Next week

- We'll look in more detail at some languages that make many of these points more concrete

Last week’s last slide:
we’ll look primarily at Smalltalk 80 and Cecil, since they cover many of the issues

Smalltalk-80: chalk talk

- Taken largely from Alan Borning

Cecil: Chambers et al. @ UW CSE

- Pure, object-oriented language
  - Classless object model
  - Type safe, garbage collected, implicit pointers, ...
  - Multi-methods
    - Dispatching on 0 or more arguments
  - Optional polymorphic static type checking
  - No type inference
  - More...

Procedures and variables

- Use method to define regular procedures
  - last expression is returned
  - can overload for different numbers of arguments

- Use let to define local and global variables
  - keyword var required for mutable variables
  - initialization required at declaration time

Example

```plaintext
let var count := 0;
method foo(a,b,c) {
  count := count + 1;
  let var d := a + b;
  let e := wuss(d,c);
  d := d + e;
  d + 5;
}
method wuss(x,y) { x – wuss(y) + 1 }
method wuss(x) { -x / 5 }
```
Closures

- Code bracketed in braces is a (no argument) closure
  - Evaluated only when invoked by eval
    - let closure := { factorial(10) + 5 };
    - eval(closure, 10) → 3628805

With arguments

let closure2 := &(n) {
    factorial(n) + 5
};
- eval(closure2, 10) → 3628805

Just like lambda, fn, \
  - anonymous, lexically scoped, largely first-class

Returning closures

- Cecil (at least a year ago) could not return closures out of their lexically enclosing scope
  - Not a language feature, but an implementation infrastructure problem
- Prevents currying, compose, closures stored in data structures, etc.

Closures for control structures

- Closures are naturally supportive of lazy control structures
  - if(test, {then_value},{else_value})
  - test1 & {test2}
  - while({test},{body})
  - for(start,stop,&(I){body})
  - do(array,&(elem){body});
  - fetch(table,key,{if_absent});
  - compare(I,j,{if_lt},{if_eq},{if_gt})

Example

method factorial (n) {
    if(n=0,
        {1},
        {n*factorial(n-1)})
}

Non-local returns

- Exit a method early with a non-local return from a nested closure
  - like return in C
  - like a limited continuation in Scheme
Example

```java
method fetch(table, key, if_absent) {
    table.do_associations(&k,v){
        if(k=key),{"v"};
    }
    eval(if_absent)
}
method fetch(table, key) {
    fetch(table, key,
        {error("key "|print_string(key)||
        "not found ")})
}
fetch(zips,"Seattle",{98195})
```

Objects, methods, fields

- To define a data structure, use `object`
  - To instantiate, use `object isa expression`
- To add methods to an object, specialize the method by adding `@Object` after the first (receiver) argument
- To add instance variable, use `field`

Example

```java
Object Point;
var field x(p@Point) := 0;
var field y(p@Point) := 0;
method area(p@Point) := {p.x*p.y};
method shift(p@Point, dx, dy) {
    p.x := p.x + dx; p.y := p.y + dy;
}
method new_point() {
    object isa Point
}
method new_point(x0,y0) {
    object isa Point {x := x0, y := y0}
}
```

Overloaded methods & dynamic dispatching

- Can overload methods in two ways
  - Same name, different number of arguments
  - Same name and number of arguments, with different specializer objects
- `method area(p@Point) {p.x*p.y}
  method area(c@Circle) {
      pi*square(c.radius)}`

Specializer overloading

- Specializer-based overloading resolved by using run-time class of received argument
  - i.e., dynamic dispatching
- `method print_area(x) {
    print(area(x));
}
let var p := new_point(3,4);
print_area(p);
new_circle(5);
print_area(p);`
Example

```c
var field x(p@Point) := 0;
method x(p@Point) { fetch p.x and return }
method set_x(p@Point,new_value) { update p.x and return }
set_x(p,p.x+1)
```

Inheritance

- Make new ADTs from old ones using `isa`
  - child/parent = subclass/superclass
  - inherit all method and field declarations
  - can add new fields and methods
    - specialized on child object
  - can override fields and method

Example

```c
Object ColorPoint isa Point;
var field color(p@ColorPoint);
method new_color_point(x0,y0,c0) {
  object isa ColorPoint {
    x := x0, y := y0, c := c0 }
  let p := new_color_point(3,4,"blue");
  print(p.color);
  p.shift(2,-2);
  print(p.x);
}
```

Overriding methods

- Parent and child can define overloaded methods
- If both apply to a call, the child’s takes precedence

```c
method draw(p@Point) {
  Display.plot_point(p.x,p.y)
}
method draw(p@ColorPoint) {
  Display.set_color(p.color);
  Display.plot_point(p.x,p.y)
}
let var p := new_point(3,4);
p.draw;
p := new_color_point(5,6,"red");
p.draw;
```

Resends

- When overriding method wants to invoke overridden method
- method draw(p@Point) {
  Display.plot_point(p.x,p.y);}
- method draw(p@ColorPoint) {
  Display.set_color(p.color);
  resend;
}

Overriding fields

- Unusual in OO languages, since the per-instance memory layout might change
- Since field accesses in Cecil are only through accessors, this is easier
  - Clients cannot tell what a message send to an accessor actually does
  - Efficiency?
- Override accessor methods with regular methods, and vice versa
Example

object Origin isa Point;
method x(@Origin) {0};
method y(@Origin) {0};

let p := ...; --Point or Origin
print(p.x); --how is x implemented?

classless object model

● With class-based object models
  – classes differ from objects
  – subclassing differs from instantiation
● Not in Cecil
  – Individual objects have their own
    implementation as part of the object
    • Methods are specialized on objects, not on
      classes
    • Objects with methods and fields specializing
      on them act like classes

More

● An individual object can inherit behavior from
  other objects
  – If there is no additional customization, then it acts like
    an instance
  – If there are new and/or overriding methods or fields,
    then it acts like a subclass
● Class-like objects can be used directly as
  instances
  – Ex: Origin object
  – Useful for constants, enumerated types, etc.
● Object creation expression instead of special
  constructors

Multiple dispatching: multi-methods

● Can specialize on more than the first argument
  method =(p1@Point,p2@Point) {
    p1.x = p2.x & {p1.y = p2.y} }
  method =(p1@ColorPoint,p2@ColorPoint) {
    resend & {p1.color = p2.color} }
  let x1 = new_point(...);
  let x2 = new_point(...);
  let y1 = new_color_point(...);
  let y2 = new_color_point(...);
  print(x1 = x2); print(x1 = y2);
  print(y1 = x2); print(y1 = y2);
Advantages of multi-methods

- Unity and generalize
  - top-level procedures (zero specialized arguments)
  - regular singly dispatched methods
  - overloading
    - dynamic, not static
- Naturally allow existing objects to be extended with new behavior

Disadvantages of multi-methods

- What’s the programming model?
  - (How do I decide when to do this and when not to?)
- What’s the encapsulation model?
- How to typecheck uses and definitions of multi-methods?
- How to implement efficiently?

Examples of multi-method uses

- Binary operations
  - Arguments drawn from an abstract domain with several possible implementations
    - equality over comparable types
    - < etc. comparison over ordered types
    - arithmetic over numbers
    - set operations (union, intersection, etc.)

More

- Cooperative operations over different types
  - display for different kinds of shapes on different kinds of output devices
    - standard implementation for each kind of shape
    - override with specific implementations for certain devices
  - operations taking flag constant objects, with different operations for different flags

Abstract objects

- Can introduce abstract objects whose sole purpose is to be inherited from
  - Not to be directly used or instantiated
  - May be only partially implemented
  - May call abstract methods that are required to be defined by concrete children

Example

```
abstract object Point;
  abstract method x(p@Point);
  abstract method y(p@Point);
  abstract method rho(p@Point);
  abstract method theta(p@Point);

method area(p@Point) { p.x * p.y }
method distance_to_origin(p@Point) { .. }
```
A concrete implementation

```plaintext
template object CartesianPoint isa Point;
field x(p@CartesianPoint) := 0;
field y(p@CartesianPoint) := 0;
field rho(p@CartesianPoint) { ... };
field theta(p@CartesianPoint) { ... };
method new_cartesian_point(x0,y0) {
  concrete object isa CartesianPoint {
    x := x0; y := y0 }
}
```

- Doesn’t reimplement the other methods (area, distance, etc.)

Another concrete implementation

```plaintext
template object PolarPoint isa Point;
field x(p@PolarPoint) { ... };
field y(p@PolarPoint) { ... };
field rho(p@PolarPoint) := 0;
field theta(p@PolarPoint) := 0
method new_polar_point(rho0,theta0) {
  concrete object isa PolarPoint {
    ... }
}
```

- Doesn’t reimplement the other methods (area, distance, etc.)

And then...

```plaintext
concrete object Origin isa Point;
method x(@Origin) { 0 }
method y(@Origin) { 0 }
method rho(@Origin) { 0 }
method theta(@Origin) { 0 }
```

- Doesn’t reimplement the other methods (area, distance, etc.)

Object roles

- **abstract**
  - potentially incomplete
  - can be inherited from
  - cannot be used directly or instantiated

- **template**
  - complete
  - can be inherited from
  - can be instantiated
  - not to be used directly

- **concrete**
  - complete
  - can be inherited from
  - can be instantiated
  - can be used directly

Multiple inheritance

- Can inherit from several parent objects
- Subclass gets union of all fields and methods inherited from parents
  ```plaintext
  object Shape;
  object Rectangle isa Shape;
  object Rhombus isa Shape;
  object Square isa Rectangle,Rhombus;
  ```

Ambiguities

- Can have ambiguities, just like with multi-methods in Cecil
  ```plaintext
  What if two parents define methods, neither of which overrides the other?
  ```
  ```plaintext
  object Rectangle isa Shape;
  method area(r@Rectangle) { ... }
  object Rhombus isa Shape;
  method area(r@Rhombus) { ... }
  object Square isa Rectangle,Rhombus;
  ```
  ```plaintext
  let s := new_square(4);
  ...area(s)... -- which method?
  ```
Can resolve

- ...by overriding method

```java
method area(s@Square) {
  resend(s@Rectangle)
}
```

Diamond-shaped inheritance

- How do we determine method in multiple inheritance if parent object is reachable in multiple ways?

```java
object Shape;
field center(s@Shape);
method is_shape(s@Shape) {...};
method is_rectangular(s@Shape) {...};
object Rectangle isa Shape;
method is_rectangular(r@Rectangle) {...};
object Rhombus isa Shape;
object Square isa Rectangle,Rhombus;
let s := new_square(3);
...is_shape(s)... is_rectangular(s)... center(s)
```

Multiple inheritance ambiguity

- Handled very differently in different languages
- Implicit resolution is common
  - Basically, build in a rule to resolve
  - CLOS linearizes the inheritance graph
    - Use whichever method you reach first
    - MOP control?
  - Python is similar, fixing a pre-order traversal of the inheritance graph

Explicit resolution

- The programmer is responsible for explicitly resolving any name clashes
  - Cecil
  - Eiffel
- If this isn’t done, the program isn’t correct

Ban name conflicts

- Ordering dependencies are often a source of problems in a language
- Explicit resolution places the burden of resolving names on the programmer
  - But may not avoid unanticipated, undesirable resolutions
Mixins

- Multiple inheritance has a nice idiomatic usage called *mixins*
  - Highly factored abstract objects
  - Generally independent axes
  - Each concrete object combines one mixin choice from each axis
- Examples axes in GUI
  - colored or not, bordered or not, titled or not
- In non-polymorphic languages, can use to create (for instance) doubly-linked lists of a given (atomic or user-defined) type

Example

```object CheckBox isa
  Square, ColorShape, BorderedShape, ShapeWithIcon, ClickableShape, ...;
```

Encapsulation

- In Cecil, there is no encapsulation associated directly with objects
- Modules are used to wrap a collection of declarations
  - Annotate those declarations with public (visible) or private (hidden)
  - Upon importing a module, only see public declarations

Example

```module PointMod {
  object Point;
  public get private set var field x(@Point);
  public get private set var field y(@Point);
  public method new_point(x0,y0) {
    ...
  }
  private method ...
}
```

When to inherit? (not Cecil-specific)

- Inheritance tends to work well when
  - subclass supports a superset of operations of superclass
    - Essentially, this is contravariance
  - subclass reuses much of the implementation of the superclass
  - subclass’ representation extends representation of superclass
  - subclass is a special kind of superclass
    - conceptually, subclass is-a superclass

Inheritance is inappropriate when

- A class has another class as a component
  - A point has-a coordinate but not a coordinate (is-a vs. has-a)
    - The interfaces aren’t related
    - Use a slot instead
- Only part of the other class’ implementation is reused
- Representation of other class needs to be altered
- When in doubt, don’t inherit!