Now it's time to use these tools for something more ambitious. We're spending the semester developing tools for defining and reasoning about a variety of programming language features. Now it's time to use these tools for something more ambitious.

A Change of Pace

Onto Objects

Plans

Final exam: Wednesday, December 14th
Next week: Chapter 19/Review
This week: Chapter 18/19

28 November

Fall 2005
Software Foundations
CIS 500
Plan:
1. Identify some characteristic "core features" of object-oriented programming
2. Develop two different analyses of these features
   a) Translation into a lower-level language
   b) Direct, high-level formalization of a simple object-oriented language

Our first goal will be to show how many of the basic features of object-oriented programming can be understood as "derived forms" in a lower-level language with a rich collection of primitive features:
- (higher-order) functions
- records
- (higher-order) functions
- super
- the binding (this)
- inheritance
- composition of state
- dynamic dispatch
- late binding (this)

For simple objects and classes, this translational analysis works very well. When we come to more complex features (in particular, classes with self), it becomes less satisfactory, leading us to the more direct treatment in the following chapter.
The Essence of Objects

Dynamic Dispatch

This question has been a subject of debate for decades. Such arguments are always contentious and seldom very interesting. However, it is easy to identify some core features that are shared by most OO languages and that, together, support a distinctive and useful programming style.

Perhaps the most basic characteristic of object-oriented programming is that operations of the same type (i.e., responding to the same set of operations) may be implemented internally in completely different ways.

Perhapsthemostbasiccharacteristicofobject-orientedprogrammingis

When we apply a function to an argument, the ensuing behavior depends on the argument itself, rather than being fixed once and for all. Dynamic dispatch: when an operation is invoked on an object, the ensuing behavior depends on the object itself, rather than being fixed once and for all.

Two objects of the same type (i.e., responding to the same set of operations) may be implemented internally in completely different ways.
Encapsulation

In most OO languages, each object consists of some internal state encapsulated with a collection of method implementations operating on that state. The encapsulation of state with methods offered by objects is a form of information hiding.

A somewhat different form of information hiding is embodied in the notion of

\[\text{Encapsulation of internal state is optional. For full encapsulation, fields must be marked \texttt{protected}.}\]

\begin{verbatim}
Example

class A {
    int x = 0;
    int m() { x = x + 1; return x; }
    int n() { x = x - 1; return x; }
}

class B extends A {
    int m() { x = x + 5; return x; }
}

class C extends A {
    int m() { x = x - 10; return x; }
}

Note: \texttt{(new B()).m()} and \texttt{(new C()).m()} invoke completely different code.

The code \texttt{(new B()).x} is not allowed.

\begin{verbatim}
Example

class A {
    protected int x = 0;
    int m() { x = x + 1; return x; }
    int n() { x = x - 1; return x; }
}

class B extends A {
    int m() { x = x + 5; return x; }
}

class C extends A {
    int m() { x = x - 10; return x; }
)

Example

In Java, encapsulation of internal state is optional. For full encapsulation, fields must be marked \texttt{protected}.

\end{verbatim}
Sidenote: Objects vs. ADTs

An ADT comprises:

- A hidden representation type $X$.
- A collection of operations for creating and manipulating elements of type $X$.

Similar to OO encapsulation in that only the operations provided by the ADT are allowed to directly manipulate elements of the abstract type. But different in that there is just one (hidden) representation type and just one implementation of the operations.

Both styles have advantages.

- Inheritance allows reusing code.
- Subtyping allows one to specialize a type.

Example

```java
// Example

class D{
    int p(AmyA){return myA.m();}
}

D d = new D();
int z = d.p(new B());
int w = d.p(new C());
```

Inheritance

Behaviors in just one place.

To avoid duplication of code, want to write the implementations of these

shared parts of their behaviors.

Objects that share parts of their interfaces will typically (though not always)

inherit.


Subtyping and Encapsulation

Objects that support all the operations in $I$ have a type better than objects that support fewer.

An interface hiding more operations is better than an inheriting one.

Both styles have advantages.

- One representation of the operations — no dynamic dispatch.
- One interface to deal with the multiple implementations of the abstract type.

Similar to OO encapsulation in that only the operations provided by the ADTs are allowed to directly manipulate them. But different in that there is just one (hidden) representation type.

A collection of operations for creating and manipulating elements of type $X$.

- A hidden representation type $X$.
- An ADT comprises:

N. B. in the OO community, the term "abstract data type" is often used as a synonym for "object type." This is unfortunate, since it confuses two rather different concepts.
Inheritance

Basic mechanism of inheritance:

A class is a data structure that can be instantiated to create new objects ("instances").

A class can be refined to create new classes ("subclasses").

Basic mechanism of inheritance:

An instance of class B has methods m, n, and o. The first two are inherited from class A.

Example:

```java
class A {
    protected int x = 0;
    int m() {
        x = x + 1;
        return x;
    }
    int n() {
        x = x - 1;
        return x;
    }
    protected int x = 0;
}
```

```java
class B extends A {
    into() {
        x = x * 10;
        return x;
    }
}
```

Example:

```java
class E {
    protected int x = 0;
    int m() {
        x = x + 1;
        return x;
    }
    int n() {
        x = x - 1;
        return this.m();
    }
    protected int x = 0;
}
```

```java
class F extends E {
    int m() {
        x = x + 100;
        return x;
    }
}
```

Late binding

Most OO languages offer an extension of the basic mechanism of classes and inheritance called late binding or open recursion.

Example:

```java
class E {
    protected int x = 0;
    int m() {
        x = x + 1;
        return x;
    }
    int n() {
        x = x - 1;
        return x;
    }
    protected int x = 0;
}
```

```java
class F extends E {
    into() {
        x = x * 10;
        return x;
    }
}
```

What does (new E()).n() return?

What does (new F()).n() return?
Calling "Super"

It is sometimes convenient to re-use the functionality of an overridden method.

Java provides a mechanism called super for this purpose.

Example

```java
class Counter {
    protected int x = 1; // Hidden state

    protected int get() {
        return x;
    }

    void inc() {
        x++;
    }

    class Counter {
        protected int x = 0; // Hidden state

        void inc() {
            x++;
        }
    }
}
```

```java
inc3(c);
```
Objects

The local variable x has type Counter = {x: Nat}

\[\text{inc} = \lambda \text{x}. \text{x}::\text{succ}(\text{x})\]
\[\text{get} = \lambda \text{x}. \text{x}::\text{unit}\]
\[\text{x}::\text{unit. Let } r = \text{x}::\text{ref} \text{ in }\]
\[\text{neomounter} = \text{newCounter} : \text{unit} \leftarrow \text{counter}\]

It will be convenient (later) to group these into a single record. Rather than a single reference cell, the states of most objects consist of a number of instance variables or fields.

Grouping Instance Variables

Rather than a single reference cell, the states of most objects consist of a number of instance variables or fields.

\[\text{inc3} = \text{newCounter. inc}::\text{unit. inc}::\text{unit. inc}::\text{unit}\]

\[\text{get} = \lambda \text{x}. \text{x}::\text{unit}\]
\[\text{inc} = \lambda \text{x}. \text{x}::\text{succ}(\text{x})\]

\[\text{newCounter} = \text{newCounter} : \text{unit} \leftarrow \text{counter}\]

\[\text{newCounter} = \text{newCounter} : \text{unit} \leftarrow \text{counter}\]

Object Generators

Rather than a single reference cell, the states of most objects consist of a number of instance variables or fields.

\[\text{inc3} = \text{newCounter. inc}::\text{unit. inc}::\text{unit. inc}::\text{unit}\]

\[\text{get} = \lambda \text{x}. \text{x}::\text{unit}\]
\[\text{inc} = \lambda \text{x}. \text{x}::\text{succ}(\text{x})\]

\[\text{newCounter} = \text{newCounter} : \text{unit} \leftarrow \text{counter}\]
Subtyping and Inheritance

Simple Classes

The definitions of `Counter` and `ResetCounter` are identical except for the `reset` method. This violates a basic principle of software engineering: each piece of behavior should be implemented in just one place in the code.

```
class Counter {
    protected int x = 1;
    int get() { return x; }
    void inc() { x++; }
}
```

```
class ResetCounter extends Counter {
    void reset() { x = 1; }
}
```

```
ResetCounter rc = new ResetCounter();
inc3(rc); rc.reset(); inc3(rc); rc.get();
```

```
rc = newResetCounter unit;
(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);
rc = newResetCounter unit!
```

```
newResetCounter = newResetCounter { x=ref1 } in { get=\_unit.!(r.x),
    inc=\_unit.r.x:=succ(!(r.x)),
    reset=\_unit.r.x:=1 };
= newResetCounter:Unit ! ResetCounter
```

```
newCounter = new Counter { x=1 } in { get=x, inc=x=\_next(x),
    reset=x=1 };
```

Subtyping and Inheritance

```
reset = \_unit. x:=1,
inc = \_unit. x:=\_next(x),
get = \_unit. ref1
newCounter = { get=\_unit. x, inc=\_unit. x=\_next(x),
    reset=\_unit. x=1 };
```
Classes

A class is a runtime data structure that can be

1. instantiated to yield new classes
2. extended to yield new classes

To avoid the problem we observed before, what we need to do is to separate

the definition of the methods from the act of binding these methods to a particular set of instance variables:

```
\text{counterClass} = \text{CounterRep} \left< \begin{array}{l}
\text{get} = \lambda x . \text{get}(\text{succ}(x)) \smallint (x) \smallint \smallint \\
\text{inc} = \lambda x . \text{inc}(x) \smallint \smallint \\
\text{reset} = \lambda x . \text{reset}(x) \smallint \smallint \\
\end{array} \right>
```

```
\text{newCounter} = \smallint \text{counterClass} r \smallint \smallint 
```

To re-use the methods of some existing object to build a new object:

```
\text{resetCounterFromCounter} = \smallint \text{Counter} \smallint \smallint 
```

This doesn't work properly because the \text{reset} method does not have access to the local variable \text{x} of the original counter.

No. This doesn't work properly because the \text{reset} method does not have

```
\text{reset} = \lambda x . \text{reset}(x) \smallint \smallint \\
\text{inc} = \lambda x . \text{inc}(x) \smallint \smallint \\
\text{get} = \lambda x . \text{get}(x) \smallint \smallint \\
\text{resetCounterFromCounter} = \smallint \smallint 
```

Note: could we just re-use the methods of some existing object to build a new object?
Defining a Subclass

resetCounterClass = r:CounterRep.
let super = counterClass in
\{get = super.get, inc = super.inc, reset = \_ : Unit.r.x := 1\};

resetCounterClass : CounterRep !
ResetCounter
newResetCounter = \_ : Unit.letr = \{x = ref1\} in resetCounterClass r;

newResetCounter : Unit !
ResetCounter

Overriding and adding instance variables

class Counter {
  protected int x = 1;
  int get() { return x; }
  void inc() { x++; }
}

class ResetCounter extends Counter {
  void reset() { x = 1; }
}

class BackupCounter extends ResetCounter {
  protected int b = 1;
  void backup() { b = x; }
  void reset() { x = b; }
}

BackupCounter = \{get: Unit ! Nat, inc: Unit ! Unit, reset: Unit ! Unit, backup: Unit ! Unit\};
BackupCounterRep = \{x: RefNat, b: RefNat\};
backupCounterClass = r:BackupCounterRep.
let super = resetCounterClass in
\{get = super.get, inc = super.inc, reset = \_ : Unit.r.x := \!(r.b), backup = \_ : Unit.r.b := \!(r.x)\};

backupCounterClass : BackupCounterRep !
BackupCounter

Notes:

- Subtyping is essential here (in the definition of super)
- New Reset (the definition of CounterPlus in BackUpCounterPlus, both extends (with BackUp) and override (with a new Reset))
We can rewrite this class so that the `get/set` functionality appears just once.

**Bad style:** The functionality of `inc` could be expressed in terms of the functionality of `get` and `set`.

```java
inc = \( \lambda \text{unit}. \; \text{r} = (\text{succ} \; \text{r}) \text{.} \)!
set = \( \lambda \text{unit}. \; \text{r} = \text{r} \text{.} \)!
get = \( \lambda \text{unit}. \; (\text{r} \text{.} \)!
```

`funnyBackupCounterClass`:

```java
funnyBackupCounterClass = BackupCounterRep
!
BackupCounter
```

In Java we would write:

```java
class SetCounter{
protected int x = 0;
int get(){return x;}
void set(int i){x = i;}
void inc(){this.set(this.get()+1);}
}
```

**Bad style:** The functionality of `inc` could be expressed in terms of the functionality of `get` and `set`.

*Can we rewrite this class so that the `get/set` functionality appears just once?*

When if counters have `get`, `set`, and `inc` methods:

```java
SetCounter = \{\text{get}: \text{unit} \rightarrow \text{Nat}, \text{set}: \text{Nat} \rightarrow \text{unit}, \text{inc}: \text{unit} \rightarrow \text{unit} \}\}
```

When if counters have `set`, `get`, and `inc` methods:

```java
-- funnyBackupCounterClass = BackUpCounterRep — BackUpCounter
backlap = super.backlap!
reset = super.reset!
inc = \( \lambda \text{unit}. \; \text{r} = (\text{succ} \; \text{r}) \text{.} \)!
set = \( \lambda \text{unit}. \; \text{r} = \text{r} \text{.} \)!
get = \( \lambda \text{unit}. \; (\text{r} \text{.} \)!
```

When if counters have `get`, `set`, and `inc` methods:

```java
funnyBackupCounterClass = BackUpCounterRep
!
BackupCounter
```

Suppose (for the sake of the example) that we wanted every call to `inc` to first make a `backup` call the `backup` and then call `inc`.

```java
backup = super.backup!
reset = super.reset!
inc = \( \lambda \text{unit}. \; \text{r} = (\text{succ} \; \text{r} \text{.} \)!
set = \( \lambda \text{unit}. \; \text{r} = \text{r} \text{.} \)!
get = \( \lambda \text{unit}. \; (\text{r} \text{.} \)!
```

When if counters have `get`, `set`, and `inc` methods:

```java
funnyBackupCounterClass = BackUpCounterRep
!
BackupCounter
```
In essence, we are switching the order of `fix` and `r:CounterRep`.

```
fix (setCounterClass r)
```

This is just a definition of a group of mutually recursive functions. The type of the `fix` expression is `setCounter`. So this does not model the behavior of `fix` (or `this`) in real OO languages.

```
Note that we have changed the types of classes from `setCounterClass` to `setCounterClass:CounterRep`.
```

...to the object creation function:

```
newSetCounter = _:Unit.let r={x=ref1} in fix(setCounterClass r);
```

In essence, we are switching the order of `fix` and `r:CounterRep`.

```
fix (setCounterClass r)
```

This is just a definition of a group of mutually recursive functions. The type of the `fix` expression is `setCounter`. So this does not model the behavior of `fix` (or `this`) in real OO languages.

```
Note that we have changed the types of classes from `setCounterClass` to `setCounterClass:CounterRep`.
```
Using this method has been called.

Let's continue the example by defining a new class of counter objects (a subclass of set-counters) that keeps a record of the number of times the set method has ever been called:

```
InstrCounter={get:Unit ! Nat, set:Nat ! Unit, inc:Unit ! Unit, accesses:Unit ! Nat};

InstrCounterRep={x:RefNat, a:RefNat};
```

\[
\text{instrCounterClass} = \text{r:InstrCounterRep.}
\text{this:InstrCounter.}
\text{let super=setCounterClassrthisin}
\{
\text{get=super.get,}
\text{set=i:Nat.(r.a:=succ(!!(r.a));super.seti),}
\text{inc=super.inc,}
\text{accesses=_:Unit.!(r.a)}
\};
\]

\[
\text{setCounterClass}:	ext{InstrCounterRep} ! \text{InstrCounter} ! \text{InstrCounter}
\]

Notes:

- the method uses both this (which is passed as a parameter) and super.
- the method calls the `super` set method twice (once as a super method and once as a method in the superclass).

One more refinement...

A small fly in the ointment...
To see why this diverges, consider a simpler example:

let \( f : \text{Nat} \rightarrow \text{Nat} \)

where \( f \) is defined as

\[
\text{let } f_0 = \text{fix } f \text{ in } \text{Nat.0}
\]

Now:

\[
\text{fix } f \text{ (Nat\rightarrow\text{Nat}) (Nat\rightarrow\text{Nat}) (Char\rightarrow\text{Char})}
\]

This works, in the sense that we can now instantiate \text{InstrCounterClass} (without diverging!), and its instances behave in the way we intended.

**Success**

One possible solution is to put a dummy abstraction in front of it:

\[
\text{let } f_0 = \text{fix } f \text{ in } \text{Nat.0}
\]

**Success**

Similarly:

\[
\text{let } f_0 = \text{fix } f \text{ in } \text{Nat.0}
\]

**Success**
Success

This works, in the sense that we can now instantiate instrCounterClass (without diverging!), and its instances behave in the way we intended.

Section 18.12 in TAPL shows how this can be repaired by using references instead of fix to "tie the knot" in the method table.

Recap

All the objects we have built in this series of examples have type Counter.

Encapsulation

But their internal representations vary widely:

Multiplerepresentations

An object is a record of functions, which maintain common internal state via a shared reference to a record of mutable instance variables.

The state is inaccessible outside of the object because there is no way to name it.

Section 18.12 in TAPL shows how this can be repaired by using references instead of fix to "tie the knot" in the method table.

Recap

Multiple representations

Encapsulation

Instead of fix to "tie the knot" in the method table.

(Hence all the "sharing" we add here has an unfortunate side effect: instead of computing the "method table" just once, when an object is created, we will compute the "method table" every time we invoke a method.)

This is one of the reasons the design of fix is so difficult. And it is one of the reasons why the design of references is so simple.
Subtyping

Subtyping between object types is just ordinary subtyping between types of records of functions. Functions like inc3 that expect Counter objects as parameters can (safely) be called with objects belonging to any subtype of Counter.

Inheritance

Inheritance is modeled by copying implementations of methods from superclasses to subcategories. Subclasses inherit these methods.

A class inherits from another class by copying its methods and variables into the new class. The new class inherits the methods and variables of the super class. This process is repeated for each super class of the class.

Note to be handed in — just for you to check your understanding.

Additional exercise

Take all the examples from this lecture (and the previous one), and recode them in Java.

[Nottobehandedin|justfortoyourtocheckyourunderstanding.]