objectEncodings

Last time, we talked about encoding objects in the typed lambda calculus with

Example from last time

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

```
class InstrCounter extends SetCounter{
  protected int a = 0;
  void set(int i){a++; super.set(i); return;}
  int accesses(){return a;}
}
```

```
fix (setCounterClass r) =
  let r = refNat in
  newSetCounter =
    (get: Unit ! Nat, set: Nat ! Unit, inc: Unit ! Unit). setCounterClass r
```

We have a little more to talk about this topic, but let's work through an

Exercise: recursion, references and subtyping.
Asmall fly in the ointment

To see why this diverges, consider a simpler example:

One more refinement...
One possible solution

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

```latex
\begin{center}
\begin{verbatim}
setCounterClass = r:CounterRep.
this:Unit ! SetCounter.
_ : Unit.
{ get = _ : Unit. !(r.x),
set = i:Nat.r.x:=i,
inc = _ : Unit. (thisunit).set(succ((thisunit).getunit)) };
\end{verbatim}
\end{center}
```

Similarly:

```latex
\begin{center}
\begin{verbatim}
InstrCounterClass = r:InstrCounterRep.
this:Unit ! InstrCounter.
_ : Unit.
let super = setCounterClassrthisunit in
{ get = super.get,
set = i:Nat.(r.a:=succ(!(r.a));super.seti),
inc = super.inc,
accesses = _ : Unit.!(r.a) };
\end{verbatim}
\end{center}
```

Success

This works, in the sense that we can now instantiate \texttt{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!

This is out of the question, and we will have to fix this somehow.

Section 18.12 in TAPL shows how this can be done by using references instead of \texttt{fix}.
Multiplerepresentations

All the objects we have built in this series of examples have type `Counter`. Functions like `inc3` that expect `Counter` objects as parameters can (safely) be called with objects belonging to any subtype of `Counter`. This style is inaccessible outside of the object because there is no way to name a shared reference to a record of mutable instance variables. An object is a record of functions, which maintain common internal state via a set called with objects belonging to any subtype of `Counter`. Functions like `inc3` that expect `Counter` objects as parameters can (safely) be called with objects belonging to any subtype of `Counter`. This style is inaccessible outside of the object because there is no way to name a shared reference to a record of mutable instance variables. An object is a record of functions, which maintain common internal state via a set
Inheritance

Classes are data structures that can be both extended and instantiated. We modeled inheritance by copying implementations of methods from superclasses to subclasses. The peculiar status of classes (which are both run-time and compile-time things) should have the same interface and be based on the same record of instance variables. Where we are...
Modeling Java
CIS 500, November 30, 2021

Models in General

- Model in general:
  - "No such thing as a perfect model" — The nature of a model is to abstract

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 the core Java features and their types and nothing else

Featherweight Java

CIS 500, November 30, 2021
Things left out

Assignment (!!)  

... Inheritance, overloading...

... Exceptions, loops, ...

... Inheritance, concurrency, class loading, inner classes, ...
Formalizing FJ

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

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

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

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);
    }

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

Conventions

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

For syntactic regularity:

- Methods always consist of a single `return` expression
- Constructors always
  - Takes same number (and types) of parameters as fields of the class
  - Assign constructor parameter to local fields
  - Call `super` constructor to assign remaining fields
  - Do nothing else

Inheritance (including open recursion through `this`)

Fields and field access

Methods and method invocation

Classes and objects

Things Left In

- Caching
- Inheritance (including open recursion through `this`)
- Fields and field access
- Methods and method invocation
- Classes and objects
Nominal types systems: Types are always named. Typenames occur everywhere needed in programs (and substitutions, etc.) and easy to extend. Recursively types hold only exactly.

Advantages of Structural Systems

- Somewhat simpler, cleaner, and more efficient to work with.
- Faster to extend (e.g. with parameter polymorphism) set of "name definitions." (e.g. some simplifying, decoupling)

Advantages of Nominal Systems

- Recursive types hold only exactly.
- Typenames everywhere needed in programs (and substitutions, etc.) and easy to extend.

Representing Objects

Formally: object values have the form new $C(v)$

So we can identify the created object with the new expression.

All this information is available in the new expression that creates an object.

Parameters passed to their constructor when they were created.

The only ways in which two objects can differ are (1) their classes and (2) the

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

But when recursive types are considered, some of this simplicity and
elegance slips away...

Caution: when recursive types are considered, some of this simplicity and

Subtyping

Syntax (terms and values)

\[ t ::= \begin{array}{l}
\text{terms} \\
\text{variable} \\
\text{field access} \\
\text{method invocation} \\
\text{cast} \\
\text{method invocation} \\
\text{object creation} \end{array} \]

\[ v ::= \begin{array}{l}
\text{values} \\
\text{object creation} \end{array} \]

Syntax (methods and classes)

\[ K ::= \begin{array}{l}
\text{constructor declarations} \\
\text{method declarations} \end{array} \]

\[ M ::= \begin{array}{l}
\text{constructor declarations} \\
\text{method declarations} \end{array} \]

\[ CL ::= \begin{array}{l}
\text{class declarations} \\
\text{class declarations} \end{array} \]

Subtyping
Subtyping

As in Java, subtyping in FJ is declared.

\[ \text{CT}(C) = \{ \text{class C extends D} \mid \text{C, D} \in \text{CT} \} \]

\[ C \leq D \leq E \]

\[ \text{semantics} \]

\[ \text{More auxiliary definitions} \]

Subtyping
Valid method overriding

Evaluation

Method body lookup

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method overriding

Valid method override
Evaluation

Projection:
newPair(newA(),newB()).snd

CIS500, November 3045

Evaluation

Casting:
(Pair)newPair(newA(),newB())

newPair(newA(),newB())

CIS500, November 3046

Evaluation

Method invocation:
newPair(newA(),newB()).setfst(newB())

newfst
7
newB();
this
7
newPair(newA(),newB())

newPair(newB(),newPair(newA(),newB()).snd)
i.e.,
newPair(newB(), newPair(newA(), newB()).

newB());
newPair(newA(),newB().setfst(newB());
this
7
newPair(newB(), newPair(newA(), newB()).

newB());
)

Pseudo:
Evaluation

((Pair)(newPair(newPair(newA(),newB()),newA()).fst.

snd
newPair(newB(), newPair(newA(), newB()).

newB());
)

((Pair)newPair(newA(),newB()).snd)

newPair(newA(),newB()).snd

newB())

CIS500, November 3048
Evaluation rules

(C) = C_f(newC(v)).f

(mbody(m; C)) = (x; t_0)(newC(v)).m(u)

(C) <: D(D)(newC(v))

Typing

Notes

FJ has no rule of subsumption (because we want to follow Java). The typing rules are algorithmic. (Where would this make a difference?)
Why two cast rules? Because that's how Java does it!

(T-Field)

\[
\frac{L \vdash (c:0 : c)}{L \vdash c : d \rightarrow c}
\]

(T-UCast)

\[
\frac{L \vdash c : d \rightarrow c \quad c : d \rightarrow c}{L \vdash c : c}
\]

(T-DCast)

\[
\frac{L \vdash c : d \rightarrow c \quad c : d \rightarrow c}{L \vdash c : c}
\]

(T-Var)

\[
\frac{L \vdash x : c \quad c \in \text{fields}(c)}{L \vdash \text{field}(c)}
\]

Why two cast rules? Because that's how Java does it!
Let's look at the second in more detail...

Java typing is algorithmic.

Why Java does it this way?

Javatypingisalgorithmic

The Javatyping 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.

Note that this rule's 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?

Why Java does it this way?
Java typing must be algorithmic. We haven't included them in FJ, but full Java has both interfaces and conditionalexpressions. The two together actually make the declarative style of typing rules unwieldy!
More standard (declarative) rule:

```
\text{bool} \ x \ 2 \ \text{true} \ x \ 2 \ \text{T}
```

Algorithmic version:

```
\text{new} \ x \ : \ y
```

Java has no joins!

Javahasnojoins

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

E.g.:

```
\text{interface} I 
\text{interface} J 
\text{interface} K \text{extends} I, J 
\text{interface} L \text{extends} I, J 
```

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

FJTyping rules

Typing rules (methods, classes)

```
class C \text{extends} D \{ x : C \} 
```

So: algebraic typing rules are really our only option.

```
\text{fields}(m, D) \land \text{super}(E) \land \text{this} \land \text{false}(C)
```

More standard (declarative) rule:
Problem: well-typed programs can get stuck.

How?

Cast failure:

(A) new Object()
Formalizing Progress

Solution: We can express $t = E \Gamma(v) \in C$ with $\Gamma \neq \emptyset$. 

Theorem: [Progress]

**Theorem** [Progress]: Suppose $t$ is a closed, well-typed normal form. Then

- Either (1) $t$ is a value, or
- For some $E$, for some $t'$, or
- For some evaluation context $E$, we can express $t = E \Gamma(v) \in C$ with $\Gamma \neq \emptyset$.

Formalizing this takes a little more work...

**Evaluation Contexts**

$E ::= 
\begin{array}{l}
\text{hole} \\
\text{field access} \\
\text{method invocation (receiver)} \\
\text{method invocation (any)} \\
\text{object creation (any)} \\
\text{object creation (arg)} \\
\text{new C (v, E, t)} \\
(t) \\
\emptyset \\
\end{array}$
Preservation

Theorem: If \( t : C \) and \( t \overset{!}{\rightarrow} t' \), then \( t' : C_0 \) for some \( C_0 < : C \).

Proof: Straightforward induction.

Surprise: Well-typed programs can step to ill-typed ones!
Correspondence with Java

Let's try to state precisely what we mean by FJ-words correspond to Java:

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).
3. A well-typed FJ-program behaves the same in FJ as in Java (if FJ is conservative).

Loosen preservation theorem

Correspondence: StupidCast typing rule

Solution: StupidCast typing rule

Add another typing rule: marked „stupid“ to

(Loose preservation theorem)

This is an example of a modeling technicality; not very interesting or deep, but

(Loose preservation theorem)

alternative approaches to casting

Correspondence: StupidCast typing rule

Solution: StupidCast typing rule

Add another typing rule: marked „stupid“ to

(Loose preservation theorem)

This is an example of a modeling technicality; not very interesting or deep, but

Correspondence: StupidCast typing rule

Solution: StupidCast typing rule

Add another typing rule: marked „stupid“ to

(Loose preservation theorem)