Foundations of Software
Fall 2022

Week 13

Plan

PREVIOUSLY: objects through translation
TODAY: Featherweight Java
NEXT: Featherweight Scala

Recap

Last week

The lectures last week developed a series of increasingly sophisticated examples of "OO-style programming" in a typed lambda-calculus.

Multiple representations

All the objects in all the examples have type `Counter` (and sometimes more specific types).

But their internal representations vary widely.

Encapsulation

An object is a record of function. Hidden, internal state is implemented via capturing variables that hold `ref`'s.

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`. To delay the binding as long as possible, this `fix` needs to happen late.

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, ...).

Java (without generics) uses nominal types only.

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` $\mathtt{C}(\overline{\mathtt{v}})$

FJ Syntax

Syntax (terms and values)

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

Syntax (methods and classes)

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

Subtyping

Subtyping

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

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

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ldots\}}}{\mathtt{C <: D}}$$

$$\mathtt{C <: C}$$

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

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

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

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

Method type lookup

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

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

Method body lookup

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

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

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

Evaluation

Projection:

$$\mathtt{new\ Pair(new\ A(),\ new\ B()).snd} \quad \longrightarrow \quad \mathtt{new\ B()}$$

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())
```

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

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 Order

FJ uses a call-by-value evaluation order (like lambda-calculus and Java):

```
    ((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

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

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

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

plus some congruence rules...

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\texttt{t}_0.\texttt{f} \longrightarrow \texttt{t}_0'.\texttt{f}} \qquad \text{(E-Field)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\texttt{t}_0.\texttt{m}(\overline{\texttt{t}}) \longrightarrow \texttt{t}_0'.\texttt{m}(\overline{\texttt{t}})} \qquad \text{(E-Invk-Recv)}$$

$$\frac{\texttt{t}_i \longrightarrow \texttt{t}_i'}{\texttt{v}_0.\texttt{m}(\overline{\texttt{v}},\ \texttt{t}_i,\ \overline{\texttt{t}}) \longrightarrow \texttt{v}_0.\texttt{m}(\overline{\texttt{v}},\ \texttt{t}_i',\ \overline{\texttt{t}})} \qquad \text{(E-Invk-Arg)}$$

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

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{(\texttt{C})\texttt{t}_0 \longrightarrow (\texttt{C})\texttt{t}_0'} \qquad \text{(E-Cast)}$$

Typing

Typing rules

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

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

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{C} <: \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \qquad \text{(T-DCast)}$$

Why two cast rules?

Typing rules

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{C} <: \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \qquad \text{(T-DCast)}$$

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

Typing rules

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

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

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

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

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

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!

But why does Java do it this way? After the break...

Typing rules (methods, classes)

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{t}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 \mathrel{<:} \mathtt{C}_0 \qquad \mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{...\}} \qquad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}_0)}{\mathtt{C}_0\ \mathtt{m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \mathtt{\{return\ t_0;\}\ OK\ in\ C}}$$

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

FJ Typing rules

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

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!

This model accounts for extensions that the model does not actually mention. We are studying the core of a *language family*.

Java conditionals

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

Java conditionals

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

Actual Java rule (algorithmic):

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

Java conditionals

More standard (declarative) rule:

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

Java conditionals

More standard (declarative) rule:

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

Algorithmic version:

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

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.

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.f`	*field access*
	`E.m(t̄)`	*method invocation (rcv)*
	`v.m(v̄,E,t̄)`	*method invocation (arg)*
	`new C(v̄,E,t̄)`	*object creation (arg)*
	`(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 $\mathtt{t} = E[\mathtt{(C)(new\ D(\bar{v}))}]$, with $\mathtt{D} \not<: \mathtt{C}$.

Proof: Straightforward induction.

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 $\mathtt{t} = E[\mathtt{(C)(new\ D(\bar{v}))}]$, with $\mathtt{D} \not<: \mathtt{C}$.

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: Standard substitution lemma, and then 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: Standard substitution lemma, and then straightforward induction. *???*

Preservation?

Preservation?

Surprise: well-typed programs *can* step to ill-typed ones! The substitition lemma does not hold.

(How?)

Preservation?

Surprise: well-typed programs *can* step to ill-typed ones! The substitition lemma does not hold.

(How?)

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

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.

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \qquad \textit{stupid warning}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \quad \text{(T-SCast)}$$

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.

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \qquad \textit{stupid warning}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}} \quad \text{(T-SCast)}$$

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.

Alternative approaches to casting

- Loosen preservation theorem
- Use big-step semantics

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. (Cf. "conservative extension" between logics.)