Programming Languages and Types

Klaus Ostermann

based on slides by Benjamin C. Pierce

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* 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:
   - 2.1 A *translation* into a lower-level language
   - 2.2 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

dynamic dispatch
encapsulation of state
inheritance
late binding (this)
super

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

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 `this`), it becomes less satisfactory, leading us to the more direct treatment in the following chapter.

Concepts

The Essence of Objects

What "is" object-oriented programming?

The Essence of Objects

What "is" object-oriented programming?

A precise definition has been the subject of debate for decades. Such arguments are always inconclusive and seldom interesting.

The Essence of Objects

What "is" object-oriented programming?

A precise definition has been the subject of debate for decades. Such arguments are always inconclusive and seldom 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 (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 (in Java)

```
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 that `(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 inaccessible from outside the object

Encapsulation

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

```
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; }
}
```

The code `(new B()).x` is not allowed.

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.

Caveat: 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 and Encapsulation

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`).

Example

```
// ... class A and subclasses B and C as above...

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

...

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

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 mechanism, called *delegation*, which allows new objects to be derived by refining the behavior of existing objects.

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 o() { x = x*10; return x; }
}
```

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

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" `this`. 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 appropriately. For this reason, it is sometimes deprecated in practice.

Examples

```
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; }
}
```

Quick check:

- 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

```
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(); }
}
```

What does `(new G()).n()` return?

Getting down to details (in the lambda-calculus)...

Simple objects with encapsulated state

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

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

Counter c = new Counter();
inc3(c);
inc3(c);
c.get();
```

How do we encode objects in the lambda-calculus?

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; 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
```

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.

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

The local variable `r` has type `CounterRep = {x: Ref Nat}`

Subtyping and Inheritance

```
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

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

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

$\Longrightarrow$ `newResetCounter : Unit → ResetCounter`

Subtyping

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

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 r = {x = ref 1} in
              {get   = c.get,
               inc   = c.inc,
               reset = λ_:Unit. r.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 r = {x = ref 1} in
              {get   = c.get,
               inc   = c.inc,
               reset = λ_:Unit. r.x:=1};
```

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

$\Longrightarrow$ 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
```

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; }
}
```

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)};
```

$\Longrightarrow$

```
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};
```

$\Longrightarrow$

```
funnyBackupCounterClass : BackupCounterRep → BackupCounter
```

Calling between methods

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

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

setCounterClass =
  λr:CounterRep.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. r.x:=(succ r.x) });
```

Calling between methods

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

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

setCounterClass =
  λr:CounterRep.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. r.x:=(succ r.x) });
```

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?

Calling between methods

In Java we would write:

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

Better...

```
setCounterClass =
  λr:CounterRep.
    fix
      (λthis: SetCounter.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. this.set (succ (this.get unit))});
```

Check: the type of the inner λ-abstraction is `SetCounter`$\rightarrow$`SetCounter`, so the type of the `fix` expression is `SetCounter`.

This is just a definition of a group of mutually recursive functions.

Note that the fixed point in

```
setCounterClass =
  λr:CounterRep.
    fix
      (λthis: SetCounter.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. this.set (succ (this.get unit))});
```

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

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

Idea: move the application of `fix` from the class definition...

```
setCounterClass =
  λr:CounterRep.
     λthis: SetCounter.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. this.set (succ(this.get unit))};
```

...to the object creation function:

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
                fix (setCounterClass r);
```

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

Note that we have changed the *types* of classes from...

```
setCounterClass =
  λr:CounterRep.
    fix
      (λthis: SetCounter.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. this.set (succ (this.get unit))});
```

$\Longrightarrow$ `setCounterClass : CounterRep → SetCounter`

... to:

```
setCounterClass =
  λr:CounterRep.
     λthis: SetCounter.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. this.set (succ(this.get unit))};
```

$\Longrightarrow$

`setCounterClass : CounterRep → SetCounter → SetCounter`

Using `this`

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: Ref Nat, a: Ref Nat};
```

```
instrCounterClass =
  λr:InstrCounterRep.
     λthis: InstrCounter.
       let super = setCounterClass r this in
         {get = super.get,
          set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
          inc = super.inc,
          accesses = λ_:Unit. !(r.a)};
⟹ instrCounterClass :
        InstrCounterRep → InstrCounter → InstrCounter
```

Notes:

- the methods use both `this` (which is passed as a parameter) and `super` (which is constructed using `this` and the instance variables)
- the `inc` in `super` will call the `set` defined here, which calls the superclass `set`
- suptyping plays a crucial role (twice) in the call to `setCounterClass`

One more refinement...

A small fly in the ointment

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

```
newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
                fix (instrCounterClass r);
```

will cause the evaluator to diverge!

Intuitively (see TAPL for details), the problem is the "unprotected" use of `this` in the call to `setCounterClass` in `instrCounterClass`:

```
instrCounterClass =
  λr:InstrCounterRep.
     λthis: InstrCounter.
       let super = setCounterClass r this in
          ...
```

To see why this diverges, consider a simpler example:

```
  ff = λf:Nat→Nat.
         let f′ = f in
         λn:Nat. 0
⟹ ff : (Nat→Nat) → (Nat→Nat)
```

Now:

```
     fix ff  ⟶  let f′ = (fix ff) in λn:Nat. 0
             ⟶  let f′ = ff (fix ff) in λn:Nat. 0
             ⟶  uh oh...
```

One possible solution

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

```
setCounterClass =
  λr:CounterRep.
  λthis: Unit→SetCounter.
    λ_:Unit.
      {get = λ_:Unit. !(r.x),
       set = λi:Nat.  r.x:=i,
       inc = λ_:Unit. (this unit).set
                                (succ((this unit).get unit))};
⟹ setCounterClass :
       CounterRep → (Unit→SetCounter) → (Unit→SetCounter)

newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
              fix (setCounterClass r) unit;
```

Similarly:

```
instrCounterClass =
  λr:InstrCounterRep.
  λthis: Unit→InstrCounter.
    λ_:Unit.
      let super = setCounterClass r this unit in
          {get = super.get,
           set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
           inc = super.inc,
           accesses = λ_:Unit. !(r.a)};

newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
               fix (instrCounterClass r) unit;
```

Success

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

Success (?)

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.

Recap

Multiple representations

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

But their internal representations vary widely.

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

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

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

- waits to be told a record `r` of instance variables and an object `this` (which should have the same interface and be based on the same record of instance variables)
- uses `r` and `this` to instantiate its superclass
- constructs a record of method implementations, copying some directly from `super` and implementing others in terms of `this` and `super`.

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

Where we are...

The essence of objects

- Dynamic dispatch
- Encapsulation of state with behavior
- Behavior-based subtyping
- Inheritance (incremental definition of behaviors)
- Access of super class
- "Open recursion" through `this`

What's missing (wrt. Java, say)

We haven't really captured the peculiar status of *classes* (which are both run-time and compile-time things) — we've captured the run-time aspect, but not the way in which classes get used as *types* in Java.

Also not *named* types with *declared* subtyping

Nor recursive types

Nor run-time type analysis (casting, etc.)

(... nor lots of other stuff)

Modeling Java

About models (of things in general)

No such thing as a "perfect model" — The nature of a model is to abstract away from details!

So models are never just "good" [or "bad"]: they are always "good [or bad] for some specific set of purposes."

Models of Java

Lots of different purposes $\longrightarrow$ lots of different kinds of models

- Source-level vs. bytecode level
- Large (inclusive) vs. small (simple) models
- Models of type system vs. models of run-time features (not entirely separate issues)
- Models of specific features (exceptions, concurrency, reflection, class loading, ...)
- Models designed for extension

Featherweight Java

Purpose: model "core OO features" and their types and *nothing else*.

History:

- Originally proposed by a Penn PhD student (Atsushi Igarashi) as a tool for analyzing GJ ("Java plus generics"), which later became Java 1.5
- Since used by many others for studying a wide variety of Java features and proposed extensions

Things left out

- Reflection, concurrency, class loading, inner classes, ...
- Exceptions, loops, ...
- Interfaces, overloading, ...
- Assignment (!!)

Things left in

- Classes and objects
- Methods and method invocation
- Fields and field access
- Inheritance (including open recursion through `this`)
- Casting

Example

```
class A extends Object { A() { super(); } }

class B extends Object { B() { super(); } }

class Pair extends Object {
  Object fst;
  Object snd;

  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd; }

  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); }
}
```

Conventions

For syntactic regularity...

- Always include superclass (even when it is `Object`)
- Always write out constructor (even when trivial)
- Always call `super` from constructor (even when no arguments are passed)
- Always explicitly name receiver object in method invocation or field access (even when it is `this`)
- Methods always consist of a single `return` expression
- Constructors always
  - Take same number (and types) of parameters as fields of the class
  - Assign constructor parameters to "local fields"
  - Call `super` constructor to assign remaining fields
  - Do nothing else

Formalizing FJ

Nominal type systems

Big dichotomy in the world of programming languages:

- *Structural* type systems:
  - What matters about a type (for typing, subtyping, etc.) is just its structure.
  - Names are just convenient (but inessential) abbreviations.
- *Nominal* type systems:
  - Types are always named.
  - Typechecker mostly manipulates names, not structures.
  - Subtyping is declared explicitly by programmer (and checked for consistency by compiler).

Advantages of Structural Systems

Somewhat simpler, cleaner, and more elegant (no need to always work wrt. a set of "name definitions")

Easier to extend (e.g. with parametric polymorphism)

(Caveat: when recursive types are considered, some of this simplicity and elegance slips away...)

Advantages of Nominal Systems

Recursive types fall out easily

Using names everywhere makes typechecking (and subtyping, etc.) easy and efficient

Type names are also useful at run-time (for casting, type testing, reflection, ...).

Clear (and compiler-checked) documentation of design intent; no accidential subtype relations.

Blame can be assigned properly if a subtype test fails.

Java (like most other mainstream languages) is a nominal system.

Representing objects

Our decision to omit assignment has a nice side effect...

The only ways in which two objects can differ are (1) their classes and (2) the parameters passed to their constructor when they were created.

All this information is available in the `new` expression that creates an object. So we can *identify* the created object with the `new` expression.

Formally: object values have the form `new C(`$\overline{\texttt{v}}$`)`

FJ Syntax

Syntax (terms and values)

`t` ::=		*terms*
	`x`	*variable*
	`t.f`	*field access*
	`t.m(`$\overline{\texttt{t}}$`)`	*method invocation*
	`new C(`$\overline{\texttt{t}}$`)`	*object creation*
	`(C) t`	*cast*
`v` ::=		*values*
	`new C(`$\overline{\texttt{v}}$`)`	*object creation*

Syntax (methods and classes)

`K` ::=		*constructor declarations*
	`C(`$\overline{\texttt{C}}$ $\overline{\texttt{f}}$`) {super(`$\overline{\texttt{f}}$`); this.`$\overline{\texttt{f}}$`=`$\overline{\texttt{f}}$`;}`	
`M` ::=		*method declarations*
	`C m(`$\overline{\texttt{C}}$ $\overline{\texttt{x}}$`) {return t;}`	
`CL` ::=		*class declarations*
	`class C extends C {`$\overline{\texttt{C}}$ $\overline{\texttt{f}}$`; K `$\overline{\texttt{M}}$`}`	

Subtyping

Subtyping

As in Java, subtyping in FJ is *declared*.

Assume we have a (global, fixed) *class table* CT mapping class names to definitions.

$$\frac{CT(\texttt{C}) = \texttt{class C extends D \{...\}}}{\texttt{C <: D}}$$

$$\texttt{C <: C}$$

$$\frac{\texttt{C <: D} \qquad \texttt{D <: E}}{\texttt{C <: E}}$$

More auxiliary definitions

From the class table, we can read off a number of other useful properties of the definitions (which we will need later for typechecking and operational semantics)...

Field(s) lookup

$$\mathit{fields}(\texttt{Object}) = \emptyset$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}}}$$

Method type lookup

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \quad \mathtt{B\ m\ (\overline{B}\ \overline{x})\ \{return\ t;\}} \in \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{B}} \rightarrow \mathtt{B}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \quad \mathtt{m}\text{ is not defined in } \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{C}) = mtype(\mathtt{m}, \mathtt{D})}$$

Method body lookup

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{B\ m\ (\overline{B}\ \overline{x})\ \{return\ t;\}} \in \overline{\mathtt{M}}}{mbody(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{t})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{m}\text{ is not defined in } \overline{\mathtt{M}}}{mbody(\mathtt{m}, \mathtt{C}) = mbody(\mathtt{m}, \mathtt{D})}$$

Valid method overriding

$$\frac{\mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{D}} \to \mathtt{D}_0 \text{ implies } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0}{\mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}_0)}$$

Evaluation

The example again

```
class A extends Object { A() { super(); } }

class B extends Object { B() { super(); } }

class Pair extends Object {
  Object fst;
  Object snd;

  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd; }

  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); }
}
```

Evaluation

Projection:

```
new Pair(new A(), new B()).snd   ⟶   new B()
```

Evaluation

Casting:

```
(Pair)new Pair(new A(), new B())
  ⟶  new Pair(new A(), new B())
```

Evaluation

Method invocation:

```
new Pair(new A(), new B()).setfst(new B())
```

$$\longrightarrow \left[\begin{array}{l} \texttt{newfst} \mapsto \texttt{new B()}, \\ \texttt{this} \mapsto \texttt{new Pair(new A(),new B())} \end{array}\right]$$

```
    new Pair(newfst, this.snd)
```

i.e., `new Pair(new B(), new Pair(new A(), new B()).snd)`

```
    ((Pair) (new Pair(new Pair(new A(),new B()), new A())
                .fst).snd
⟶  ((Pair)new Pair(new A(),new B())).snd
⟶  new Pair(new A(), new B()).snd
⟶  new B()
```

Evaluation rules

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}))\,.\mathtt{f}_i \longrightarrow \mathtt{v}_i} \qquad (\text{E-ProjNew})$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{t}_0)}{(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}))\,.\mathtt{m}(\overline{\mathtt{u}}) \longrightarrow [\overline{\mathtt{x}} \mapsto \overline{\mathtt{u}}, \mathtt{this} \mapsto \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})]\mathtt{t}_0} \qquad (\text{E-InvkNew})$$

$$\frac{\mathtt{C} <: \mathtt{D}}{(\mathtt{D})(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})) \longrightarrow \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})} \qquad (\text{E-CastNew})$$

plus some congruence rules...

$$\frac{t_0 \longrightarrow t_0'}{t_0.f \longrightarrow t_0'.f} \quad \text{(E-Field)}$$

$$\frac{t_0 \longrightarrow t_0'}{t_0.m(\overline{t}) \longrightarrow t_0'.m(\overline{t})} \quad \text{(E-Invk-Recv)}$$

$$\frac{t_i \longrightarrow t_i'}{v_0.m(\overline{v},\ t_i,\ \overline{t}) \longrightarrow v_0.m(\overline{v},\ t_i',\ \overline{t})} \quad \text{(E-Invk-Arg)}$$

$$\frac{t_i \longrightarrow t_i'}{\texttt{new C}(\overline{v},\ t_i,\ \overline{t}) \longrightarrow \texttt{new C}(\overline{v},\ t_i',\ \overline{t})} \quad \text{(E-New-Arg)}$$

$$\frac{t_0 \longrightarrow t_0'}{(C)t_0 \longrightarrow (C)t_0'} \quad \text{(E-Cast)}$$

Typing

Typing rules

$$\frac{\mathtt{x}:\mathtt{C} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{C}} \qquad \text{(T-Var)}$$

Typing rules

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{f}_i : \mathtt{C}_i} \qquad \text{(T-Field)}$$

Typing rules

$$\frac{\Gamma \vdash t_0 : D \qquad D <: C}{\Gamma \vdash (C)t_0 : C} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash t_0 : D \qquad C <: D \qquad C \neq D}{\Gamma \vdash (C)t_0 : C} \qquad \text{(T-DCast)}$$

Why two cast rules?

Typing rules

$$\frac{\Gamma \vdash t_0 : D \qquad D <: C}{\Gamma \vdash (C)t_0 : C} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash t_0 : D \qquad C <: D \qquad C \neq D}{\Gamma \vdash (C)t_0 : C} \qquad \text{(T-DCast)}$$

Why two cast rules? Because that's how Java does it!

Typing rules

$$\frac{\Gamma \vdash t_0 : C_0 \qquad mtype(m, C_0) = \overline{D} \to C \qquad \Gamma \vdash \overline{t} : \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash t_0.m(\overline{t}) : C} \quad \text{(T-Invk)}$$

Note that this rule "has subsumption built in" — i.e., the typing relation in FJ is written in the *algorithmic* style of TAPL chapter 16, not the declarative style of chapter 15.

Typing rules

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \qquad \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) : \mathtt{C}} \quad \text{(T-Invk)}$$

Note that this rule "has subsumption built in" — i.e., the typing relation in FJ is written in the *algorithmic* style of TAPL chapter 16, not the declarative style of chapter 15.

Why? Because Java does it this way!

Typing rules

$$\frac{\Gamma \vdash t_0 : C_0 \qquad mtype(m, C_0) = \overline{D} \to C \qquad \Gamma \vdash \overline{t} : \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash t_0.m(\overline{t}) : C} \quad \text{(T-Invk)}$$

Note that this rule "has subsumption built in" — i.e., the typing relation in FJ is written in the *algorithmic* style of TAPL chapter 16, not the declarative style of chapter 15.

Why? Because Java does it this way!

But why does Java do it this way??

Java typing is algorithmic

The Java typing relation is defined in the algorithmic style, for (at least) two reasons:

1. In order to perform static *overloading resolution*, we need to be able to speak of "the type" of an expression
2. We would otherwise run into trouble with typing of conditional expressions

Let's look at the second in more detail...

Java typing must be algorithmic

We haven't included them in FJ, but full Java has both *interfaces* and *conditional expressions*.

The two together actually make the declarative style of typing rules unworkable!

Java conditionals

$$\frac{t_1 \in \texttt{bool} \qquad t_2 \in T_2 \qquad t_3 \in T_3}{t_1 \ \texttt{?} \ t_2 \ \texttt{:} \ t_3 \in \texttt{?}}$$

Java conditionals

$$\frac{\mathtt{t}_1 \in \mathtt{bool} \qquad \mathtt{t}_2 \in \mathtt{T}_2 \qquad \mathtt{t}_3 \in \mathtt{T}_3}{\mathtt{t}_1 \ \mathtt{?}\ \mathtt{t}_2 \ \mathtt{:}\ \mathtt{t}_3 \in \mathtt{?}}$$

Actual Java rule (algorithmic):

$$\frac{\mathtt{t}_1 \in \mathtt{bool} \qquad \mathtt{t}_2 \in \mathtt{T}_2 \qquad \mathtt{t}_3 \in \mathtt{T}_3}{\mathtt{t}_1 \ \mathtt{?}\ \mathtt{t}_2 \ \mathtt{:}\ \mathtt{t}_3 \in \mathit{min}(\mathtt{T}_2, \mathtt{T}_3)}$$

Java conditionals

More standard (declarative) rule:

$$\frac{\texttt{t}_1 \in \texttt{bool} \qquad \texttt{t}_2 \in \texttt{T} \qquad \texttt{t}_3 \in \texttt{T}}{\texttt{t}_1 \ \texttt{?} \ \texttt{t}_2 \ \texttt{:} \ \texttt{t}_3 \in \texttt{T}}$$

Java conditionals

More standard (declarative) rule:

$$\frac{t_1 \in \texttt{bool} \qquad t_2 \in T \qquad t_3 \in T}{t_1 \ \texttt{?} \ t_2 \ \texttt{:} \ t_3 \in T}$$

Algorithmic version:

$$\frac{t_1 \in \texttt{bool} \qquad t_2 \in T_2 \qquad t_3 \in T_3}{t_1 \ \texttt{?} \ t_2 \ \texttt{:} \ t_3 \in T_2 \vee T_3}$$

Requires joins!

Java has no joins

But, in full Java (with interfaces), there are types that have no join!

E.g.:

```
interface I {...}
interface J {...}
interface K extends I,J {...}
interface L extends I,J {...}
```

`K` and `L` have no join (least upper bound) — both `I` and `J` are common upper bounds, but neither of these is less than the other.

So: algorithmic typing rules are really our only option.

FJ Typing rules

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{t}}) : \mathtt{C}} \qquad \text{(T-New)}$$

Typing rules (methods, classes)

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{t}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 \mathtt{<:} \mathtt{C}_0 \qquad CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{...\}} \qquad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}_0)}{\mathtt{C}_0\ \mathtt{m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\mathtt{return\ t}_0\mathtt{;}\}\ \mathtt{OK\ in\ C}}$$

$$\frac{\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{g}})\mathtt{;}\ \mathtt{this.}\overline{\mathtt{f}} = \overline{\mathtt{f}}\mathtt{;}\} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ in\ C}}{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}}$$

Properties

Progress

Progress

Problem: well-typed programs *can* get stuck.

How?

Progress

Problem: well-typed programs *can* get stuck.

How?

Cast failure:

```
(A)(new Object())
```

Formalizing Progress

Solution: Weaken the statement of the progress theorem to

> *A well-typed FJ term is either a value or can reduce one step* or *is stuck at a failing cast.*

Formalizing this takes a little more work...

Evaluation Contexts

E ::=		*evaluation contexts*
	$[]$	*hole*
	$E.\mathtt{f}$	*field access*
	$E.\mathtt{m}(\overline{\mathtt{t}})$	*method invocation (rcv)*
	$\mathtt{v}.\mathtt{m}(\overline{\mathtt{v}}, E, \overline{\mathtt{t}})$	*method invocation (arg)*
	$\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}, E, \overline{\mathtt{t}})$	*object creation (arg)*
	$(\mathtt{C})E$	*cast*

Evaluation contexts capture the notion of the "next subterm to be reduced," in the sense that, if $\mathtt{t} \longrightarrow \mathtt{t}'$, then we can express $\mathtt{t}$ and $\mathtt{t}'$ as $\mathtt{t} = E[\mathtt{r}]$ and $\mathtt{t}' = E[\mathtt{r}']$ for a unique E, $\mathtt{r}$, and $\mathtt{r}'$, with $\mathtt{r} \longrightarrow \mathtt{r}'$ by one of the computation rules E-ProjNew, E-InvkNew, or E-CastNew.

Progress

Theorem [Progress]: Suppose $\mathtt{t}$ is a closed, well-typed normal form. Then either (1) $\mathtt{t}$ is a value, or (2) $\mathtt{t} \longrightarrow \mathtt{t}'$ for some $\mathtt{t}'$, or (3) for some evaluation context E, we can express $\mathtt{t}$ as $\mathtt{t} = E[\mathtt{(C)(new\ D(\overline{v}))}]$, with $\mathtt{D} \not<: \mathtt{C}$.

Preservation

Theorem [Preservation]: If $\Gamma \vdash \mathtt{t} : \mathtt{C}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathtt{C}'$ for some $\mathtt{C}' <: \mathtt{C}$.

Proof: Straightforward induction.

Preservation

Theorem [Preservation]: If $\Gamma \vdash \mathtt{t} : \mathtt{C}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathtt{C}'$ for some $\mathtt{C}' <: \mathtt{C}$.

Proof: Straightforward induction. *???*

Preservation?

Preservation?

Surprise: well-typed programs *can* step to ill-typed ones!

(How?)

Preservation?

Surprise: well-typed programs *can* step to ill-typed ones!

(How?)

$$\texttt{(A)}\underline{\texttt{(Object)new B()}} \longrightarrow \texttt{(A)new B()}$$

Solution: "Stupid Cast" typing rule

Add another typing rule, marked "stupid" to indicate that an implementation should generate a warning if this rule is used.

$$\frac{\Gamma \vdash t_0 : \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \qquad \textit{stupid warning}}{\Gamma \vdash (\mathtt{C})t_0 : \mathtt{C}} \qquad \text{(T-SCast)}$$

Solution: "Stupid Cast" typing rule

Add another typing rule, marked "stupid" to indicate that an implementation should generate a warning if this rule is used.

$$\frac{\Gamma \vdash t_0 : \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \qquad \textit{stupid warning}}{\Gamma \vdash (\mathtt{C})t_0 : \mathtt{C}} \qquad \text{(T-SCast)}$$

This is an example of a modeling technicality; not very interesting or deep, but we have to get it right if we're going to claim that the model is an accurate representation of (this fragment of) Java.

Correspondence with Java

Let's try to state precisely what we mean by "FJ corresponds to Java":

Claim:

1. Every syntactically well-formed FJ program is also a syntactically well-formed Java program.
2. A syntactically well-formed FJ program is typable in FJ (without using the T-SCAST rule.) iff it is typable in Java.
3. A well-typed FJ program behaves the same in FJ as in Java. (E.g., evaluating it in FJ diverges iff compiling and running it in Java diverges.)

Of course, without a formalization of full Java, we cannot *prove* this claim. But it's still very useful to say precisely what we are trying to accomplish—e.g., it provides a rigorous way of judging counterexamples.

Alternative approaches to casting

- Loosen preservation theorem
- Use big-step semantics

More on Evaluation Contexts

Progress for FJ

Theorem [Progress]: Suppose $\mathtt{t}$ is a closed, well-typed normal form. Then either

1. $\mathtt{t}$ is a value, or
2. $\mathtt{t} \longrightarrow \mathtt{t}'$ for some $\mathtt{t}'$, or
3. for some evaluation context E, we can express $\mathtt{t}$ as

$$\mathtt{t} = E[\mathtt{(C)(new\ D(\overline{v}))}]$$

with $\mathtt{D} \not<: \mathtt{C}$.

Evaluation Contexts

E ::=	*evaluation contexts*
$[]$	*hole*
E.f	*field access*
E.m($\overline{\text{t}}$)	*method invocation (rcv)*
v.m($\overline{\text{v}}$,E,$\overline{\text{t}}$)	*method invocation (arg)*
new C($\overline{\text{v}}$,E,$\overline{\text{t}}$)	*object creation (arg)*
(C)E	*cast*

E.g.,

```
[].fst
[].fst.snd
new C(new D(), [].fst.snd, new E())
```

Evaluation Contexts

E[t] denotes "the term obtained by filling the hole in E with t."

E.g., if E = (A)[], then

$$E[\texttt{(new Pair(new A(), new B())).fst}]$$
$$=$$
$$\texttt{(A)((new Pair(new A(), new B())).fst)}$$

Evaluation Contexts

Evaluation contexts capture the notion of the "next subterm to be reduced":

- By ordinary evaluation relation:

  $$\texttt{(A)((new Pair(new A(), new B())).fst)} \longrightarrow \texttt{(A)(new A())}$$

  by E-CAST with subderivation E-PROJNEW.

- By evaluation contexts:

  $E = \texttt{(A)[]}$
  $\texttt{r} = \texttt{(new Pair(new A(), new B())).fst}$
  $\texttt{r}' = \texttt{new A()}$
  $\texttt{r} \longrightarrow \texttt{r}'$ by E-PROJNEW
  $E[\texttt{r}] = \texttt{(A)((new Pair(new A(), new B())).fst)}$
  $E[\texttt{r}'] = \texttt{(A)(new A())}$

Precisely...

Claim 1: If $\mathtt{r} \longrightarrow \mathtt{r}'$ by one of the computation rules E-ProjNew, E-InvkNew, or E-CastNew and E is an arbitrary evaluation context, then $E[\mathtt{r}] \longrightarrow E[\mathtt{r}']$ by the ordinary evaluation relation.

Claim 2: If $\mathtt{t} \longrightarrow \mathtt{t}'$ by the ordinary evaluation relation, then there are unique E, $\mathtt{r}$, and $\mathtt{r}'$ such that

1. $\mathtt{t} = E[\mathtt{r}]$,
2. $\mathtt{t}' = E[\mathtt{r}']$, and
3. $\mathtt{r} \longrightarrow \mathtt{r}'$ by one of the computation rules E-ProjNew, E-InvkNew, or E-CastNew.

Hence...

Evaluation contexts are an alternative to congruence rules: Just add the rule $\frac{\mathtt{r} \longrightarrow \mathtt{r}'}{E[\mathtt{r}] \longrightarrow E[\mathtt{r}']}$.

Evaluation contexts are also quite useful for formalizing advanced control operators - the evaluation context is a representation of the current continuation.

They are also useful to formulate contextual/observational equivalence of terms.