Language Design
and
Overview of COOL

CS143 Lecture 2

(by Alex Aiken with minor edits by David Dill)

Grade Weights

- Project 50%
  - I, II 10% each
  - III, IV 15% each

- Midterm 15%

- Final 25%

- Written Assignments 10%
  - 2.5% each

Lecture Outline

- Today's topic: language design
- Why are there new languages?
- Good-language criteria
- History of ideas:
    - Abstraction
    - Types
    - Reuse
- Cool
- The Course Project

Programming Language Economics 101

- ## Languages are adopted to fill a void
    - Enable a previously difficult/impossible application
    - Orthogonal to language design quality (almost)

- ## Programmer training is the dominant cost
    - Languages with many users are replaced rarely
    - Popular languages become ossified
    - But easy to start in a new niche . . .

Why So Many Languages?

- Application domains have distinctive and conflicting needs

- Examples:

Topic: Language Design

- No universally accepted metrics for design

- Claim: "A good language is one people use"

Language Evaluation Criteria

Characteristic	Criteria		
	Readability	Writeability	Reliability
Simplicity	*	*	*
Data types	*	*	*
Syntax design	*	*	*
Abstraction		*	*
Expressivity		*	*
Type checking			*
Exception handling			*

History of Ideas: Abstraction

- Abstraction = detached from concrete details

- Abstraction necessary to build software systems

- Modes of abstraction
  - Via languages/compilers:
    - Higher-level code, few machine dependencies
  - Via subroutines
    - Abstract interface to behavior
  - Via modules
    - Export interfaces; hide implementation
  - Via abstract data types
    - Bundle data with its operations

History of Ideas: Types

- ## Originally, few types
  - FORTRAN: scalars, arrays
  - LISP: no static type distinctions

- ## Realization: Types help
  - Allow the programmer to express abstraction
  - Allow the compiler to check against many frequent errors
  - Sometimes to the point that programs are guaranteed "safe"

- ## More recently
  - Lots of interest in types
  - Experiments with various forms of parameterization
  - Best developed in functional programming

History of Ideas: Reuse

- Reuse = exploit common patterns in software systems
  - Goal: mass-produced software components
  - Reuse is difficult

- Two popular approaches
  - Type parameterization (List(int), List(double))
  - Classes and inheritance: C++ derived classes
  - Combined in C++, Java

- Inheritance allows
  - Specialization of existing abstraction
  - Extension, modification, hiding behavior

Trends

- ## Language design
  - Many new special-purpose languages
  - Popular languages to stay

- ## Compilers
  - More needed and more complex
  - Driven by increasing gap between
    - new languages
    - new architectures
  - Venerable and healthy area

Why Study Languages and Compilers ?

5. Increase capacity of expression

4. Improve understanding of program behavior

3. Increase ability to learn new languages

2. Learn to build a large and reliable system

1. See many basic CS concepts at work

Cool Overview

- Classroom Object Oriented Language

- Designed to
  - Be implementable in a short time
  - Give a taste of implementation of modern
    - Abstraction
    - Static typing
    - Reuse (inheritance)
    - Memory management
    - And more …

- But many things are left out

A Simple Example

```
class Point {
    x : Int ← 0;
    y : Int ← 0;
};
```

- Cool programs are sets of class definitions
  - A special class **Main** with a special method **main**
  - No separate notion of subroutine

- class = a collection of attributes and methods
- Instances of a class are objects

Cool Objects

```
class Point {
    x : Int ← 0;
    y : Int; (* use default value *)
};
```

- The expression "new Point" creates a new object of class Point

- An object can be thought of as a record with a slot for each attribute

x	y
0	0

Methods

- A class can also define methods for manipulating the attributes

```
class Point {
    x : Int ← 0;
    y : Int ← 0;
    movePoint(newx : Int, newy : Int): Point {
        { x ← newx;
          y ← newy;
          self;
        } -- close block expression
    }; -- close method
}; -- close class
```

- Methods can refer to the current object using self

Information Hiding in Cool

- Methods are global

- Attributes are local to a class
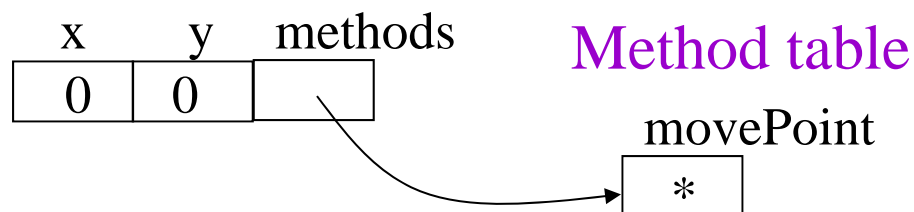 - They can only be accessed by the class's methods

- Example:

```
class Point {
    . . .
    x () : Int { x };
    setx (newx : Int) : Int { x ← newx };
};
```

Methods

- Each object knows how to access the code of a method
- As if the object contains a slot pointing to the code

x	y	movePoint
0	0	*

- In reality implementations save space by sharing these pointers among instances of the same class

x	y	methods
0	0	

Method table

movePoint
*

Inheritance

- We can extend points to colored points using subclassing => class hierarchy

```
class ColorPoint inherits Point {
   color : Int ← 0;
   movePoint(newx : Int, newy : Int): Point {
      { color ← 0;
      x ← newx; y ← newy;
      self;
      }
   };
};
```

x	y	color	movePoint
0	0	0	*

Cool Types

- Every class is a type
- Base classes:
  - Int   for integers
  - Bool   for boolean values: true, false
  - String   for strings
  - Object   root of the class hierarchy

- All variables must be declared
  - compiler infers types for expressions

Cool Type Checking

```
x : A;
x ← new B;
```

- Is well typed if *A* is an ancestor of *B* in the class hierarchy
  - Anywhere an *A* is expected a *B* can be used

- Type safety:
  - A well-typed program cannot result in runtime type errors
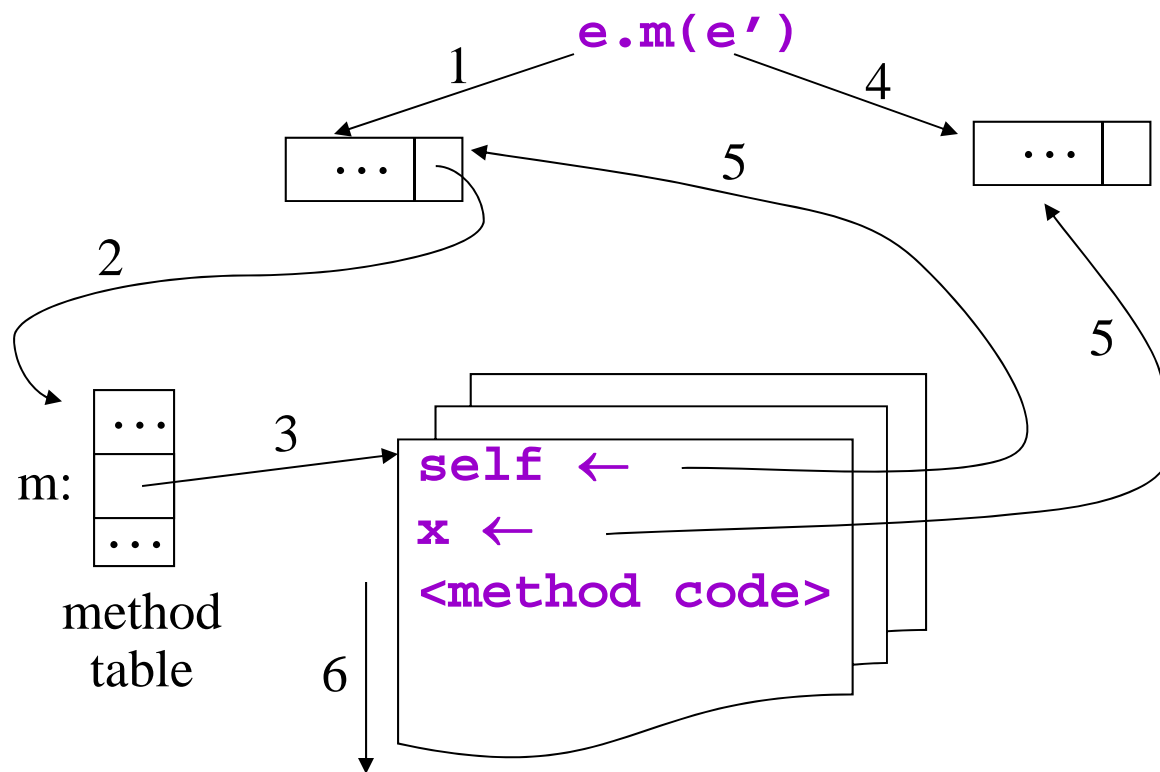
Method Invocation and Inheritance

- Methods are invoked by dispatch

- Understanding dispatch in the presence of inheritance is a subtle aspect of OO languages

```
p : Point;
p ← new ColorPoint;
p.movePoint(1,2);
```

- p has static type Point
- p has dynamic type ColorPoint
- p.movePoint must invoke the ColorPoint version

Method Invocation

- Example: invoke one-argument method m(x)

e.m(e')

1. Eval. e
2. Find class of e
3. Find code of m
4. Eval. argum.
5. Bind self and x
6. Run method

m:

method table

```
self ←
x ←
<method code>
```

Other Expressions

- ## Expression language
  - every expression has a type and a value
  - Loops: while E loop E pool
  - Conditionals   if E then E else E fi
  - Case statement   case E of x : Type $\Rightarrow$ E; ... esac
  - Arithmetic, logical operations
  - Assignment   x $\leftarrow$ E
  - Primitive I/O   out_string(s), in_string(), ...

- ## Missing features:
  - arrays, floating point operations, exceptions, ...

Cool Memory Management

- Memory is allocated every time new is invoked

- Memory is deallocated automatically when an object is not reachable anymore
  - Done by the garbage collector (GC)
  - There is a Cool GC

Course Project

- A complete compiler
  - Cool ==> MIPS assembly language
  - No optimizations

- Split in 4 programming assignments (PAs)

- There is adequate time to complete assignments
  - But <u>start early</u> and please follow directions

- Individual or team
  - max. 2 students