	*CCC);*
***dispatch** f(AA,*	*BBB,*	*C);*
***dispatch** f(A,*	*BBB,*	*C);*
***dispatch** f(AA,*	*BB,*	*CC);*
***dispatch** f(AAA,*	*BBB,*	*CCC);*
***dispatch** f(AA,*	*B,*	*CCC);*

a:A = new AAA;
b:B = new BB;
c:C = new CCC;
f(a,b,c) which method version is invoked?

Step I: order on A

f(AAA,	*BB,*	*CCC);*
f(AAA,	*BBB,*	*CCC);*
f(AA,	*BBB,*	*C);*
f(AA,	*BB,*	*CC);*
f(AA,	*B,*	*CCC);*
f(A,	*B,*	*C);*
f(A,	*BBB,*	*C);*

Single, double and multiple dispatch (10/10)

Step II: order on B

```
f(AAA,    BBB,    CCC);
f(AAA,    BB,     CCC);
f(AA,     BBB,    C);
f(AA,     BB,     CC);
f(AA,     B,      CCC);
f(A,      BBB,    C);
f(A,      B,      C);
```

Step II: order on C (no change)

```
f(AAA,    BBB,    CCC);
f(AAA,    BB,     CCC);
f(AA,     BBB,    C);
f(AA,     BB,     CC);
f(AA,     B,      CCC);
f(A,      BBB,    C);
f(A,      B,      C);
```

f(<u>AAA</u>, BB, CCC) is handled with three steps of dispatching (left to right) using vtable trees

1: [AAA] vtable

```
f(AAA,    BBB,    CCC);
f(AAA,    BB,     CCC);
f(AA,     BBB,    C);
f(AA,     BB,     CC);
f(AA,     B,      CCC);
f(A,      BBB,    C);
f(A,      B,      C);
```

2: [AAA][BB] vtable

```
f(AAA,    BB,     CCC);
f(AA,     BB,     CC);
f(AA,     B,      CCC);
f(A,      B,      C);
```

4: [AAA][BB][CCC][0] method

```
f(AAA,    BB,     CCC);
```

3: [AAA][BB][CCC] vtable

```
f(AAA,    BB,     CCC);
f(AA,     BB,     CC);
f(AA,     B,      CCC);
f(A,      B,      C);
```

Multiple dispatching on N arguments requires N indirections to detect the target vtable in the vtable tree

Reifying the dispatcher (1/7)

- In most object languages the field access operator is hidden and represents inaccessible underlying language mechanisms
- But interesting programming patterns arise if languages reify their dispatcher
  - meaning they allow programmers set their own function to map field keys to field storage
- Terms relating to dispatcher reification are *dot overloading* and *editable vtables*
- The technique is uncommon to class-based languages and is better suited to object-based languages

Reifying the dispatcher (2/7)

- The C++ language allows a limited form of reification with arrow overloading for objects and references (*but not for pointers!*)

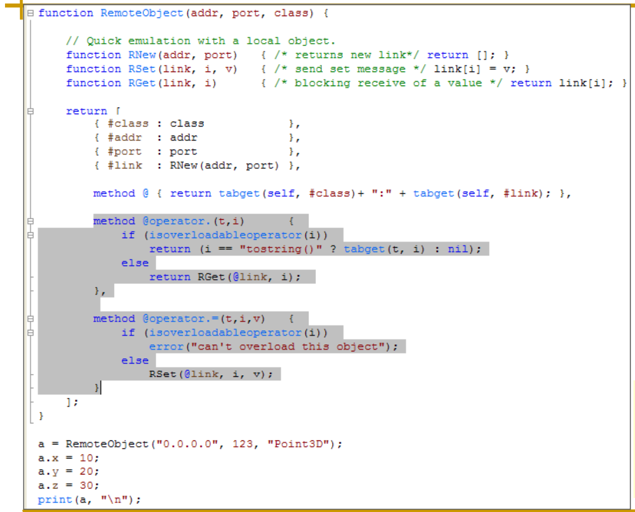
```
struct B { int x, y; };
struct A {
   B b;
   B* operator->(void) { return & b; }
};
void f (A& a) {
        a->x = 10;  } // Notice 'a' is not A*
        a.operator->()->x = 10; // Compile-time interpretation
}
```

- The previous feature is extensively used to allow intuitive syntactic access to the current element of an iterator in STL

Reifying the dispatcher (3/7)

- Reification of the dispatcher is more easy in untyped object-based languages where no compile-time field binding is applied
- Since everything is late bound one could support reification of the late binding function
- Normally this would apply to both read / write versions, thus two dispatchers are needed
- With a language reifying the dispatcher clients are allowed to implement their own object models and inheritance policies

Reifying the dispatcher (4/7)



```
function RemoteObject(addr, port, class) {

    // Quick emulation with a local object.
    function RNew(addr, port)    { /* returns new link*/ return []; }
    function RSet(link, i, v)    { /* send set message */ link[i] = v; }
    function RGet(link, i)       { /* blocking receive of a value */ return link[i]; }

    return [
        { #class : class            },
        { #addr  : addr             },
        { #port  : port             },
        { #link  : RNew(addr, port) },

        method @ { return tabget(self, #class)+ ":" + tabget(self, #link); },

        method @operator.(t,i)      {
            if (isoverloadableoperator(i))
                return (i == "tostring()" ? tabget(t, i) : nil);
            else
                return RGet(@link, i);
        },

        method @operator.=(t,i,v)   {
            if (isoverloadableoperator(i))
                error("can't overload this object");
            else
                RSet(@link, i, v);
        }
    ];
}

a = RemoteObject("0.0.0.0", 123, "Point3D");
a.x = 10;
a.y = 20;
a.z = 30;
print(a, "\n");
```

•*RemoteObject* is a constructor function producing an object whose implementation is at a remote side.
The vtable overloading, reader version as **@operator.** and writer version as **@operator.=**, makes remote binding entirely transparent to clients.

•The *RNew*, *RSet* and *RGet* inner functions are those to handle remote communication for object creation and editing.

•The vtable overloading in Delta works in cooperation to inheritance and delegation as it affects only local lookup to objects.

Reifying the dispatcher (5/7)

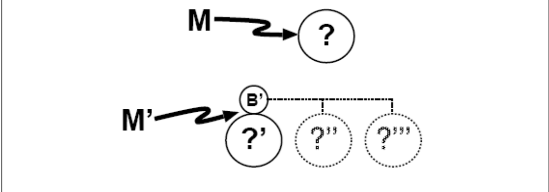


Figure 3. Minimal object model. An object is some unknown state *?* on which a method M can be invoked by name. To implement this model we need a mapping from method names to method implementations. So, to invoke a method M' in the object *?'* we find the corresponding method implementation in a behaviour description B'. Hence an object is a tuple of behaviour B' and state *?'*. Since behaviour is separate from the object it describes, it is possible to share any given behaviour B' between several distinct objects *?', ?'', ?''', …*

objects being pairs of state and sharable behavior actually committing method invocation requests

The runtime mapping of a method names to a method implementations is called *dispatcher* while the respective syntax is called *selector*.

Reifying the dispatcher (6/7)

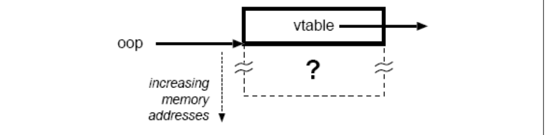


Figure 4. Implementation of minimal object. An object pointer (oop) points to the start of the object's internal state (if any). The object's behaviour is described by a *virtual table* (vtable). A pointer to the vtable is placed one word before the object's state.

an example object memory layout for dynamic vtables on the minimal object model (minimal = we pose no restriction on how state is represented – can be memory region or field dictionary)

Reifying the dispatcher (7/7)

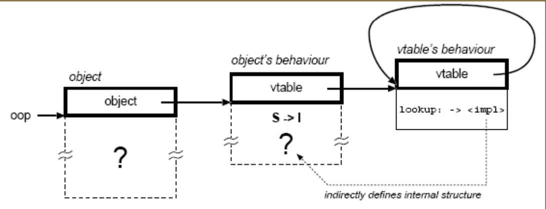


Figure 5. Everything is an object. Every object has a vtable that describes its behaviour. A method is looked up in a vtable by invoking its `lookup:` method. Hence there is a 'vtable vtable' that provides an implementation of `lookup:` for all vtables in the system, including for itself. The implementation of this `lookup:` method is the only thing in the object model that imposes internal structure on vtables. (If the figure seems confusing, try thinking of the word in the solid box as the 'type' of the object to which it is attached.)

an example object memory layout for dynamic vtables on the minimal object model (the basic vtable viewed as an object offering 'lookup' is for bootstrapping reasons)