Now it’s time to use those tools for something more ambitious.

We’ve spent the semester developing tools for defining and reasoning about a

A Change of Pace

Onto Objects

Plans

Course status: add together 2 midterm grades
Final exam: Monday, December 20th
Next week: Chapter 19/Review
This week: Chapter 18/19

69 - D/F (24%)
90-128 - B (49%)
129-160 - A (24%)
69-89 - C (24%)
< 69 - D/F (7%)

On to Objects

Fall 2004
Software Foundations
CIS 500
Plan:

1. Identify some characteristic core features of object-oriented programming
2. Develop two different analyses of these features

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

(1) A direct, high-level formalization of a simple object-oriented language
(2) A translation into a lower-level language

The Translational Analysis

The first step 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:

- higher-order functions
- (higher-order) functions
- records
- records
- self (this) and super
- inheritance
- subtyping
- dynamic dispatch
- objects
- encapsulation

For simple objects and classes, this translational analysis works very well.

For more complex objects and classes, the more direct treatment in the following chapter becomes less satisfactory, leading us to the more direct treatment in the

History

CIS 500, 29 November
The Essence of Objects

What is object-oriented programming?

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

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 (as when we apply a function to an argument).

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

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: (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,

```java
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; }
}
```

Sidenote: encapsulation

Encapsulation is arguably a little less fundamental than dynamic dispatch. Although their basic mechanisms are quite different, the higher-level

```
A abstract data type (ADT).
A somewhat different form of information hiding is embodied in the notion of

The encapsulation of state with methods offered by objects is a form of

Sidenote: Objects vs. ADTs

The encapsulation of state with methods offered by objects is a form of

information hiding.

Sidenote: Objects vs. ADTs

An ADT comprises:

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

Both styles have advantages.

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

t in that there is just one (hidden) representation type and just one implemen-
tation of the operations | no dynamic dispatch.

Inheritance are implemented similarly to those in „mainstream“ OO languages.

Although their basic mechanisms are quite different, the higher-level

```
public class A {
    public int x = 10;
    public int m() { return x; }
}
public class B extends A {
    public int m() { return x; }
}
```

In Java, encapsulation of internal state is optional. For full encapsulation,

```
Example

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

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

t in that there is just one (hidden) representation type and just one implemen-
tation of the operations | no dynamic dispatch.

Inheritance are implemented similarly to those in „mainstream“ OO languages.

Although their basic mechanisms are quite different, the higher-level

```
public class A {
    public int x = 10;
    public int m() { return x; }
}
public class B extends A {
    public int m() { return x; }
}
```

In Java, encapsulation of internal state is optional. For full encapsulation,

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

Inheritance are implemented similarly to those in „mainstream“ OO languages.

Although their basic mechanisms are quite different, the higher-level

```
public class A {
    public int x = 10;
    public int m() { return x; }
}
public class B extends A {
    public int m() { return x; }
}
```

In Java, encapsulation of internal state is optional. For full encapsulation,

```
class A {
    int x = 10;
    int m() { return x; }
}
class B extends A {
    int m() { return x; }
}
class C extends A {
    int m() { return x; }
}
```
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. Subtyping (or "interface" in Smalltalk terminology) of an object is just the specialization of its type.

Example

```java
class D {
    int p(AmyA){return myA.m();}
}...Dd=new D();
int z=d.p(new B());
int w=d.p(new C());
```

Inheritance

Objectsthat share partsof their interfaces will typically (though not always) share partsof their behaviors.

To avoid duplication of code, want to write the implementations of those operatings in just one place.

Basic mechanism of inheritance:

A class is a data structure that can be

- extended to create new classes ("subclasses")
- instantiated to create new objects ("instances")

N.B.: Some OOP languages offer an alternative (but fundamentally similar) mechanism called delegation, which allows new objects to be derived from existing ones (and to be treated as if they were part of the original data structure).

Inheritance

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

An interface inherits more operations if it supports at least the operations in I.

Inheritance

By refining the interface of existing objects.

```
if x = d.p(new C())
    if y = d.p(new B())
        d = new D()
    ...
```

```java
    class D {
        return myA.m();
    }
```
Late binding allows a method within a class to call another method via a special `pseudo-variable` `self`. If the second method is overridden by some inheritance called late binding or open recursion.

Most OO languages offer an extension of the basic late binding of classes and methods with a mechanism called super for this purpose.

Java provides a mechanism called `super` for this purpose.

If is sometimes convenient to re-use the functionality of an overridden method. `Calling super`
What does (new G()).n() return?

Objects

Example

Getting down to details

(in the lambda-calculus)...
Objects

```java
Objects
```

```java
Subtyping and Inheritance

```java
Subtyping and Inheritance
```

```java
Object Generators

```java
Object Generators
```

```java
Counter

```java
Counter

```java
Subtyping and Inheritance

```java
Subtyping and Inheritance
```

```java
Object Generators

```java
Object Generators
```

```java
Counter

```java
Counter

```java
Subtyping and Inheritance

```java
Subtyping and Inheritance
```

```java
Object Generators

```java
Object Generators
```
Subtyping

ResetCounter = {get:Unit, inc:Unit, reset:Unit};

newResetCounter = (_:Unit.let x = ref1 in
{get = _:Unit.!x,
inc = _:Unit.x := succ(!x),
reset = _:Unit.x := 1});

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

c = let r = {x = ref1} in
{get = _:Unit.!(r.x),
inc = _:Unit.r.x := succ(!(r.x))};

CounterRep = {x:RefNat};

CIS 500, 29 November

Subtyping

newResetcounter : Unit → Resetcounter

gerestcounter = {get:Unit, inc:Unit, reset:Unit};

4 \iff
touch rc; reset rc; touch rc; rc.get unit;
rc = newresetcounter unit!
Simple Classes

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.

This doesn’t work properly because the `reset` method does not have access to the instance variable `x` of the original counter.

```ocaml
reset = ??:unit. x:=1
inc = c.inc
get = c.get
```

Reusing Methods

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

No: This doesn’t work properly because the `reset` method does not have access to the instance variable `x` of the original counter.

```ocaml
resetCounterFromCounter = c:Counter.let x=ref 1 in
{get=c.get;
 inc=c.inc;
 reset=?:unit.x:=1};
```

Classes

A class is a runtime data structure that can be

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

Each of these operations is specific to classes. Classes are special kinds of objects.
To avoid the problem we observed before, what we need to do is separate
the definition of the methods from the act of binding those methods to a particular set of instance variables.

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

```
newResetCounter = new ResetCounter;
```

```
BackupCounter = {get:Unit Nat, inc:Unit Nat, reset:Unit Nat, backup:Unit Nat};
BackupCounterRep = {x:Ref Nat, b:Ref Nat};
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)};
```

```
newBackupCounter = new BackupCounter;
```

```
```
```
Can we 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.

\[
\begin{aligned}
\text{func} = \lambda v. \text{unit}, \ i.x := (\text{succ } i.x) \}(); \\
\text{set} = \lambda v. \text{unit}, \ i.x := i, \\
\text{get} = \lambda v. \text{unit}, \ (i.x) \\
\text{v} : \text{CounterRep}. \\
\text{setCounter} \rightarrow \text{CounterRep} \\
\end{aligned}
\]

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

\[
\text{CIS500, 29 November 46}
\]

Calling between methods

Suppose (for the sake of the example) that we wanted every call to \( inc \) to first
backup the current state. We can avoid copying the code for \( backup \) by
backing up the current state. We can avoid copying the code for \( backup \) by
backing up the current state.

\[
\text{func} = \lambda v. \text{unit}, \ i.b := (\text{succ } i.b) \}(); \\
\text{set} = \lambda v. \text{unit}, \ i.x := i, \\
\text{get} = \lambda v. \text{unit}, \ (i.x) \\
\text{v} : \text{CounterRep}. \\
\text{setCounter} \rightarrow \text{CounterRep} \\
\]

Bad style: The functionality of \( inc \) could be expressed in terms of the
functionality of \( get \) and \( set \).

Can we write this class so that the \( get \)/\( set \) functionality appears just once?

\[
\begin{aligned}
\text{func} = \lambda v. \text{unit}, \ i.x := (\text{succ } i.x) \}(); \\
\text{get} = \lambda v. \text{unit}, \ (i.x) \\
\text{set} = \lambda v. \text{unit}, \ i.x := i, \\
\text{v} : \text{CounterRep}. \\
\text{setCounter} \rightarrow \text{CounterRep} \\
\end{aligned}
\]

Calling between methods

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

\[
\text{CIS500, 29 November 47}
\]

Calling between methods

\[
\text{func} = \lambda v. \text{unit}, \ i.b := (\text{succ } i.b) \}(); \\
\text{get} = \lambda v. \text{unit}, \ (i.x) \\
\text{set} = \lambda v. \text{unit}, \ i.x := i, \\
\text{v} : \text{CounterRep}. \\
\text{setCounter} \rightarrow \text{CounterRep} \\
\]

Calling between methods

Suppose (for the sake of the example) that we wanted every call to \( inc \) to first
backup the current state. We can avoid copying the code for \( backup \) by
backing up the current state. We can avoid copying the code for \( backup \) by
backing up the current state. We can avoid copying the code for \( backup \) by
backing up the current state.
setCounterClass = r:CounterRep.
fix (self:SetCounter.
{get= _:Unit.!(r.x),
set=i:Nat.r.x:=i,
inc= _:Unit.self.set(succ(self.getunit))});

Check: the type of the inner fix expression is SetCounter.

setCounterClass: CounterRep =

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

setCounterClass: CounterRep ! SetCounter

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

We fixed the lack of "self" (or this) in real OO languages.

setCounterClass: CounterRep =
fix (setCounterClass r)

Note that the fixed point in setCounterClass is.

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

setCounterClass: CounterRep ! SetCounter

This is just a definition of a set (record) of mutually recursive functions. We

We fixed the lack of "self" (or this) in real OO languages.

setCounterClass: CounterRep =
fix (setCounterClass r)

Note that the fixed point in setCounterClass is.

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

setCounterClass: CounterRep =
fix (setCounterClass r)

Note that the fixed point in setCounterClass is.
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.

Let's define a new class of counters that keeps a record of the number of times the `set` method has ever been called.

One more refinement...

A small fly in the ointment

...
To see why this diverges, consider a simpler example:

\[ f : \text{Nat} \to \text{Nat}. \]

\[
\begin{align*}
\text{let } f_0 &= \text{fix } f \\
&\text{in } f_0 : \text{Nat} \to \text{Nat}
\end{align*}
\]

Now:

\[ \text{fix } f \]

\[ f (\text{fix } f) \]

\[ \text{fix } f (f (\text{fix } f)) \]

\[ f_0 (f_0 (f_0 (f_0 (f_0 \ldots)))) \]

This works, in the sense that we can now instantiate \( \text{InstrCounter} \), and its instances behave in the way we intended. Without doing anything, and its instances behave in the way we intended.

To see why this diverges, consider a simpler example:

\[ f : \text{Nat} \to \text{Nat}. \]

\[
\begin{align*}
\text{let } f_0 &= \text{fix } f \\
&\text{in } f_0 : \text{Nat} \to \text{Nat}
\end{align*}
\]

Now:

\[ \text{fix } f \]

\[ f (\text{fix } f) \]

\[ \text{fix } f (f (\text{fix } f)) \]

\[ f_0 (f_0 (f_0 (f_0 (f_0 \ldots)))) \]

This works, in the sense that we can now instantiate \( \text{InstrCounter} \), and its instances behave in the way we intended. Without doing anything, and its instances behave in the way we intended.
Success

This works, in the sense that we can now instantiate an instance of `Counter`, 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 a method is invoked.

Section 18.12 in TAPL shows how this can be repaired by using references instead of fix. Instead of fix to "te the knot" in the method table, we invoke a method computing the "method table" just once when an object is created, we will now re-compute it every time we invoke a method.
Subtyping

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

Inheritance

Classes are data structures that can be both extended and instantiated. We model inheritance by copying implementations of methods from superclasses to subclasses.

Additional exercise

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

Not to be handed in — just for you to check your understanding.