Announcements

My office hours this week: Friday 1:30-3:00 PM

Object Encodings

Example from last time:

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

```java
class InstrumentCounter extends SetCounter {
  protected int a = 0;
  void set(int i) { a++; super.set(i); // protected
  int accesses() { return a; }
  void inc() { this.set(this.get() + 1); // protected
  }
}
```

We have a little more to talk about this topic, but let's work through an example to see where we are.

We have a lot more to talk about this topic, but let's work through an example to see where we are.

Records, recursion, references and subtyping.

Last time, we talked about object encodings in the typed lambda calculus with

Object Encodings
A small fly in the ointment

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

\[
\text{newInstrCounter} = \text{_:Unit}.\text{let r={x=ref1, a=ref0} in fix(instrCounterClassr)};
\]

will cause the evaluator to diverge! Intuitively (see TAPL for details), the problem is the "unprotected" use of self in the call to setCounterClass in instrCounterClass:

\[
\text{let super = setCounterClassrself in...}
\]

One more refinement...
Toseewhythisdiverges,considerasimplerexample:

```
f: Nat ! Nat.

let f_0 = fin

ff: (Nat ! Nat) ! (Nat ! Nat)
```

Now:

```c
fix ff
```

uhoh...

One possible solution

Idea: "dealy" set by putting a dummy abstraction in front of it.

Similarity:

Similarly:

```
instrCounterClass = r:InstrCounterRep.
self:Unit ! InstrCounter.
let super = setCounterClass r self

{get = super.get,
set = i:Nat. (r.a := succ(!(r.a)); super.set i),
inc = super.inc,
accesses = _:Unit.!(r.a)}
```

Success

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

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.

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.
Success

This works, in the sense that we can now instantiate `IncrementalCounter`. However, all the “delaying” we added has an unintended 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. Nonetheless, the “method table” is accessible outside of the object because there is no way to name it. Instead of `fix`, let’s use a record of mutable instance variables.

Recap

Encapsulation

Multiple representations

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

uses r and self to instantiate its superclass

constructs 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.

The (an) essence of objects

- Multiple representations
- Encapsulation of state with behavior
- Subtyping
- Inheritance (incremental definition of behaviors)
- “Open recursion” through self

Where we are...
What’s missing

The peculiar status of classes (which are both run-time and compile-time things)

- Named types with declared subtyping
- Recursive types
- Run-time type analysis (casting, etc.)
- ...lots of other stuff

Models of Java

- Lots of different purposes → lots of different kinds of models
  - Source-level vs. byte-code 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

No such thing as a “perfect model” — the nature of a model is to abstract away from details!

So models are never just “good”; they are always “good for some specific set of purposes.”

Models in General

- Lotsofdierentpurposes
  - Lotsofdierentkindsofmodels
    - Source-levelvs.byte-codelevel
    - 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

ModelsinGeneral

- Lotsofdierentpurposes
  - Lotsofdierentkindsofmodels
    - Source-levelvs.byte-codelevel
    - 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

ModelsinGeneral
FeatherweightJava

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

History:

Originally proposed by a Penn PhD student (Vuvinh Tranh) as a tool for analyzing C++ "Java plus Generics"

Frequently used by many others for studying a wide variety of Java features and

... Exceptions, loops, Reflection, concurrency, class loading, inner classes, ...

Things left out

... Reflection, concurrency, class loading, inner classes, ...

Exceptions, loops, ...

Things left out

... Reflection, concurrency, class loading, inner classes, ...

Exceptions, loops, ...

Things left out

... Reflection, concurrency, class loading, inner classes, ...

Exceptions, loops, ...

Things left out
Things left out

Conventions
- Do nothing else
- Call super constructor to assign remaining fields
- Assign constructor parameters to local fields
- Take the same number (and types) of parameters as fields of the class
- Constructors always consist of a single return expression (even when it is this)
- Methods always consist of a single return expression

Example

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

Conventions
- For syntactic regularity...
- Always includes superclass (even when it is Object)
- Always write out constructor (even when no arguments are passed)
- Always explicitly name receiver object in method invocation or field access
- Always call super from constructor (even when no arguments are passed)
- Always write out constructor (even when trivial)
- Always include superclass (even when it is Object)
- Methods always consist of a single return expression
- Constructors always take a number (and types) of parameters as fields of the class
- Assign constructor parameter to local fields
- Call super constructor to assign remaining fields
- "When inheritance is needed, open recursion through this"

Things left in

Cases
- Assignment (ii)
- Inheritance, overriding
- "..."
- Exceptions, loops, ...
Nominal types systems

Big dichotomy in the world of programming languages:

1. Structural types systems:
   - What matters about a type (for typing, subtyping, etc.) is just its structure.
   - Names are just convenient (but essential) abbreviations.

2. Nominal types systems:
   - Types are always named.
   - Typing names everywhere makes subtyping and checking efficient.

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

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

In many systems everywhere makes subtyping and checking easy and efficient.

Recursive types fall out easily.

Advantages of Structural Systems

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

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

Advantages of Nominal Systems

Recursive types fall out easily.

Using names everywhere makes typing and subtyping easy and efficient.

Typenames are also useful at run-time (for casting, testing, reflection, ...).

Some nominal systems cleaner, and more elegant (no need to always work with a set of "name definitions"), and more elegant.

Formalizing FJ
Representing objects

Our decision to omit assignment has an 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.

Syntax (terms and values)

\[
\begin{align*}
t &::= \text{terms} \\
x &::= \text{variable} \\
t.f &::= \text{field access} \\
t.m(t) &::= \text{method invocation} \\
\text{newC}(t) &::= \text{object creation} \\
(C)t &::= \text{cast}
\end{align*}
\]

Syntax (methods and classes)

\[
\begin{align*}
K &::= \text{constructor declarations} \\
M &::= \text{method declarations} \\
CL &::= \text{class declarations}
\end{align*}
\]

\[
\begin{align*}
\text{class C extends C \{ I \ X \}} \\
\text{class declarations} &::= C \\
\text{method declarations} &::= M \\
\text{constructor declarations} &::= K
\end{align*}
\]

Formally: object values have the form new (\(A\))
As in Java, subtyping in FJ is declared. Assume we have a (global, fixed) class table CT mapping class names to definitions.

\[
\begin{align*}
C<T &< D, \quad C<T &< C, \quad C<T &< E, \\
D<T &< E, \quad D<T &< D, \quad D<T &< E
\end{align*}
\]

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)...

Fields lookup

\[
\begin{align*}
\text{fields}(\text{object}) &\rightarrow \emptyset \\
\text{fields}(D) &\rightarrow D \in I \\
\text{fields}(C) &\rightarrow \text{fields}(D) \rightarrow D \in I
\end{align*}
\]
Valid method overriding

\[ \text{mtype}(\text{m}; \text{D}) = \text{D} \rightarrow \text{implies} \quad \text{C} = \text{D} \quad \text{and} \quad \text{C}_0 = \text{D}_0 \]

\[ \text{mbody}(\text{m}; \text{C}) = \text{mbody}(\text{m}; \text{D}) \]

Evaluation

Method body lookup

\[ \text{methodbodylookup}(\text{C}) = \text{class C extends D} \{ \text{f; } \text{M} \} \]

\[ \text{Bm}(\text{B}; \text{x}) = \text{return t; } \]

\[ \text{mbody}(\text{m}; \text{C}) = \text{x; t} \]

\[ \text{mbody}(\text{m}; \text{D}) = \text{x; t} \]

Valid method lookup

\[ \text{methodlookup}(\text{C}) = \text{class C extends D} \{ \text{f; } \text{M} \} \]

\[ \text{Bm}(\text{B}; \text{x}) = \text{return t; } \]

\[ \text{mbody}(\text{m}; \text{C}) = \text{x; t} \]

\[ \text{mbody}(\text{m}; \text{D}) = \text{x; t} \]
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())

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

Evaluation

new Pair(new A(), new B()) → new Pair(new A(), new B())
Typhing

Evaluation Rules

Typing

plus some congruence rules...

(E-CAST)
\[ (c)_\alpha \leftarrow (d)_\beta \]
\[ \gamma \leftarrow \alpha \]
\[ \delta \leftarrow \beta \]
\[ c \rightarrow d \]

(E-NEW-ARG)
\[ \gamma \leftarrow \alpha \]
\[ \delta \leftarrow \beta \]
\[ new(C, \alpha, \beta, \gamma, \delta) \leftarrow new(C, \alpha, \beta, \gamma, \delta) \]

(E-INVK-ARG)
\[ \gamma \leftarrow \alpha \]
\[ \delta \leftarrow \beta \]
\[ v_0 \cdot new(C, \alpha, \beta, \gamma, \delta) \leftarrow v_0 \cdot new(C, \alpha, \beta, \gamma, \delta) \]

(E-INVK-Recv)
\[ \gamma \leftarrow \alpha \]
\[ \delta \leftarrow \beta \]
\[ t_0 \cdot new(C, \alpha, \beta, \gamma, \delta) \leftarrow t_0 \cdot new(C, \alpha, \beta, \gamma, \delta) \]

(E-INVK-NEW)
\[ \gamma \leftarrow \alpha \]
\[ \delta \leftarrow \beta \]
\[ new(C, \alpha, \beta, \gamma, \delta) \leftarrow new(C, \alpha, \beta, \gamma, \delta) \]

(E-NEW-NEW)
\[ \gamma \leftarrow \alpha \]
\[ \delta \leftarrow \beta \]
\[ new(C, \alpha, \beta, \gamma, \delta) \leftarrow new(C, \alpha, \beta, \gamma, \delta) \]

(E-CAST)
\[ \gamma \leftarrow \alpha \]
\[ \delta \leftarrow \beta \]
\[ new(C, \alpha, \beta, \gamma, \delta) \leftarrow new(C, \alpha, \beta, \gamma, \delta) \]

(E-INVK)
\[ \gamma \leftarrow \alpha \]
\[ \delta \leftarrow \beta \]
\[ new(C, \alpha, \beta, \gamma, \delta) \leftarrow new(C, \alpha, \beta, \gamma, \delta) \]

(E-NEW)
\[ \gamma \leftarrow \alpha \]
\[ \delta \leftarrow \beta \]
\[ new(C, \alpha, \beta, \gamma, \delta) \leftarrow new(C, \alpha, \beta, \gamma, \delta) \]
Why two cast rules?

(T-cast)

\[ L \vdash c : C \quad \text{fields}(C) \subseteq \{ f \} \]

\[ L \vdash f : C_0 \quad \text{fields}(C) \subseteq \{ f \} \]

Typing rules

(T-Field)

\[ L \vdash c : C \quad \text{fields}(C) \subseteq \{ f \} \]

\[ L \vdash f : C_0 \quad \text{fields}(C) \subseteq \{ f \} \]

Typing rules

(T-Cast)

\[ L \vdash (c : C) \]

\[ L \vdash \text{fields}(C) \subseteq \{ f \} \]

\[ L \vdash f : C_0 \]

Typing rules

(T-Var)

\[ L \vdash x : C \]

\[ x \in L \]

Typing rules

Notes

FJ has no rule of substitution (because we want to follow Java). The typing rules are algorithmic.

Where would this make a difference?...
Typing rules

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

Why does Java do it this way?

Why? Because Java does it this way!

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.

But why does Java do it this way?

Why? Because Java does it this way!

Typing rules

Typing rules

Typing rules

Typing rules
Java typing must be algorithmic!

The two together actually make the declarative style of typing rules unwieldy.

We haven’t included them in P4, but full Java has both interfaces and method signatures.

### Java conditionals

<table>
<thead>
<tr>
<th>truth</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Actual Java rule (algorithmic):

```
if (t1) {
  t2;
} else {
  t3;
}
```

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

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

```
if (t1) {
  t2;
} else {
  t3;
}
```

We haven’t included them in P4, but full Java has both interfaces and method signatures.

### Java conditionals

<table>
<thead>
<tr>
<th>truth</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Actual Java rule (algorithmic):

```
if (t1) {
  t2;
} else {
  t3;
}
```

Java conditionals

<table>
<thead>
<tr>
<th>truth</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Actual Java rule (algorithmic):

```
if (t1) {
  t2;
} else {
  t3;
}
```

Java conditionals

<table>
<thead>
<tr>
<th>truth</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Actual Java rule (algorithmic):

```
if (t1) {
  t2;
} else {
  t3;
}
```

Java conditionals

<table>
<thead>
<tr>
<th>truth</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Actual Java rule (algorithmic):

```
if (t1) {
  t2;
} else {
  t3;
}
```

Java conditionals

<table>
<thead>
<tr>
<th>truth</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Actual Java rule (algorithmic):

```
if (t1) {
  t2;
} else {
  t3;
}
```

Java conditionals

<table>
<thead>
<tr>
<th>truth</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Actual Java rule (algorithmic):

```
if (t1) {
  t2;
} else {
  t3;
}
```

Java conditionals

<table>
<thead>
<tr>
<th>truth</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
More standard (declarative) rules:

Algorithmic version:

Requires joins:

Algorithmic version:

More standard (declarative) rules:

Java has no joins.
Typing rules (methods, classes)

\[
\begin{array}{l}
\frac{}{\text{this} : C \vdash t : E_0 \quad E_0 <: C_0} & \text{C}^T(C) - \text{class } C \text{ extends } D \ldots \\
\frac{\text{override}(m, D, C \rightarrow C_0)}{m : C \vdash t : E} & \text{\text{field}(D) - } D \vdash m : E \text{ ok in } C \\
\frac{C \rightarrow C \rightarrow (\text{super}(g), C \rightarrow F)}{F = \text{class } C \text{ extends } D \ldots \text{ok } C} & \text{CIS500, 1 December} \\
\end{array}
\]

Theorem [Preservation]: If \( \Gamma \vdash t : C \) and \( t \rightarrow t' \), then \( \Gamma \vdash t' : C' \) for some \( C' <: C \).

Proof: Straightforward induction.

Properties

Preservation

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

Proof: Straightforward induction.
Preservation?

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

(How?)

Solution: “Stupid Cast” typing rule

Add another typing rule, marked “stupid” to

\[
\Gamma \vdash t_0 : D \\
\Gamma \vdash \text{stupid warning} \\
\Gamma \vdash (C)t_0 : C
\]

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.
Solution: "Stupid Case" typing rule

Alternative approaches to casting

Loosen presentation theorem

Use big-step semantics

Progress

Correspondence with Java

Alternative approaches to casting

Loosen presentation theorem

Use big-step semantics

Progress

Correspondence with Java
Problem: well-typed programs can get stuck. How?

Solution: Weaken the statement of the progress theorem to

\[ \text{progress} \]
Theorem \[\text{Progress}\]: Suppose \( t \) is a closed, well-typed normal form. Then either (1) \( t \) is a value, or (2) \( t \rightarrow t_0 \) for some \( t_0 \), or (3) for some evaluation context \( E \), we can express \( t \) as \( E[\text{new} D(v)] \) with \( D \not< C \).