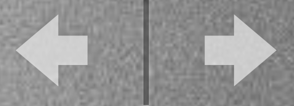
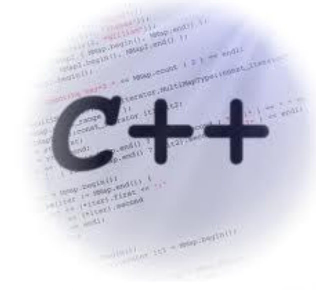
Microscopic Lens: Formalisms of OO Programming Languages

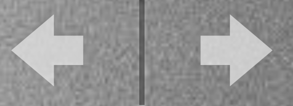
Mario Barrenechea
mario.barrenechea@colorado.edu



What does OO mean?

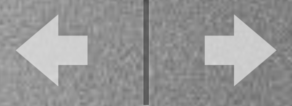
- *"Object-Oriented":*
  - begs several questions:
    - How do mainstream languages {Java, C++, Python, ...} support objects?
    - Why do we need to formalize OO languages? Is there even a need?
  - brings semantically unique changes "under the hood":
    - Let's look at these changes in detail.





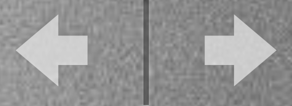
Formalisms?

- It's important to be formal about our ideas in order to prove them *sound* and *correct*.
  - "Soundness" *iff* all expressions w.r.t (with respect to) a feature F follow its inference rules truthfully.
  - "Correctness" *iff* all expressions that are true w.r.t F can be proved.
  - Otherwise, we wouldn't be 100% sure that the features we rely on should not be relied on!
  - We wish to do this for the **semantics** of our programming language. Both soundness and completeness are converses of each other, but together they allow us to add features in our languages with confidence.

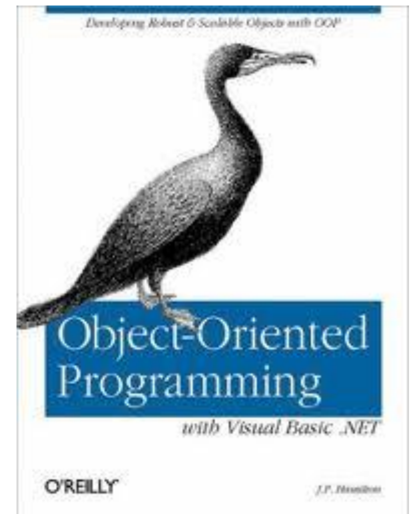


A Programming Language

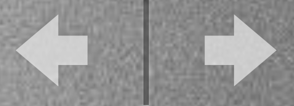
- A **programming language** is a well-defined mechanism for communicating higher-order instructions to a computer:
  - Every language has a **syntax** (grammar): In Backus-Naur Form (BNF), it can be expressed as a set of permissible tokens that can together express a program in that language:
    - Ex: `p ::= int | char | boolean | short | long | byte | double | float`
      - What values can p take on? What is p?
  - Every language is defined by its **semantics**. They convey the meaning behind expressions formed by the syntax of the language.
    - Three types of semantics studied: {Denotational, Operational, Axiomatic}
    - I won't define them, but most of this discussion revolves around the semantics of OO languages.



OO Features

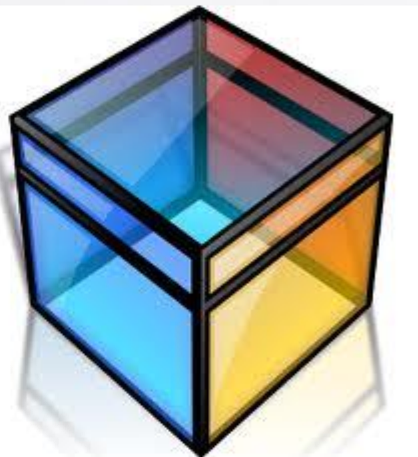


- From a high level, we take a lot of OO features for granted:
  - {Subtyping, Inheritance, Polymorphism, Generics, …}
  - Formalizing these ideas concretely can be a lot to swallow (and is obviously not exhaustive), but it enables us to see how these high-level OO language features are constructed.
  - We'll start by building some concepts of objects and their related features.
  - Then, we will examine **subtyping, polymoprhism,** and **inheritance** features in formal detail. We'll also draw from some relevant literature to reinforce our understanding of them.



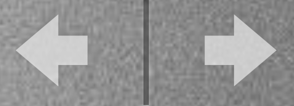
What is an Object?

- OOA&D: A representation of data with responsibility that exhibits a specific behavior!
    - A little more concretely: A data structure with representation (encapsulation) of internal state, access to that state via methods (functions), and a type. All of this specifies behavior.
    - A **type** is a parameterization to describe an object as a first-class entity within a program. If we want to pass an object around, we need to define its type.
        - **Type systems** are used within programming languages to define type hierarchies where OO functionality (i.e. polymorphism) is possible.



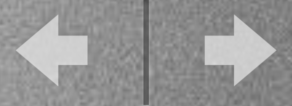
What happens when an object is casted to a wrong type during runtime?

- IllegalCastException in Java. However, if we didn't have these boundaries, languages like Java wouldn't be type safe!



An Abstraction for the Object

- Let's restrict our view of what an object is, which can be *represented in a number of different ways.*
- One way to model an object is to describe it as a **record** of attributes, which is a data structure with an aggregate type of all of its fields:
  - `R = {a := 5, b:= true, c:=` **λ**`x:Int.x} : {a: Int, b: Bool, c: Int->Int}`
- Generally, "a : T" denotes the variable "a" having the type "T".
- The attributes "a" and "b" are of type integer and boolean respectively, and "c" is a lambda function of type "Int -> Int", since it accepts an Integer argument and returns it.
- "Int -> Int" can be thought of as the type for the lambda expression **λ**x.x, which is an idempotent function ("identity function"). It simply returns its own argument.



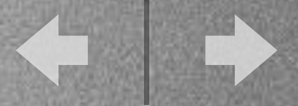
- So, with this object abstraction, we can do things with R just like we can do things with objects in Java, C++, or Python. For example:

```
>> Let R = {a:= 5, b:= True, c:= λx:Int.x} in

>> print R.a;

>> Let d = (R.c)(R.a) in

>> print d;
```

R ⟶

Attribute	Value
a	5
b	True
c	λx.x

- What does this program do? (don't worry about program syntax)
  - We create a record `R` with three attributes (as shown previously) and then print "a" to the output stream.
  - Then, we create a local variable "d" that is assigned to the **function application** (function call) of `R.c` with `R.a`, which will yield 5. Then, we print "d" to the output stream.

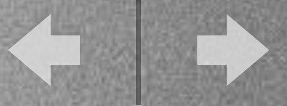


Functions

- What's this lambda function?
  - The lambda function is a mathematical notation used in the **lambda calculus** invented by Church in the 1920's to reason about computable expressions (i.e. functions). It is a Turing-complete language.
  - A lambda function (λx.t) is the basic **lambda abstraction**, which is the core value of the lambda calculus grammar:

t ::=	x	**variable**
	| t t	**application**
	| λx.t	**abstraction**

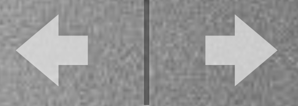
  - The lambda calculus is like a "computational substrate" that allows us to reason about functions in our programming languages [4].



- Lambda functions can be assigned to variables, assuming we have function references:

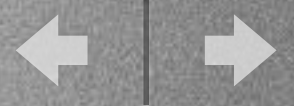
```
>> Let R.c = λn . If (n == 0) return 1 else return n *
factorial (n - 1) in

>> print (R.c) (R.a);
```

- What does this program do?
  - We access the field “c” within our record object **R** and assign it to have the value of a new lambda function, which computes the factorial of the input argument n.
  - In this case, we know from the previous example that `R.a = 5`, so the function application of `(R.c)(R.a)` can be expressed as `(λn . Factorial) (R.a)`, which returns factorial (5) = 120. We print the answer to the output stream.



Objects vs. Records

- We treat our object R like a record, but objects and records are subtly different.
  - Records are compound data structures that hold fields and functions (as we have seen with the lambda calculus). They are accessible and modifiable, and objects are essentially implementations of a record. Think of records as C-style structs.
  - But what makes an object different than a record?
    - An object is constructed from a **class**, a blueprint that defines all of its instance variables and methods that are accessible in the scope where the object was constructed. Records are not constructed from a class.
    - Records do not have a uniform type. Instead, their type is constructed from the sum of its parts (ex. `R={a:int, b:Bool, c: Int -> Int}`). Behind the scenes, objects mimic types like records [5], but in practice their class defines their type.
    - Furthermore, object attributes are accessed at runtime using a lookup algorithm called **dynamic dispatching**. It maps the message call (such as R.c) with the actual attribute found within the object and returns it, a technique that cannot be done at compile-time. Therefore, it is a run-time mapping.

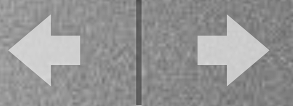


Dynamic Dispatching

- Dynamic Dispatching allows for **polymorphic** behavior, when an object B extends A (B : A), and a call is made to a function in B's class that is implemented in A.

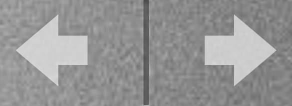
```
newA = λ_:Unit . let x = ref in {doFunctionA() := (λ_:Unit.return 1)};

newB = λ_:Unit . let x = ref in {doFunctionA() := (λ_:Unit.return 1),
    doFunctionB() := (λ_:Unit.return 2)};

>> let b = newB in print b.doFunctionA();
```

- This shows that we initialize an object "b" of type B (b: B) via a lambda function that generates objects of type B (called a **constructor**). Then, with "b", we call a method found in B's parent class A. The print command will print out "1" to the output stream.
- The underscore ("_") accounts for any arbitrary input parameter of type `Unit`. Since our constructor functions `newA` and `newB` don't account for any input parameters, we don't have to care about them.



Subtyping

- Since the `newB` constructor extends the `newA` constructor by including a `doFunctionB()`, we can say that `B <: A`. In other words, B is a **subtype** of A.
- **Subtyping** (also known as polymorphism) is an OO feature that allows the programmer to instantiate types that can be upcasted to higher types. This gives power to the programmer to treat in the same way instances of classes that may exhibit different behavior because their types vary along the type hierarchy.
- In our previous example, we had our object b : B call a function in its parent class A because B <: A. Subtyping allows the safe handling of type casting of B to A because the type system recognized that the types did in fact match!
- In Java, our object hierarchy defines the type hierarchy of all types. The top of the hierarchy is the `Object`, and the bottom is `void`. In C++, there is no type hierarchy. How about in Python?



More familiar ways to abstract objects?

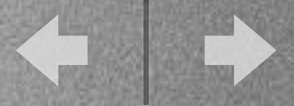
- In [3], Felleisen and Flatt describe a subset of the Java language called MiniJava, which makes it easier to understand how OO features like subtyping and inheritance are implemented.
- Suppose we wanted to create a class for Fish and an extension of that class called ColoredFish. The class structure is represented in the next slide.

`P = c c .... M`	**Program**
`c = class c extends c {f f ... m m}`	**Class Declaration**
`f = T f = V`	**Field Declaration**
`m = T m (T X, ... ,T X) { M }`	**Method Declaration**

`c =` A class name or `Object`, `f =` A field name, `m =` A method name, `X =` A variable name or `this`, `T =` A type, `M =` An expression, such as `X`, `null`, `M:c.f` (field access), `M:c.M (M,… M)` (method call), `(c)M` (cast), etc…

- **MiniJava Syntax:**

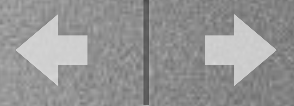
```
class Fish extends Object {
    num size = 1
    num getWeight(){this:Fish.size}
    num grow(num a){this:Fish.size := (+ a this:Fish.size)}
    num eat(Fish f){this:Fish.grow(f:Fish.getWeight())}
}//end class
Class ColoredFish extends Fish {
    num color = 7
    num getWeight(){* super:Fish.getWeight() this:ColoredFish.color}
}//end class
```



- A Program P in MiniJava consists of class declarations `c1, c2, c3, … cN`, followed by program expressions to be run. There is no main method in MiniJava.

```
Fish f  = new Fish();

ColoredFish coloredF = new ColoredFish();

F.eat(coloredF);
```

- After the class declarations, we instantiate two new variables `f` and `coloredF` of type `Fish` and `ColoredFish`, and then we call `f`'s method `eat()`, which accepts a `Fish` type and thus increments the size of the underlying `Fish` (`f`).
- **Subtyping** is precisely shown here; MiniJava accepted the `coloredF : ColoredFish` into the `eat()` method because `coloredF : ColoredFish <: Fish`. **Inheritance** allows the reuse of superclass functionality when `getWeight()` was called from the `ColoredFish` class. **Method overriding** was shown when `ColoredFish` implemented `getWeight()` again.



MiniJava Rules

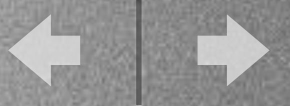
- Among some of the properties that are proved sound in [4], here are a few that are straightforward:

$$ClassesOnce\ (P)\ iff$$
$$(c_1, c_2, \ldots, c_n\ \in P) \rightarrow c_i\ \neq c_j$$

$$ObjectBuiltIn\ (P)\ iff$$
$$class\ \texttt{Object} \notin P$$

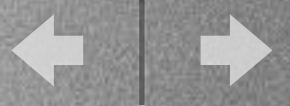
$$ClassesDefined\ (P)\ iff$$
$$(c\ \in P) \rightarrow c = \texttt{Object}\ or\ (class\ c\ \in P)$$

- The ClassesOnce property shows that for all classes defined in a program P, there should not be any duplicate classes.
- ObjectBuiltIn is the property that ensures the class Object is not defined within P. Object is a built-in type in MiniJava, so there's no need to define it again.
- ClassesDefined is the property ensuring that, for every class defined in P, that is either the `Object` class or is in P.



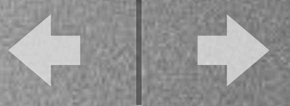
Conclusion

- To wrap things up:
  - We learned the importance of formalizing our programming languages.
  - We looked a little deeper at what a programming language is (syntax and semantics).
  - We have a better idea of what an object is, and how it’s different than a record.
  - We learned a little bit about functions and how they are formally represented with the lambda calculus.
  - We saw some neat programs that utilized these features and demonstrated several OO concepts like subtyping and object construction.
  - We learned about dynamic dispatching, and how polymorphism can be shown through types.
  - We looked at a micro-language called MiniJava that demonstrated inheritance and subtyping features. We also examined only a subset of rules that are important to verify for MiniJava.



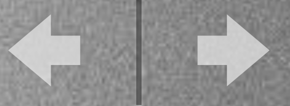
Final Notes

- Hopefully, this discussion has informed you of the importance in formalizing our OO programming languages.
  - Programming Languages are designed and implemented all of the time, always pushing the envelope for developer usability and more advanced features.
  - Imagine proving properties (like shown in MiniJava) for a giant system (10 KLOC). In [2], Andronick did this for a microkernel written in Haskell!
  - One of the big debates in programming languages is between formal verification of software and software testing. On one hand, formal verification will ensure that the software is 100% correct, but it gets extremely hard to verify large systems. On the other hand, with testing, there are tools that help us do it with ease, you never really know when you’re done…



But wait, there’s more!

- There’s just so much more to discover about OO programming languages and their formal details.
    - Abadi and Cardelli [1] have developed the object calculus to formally explain objects and classes in their own terms.
    - Remy [5] extends the object calculus with a more flexible type structure for objects, but it’s a good reference for understanding the difference between objects and records.
    - Harper [4] gives a very well-rounded overview of programming languages, type theory, lambda calculus, classes and objects, and much more.



References

[1] Abadi, M., & Cardelli, L. (1996). A Theory of Objects Object-Oriented Features. *Systems Research.*

[2] Andronick, J. (2011). From a proven correct microkernel to trustworthy large systems. *Formal Verification of Object-Oriented Software*, 1–9. Springer. Retrieved from http://www.springerlink.com/index/A131647671562717.pdf

[3] Felleisen, M. (2006). Programming languages and lambda calculi. *lecture notes. http://www. ccs. neu. Retrieved from http://www.math.northwestern.edu/~richter/mono.pdf*

[4] Harper, R. (2009). Practical foundations for programming languages. *Draft of February*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.132.5853&rep=rep1&type=pf

[5] Rémy, D. (1998). From classes to objects via subtyping. *Programming Languages and Systems, 200–220. Springer. Retrieved from http://www.springerlink.com/index/1776127254617303.pdf*

- Wikipedia, only for general overview and confirmation of understanding.