CIS 500
Software Foundations

Fall 2007

Lecture 23

Administrivia

- HW12 due next Monday, 10AM
- Final exam: Tuesday, Dec 18, 12-2
  - Location: Moore 216 (*not Wu and Chen!*)
- Review questions for the last part of the class will be available about a week before the exam
- My office hours:
  - 5–6:30 this afternoon
  - Then not until next semester
- Leonid's office hours next week will be posted to the class mailing list

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 ⟶ 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 (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.

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

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

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

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

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

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

Method type lookup

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

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

Method body lookup

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

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

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())
⟶ [ newfst ↦ new B(),
    this ↦ new Pair(new A(),new B()) ]
  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

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

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

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

plus some congruence rules...

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

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

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

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

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

Typing

Typing rules

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

Typing rules

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

Typing rules

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

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

Why two cast rules?

Typing rules

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

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

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

Typing rules

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

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

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

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!

Typing rules

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

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 (see discussion in TAPL).

FJ Typing rules

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

Typing rules (methods, classes)

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

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

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)*
	$\mathtt{v.m}(\overline{\mathtt{v}}, E, \overline{\mathtt{t}})$	*method invocation (arg)*
	$\mathtt{new\ C}(\overline{\mathtt{v}}, E, \overline{\mathtt{t}})$	*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 $\mathtt{t} = E[\mathtt{(C)(new\ D}(\overline{\mathtt{v}})\mathtt{)}]$, 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?)

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

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.

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

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.

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

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

Recap...

What is "software foundations"?

Software foundations (a.k.a. "theory of programming languages") is the study of the *meaning* of programs.

A main goal is finding ways to describe program behaviors that are both *precise* and *abstract*.

Why study software foundations?

- To be able to prove specific facts about particular programs (i.e., program verification)
- To develop intuitions for informal reasoning about programs
- To prove general facts about all the programs in a given programming language (e.g., safety or security properties)
- As a foundation for a wide range of static analyses (e.g., Microsoft's current suite of tools for checking device drivers)
- To understand language features (and their interactions) deeply and develop principles for better language design

What I hope you got out of the course

- A more sophisticated perspective on programs, programming languages, and the activity of programming
  - How to view programs and whole languages as formal, mathematical objects
  - How to make and prove rigorous claims about them
  - Detailed study of a range of basic language features
- Deep intuitions about key language properties such as type safety
- Familiarity with today's best practices for language design, description, and analysis
- Fun hacking Coq

What next?

The rest of TAPL

Many more core topics are covered in TAPL.

- References (mutable state)
- Exceptions
- Recursive types (including a rigorous treatment of induction and co-induction)
- Parametric polymorphism
  - Universal and existential types
  - Bounded quantification
  - Refinement of the imperative object model
  - ML-style type inference
- Type operators
  - Higher-order bounded quantification
  - A purely functional object model

The Research Literature

With this course under your belt, you are ready to directly address research papers in programming languages.

This is a big area, and each sub-area has its own special techniques and notations, but you now have all the basic intuitions needed to study these on your own.

The End