EEECS 490 – Lecture 15

Object-Oriented Programming

Announcements

- Project 3 due tomorrow at 8pm
- Midterm Tuesday 10/31 during class time
  - **Will be in 1109 FXB, not in this room**
  - Covers lectures 1-12
  - You are allowed one 8.5x11" note sheet, double sided
  - Review session: Sunday 10/29 2-4pm in 1690 BBB
- Mid-semester survey due Fri 11/3 at 8pm
  - http://survey2.eecs490.org

Review: Types and Type Judgments

- Our language has two types: Int and $Bool$
- We determine the type of a ***term*** in the program based on its syntactic form and the types of its subterms
- A ***typing relation*** or ***type judgment*** has the form

$$t : T$$

and it specifies that term t has type T

Review: Typing Rule

- Typing rules have the following familiar form:

Premise

$$\frac{t_1 \;:\; T_1 \quad \dots \quad \boxed{t_k \;:\; T_k}}{t \;:\; T}$$

Conclusion

Type Judgment

- This is a conditional rule that means:
  - **If** t_1 has type T_1, …, and **if** t_k has type T_k
  - **Then** t has type T

- This specifies a formula for computing the type of a term in a compiler
  - If the compiler sees a term of the form t, it can compute the type of t by computing the types of $t_1, \dots, t_k$ that are in the premises

Review: Subsumption Rule

- The ***subsumption rule*** allows a term to be typed as a supertype of its actual type:

$$\frac{\Gamma \vdash s : S \qquad S <: T}{\Gamma \vdash s : T}$$

- The rule encodes a notion of substitutability, allowing a subtype to be used where a supertype is expected:

$$\frac{\Gamma \vdash f : Float \to Float \qquad \dfrac{\Gamma \vdash x : Int \qquad Int <: Float}{\Gamma \vdash x : Float}}{\Gamma \vdash (f\ x) : Float}$$

Joins

- We need to rewrite the arithmetic rules to work with both $Ints$ and $Floats$
- The result type should be the ***least upper bound***, or ***join***, of the operand types
  - The join $T = T_1 \sqcup T_2$ is the minimal type T such that $T_1 <: T$ and $T_2 <: T$

$$Int = Int \sqcup Int$$
$$Float = Int \sqcup Float$$
$$Float = Float \sqcup Float$$

- Rule for addition:

Require operand type to be a number

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_1 <: Float \quad T_2 <: Float \quad T = T_1 \sqcup T_2}{\Gamma \vdash (t_1 + t_2) : T}$$

The Top Type

- Many languages have a Top type (also written as ⊤), that is a supertype of every other type:

$$S <: Top$$

  - Example: `object` in Python

- Adding Top to our language ensures that every pair of types has a join[1]

- We can then relax the rule for conditionals:

$$\frac{\Gamma \vdash t_1 : Bool \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3 \qquad T = T_2 \sqcup T_3}{\Gamma \vdash (\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3) : T}$$

[1]This is not necessarily true for other languages.

Contravariant Parameters

- A function that takes in a more general parameter type should be substitutable for a function that takes in a more specific parameter type

- For example, the following should be valid:

  $$((\textbf{lambda}\ f : Int \rightarrow Bool\ .\ (f\ 3))\ (\textbf{lambda}\ x : Float\ .\ \textbf{true}))$$

- Thus, if $T_1 <: S_1$, then it should be that $S_1 \rightarrow U <: T_1 \rightarrow U$

- This permits a ***contravariant*** parameter type, since the direction of <: is switched between the parameter and function types

Covariant Return Types

- A function that takes returns a more specific type should be substitutable for a function returns a more general type

- For example, the following should be valid:

  $$((\textbf{lambda}\ f : Int \to Float\ .\ (f\ 3))\ (\textbf{lambda}\ x : Int\ .\ x))$$

- Thus, if $S_2 <: T_2$, then it should be that $U \to S_2 <: U \to T_2$

- This permits a ***covariant*** return type, since the direction of <: is the same between the return and function types

Subtyping for Functions

- In general, a function is substitutable for another if the parameter types are contravariant and the return types are covariant:

$$\big(\big(\mathbf{lambda}\ f : Int \rightarrow Float\ .\ (f\ 3)\big)\ (\mathbf{lambda}\ x : Float\ .\ 0)\big)$$

- Rule for subtyping functions:

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

Data Abstraction

- Abstraction separates what something is from how it works
- ***Abstract data types (ADTs)*** separate the interface of a data type from its implementation
- ***Encapsulation*** is an important, though not universal, property of an ADT, binding the data the ADT represents along with the functions that operate on that data
- The course notes build a hierarchy of ADTs, beginning with immutable pairs all the way up to an abstraction similar to that provided by object-oriented programming
  - Required reading, like the rest of the notes!

Message Passing

- Higher-order functions can provide encapsulation in a functional ADT, allowing us to pass a ***message*** requesting a particular behavior
- A ***dispatch function*** takes the appropriate action given a message

```
>>> p = mutable_pair(3, 4)
>>> p('first')
3
>>> p('second')
4
>>> p('set_first', 5)
>>> p('set_second', 6)
>>> p('first')
5
>>> p('second')
6
```

```
def mutable_pair(x, y):
    def dispatch(message, value=None):
        nonlocal x, y
        if message == 'first':
            return x
        elif message == 'second':
            return y
        elif message == 'set_first':
            x = value
        elif message == 'set_second':
            y = value
    return dispatch
```

Dispatch Dictionaries

- A ***dispatch dictionary*** stores a mapping of messages to functions that perform the specified behavior
  - The dispatch function now just looks up a message in the dictionary and returns the corresponding function

Member variable

```
def account(initial_balance):
    ...
    dispatch = dictionary()
    dispatch('setitem', 'balance', initial_balance)
    dispatch('setitem', 'deposit', deposit)
    dispatch('setitem', 'withdraw', withdraw)
    dispatch('setitem', 'get_balance', get_balance)

    def dispatch_message(message):
        return dispatch('getitem', message)

    return dispatch_message
```

Object-Oriented Programming

- Object-oriented languages provide a systematic mechanism for defining abstract data types
- Fundamental features:
  - ***Encapsulation***: bundling together data of an ADT along with the functions that operate on the data
  - ***Information hiding***: restricting access to the implementation details of an ADT
  - ***Inheritance***: reusing code of an existing ADT when defining a new one
  - ***Subtype polymorphism***: using an instance of a derived ADT where a base ADT is expected
    - Requires some form of ***dynamic binding***, where the derived functionality is used at runtime

The term "encapsulation" is often used to encompass information hiding as well.

Terminology

- A ***class*** defines a pattern for the instances of an ADT
  - Specifies the data included and the functions that operate on that data
- An ***object*** is an instance of a class
- The individual data items and functions that comprise a class are its ***members***
- Data members are also called ***fields*** or ***attributes***
- Member functions are usually called ***methods***

```
struct Foo {
  int x;
  Foo(int x_);
  int bar(int y);
};
```

Field

Constructor

Method

Static Fields

- Each object has its own set of instance fields
- ***Static fields*** are associated with a class, and there is only one copy shared by all instances of the class
  - Can generally be accessed directly through class or indirectly through an instance
- Example in Java:

```
class Foo {
  static int bar = 3;
}

class Main {
  public static void main(String[] args) {
    System.out.println(Foo.bar);
    System.out.println(new Foo().bar);
  }
}
```

Access through class

Access through instance

Static Fields in C++

- Example:

```
struct Foo {
  static int bar;
};

int Foo::bar = 3;

int main() {
  cout << Foo::bar << endl;
  cout << Foo().bar << endl;
}
```

Out-of-line definition required to designate storage

Access through class uses scope-resolution operator

Access through instance uses dot operator

Static Fields in Python

- In Python, variables defined directly within the class definition are automatically static fields

```
class Foo:
    bar = 3

print(Foo.bar)
print(Foo().bar)
```

- Instance fields have to be defined through `self`

```
class Baz:
    def __init__(self):
        self.bar = 3
```

Access Control

- Information hiding requires ability to restrict access to members of a class
- Access modifiers, in languages that have them, allow the programmer to specify what code has access

	`public`	`private`	`protected`		`internal` in C#, Java default	Python
			C++, C#	Java		
Same instance	X	X	X	X	X	X
Same class	X	X	X	X	X	X
Derived classes	X		X	X		X
Code in same package	X			X	X	X
Global access	X					X

In Ruby, field access is restricted to the same instance.

Instance Methods

- Instance methods take in the instance on which to operate as a parameter
  - Often named `self` or `this`
  - Usually an implicit parameter
- Example in C++:

```
class Foo {
  int x;
public:
  Foo(int x_) : x(x_) {}
  int get_x() { return this->x; }
};

Foo f(3);
f.get_x();
```

this-> can be elided if x not hidden by local variable

Object that receives method call

Address of object implicitly passed as this

Methods in Python

- In Python, instance methods must take the instance as an explicit parameter
  - Named `self` by convention
- Example:

```
class Foo:
    def __init__(self, x):
        self.x = x
    def get_x(self):
        return self.x

f = foo(3)
f.get_x()
```

Object passed to first parameter

Cannot elide `self`.

Object that receives method call

Break Time!

- We'll start again in five minutes.

Static Methods

- ***Static methods*** do not operate on an instance, so they do not have access to instance members
- In many languages, the `static` keyword denotes a static method
- In Python, the `@staticmethod` decorator must be used to enable access through both a class and instance

```
class Baz:
    @staticmethod
    def name():
        return 'Baz'

print(Baz.name())
print(Baz().name())
```

Class Methods in Python

- Python also allows the definition of ***class methods***, which take in the class as an argument

```
class Baz:
    @classmethod
    def name(cls):
        return cls.__name__

class Fie(Baz):
    pass

print(Baz.name())      # prints Baz
print(Baz().name())    # prints Baz
print(Fie.name())      # prints Fie
print(Fie().name())    # prints Fie
```

Property Methods

- Some languages enable ***property methods*** to be defined, which have the syntax of field access but invoke methods
  - Abstract the interface of a field from its implementation

```
>>> c = Complex(1, math.sqrt(3))
>>> c.magnitude
2.0
>>> c.angle / math.pi
0.3333333333333333
```

- Example in Python:

```
class Complex(object):
    def __init__(self, real, imag):
        self.real, self.imag = real, imag

    @property
    def magnitude(self):
        return (self.real ** 2 +
                self.imag ** 2) ** 0.5

    @property
    def angle(self):
        return math.atan2(self.imag, self.real)
```

Property Setters

- In Python, the **`@property`** decorator only specifies a getter property method
- A setter can be defined with **`@<method>.setter`**, where **`<method>`** is the name of the property method

```
@magnitude.setter
def magnitude(self, mag):
    old_angle = self.angle
    self.real = mag * math.cos(old_angle)
    self.imag = mag * math.sin(old_angle)
```

```
>>> c.magnitude = math.sqrt(2)
>>> c.angle = math.pi / 4
>>> c.real
1.0000000000000002
>>> c.imag
1.0
```

Nested and Local Classes

- Many languages allow classes to be defined within another class or within a local scope
- Languages in which classes are first-class entities, such as Python, allow classes to be created dynamically
  - Generally have access to all variables in scope
- In other languages, such as C++, the primary purpose of a nested or local class is to limit the scope in which it may be used
  - In C++, a nested class has access to the private members of its enclosing class, but not vice versa
  - Local classes in C++ do not have access to local variables

Nested Classes in Java

- In Java, local classes have access to *effectively final* local variables
- Nested and local classes defined at non-static scope are associated with an instance of the enclosing class and have access to its members

```
class Outer {
  private int x;
  Outer(int x_) { x = x_; }
  class Inner {
    private int y;
    Inner(int y_) { y = y_; }
    int get() { return x + y; }
  }
}
```

```
Outer out = new Outer(3);
Outer.Inner inn = out.new Inner(4);
System.out.println(inn.get());
```

OOP and Message Passing

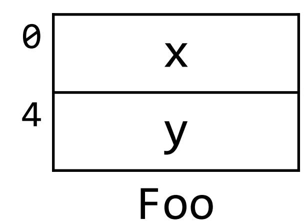
- Conceptually, object-oriented programming consists of passing messages to objects, which then respond to the message
  - Member access on an object can be thought of as sending a message to the object
- Languages differ in:
  - Whether the set of messages an object responds to (i.e. its members) is fixed at compile time
  - Whether the actual message to be sent to an object must be known at compile time

Record[1]-Based Implementation

- In languages that prioritize efficiency, the members of an object are known at compile time
- Fields of an object are stored directly within the memory of the object, at offsets that can be computed at compile time
- Field access can be translated by the compiler to an offset into the object

```
class Foo {
public:
  int x, y;
  Foo(int x_, int y_);
};

Foo f(3, 4);
cout << (f.x + f.y);
```



[1]Records are called *structs* in C++.

Dictionary-Based Implementation

- In languages that allow members to be added to an object at runtime, an object's members are usually stored in a dictionary
  - Similar to our message-passing implementation from the notes
- A well-defined lookup process specifies how to lookup a member
  - In Python, check instance dictionary first, then class

```
class Foo:
    y = 2
    def __init__(self, x):
        self.x = x

f = Foo(3)
print(f.x, f.y, Foo.y)   # prints 3 2 2
f.y = 4
print(f.x, f.y, Foo.y)   # prints 3 4 2
```

Adds binding to instance dictionary

Slots in Python

- Python actually takes a hybrid approach, using a dictionary by default but allowing a record-like representation as well

```
class Complex(object):
    __slots__ = ('real', 'imag')
    def __init__(self, real, imag):
        self.real, self.imag = real, imag

    @property
    def magnitude(self):
        return (self.real ** 2 +
                self.imag ** 2) ** 0.5

    @property
    def angle(self):
        return math.atan2(self.imag, self.real)
```

__slots__ used to specify fields in dictionary-less objects

Objects that are dictionary-less lose the ability to add instance attributes at runtime.

Dynamic Messages

- Dictionary-based languages generally provide a means for constructing and sending a message to an object at runtime

- Example in Python:

```
>>> x = [1, 2, 3]
>>> x.__getattribute__('append')(4)
>>> x
[1, 2, 3, 4]
```

Java Reflection

- In Java, the powerful *reflection* API allows inspection of classes and objects at runtime
- Reflection can be used to construct and invoke a dynamic message

```
import java.lang.reflect.Method;

class Main {
  public static void main(String[] args)
    throws Exception {
    String s = "Hello World";
    Method m =
      String.class.getMethod("length", null);
    System.out.println(m.invoke(s)); // prints 11
  }
}
```