29 November
Fall 2004
Software Foundations
CIS 500
This week: Chapter 18/19
Next week: Chapter 19/Review
Final exam: Monday, December 20th
Course status: add together 2 midterm grades

≥ 90  A  (24%)
90-128  B  (49%)
69-89  C  (20%)
< 69  D/F  (7%)

Plans

CIS 500, 29 November
On to Objects
A Change of Pace

We’ve spent the semester developing tools for defining and reasoning about a variety of programming language features. Now it’s time to use those tools for something more ambitious.
Plan:

1. Identify some characteristic "core features" of object-oriented programming
2. Develop two different analyses of these features:
   (a) A direct, high-level formalization of a simple object-oriented language
   (b) A translation into a lower-level language

Case study: object-oriented programming
Our first goal will be to show how many of the basic features of object-oriented languages can be understood as "derived forms" in a lower-level language with a rich collection of primitive features:

- late binding
- (higher-order) functions
- supports
- references
- recursion
- subtyping
- records
- self
- objects
- super
- inheritance
- encapsulation of state
- dynamic dispatch
- objects
- languages

The Translational Analysis
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.
Concepts
What is "object-oriented programming"? The Essence of Objects
What is “object-oriented programming”?

The Essence of Objects

This question has been a subject of debate for decades. Such arguments are always inconclusive and seldom very interesting.
The Essence of Objects

What is "object-oriented programming"?

Style.

Languages and that, together, support a distinctive and useful programming style. However, it is easy to identify some core features that are shared by most OO languages; regardless of how inclusive and seldom very interesting.

This question has been a subject of debate for decades. Such arguments are always inconclusive and seldom very interesting.
Dynamic dispatch

Perhaps the most basic characteristic of object-oriented programming is 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.
Example

```java
Note: (new B()).m() and (new C()).m() invoke completely different code.

```
Encapsulation

In most OO languages, each object consists of some internal state encapsulated with a collection of method implementations operating on that state.

- state directly accessible to methods
- state invisible / inaccessible from outside the object
Example

In Java, encapsulation of internal state is optional. For full encapsulation, fields must be marked protected:

```java
class A {
    protected int x = 0;

    int m() {
        x = x - 10;
        return x;
    }
}

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

class B extends A {
    int m() {
        x = x + 5;
        return x;
    }
}
```
Encapsulation is arguably a little less fundamental than dynamic dispatch, in the sense that there are several OO languages (e.g., CLOS, Dylan, and Cecil) that do not encapsulate state with methods. Although their basic mechanisms are quite different, the higher-level programming idioms (classes, inheritance, etc.) arising in multi-method polymorphism are surprisingly similar to those in mainstream OO languages. Languages are based, instead, on multi-methods, a form of ad-hoc dynamic dispatch of other objects of the same class...

(Side note for Java experts: we’re also eliding some subtleties involving accessing the protected fields of other objects of the same class...)

Encapsulation
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 an abstract data type (ADT).
An ADT comprises:

- A collection of operations for creating and manipulating elements of type \( X \).
- A hidden representation type \( X \).

Both styles have advantages.

But different in that there is just one (hidden) representation type and just one implementation of the operations — no dynamic dispatch.

Similar to OO encapsulation in that only the operations provided by the ADT are allowed to directly manipulate elements of the abstract type.

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

In the OO community, the term "abstract data type" is often used as more or less a synonym for "object type." This is unfortunate, since it confuses two rather different concepts.

\( \text{CIS 500, 29 November} \)
Subtyping

The type (or interface in Smalltalk terminology) of an object is just the set of operations that can be performed on it (and the types of their parameters and results); it does not include the internal representation. Any interface listing more operations is "better" than one listing fewer.

This gives rise to a natural and useful form of polymorphism: we can write one piece of code that operates uniformly on any object whose interface is "at least as good as" I (i.e., any object that supports at least the operations in I).

Objects interfaces fit naturally into a subtype relation. An interface listing "at least as good as" I (i.e., any object that supports at least the operations in I) subtypes any interface listing fewer operations.
```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());

... 

{ 
    { int p(A myA) { return myA.m(); } 
        class D 
    } 
} 
```
Inheritance

Objectsthatsharepartsoftheirinterfaceswilltypically(thoughnotalways)
sharepartsoftheirbehaviors.

Toavoidduplicationofcode,wanttowritetheimplementationsofthese
behaviorsinjustoneplace.

Objecesthatsharepartsoftheirinterfaceswilltypically(notalways)
Inheritance

Basic mechanism of inheritance: classes

- Instatiated to create new objects ("instances")
- Refined to create new classes ("subclasses")

By refining the behavior of existing objects, a similar mechanism, called delegation, which allows new objects to be derived.

N.b.: some OO languages offer an alternative (but fundamentally similar) mechanism called delegation, which allows new objects to be derived by refining the behavior of existing objects.
An instance of B has methods m, n, and o. The first two are inherited from A.
Late binding is rather tricky, both to define (as we will see) and to use tastefully. For this reason, it is sometimes deprecated in practice.

Though quite useful in many situations, late binding is rather tricky, both to define and to use tastefully. For this reason, it is sometimes deprecated in practice.

Late binding allows a method within a class to call another method via a special “pseudo-variable” \texttt{self}. If the second method is overridden by some subclass, then the behavior of the first method automatically changes as well.

Late binding is an extension of the basic mechanism of classes and inheritance called \texttt{late binding} or \texttt{open recursion}.

Most OO languages offer an extension of the basic mechanism of classes and
Examples

```java
class E {
  protected int x = 0;
  int m() {
    x = x + 1;
    return x;
  }
  int n() {
    x = x - 1;
    return this.m();
  }
}
class F extends E {
  int m() {
    x = x + 100;
    return x;
  }
}
```

What does \( new \ E() \) \( .m() \) return?

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

Java provides a mechanism called super for this purpose. It is sometimes convenient to "re-use" the functionality of an overridden method.
What does (new G()).n() return?

```java
class E {
    protected int x = 0;

    int m() {
        x = x + 1;
        return x;
    }

    int n() {
        x = x - 1;
        return this.m();
    }
}
class G extends E {

    int m() {
        x = x + 100;
        return super.m();
    }
}
```
Getting down to details (in the lambda-calculus)
Objects

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

Counter c = new Counter();

void inc3(Counter c) {
  c.inc(); c.inc(); c.inc();
}

Counter c = new Counter();
inc3(c);
inc3(c);

{ 
  c.inc();
  c.inc();
  c.inc();
}
void inc3(Counter c) {
  { 
  void inc() { x++; c; }
  }
}

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

```
How do we encode objects in the lambda-calculus?
Objects

\[
\begin{aligned}
\text{c} = & \text{let } x = \text{ref 1 in} \\
\{&\text{get} = \text{Unit!} \langle x \rangle, \\
&\text{inc} = \text{Unit} \langle x \rangle := \text{succ}(!x)\}; \\
\text{c} : \text{Counter}
\end{aligned}
\]

where

\[
\begin{aligned}
\text{Counter} = & \{\text{get} : \text{Unit} \langle \text{Nat} \rangle, \\
&\text{inc} : \text{Unit} \langle \text{Unit} \rangle\}
\end{aligned}
\]
Objects

\[\text{inc3} = \text{Counter} : (\text{c}.\text{inc unit}; \text{c}.\text{inc unit}; \text{c}.\text{inc unit});\]

\[\text{inc3} : \text{Counter} \leftarrow \text{Unit}\]

\[\text{inc3} \leftarrow \text{Counter} : (\text{c}.\text{inc unit}; \text{c}.\text{inc unit}; \text{c}.\text{inc unit});\]
newCounter = \_ : Unit. let x = ref 1 in { get = \_ : Unit. !x, 
succ = \_ : Unit. x := succ(!x) };
inc = \_ : Unit. x := succ(!x),
get = \_ : Unit. x,
\_ : Unit. let x = ref 1 in 
newCounter =

Object Generators
Subtyping and Inheritance

```java
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();
```
Subtyping
Subtyping

rc = newResetCounter unit;

(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);

\[ \Rightarrow 4 \]
Rather than a single reference cell, the states of most objects consist of a number of instance variables or fields.

It will be convenient (later) to group these into a single record.

CounterRep = {x: Ref Nat | {get = \_: Unit. !(r.x),
inc = \_: Unit. r.x := succ(!(r.x))}}

Grouping Instance Variables
The definitions of newCounter and newResetCounter 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.
Idea: could we just re-use the methods of some existing object to build a new object?

```plaintext
reset = ∀:Unit. x:=1;
    inc = c.inc,
    get = c.get,

let x = ref 1 in
resetCounter = resetCounterFromCounter
```

No: This doesn't work properly because the `reset` method does not have access to the instance variable `x` of the original counter.
Idea: could we just re-use the methods of some existing object to build a new object?

**Reusing Methods**

```plaintext
reset = ∀:unit. x:=1;
inc = c.inc;
get = c.get;

class Counter. Let x = ref 1 in
resetCounterPromCounter = resetCounterPromCounter

No: This doesn't work properly because the reset method does not have access to the instance variable x of the original counter.
```
A class is a run-time data structure that can be

1. instantiated to yield new objects
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:

\[ \text{newCounter} \leftarrow \text{unit : Counter } \]

\[ \text{counterClass} r \]

\[ \forall x : \text{Unit}. \text{let } r = \{x = \text{ref 1} \} \text{ in} \]

\[ \text{newCounter} = \text{newCounter} \]

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.
Defining a Subclass

```plaintext
let super = counterClass in resetCounterClass =

\(\text{Let } x = \text{ref } 1 \text{ in resetCounterClass } r\};
\(\forall : \text{Unit} \ x := 1\);
\(\text{newResetCounter } = \text{resetCounter}\)

\text{newResetCounter } = \text{resetCounterClass : CounterRep } = \text{resetCounter}

let super = counterClass in

let super = counterClass in

let super = counterClass in

let super = counterClass in
```
Adding instance variables

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

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

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

CIS 500, 29 November
Adding instance variables

In general, when we define a subclass we will want to add new instances variables to its representation.

Variables to its representation.
backupCounterClass = r:BackupCounterRep.

let super = resetCounterClass in

\{ (.x, .r) \mapsto \{
  incr = super.incr,
  reset = super.reset,
  get = super.get,
  new = super.new,
  backup = new Unit.(x, .r).
\}

\( \forall .(x, .r) : \text{resetCounterClass} \)
Calling super

Suppose (for the sake of the example) that we wanted every call to inc to first

makings inc use the backup and inc methods from super.

We can avoid copying the code for backup by back up the current state. We can avoid copying the code for backup

funnyBackupCounter : backupCounterClass <-

funnyBackupCounterClass : BackupCounterRep

let super = backupCounterRep in

\text{funnyBackupCounterClass: BackupCounterRep}
Let's define a class of counters with `get`, `set`, `inc` and `dec` methods:

```scala
class SetCounter {
  val counterRep = {
    get = λ :unit. ¡r.x:=i
    set = λ :nat. ¡r.x:=i
    inc = λ :unit. ¡r.x:succ r.x
  } "setCounterClass = setCounterClass"
}
```

Let's define a class of counters with `get`, `set`, `inc` and `dec` methods:
Calling between methods

Let's define a class of counters with `get`, `set`, and `inc` methods:

```haskell
class SetCounter { get :: Unit → Nat, set :: Nat → Unit, inc :: Unit → Unit; }
```

```haskell
setCounterClass = r :: CounterRep. { get = ∀x :: Unit. r.x, set = ∀i :: Nat. r.x := i, inc = ∀x :: Unit. r.x := (suc r.x) } } r
```

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

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

```haskell
inc = ∀x :: Unit. r.x := (suc r.x) } r

set = ∀i :: Nat. r.x := i

get = ∀x :: Unit. r.x

setCounterClass = r :: CounterRep. { setCounter = get :: Unit → Nat, set :: Nat → Unit, inc :: Unit → Unit; }
```

```haskell
setCounter = { get :: Unit → Nat, set :: Nat → Unit, inc :: Unit → Unit; }
```
This is just a definition of a set (record) of mutually recursive functions. We say something similar in the \texttt{seven/twod} example in 11.11.

We first define a set (record) of mutually recursive functions. (We saw something similar in the \texttt{seven/twod} example in 11.11.)

\begin{verbatim}
setCounter = r:CounterRep.
fix (setCounter){
  get = \forall :Unit. set.(succ.(self.getUnit))
  set = \forall :Nat. r.x:=!
  inc = \forall :Unit. self.set(succ(self.getUnit))
}
\end{verbatim}

This is just a definition of a set (record) of mutually recursive functions. (We saw something similar in the \texttt{seven/twod} example in 11.11.)

We first define a set (record) of mutually recursive functions. (We saw something similar in the \texttt{seven/twod} example in 11.11.)

\begin{verbatim}
setCounter = r:CounterRep.
fix (setCounter){
  get = \forall :Unit. set.(succ.(self.getUnit))
  set = \forall :Nat. r.x:=!
  inc = \forall :Unit. self.set(succ(self.getUnit))
}
\end{verbatim}

Better
So this does not model the behavior of self (or this) in real OO languages.

is "closed" — we "tie the knot" when we build the record.

\[
\text{fix } \text{Counter} = \text{self}.
\text{set} = \text{unit}.
\text{get} = \text{unit}.
\text{inc} = \text{unit}.
\text{set} (\text{suc} (\text{get} (\text{set} (\text{unit}))))
\]

Note that the fixed point in SetCounterClass
In essence, we are switching the order of \( \text{fix} \) and \( \text{CounterRep} \):

\[
\text{fix} \ (\text{setCounterClass} \ r) = \\
\forall \text{unit} . \ \text{let} \ r' = \{ x = \text{ref} \ i \} \ \text{in} \\
\text{newSetCounter} = \\
\text{setCounter} = \text{setCounterClass} \ r
\]

...to the object creation function:

\[
\text{newSetCounter} = \\
\text{set} = \forall \text{unit} . \ \text{let} . \ \text{set} (\text{succ}(\text{self}.\text{get} . \text{unit})) \} \\
\text{get} = \forall \text{unit} . \ \text{let} . \ (i : \text{r} . x) \} \\
\text{set} = \forall \text{nat} . \ \text{r} . x : i \} \\
\text{inc} = \forall \text{unit} . \ \text{let} . \ \text{set} . \ (\text{succ}(\text{self}.\text{get} . \text{unit})) \} \\
\text{self} : \text{SetCounter}. \\
\text{CounterRep} = \\
\text{setCounterClass} = \\
\text{Idea: move the application of fix from the class definition...}
Note that we have changed the types of classes from...

```
s: SetCounterClass := r:CounterRep.
self:SetCounter.
{get=\_:Unit.!(r.x),
set=\_i:Nat.r.x:=i,
inc=\_:Unit.self.set(succ(self.getunit))};

x: SetCounter.
x: CounterRep.
s: SetCounterClass := x: CounterRep.
```

```
fix
x: CounterRep.
s: SetCounterClass := x: CounterRep.
```

...to:

```
s: SetCounterClass := r:CounterRep.
self:SetCounter.
{get=\_:Unit.!(r.x),
set=\_i:Nat.r.x:=i,
inc=\_:Unit.self.set(succ(self.getunit))};

x: SetCounter.
x: CounterRep.
s: SetCounterClass := x: CounterRep.
```
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.

Using self

InstrCounter = \{get:Unit!\nat, set:Nat\!

InstrCounterRep = \{x:RefNat, a:RefNat\!

inc:Unit\!

accesses:Unit\!

set:Nat\!

get:Unit\!

set:Unit\!

unit\!

Nat\!

unit\!

Nat\!

unit\!

Nat\!\}
Supporting play a crucial role (twice) in the call to `setCounterClass`:

```plaintext
let super = setCounterClass r self in
{get = super.get,
set = i:Nat.(r.a:=succ(!(r.a));super.set i),
inc = super.inc,
accesses = super.set i (access) (get i);)
}
```

The `inc` in `super` will call the `set` defined here, which calls the superclass

(Which is constructed using `self` and the instance variables)

The methods use both `self` (which is passed as a parameter) and `super`:

Notes:

- `setCounterClass : InstrCounterRep => InstrCounter`  
- `InstrCounter <- InstrCounterRep`  

```plaintext
let super = setCounterClass r self in
{get = super.get,
set = i:Nat.(r.a:=succ(i);super.set i),
inc = super.inc,
accesses = super.set i (access) (get i);)
}
```

```plaintext
let super = setCounterClass r self in
{get = super.get,
set = i:Nat.(r.a:=succ(i);super.set i),
inc = super.inc,
accesses = super.set i (access) (get i);)
}
```

```plaintext
let super = setCounterClass r self in
{get = super.get,
set = i:Nat.(r.a:=succ(i);super.set i),
inc = super.inc,
accesses = super.set i (access) (get i);)
}
```
One more refinement...
Asmall y in the ointment

The implementation we have given for instrumented counters is not very useful because calling the object creation function 

\[ \text{newInstrCounter} = \_\text{Unit.letr} = \{x=\text{ref1}, a=\text{ref0}\} \text{in instrCounterClassr}; \]

will cause the evaluator to diverge! Intuitively (see TAPL for details), the problem is the "unprotected" use of

\[ \text{fix (InstrCounterClass r)}; \]

\[ \forall \text{unit. Let r = } \{x=\text{ref1}, a=\text{ref0}\} \text{ in } \text{newInstrCounter = InstrCounter}. \]

The implementation we have given for instrumented counters is not very useful.

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

To see why this diverges, consider a simpler example:
One possible solution

Idea: "delay" self by putting a dummy abstraction in front of it...

\[
\text{setCounterClass} = \lambda r : \text{CounterRep}. \\
\quad \lambda \text{self} : \text{Unit} \rightarrow \text{SetCounter}. \\
\quad \lambda x : \text{Nat}. \\
\quad \begin{cases}
\quad \text{get} = \lambda \text{unit}. \downarrow (r.x), \\
\quad \text{set} = \lambda i : \text{Nat}. \quad r.x := i, \\
\quad \text{inc} = \lambda \text{unit}. (\text{self unit}).\text{set}(\text{succ}(\text{self unit}).\text{get unit}))
\end{cases}
\]

\[
\text{newSetCounter} = \lambda \text{Unit}. \text{let } r = \{x = \text{ref 1}\} \text{ in } \text{fix (setCounterClass } r\text{)} \text{ unit};
\]
Similarly:

```latex
\textit{similarly}:
\begin{align*}
\text{fix (InstrCounterClass } r) & \text{ unit;} \\
\forall \text{ unit. Let } r = \{ x = \text{ret } 1, a = \text{ret } 0 \} \text{ in } \\
\text{newInstrCounter} = \\
\{ \text{ accesses = } \forall \text{ unit. } i(r.a) \}; \\
\text{ inc = super.inc,} \\
\text{ inc = super.inc,} \\
\text{ set = } \forall \text{ unit. } i(r.a = \text{succ} (r.a)); \text{ super.set } i, \}
\text{ get = super.get,} \\
\text{ get = super.get,} \\
\text{ Let super = setCounterClass } r \text{ self unit in } \\
\forall \text{ unit.} \\
\forall \text{ self : Unit} \hookrightarrow \text{InstrCounter,} \\
\forall r : \text{InstrCounterRep.} \\
\text{InstrCounterClass } = \\
\text{Similarly:}
\end{align*}
```
This works, in the sense that we can now instantiate `InstrCounterClass` without diverging! and its instances behave in the way we intended.

However, all the "delaying" we added has an unfortunate side effect: instead of computing the "method table" just once, when an object is created, we will now re-compute it every time we invoke a method!

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.
Instead of \texttt{fix} to "tie the knot" in the method table.

Section 18.12 in TAPL shows how this can be repaired by using references
now re-compute it every time we invoke a method!
computing the "method table" just once, when an object is created, we will
however, all the "delayering" we added has an unfortunate side effect: instead of
(without diverging!); and its instances behave in the way we intended.

\texttt{Success (:)}

This works, in the sense that we can now instantiate \texttt{InstantCounterClass}
But their internal representations vary widely.

All the objects we have built in this series of examples have type `counter`. Multiple representations...
Encapsulation

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

This state is inaccessible outside of the object because there is no way to name it. (Instance variables can only be named from inside the methods.)
Subtyping between object types is just ordinary subtyping between types of records of functions. Functions like \texttt{inc3} that expect \texttt{Counter} objects as parameters can (safely) be called with objects belonging to any subtype of \texttt{Counter}. Subtyping between object types is just ordinary subtyping between types of records of functions.
Inheritance

Classes are data structures that can be both extended and instantiated.

We modeled inheritance by copying implementations of methods from superclasses to subclasses.

Each class

uses and self to instantiate its superclass

consists of a record of method implementations, copying some directly

from super and implementing others in terms of self and super.

The self parameter is "resolved" at object creation time using fix.

wait to be told a record of instance variables and an object self (which

should have the same interface and be based on the same record of

instance variables)

wait for a record of instance variables and an object self (which

CIS 500, 29 November
Additionalexercise

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

[Nottobe handed in — just for you to check your understanding.]