CIS 500

Software Foundations

Fall 2002

18 November

Administrivia

- Prof. Pierce's 3PM recitation this afternoon cancelled — please go to Max K. in Towne 307 instead
- Midterm results available in 556
- Rough grade breakdown:

  65-80: A

  50-64: B

  35-49: C

  $\leq$34: D/F

  58+ points is on-target for WPE-I
- Grading questions? See your TA.

On to Objects

A Change of Pace

We've spent the past 10 weeks developing tools for defining and reasoning about programming language features.

Now it's time to use these tools for something more ambitious.

Case study: object-oriented programming

Plan:

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

The Translational Analysis

Our first goal will be to show how many of the basic features of object-oriented languages

- objects
- dynamic dispatch
- encapsulation of state
- inheritance
- self (this) and super
- late binding

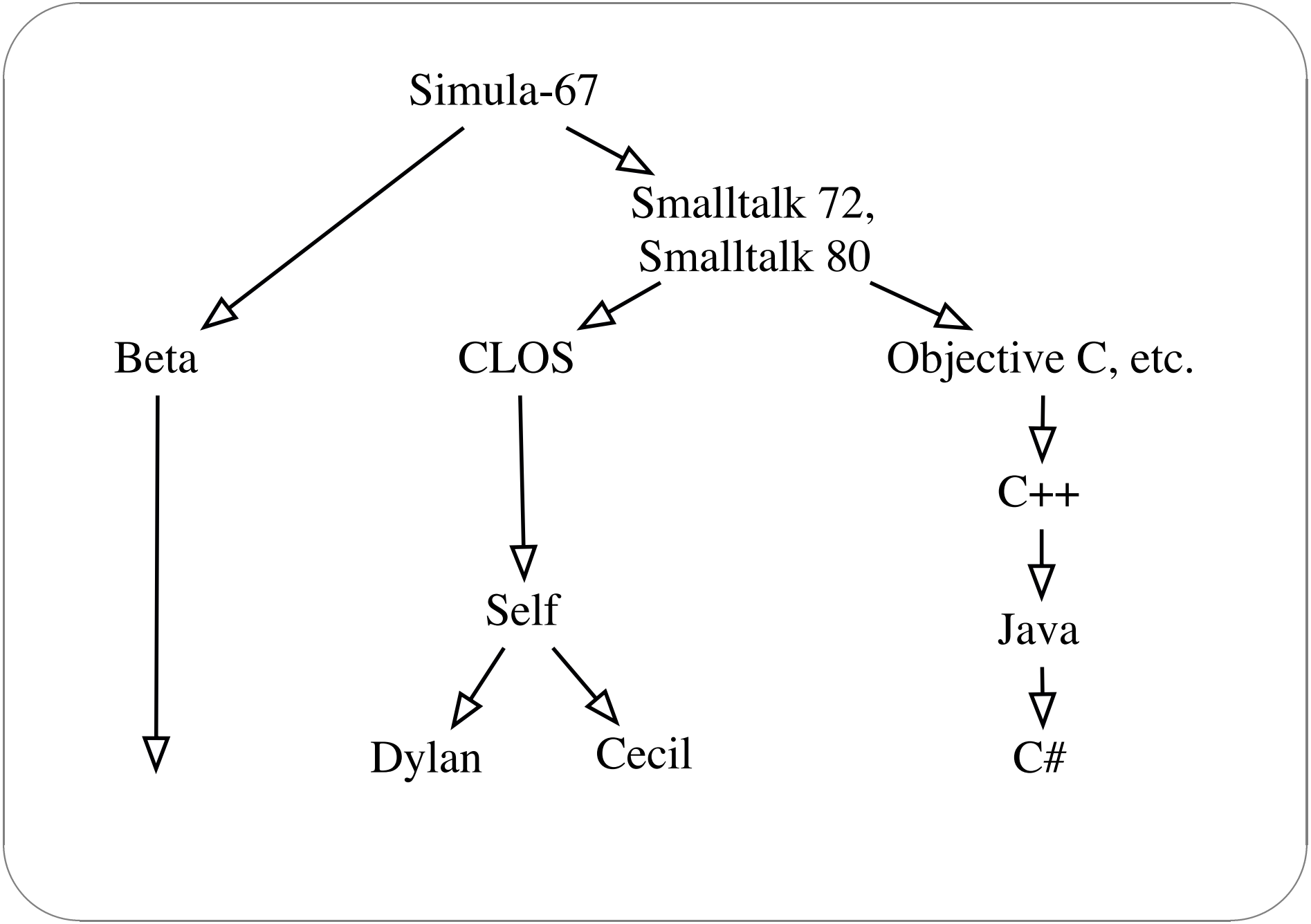
can be understood as "derived forms" in a lower-level language with a rich collection of primitive features:

- (higher-order) functions
- records
- references
- recursion
- subtyping

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.

History


Simula-67
Smalltalk 72,
Smalltalk 80
Beta
CLOS
Objective C, etc.
C++
Self
Java
Dylan
Cecil
C#

Concepts

The Essence of Objects

What "is" object-oriented programming?

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.

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.

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.

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

Side note: encapsulation

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.

These languages are based, instead, on multi-methods, a form of ad-hoc polymorphism.

Although their basic mechanisms are quite different, the higher-level programming idioms (classes, inheritance, etc.) arising in multi-method languages are surprisingly similar to those in "mainstream" OO languages.

Side note: Objects vs. ADTs

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).

Side note: 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 — no dynamic dispatch.

Both styles have advantages.

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.

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.

Object interfaces fit naturally into a subtype relation.

> An interface listing more operations is "better" than one listing fewer operations.

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`).

Inheritance

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

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

$\Longrightarrow$ inheritance

Inheritance

Basic mechanism of inheritance: classes

A class is a data structure that can be

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

N.b.: some OO languages offer an alternative (but fundamentally fairly similar) mechanism, called delegation, which allows new objects to be derived by refining the behavior of existing objects.

Late binding

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

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 subclass, then the behavior of the first method automatically changes as well.

Though quite useful in many situations, 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.

Getting down to details...

Objects

```
c = let x = ref 1 in
        {get = λ_:Unit. !x,
         inc = λ_:Unit. x:=succ(!x)};
```

$\Longrightarrow$ `c : Counter`

where

`Counter = {get:Unit→Nat, inc:Unit→Unit}`

Objects

```
inc3 = λc:Counter. (c.inc unit; c.inc unit; c.inc unit);
⟹ inc3 : Counter → Unit

(inc3 c; c.get unit);
⟹ 7
```

Object Generators

```
newCounter =
  λ_:Unit. let x = ref 1 in
             {get = λ_:Unit. !x,
              inc = λ_:Unit. x:=succ(!x)};

⟹ newCounter : Unit → Counter
```

Subtyping

```
ResetCounter = {get:Unit→Nat, inc:Unit→Unit, reset:Unit→Unit};


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

⟹ newResetCounter : Unit → ResetCounter
```

Subtyping

```
rc = newResetCounter unit;


(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);
⟹ 4
```

Grouping Instance Variables

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.

```
c = let r = {x=ref 1} in
      {get = λ_:Unit. !(r.x),
       inc = λ_:Unit. r.x:=succ(!(r.x))};

CounterRep = {x: Ref Nat};
```

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.

Reusing Methods

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

```
resetCounterFromCounter =
  λc:Counter. let x = ref 1 in
              {get   = c.get,
               inc   = c.inc,
               reset = λ_:Unit. x:=1};
```

Reusing Methods

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

```
resetCounterFromCounter =
  λc:Counter. let x = ref 1 in
              {get   = c.get,
               inc   = c.inc,
               reset = λ_:Unit. x:=1};
```

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

⟹ classes

Classes

A class is a run-time data structure that can be

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

Classes

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

```
counterClass =
  λr:CounterRep.
    {get = λ_:Unit. !(r.x),
     inc = λ_:Unit. r.x:=succ(!(r.x))};
⟹ counterClass : CounterRep → Counter
```

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

```
newCounter =
  λ_:Unit. let r = {x=ref 1} in
             counterClass r;
⟹ newCounter : Unit → Counter
```

Defining a Subclass

```
resetCounterClass =
  λr:CounterRep.
    let super = counterClass r in
      {get   = super.get,
       inc   = super.inc,
       reset = λ_:Unit. r.x:=1};
⟹ resetCounterClass : CounterRep → ResetCounter

newResetCounter =
  λ_:Unit. let r = {x=ref 1} in resetCounterClass r;
⟹ newResetCounter : Unit → ResetCounter
```

Adding instance variables

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

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


backupCounterClass =
  λr:BackupCounterRep.
    let super = resetCounterClass r in
       {get    = super.get,
        inc    = super.inc,
        reset  = λ_:Unit. r.x:=!(r.b),
        backup = λ_:Unit. r.b:=!(r.x)};
⟹ backupCounterClass : BackupCounterRep → BackupCounter
```

Notes:

- `backupCounterClass` both extends (with `backup`) and overrides (with a new `reset`) the definition of `counterClass`
- subtyping is essential here (in the definition of `super`)

```
backupCounterClass =
  λr:BackupCounterRep.
    let super = resetCounterClass r in
       {get    = super.get,
        inc    = super.inc,
        reset  = λ_:Unit. r.x:=!(r.b),
        backup = λ_:Unit. r.b:=!(r.x)};
```

Calling super

Suppose (for the sake of the example) that we wanted every call to `inc` to first back up the current state. We can avoid copying the code for `backup` by making `inc` use the `backup` and `inc` methods from `super`.

```
funnyBackupCounterClass =
  λr:BackupCounterRep.
    let super = backupCounterClass r in
       {get = super.get,
        inc = λ_:Unit. (super.backup unit; super.inc unit),
        reset = super.reset,
        backup = super.backup};

⟹ funnyBackupCounterClass : BackupCounterRep → BackupCounter
```