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

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

FJ Syntax

Syntax (terms and values)

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

Syntax (methods and classes)

$\mathtt{K} ::=$ *constructor declarations*

$\mathtt{C(\overline{C}\ \overline{f})\ \{super(\overline{f});\ this.\overline{f}=\overline{f};\}}$

$\mathtt{M} ::=$ *method declarations*

$\mathtt{C\ m(\overline{C}\ \overline{x})\ \{return\ t;\}}$

$\mathtt{CL} ::=$ *class declarations*

$\mathtt{class\ C\ extends\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}}$

Subtyping

Subtyping

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

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

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

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

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

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

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

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

Method type lookup

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

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

Method body lookup

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

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

Valid method overriding

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

```
class Object {
  int cmp(Object other) {
    return ...
  }
}

class Pair extends Object {
  int cmp(Object other) {
    return ... (Pair)other ...
  }
}
```

override: can't change method signature (argument types and return type), which is why you need downcasting

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:

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

Evaluation

Casting:

$$\begin{array}{l}\texttt{(Pair)new Pair(new A(), new B())} \\ \longrightarrow \ \texttt{new Pair(new A(), new B())}\end{array}$$

Evaluation

Method invocation:

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

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

$$\texttt{new Pair(newfst, this.snd)}$$

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

$$
\begin{array}{ll}
 & \texttt{((Pair) }\underline{\texttt{(new Pair(new Pair(new A(),new B()), new A())}}\\
 & \qquad\quad \underline{\texttt{.fst}}\texttt{).snd}\\
\longrightarrow & \underline{\texttt{((Pair)new Pair(new A(),new B()))}}\texttt{.snd}\\
\longrightarrow & \underline{\texttt{new Pair(new A(), new B()).snd}}\\
\longrightarrow & \texttt{new B()}
\end{array}
$$

Evaluation rules

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

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

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

plus some congruence rules...

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

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

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

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

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

Typing

Typing rules

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

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

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

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

Why two cast rules?

Typing rules

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

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

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

Typing rules

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

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

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t_0} : \mathtt{C_0} \\ \mathit{mtype}(\mathtt{m}, \mathtt{C_0}) = \overline{\mathtt{D}} \to \mathtt{C} \\ \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{t_0.m(}\overline{\mathtt{t}}\mathtt{)} : \mathtt{C}} \quad \text{(T-INVK)}$$

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

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

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.

Why? Because Java does it this way!

But why does Java do it this way??

FJ Typing rules

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

Typing rules (methods, classes)

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{t_0} : \mathtt{E_0} \qquad \mathtt{E_0} \; \mathtt{<:} \; \mathtt{C_0} \\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{...\}} \\ \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \longrightarrow \mathtt{C_0})}{\mathtt{C_0\ m\ (}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\ \{return\ t_0;\}\ OK\ in\ C}}$$

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

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 (cf. congruence rules)

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

Evaluation contexts capture the notion of the "next subterm to be reduced," in the sense that, if $\texttt{t} \longrightarrow \texttt{t}'$, then we can express `t` and $\texttt{t}'$ as $\texttt{t} = E[\texttt{r}]$ and $\texttt{t}' = E[\texttt{r}']$ for a unique E, `r`, and $\texttt{r}'$, with $\texttt{r} \longrightarrow \texttt{r}'$ 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 $\mathtt{t} = E[\mathtt{(C)(new\ D(}\overline{\mathtt{v}}\mathtt{))}]$, 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.

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

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.

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

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

Alternative approaches to casting

- Loosen preservation theorem
- Use big-step semantics