Prototype-based languages

- So far, have discussed class-based languages
  - Classes are second-class values, objects are first-class
  - Objects only produced via classes
- Another option: object- or prototype-based languages
  - No classes (can be simulated)
  - Can clone other objects, overriding fields
  - Examples: SELF, Cecil

Object calculus

- Can explain semantics of OO languages more simply with more powerful construct than recursive records: object calculus
  - Abadi & Cardelli, Ch. 7-8
- New primitive object expression for object creation:

  \[
  \{ x_1.l_1 = e_1, ..., x_n.l_n = e_n \}
  \]

  - Idea: \( x_i \) stands for name of object (receiver/self) in expression \( e_i \) (implicit recursion)
  - Can extend object expression, automatically rebind recursion:

    \[
    \text{new}_{\text{point}}(x_x, y_y) = \{ s.x = x_x, s.y = y_y, \\
    s.\text{movex} = \lambda d:\text{int} . s + \{ r.x = s.x + d \} \}
    \]

Prototype example

In untyped object calculus:

- \( \text{point} = \{ \text{movex} = \lambda d . p + \{ q.x = p.x+d, q.y = p.y \} \}
- \( \text{constr}_{\text{point}} = \lambda p.x, y. p + \{ p.x = x, p.y = y \} \)
- \( \text{new}_{\text{point}} = \lambda x, y. \text{constr}_{\text{point}}(\text{point}) \)
- \( \text{colored}_{\text{point}} = \lambda \text{cp.draw} = \ldots, \text{cp.color} \ldots \)
- \( \text{new}_{\text{cp}} = \lambda x, y, c. \text{constr}_{\text{cp}}(\text{colored}_{\text{point}}, x, y, c) \)
- \( a_{\text{cp}} = \text{Make}_{\text{cp}}(10, 10, \text{red}) = \{ \text{movex} = \ldots, \text{cp.draw} = \ldots, \text{cp.color} = \text{red} \} \)

Inheritance without classes!
Methodology: template, traits superobjects

Typed object calculus

\[
\begin{align*}
e & ::= \ldots | x.i | o.e | o + \{ x.i = e' \} \\
v, o & ::= \{ x, l_i = e_i \} (n \geq 0) \\
\tau & ::= \ldots | \{ \{ \xi \}_{i=1..n} \} \quad \text{(object type)}
\end{align*}
\]

\[
\begin{align*}
o.l & \mapsto e_i(o/x_i) \\
\frac{o + \{ x.i = e \} \mapsto \{ x.l_i = e, x.l_j = e_i \ (i \neq j) \} \quad (j \in 1..n)}{\text{object type}}
\end{align*}
\]

\[
\begin{align*}
\Gamma & : x_1 : \tau_1 \cdots x_n : \tau_n \\
\Gamma & + o : \tau_o \\
\frac{o = \{ x_i.l_i = e_i \ (i \in 1..n) \} \quad (o = \{ x_i.l_i = e_i \ (i \in 1..n) \})}{\text{object type}}
\end{align*}
\]

Multimethods

- Object provide possible extensibility at each method invocation \( o.m(a, b, c) \)
  - Different class for “o” permits different code to be substituted (via subtyping)
  - Object dispatch selects correct code to run
  - Run-time types of 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
Intersecting shapes

```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 }

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

Parameterized Types

- Have introduced a number of type constructors:
  - `→, +, [], ref, array, ...
- Can think of type constructors as functions from types to types:
  - `→ :: type –→ type, ref :: type –→ type &c.
- Can we allow the programmer to define their own type constructors?
- Data structures:
  - `Hashtable[Key, Value]
  - `HashSet[int]

Implementation

```java
class HashSet[T] implements Collection[T] {
    private HashMap[T,T] m;
    public boolean add(T x) {…}
    public boolean contains(T x) {
        return m.containsKey(x);
    }
    public Iterator[T] iterator() {…}
    public boolean remove(T x) {…}
    ...
}
```

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`

`Fω`:

\[
K \in \text{Kind} :::= \text{type} | K \rightarrow K
\]

\[
\tau :::= X | B | \tau_1 \rightarrow \tau_2 | \tau_1 \tau_2 | \lambda X : \tau \tau
\]

—A copy of the lambda calculus “one level up” with `type` as the base kind

Types, Terms, Kinds

```
type               types
-------------------
type               kinds
int               int → int
\(\lambda X : \text{type} \rightarrow \text{type}\) int
\(\forall X : X \rightarrow X\)

terms
-------------------
\(\lambda x : \text{int} \cdot x\)
\(\Lambda X . \lambda x : X \cdot x\)
```

PolyJ

- Java + parametric polymorphism, parameterized types (also: GJ):
  - `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]`

---

---
**Fω**: Higher-order polymorphism

\[ \begin{align*}
K & ::= \text{type} \mid K \rightarrow K \\
\tau & ::= X \mid B \mid \tau_1 \rightarrow \tau_2 \mid \lambda X:K.\tau \\
e & ::= x \mid \lambda x:\tau.e \mid e_1 e_2 \\
\Delta & ::= \emptyset \mid \Delta, X:K \\
\Gamma & ::= \emptyset \mid \Gamma, x:\tau
\end{align*} \]

Typing rules

\[ \frac{\Delta \Gamma, x: \tau \vdash \Delta \Gamma \vdash e : \tau'}{\Delta \Gamma \vdash (\lambda x:\tau.e) : \tau \rightarrow \tau'} \]

Kinding rules

\[ (\Delta \vdash \tau :: K) \]

\[ \begin{align*}
\Delta, X:K & \vdash X:K \\
\tau & ::= X \mid B \mid \tau_1 \rightarrow \tau_2 \\
\Delta & ::= \emptyset \mid \Delta, X:K \\
\Gamma & ::= \emptyset \mid \Gamma, x:\tau
\end{align*} \]

Typing judgment: \[ \Delta; \Gamma \vdash e : \tau \]

Kinding judgment: \[ \Delta \vdash \tau :: K \]

Type equivalence: \[ \Delta \vdash \tau_1 \equiv \tau_2 :: K \]

Typing rules

\[ \Delta, \Delta' \vdash \Delta \Delta' : \tau \]

Kinding rules

\[ \Delta \vdash \lambda X:K.\tau :: K' \]

\[ \Delta \vdash \tau_1 :: K' \]

\[ \Delta \vdash \tau_2 :: K' \]

Many ways to produce same type... how to decide type equivalence?

Strong normalization: expansion will terminate!

**Bounded type parameters**

```java
class HashMap[K,V] implements Map[K,V] { 
    bool add(K key, V value) { int i = key.hashCode(); … } 
}
```

- 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) = \[ \lambda K \text{Hashable} : \tau.\lambda V : \text{type}.\mu \{ \text{add: } K^*V \rightarrow \text{bool},… \} \]

**Bounded polymorphism**

```java
class HashMap[K<:Comparable[K],V] implements Map[K,V]{
    static Hashmap() {…}
    bool add(K key, V value) { int i = key.hashCode(); … }
}
```

- Defines parameterized type ObjectT(HashMap): type of objects
- What is value of class object?
  - `\lambda K<:Comparable[K]::\tau.\lambda V::\text{type}.\mu \{ \text{add: } K^*V \rightarrow \text{bool},… \}`

Bounded polymorphism