CS345H: Programming Languages

Lecture 17: Introduction to Object-Oriented Languages

Thomas Dillig

Overview

- Today, we will study Object Oriented Programming
- Not in the sense of how to use it, but to understand the fundamental aspects of this paradigm
- We will also explore how to formalize some aspects of it

Subtyping

- Before we actually look at object-orientation, let's start with a more basic observation that precedes object oriented thinking
- If we have expression e_1 with type τ_1 and expression e_2 with type τ_2, we can sometimes safely use e_2 instead of e_1
- Example: $\tau_1 = Int \rightarrow Int$ and $\tau_2 = \alpha \rightarrow Int$
- We have seen this before: Polymorphism

Another Kind of Polymorphism

- However, there is also another kind of polymorphism that arises from records called subtype polymorphism:
- Consider the following two data types in Java:

```
class X {
  public int a;
};
class Y {
  public int a;
  public int b;
};
```

- Here, anyone who expects something of type X can work just as well with something of type Y
- Why? Because the fields of Y are a superset of the fields of X

Subtyping Polymorphism

- Intuitively, if B is a subtype of A, it has to be safe to use a B wherever an A is expected
- But the exact definition of subtype depends on the language!
- Observe that subtyping is really another kind of polymorphism as it allows us to write code that works with more than one type

Structural Suptyping

- One possible definition of subtype: B is a subtype of A if B has all fields A has
- This is called structural subtyping
- With this, there is no need to declare that one type is a supertype of another, it is just a direct function of the structure of a type
- This is also known as "duck typing"

Nominative Subtyping

- Contrasting approach: Programmer must explicitly declare subtype relationships
- Usually, this is combined with language mechanisms that ensure the resulting type is actually compatible
- This is know as nominative subtyping

Classes

- Next Key Idea using in object-oriented languages: Classes
- A class combines data with functions that operate on this data
- Crucially, the methods operating on data are bundled together with the data and can access the data
- You can view a class as a type that includes both data and functions to manipulate this data
- We create an instance of a class to use it
- Every instance has its own data

Classes Example

- Consider the following program:

```
struct point {
  int x;
  int y;
};
void inc_x(point* p) {
  p->x++;
}

point *p = new point;
inc_x(p);
```


Classes Example Cont.

- Here is the same program rewritten with classes:

```
class point {
  int x;
  int y;
  void inc_x() {
    this->x++;
  }
};

point *p = new point;
p->inc_x();
```

- Here, p is an object
- Object = instantiated class

Terminology of OO

- Class: A kind of type that combines data and functions to operate on this data
- Instance of a class: An object build from the class type
- Important: There is one class X in a program (even before running it), but potentially many instances of X at run-time
- Functions in classes are called methods
- Data fields are (sometimes) called instance variables
- Important: Every instance of a class has its own set of instance variables!
- Constructor: A special method that is run on instance creation

Classes Example Extended

- Here is a slightly extended version of the program

```
class point {
  int x;
  int y;
  point() { this->x = 0; this->y = 0; }
  void inc_x() {
    this->x++;
  }
};

point *p = new point();
point *q = new point();
p->inc_x();
```

- What is the value of y->x? 0

Objects and Subtyping

- Classes combine very powerfully with subtyping
- But subtyping now also needs to take methods into account!
- Specifically, any subtype must also have (at least) the same methods as the original type
- This way, we can use the subtype in any context that expects the original type

Virtual Methods and Subtyping

- But this has one problem. In general, we may need to change the implementation of an existing method when defining a subclass
- Allowing redefinition of methods is known as virtual methods
- This is another key OO feature

Virtual Method Example

- Consider the following code:

```
class point {
  int x;
  int y;
  point() { this->x = 0; this->y = 0; }
  virtual void inc_x() {
    this->x++;
  }
};
class bigpoint:public point {
  int x;
  int y;
  virtual void inc_x() {
    this->x+=2;
  }
};
point *p = new bigpoint();
p->inc_x();
```


Static vs. Dynamic Types

- Allowing subtyping of classes means that every expression in a program has two kinds of types:
  1. Static Type: This is the type computed by the type checker at compile time. For example, the static type of p in the example is `point`.
  2. Dynamic Type: This is the type of object a variable holds at run-time. For example, the dynamic type of p in the example is `bigpoint`.
- The dynamic type decides which virtual method is called!
- This is sometimes called the essence of OO

Relationship between Static and Dynamic Type

- The dynamic type of an object must always be a subtype of its static type in any well-typed program
- Observe that the semantics of a program with virtual methods require knowing the type of the object at run-time
- This is a new use of typing: So far, we only used types to prevent run-time errors
- In OOP, we also use types at run-time to decide which method to invoke!
- You can think of this operationally as an implicit run-time check on a type tag that decides which version of a method is called

Encapsulation

- The last major OOP feature we have not yet discussed in encapsulation
- This means that class data can be made non-accessible to clients of the class
- Example: Declaring an instance variable `private`
- Fortunately, this only rejects some programs at compile time but does not change semantics

Why Encapsulation?

- But why do we want to restrict access to object data?
- The idea is to force clients of a class to only rely on its public interface
- We are therefore free to change the implementation of classes without affecting its clients
- And we are free to pass any subtype with the same public interface
- Actually, old idea: This is also known as abstract data types (ADT) and predates OOP

Essential OOP Features

- The following four features are usually considered necessary for a language to be object-oriented:
  1. Subtyping
  2. Classes
  3. Virtual Methods
  4. Encapsulation

Why OOP?

- The allure of the OOP model is that it allows you to view your program as a collection of interacting entities (objects) instead of a collection of data and sets of functions
- This often allows for much cleaner and more extensible code
- If the problem you are solving fits into the OO model!

Uses of OOP

- OOP is a great fit for:
  - GUI toolkits
  - Data Structures
  - ....
- In fact, OOP fits so many problems reasonably well that it has become the default paradigm used in most software
- In fact, many modern languages, such as Java, force an OOP style
- However, not every problem maps well into objects!

History of OOP

- The idea of objects and subtyping originated at MIT in the 1950s and 1960s
- This was in the context of AI research in LISP
- Over the years, various features that we would call object-oriented today made their way into various LISP dialects

Object Orientation: History Cont.

- First object-oriented language: Simula 67
  - Developed by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo
  - Simula was designed as an special-purpose language for discrete event simulations
  - But it as certainly not designed as a general-purpose programming language

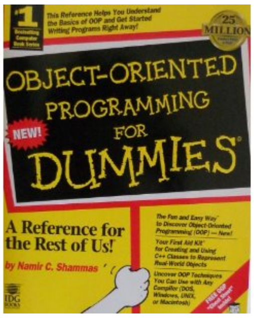
Features Combined in Simula

- Subtyping
- Classes
- virtual methods
- Garbage Collection

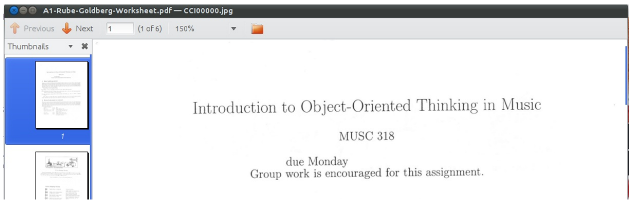
From Simlula to Smalltalk

- Simula was extremely successful in speeding up writing of discrete event simulations
- In fact, so successful that a group at Xerox PARC decided to create a general-purpose language based on this paradigm called Smalltalk
- Smalltalk coined the term "object oriented"
- And lead to a huge wave of OO languages
- This was an huge fad in the late 90's

OO in the Real World



OO in the Real World



Other OO languages

- C++
- Java
- C#
- Pretty much any newer imperative language (and plenty functional ones as well)

The OO paradigm in the wild

- Object oriented programming is one of the very few techniques that actually seems to make it easer for humans to build software
- But it does not work well for every problem
- However, since it is so pervasive, this tends to be forgotten

Summay

- We have looked at the four aspects that define object-oriented programming
- Next time: Some issues with semantics and typing in OO languages