T(C) = ClassT(C) = { Ccon: MethodsT(C) }

Another option:
class C
C new ObjProtT(C) = µ

Advanced Programming Languages

Java Constructors
class C extends D implements I {
    constructor C(x; t1) = D(e2); ... l1 = e1 ...
    public methods ... m1 = λx1:t1.e1 ...
    protected fields ... l2: τ2 ...
}

ObjProtT(C) = µC.[ ... l2: τ2 ... m2:τ2→τ′2 ... ]
ClassT(C) = { Ccon: MethodsT(C) } ...
MethodsT(C) = µC.[ ... m2:τ2→τ′2 ... ]

new C(e2) ⇒ C con([ ... this. m2 = λx2:τ2.e2 ... ], e1) C con= λthis.x2. D con(this, e2) with ... with this. l2 = e2

Typed object calculus

C++ constructors
class C extends D implements I {
    constructor C(x; t1) = D(e2); ... l1 = e1 ...
    public methods ... m1 = λx1:t1.e1 ...
    protected fields ... l2: τ2 ...
    protected fields ... l′2: τ′2 ...
}

ObjProtT(C) = µC.[ ... l2: τ2 ... m2:τ2→τ′2 ... ]
ClassT(C) = { Ccon: τ2→ObjProtT(C) }
MethodsT(C) = µC.[ ... m2:τ2→τ′2 ... ]

new C(e2) ⇒ C con([ ... this. m2 = λx2:τ2.e2 ... ], e1) C con= λthis.x2. D con(this, e2) with ... with this. l2 = e2

C++ constructors

Prototype-based languages

So far: class-based languages
  – Classes are second-class values, objects are first-class
  – Objects only produced via classes

Another option: object- or prototype-based languages (ala object calculus!)
  – No classes (can be simulated, as shown)
  – Can clone other objects, overriding fields
  – Examples: SELF, Cecil, object calculus

Prototype example

In untyped object calculus:

point = [p.movev = λd. p with q.x = p.x+d, q.y=p.y]
Inh_point = λP.x,y. (P with p.x = x, p.y=y)
Make_point = λx,y. Inh_point(point, x, y)
colored_point = point with cp.draw = ... cp.color...
Inh_cp = λP,x,y,c.(Inh_point(P,x,y) with p.color = c)
Make_cp = λx,y,c. Inh_cp(colored_point, x, y)

Inheritance without classes!
Multimethods

- Object provide possible extensibility at each method invocation o.m(a,b,c)
  - Different class for “o” permits different code to be substituted after the fact
  - Object dispatch selects correct code to run
  - Different classes for a, b, c have no effect on choice of code: not the method receiver
- Multimethods/generic functions (CLOS, Dylan, Cecil) : can dispatch on any argument

Shape example

```java
class Shape {
    boolean intersects(Shape s);
}
Class Triangle extends Shape {
    boolean intersects(Shape s) {
        typecase (s) {
            Box b => ... triangle/box code
            Triangle t => triangle/triangle code
            Circle c => triangle/circle code
        }
    }
}

Generic functions:
intersects(Box b, Triangle t) { triangle/box code }
intersects(Triangle t1, Triangle t2) { triangle/triangle }
intersects(Circle c, Triangle t) { Triangle/circle }
... extensible!
```

But... semantics difficult to define (what is scope of generic function? Ambiguities!), type-checking problematic

Polymorphic Types

- Have introduced a number of type constructors: →, +, { }, [ ], ref, array, ...
- Can think of type constructors as functions from types to types:
  → :: type → type, + :: type → type → type, &c.
- Can we allow the programmer to define their own type constructors? Is this useful?
- Yes: data structures

```java
shape

class Shape {
    boolean intersects(Shape s);
    }
Class Triangle extends Shape {
    boolean intersects(Shape s) {
        typecase (s) {
            Box b => ... triangle/box code
            Triangle t => triangle/triangle code
            Circle c => triangle/circle code
        }
    }
}

Generic functions:
intersects(Box b, Triangle t) { triangle/box code }
intersects(Triangle t1, Triangle t2) { triangle/triangle }
intersects(Circle c, Triangle t) { Triangle/circle }
... extensible!
```

But... semantics difficult to define (what is scope of generic function? Ambiguities!), type-checking problematic

PolyJ

- Java + parametric polymorphism, parameterized types:

```
interface Collection<T> {
    public boolean add(T x);
    public boolean contains(T x);
    public Iterator<T> iterator();
    public boolean remove(T x);
    ...}
```

- Collection: type → type
- Collection[int] : type
- HashSet[int] ≤ Set[int] ≤ Collection[int]

Kinds

- How to prevent ill-formed types like Collection[Collection]?
- Need to keep track of identifiers like Collection, Hashtable, etc. and keep track of their kind
- Fix:

```
K ∈ Kind ::= type | K → K
τ ::= X | B | τ₁ → τ₂ | τ₁ τ₂ | λX::K.τ
```

A copy of the lambda calculus “one level up”
### Types, Terms, Kinds

- **Types**: `type`, `type→type`  
- **Terms**: `λx:int.x`, `∀X.X→X`  
- **Kinds**: `λX::type.X→X`, `∀X.X→X`

### Fω

- **Kinds**: `K ::= type | K→K`  
- **Terms**: `τ ::= X | B | τ₁→τ₂ | λX::K.τ`  
- **Env**: `e ::= x | λx:τ.e | e₁ e₂`  
- **Context**: `Δ ::= ∅ | Δ, X::K`  
- **Glob**: `Γ ::= ∅ | Γ, x:τ`

### Type judgment

- **Type judgment**: `Δ;Γ / e:τ`  

### Kind judgment

- **Kind judgment**: `Δ / e:τ :: K`  

### Type equivalence

- **Type equivalence**: `Δ / e₁ ≅ e₂ :: K`

### Typing rules

- **Typing rules**:  
  - `Δ;Γ, x:τ / e₁:τ₁, e₂:τ₂ / e₁ e₂:τ`  

### Kinding rules

- **Kinding rules** (`Δ ⊢ τ :: K`):  
  - Just the `λ`-rules...

### Bounded type parameters

- **Bounded type parameters**:  
  - Hash table code must be able to compute hash value for values of type `K`: can’t apply `HashMap` to every type!  
  - Key type `K` okay if subtype of `interface Hashable { int hashCode(); }`:  
  - `K` is a **bounded** parameter: `ObjectT(HashMap) = λK≤Hashable::type.AV::type.{add: K*V→bool}`