Advanced Topics in Programming Languages **Spring Semester, 2012**

Lecture 12: Featherweight Java

Lecturer: Mooly Sagiv *Scribe: Adam Morrison*

12.1 Introduction

In this lecture we apply the ideas from previous lectures to directly define and reason about *Featherweight Java* (FJ), an object-oriented language based on Java. FJ is a subset of Java (every FJ program is a Java program) designed with the goal of making its type safety proof as short as possible, while still capturing the essence of Java's safety arguments. FJ therefore incorporates many of Java's interesting aspects, such as the "everything is an object" philosophy and dynamic casting, but drops any feature that would make the safety proof longer without making it significantly different.

For example, one central feature that FJ omits is *mutation* (i.e., the assignment operator), as supporting it would require modeling the heap. The features of Java that FJ does model include mutually recursive class definitions, object creation, field access, method invocation, method override, subtyping and casting.

As a compact language model, FJ is useful for modeling extensions of Java. Adapting a language extension to FJ focuses attention on essential aspects of the extension. The original FJ paper [1] demonstrated this by adding generic classes to FJ.

12.2 Nominal vs. structural types systems

FJ's type system differs from what we saw in previous lectures. Until now we have been working with *structural* type systems, in which a type is determined by its structure. In a structural system the following types are equivalent:

$$\texttt{Euros} = \{\texttt{amount} : \texttt{Float}\} \quad \texttt{Dollars} = \{\texttt{amount} : \texttt{Float}\}$$

The names `Euros` and `Dollar` are simply cosmetic abbreviations. Consequentially, in a structural system subtyping is also determined by the structure of the type. For example, the rules used in previous lectures imply that $\texttt{Product} = \{\texttt{name} : \texttt{String}, \texttt{price} : \texttt{Float}\}$ is a subtype of $\texttt{Employee} = \{\texttt{name} : \texttt{String}\}$ even though intuitively this seems strange.

In contrast, FJ has a *nominal* type system in which the *name* of a type is part of its definition, so `Euros` and `Dollars` are distinct types. As a result, it is up to the programmer to explicitly identify the subtyping relation by declaring which classes a newly defined class extends. The compiler verifies these declarations, i.e., that the fields and methods of the new class really do extend its superclasses.

Nominal systems simplify many practical aspects of the programming language, perhaps explaining why they are present in many popular languages. First, representing recursive types (such as a `Node` in a linked list that has a field `next` of the same type) and mutually-recursive types is straightforward. In a structural system representing such types requires complex formal mechanisms. Second, the explicit subtype declarations make it easy for the compiler to represent and check the subtyping relation. A structural system requires either performing a full structural test for each subtype test, or using more sophisticated representation techniques. The compiler in a nominal system can embed the representation of the types into the program by tagging each object with a header pointing to metadata describing the object's type and pointing to its supertypes. This simplifies the implementation of languages features such as run-time type testing (e.g., `instanceOf`), printing structures and reflection. Structural system can embed a similar representation but it constitutes an additional mechanism, since there are no compile-time type names that the run-time tags can match to. Finally, nominal systems prevent *spurious subsumption* – mixing of different types that are structurally compatible, such as passing `Euros` to a `DollarsToEuros` function. For a structural system, the program needs to use single-field variants (Lecture 10) or abstract data types to avoid this problem.

Interestingly, most research works deal with structural systems. Working with types in a nominal system requires working with respect to some global collection of type names and their definitions, which complicates definitions and proofs. In a structural system (without recursive types) a type can be treated as a closed entity. Furthermore, advanced language features for type abstraction, such as parametric polymorphism, abstract data types, user-defined type operators, functors, etc. do not fit cleanly into nominal systems.

12.3 Syntax

We present the formal definition of FJ's syntax. As will be seen, we simplify the formal definition by making some requirements on the structure of FJ code, e.g., that fields are always initialized in their declaration order.

Metavariables: B, C, D and E range over class names; f and g range over field names; m ranges over method names, v and u range over values, and x ranges over parameter names.

Notation: For $\texttt{z} \in \{\texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{f}, \texttt{g}, \texttt{t}, \texttt{x}, \texttt{v}\}$ we write $\bar{\texttt{z}}$ as shorthand for $\texttt{z}_1, \ldots, \texttt{z}_n$ for some n. For method declaration M, we write $\bar{\texttt{M}}$ as shorthand for $\texttt{M}_1 \ldots \texttt{M}_n$ (no commas). Pairs of sequences are abbreviated similarly, with "$\bar{\texttt{C}}\ \bar{\texttt{f}}$" standing for $\texttt{C}_1\ \texttt{f}_1, \ldots, \texttt{C}_n\ \texttt{f}_n$, "$\bar{\texttt{C}}\ \bar{\texttt{f}};$" standing for $\texttt{C}_1\ \texttt{f}_1; \ldots; \texttt{C}_n\ \texttt{f}_n;$, and "$\texttt{this}.\bar{\texttt{f}} = \bar{\texttt{f}};$" for $\texttt{this}.\texttt{f}_1 = \texttt{f}_1; \ldots; \texttt{this}.\texttt{f}_n = \texttt{f}_n;$. We assume such sequences do not contain duplicate names.

Featherweight Java syntax

CL	::=		*Class declarations*
		class C extends C $\{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \texttt{K}\ \bar{\texttt{M}}\}$	Introduces a class C with superclass D that has (1) fields $\bar{\texttt{f}}$ of types $\bar{\texttt{C}}$, which have distinct names from D's fields, (2) one constructor K and a set of methods $\bar{\texttt{M}}$, which can override some of D's methods.
K	::=		*Constructor declarations*
		$\texttt{C}(\bar{\texttt{C}}\ \bar{\texttt{f}})\ \{\texttt{super}(\bar{\texttt{f}});\ \texttt{this}.\bar{\texttt{f}} = \bar{\texttt{f}};\}$	The constructor takes as many parameters as there are instance variables and in the same order as they are declared. It first calls the superclass constructor and then assigns the parameters to the instance's fields.
M	::=		*Method declarations*
		$\texttt{C m}(\bar{\texttt{C}}\ \bar{\texttt{x}})\ \{\texttt{return t};\}$	A method named m with result type D and parameters x of types C. Since we omit assignments, the method only returns a term t where the variables x are bound.
t	::=		*Terms*
		x	*Variables:* We assume that the set of variables includes the special variable this which is never used as an argument to a method.
		t.f	*Field access*
		$\texttt{t.m}(\bar{\texttt{t}})$	*Method invocation*
		$\texttt{new C}(\bar{\texttt{t}})$	*Object creation*
		(C)t	*Cast*
v	::=		*Values*
		$\texttt{new C}(\bar{\texttt{v}})$	*Object creation*

12.4 Subtyping

We define a *class table*, CT, that maps from class names to class declarations, e.g.,

$$CT(\texttt{Cat}) = \texttt{class Cat extends Animal \{ ... \}}$$

But since `Object` is not explicitly defined, how shall we define $CT(\texttt{Object})$? We solve this technical problem by recognizing that `Object` is a special object which will not have a class table entry (i.e., $\texttt{Object} \notin dom(CT)$). We will see this distinction as we add a special case for `Object` when looking up field names, since `Object` has no fields.

A FJ program is a pair $(CT, \texttt{t})$ consisting of the term to be evaluated, `t`, and the class table with respect to which `t` is evaluated. From now on we will assume a fixed class table CT.

We derive the typing relation, <:, from the class table:

Definition of <: relation

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

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

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

We assume that the <: relation is antisymmetric (i.e., the only allowed cycles are of the form $\texttt{C} <: \texttt{C}$).

12.5 Operational semantics

To define the operational semantics (and later the typing rules) we require some auxiliary definitions.

Field lookup: $fields(\texttt{C})$ is the sequence $\bar{\texttt{C}}\ \bar{\texttt{f}}$ of all of `C`'s fields (including those inherited from `C`'s superclasses). Formally:

$$fields(\texttt{Object}) = \epsilon$$

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

Methods: The type of method $\mathtt{m}$ in class $\mathtt{C}$, $\mathit{mtype}(\mathtt{m},\mathtt{C})$, is a pair consisting of a sequence of argument types, $\bar{\mathtt{B}}$, and the type of the result, $\mathtt{B}$. We use the notation $\bar{\mathtt{B}} \mapsto \mathtt{B}$. Similarly, the body of the method, $\mathit{mbody}(\mathtt{m},\mathtt{C})$ is the pair $(\bar{\mathtt{x}},\mathtt{t})$ of the sequence of parameters $\bar{\mathtt{x}}$ and a term $\mathtt{t}$. Formally:

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

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

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

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

Method override The predicate $\mathit{override}(\mathtt{m},\mathtt{D},\bar{\mathtt{C}} \to \mathtt{C}_0)$ if true if method $\mathtt{m}$ with argument types $\bar{\mathtt{C}}$ and result $\mathtt{C}_0$ may be defined in class $\mathtt{D}$. It checks that either $\mathtt{m}$ is not defined in $\mathtt{D}$'s superclasses, or that if it is, then $\mathtt{m}$'s arguments and return type match those of the classes it extends. Formally:

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

Semantics We now formally define a small step operation semantics for FJ. We will use the following class table as a running example to illustrate the evaluation rules:

```
class A extends Object { A() { super(); } }
class B extends Object { B() { super(); } }
class Pair extends Object {
    Object first;
    Object second;
    Pair(Object first, Object second) {
        super(); this.first = first; this.second = second; }
    Pair SetFirst(Object newFirst) {
        return new Pair(newFirst, this.second); }
}
```

Field projection:

$$\frac{\mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}}}{(\mathtt{new\ C}\,(\bar{\mathtt{v}}))\,.\mathtt{f_i} \longrightarrow \mathtt{v_i}} \qquad \text{(E-PROJNEW)}$$

```
new Pair(new A(), new Pair(new A(), new B())).second
⟶
new Pair(new A(), new B())
```

$$\frac{\mathtt{t_0} \longrightarrow \mathtt{t_0'}}{\mathtt{t_0.f} \longrightarrow \mathtt{t_0'.f}} \qquad \text{(E-FIELD)}$$

```
new Pair(new A(), new Pair(new A(), new B())).second.first
⟶
new Pair(new A(), new B()).first
```

Method invocation: We use standard call-by-value semantics. When we have all the values, we bind the actual parameters to the formal parameters and evaluate the term. Notice that we exploit the fact that FJ has no side effects: If a formal parameter appears more than once in the body, the argument value may be duplicated. But since there are no side effect, this cannot be observed.

$$\frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\bar{\texttt{x}}, \texttt{t}_0)}{(\texttt{new C}\,(\bar{\texttt{v}}))\,\texttt{.m}(\bar{\texttt{u}}) \longrightarrow [\bar{\texttt{x}} \mapsto \bar{\texttt{u}}, \texttt{this} \mapsto \texttt{new C}(\bar{\texttt{v}})]\,\texttt{t}_0} \qquad \text{(E-INVKNEW)}$$

```
new Pair(new A(), new B()).setFirst(new B())
```

$$\longrightarrow$$

$$[\ \texttt{newFirst} \mapsto \texttt{new B}(), \texttt{this} \mapsto \texttt{new Pair(new A}(), \texttt{new B}())\ ]$$

```
new Pair(newFirst, this.second)
```

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}'_0}{\texttt{t}_0\texttt{.m}(\bar{\texttt{t}}) \longrightarrow \texttt{t}'_0\texttt{.m}(\bar{\texttt{t}})} \qquad \text{(E-INVK-RECV)}$$

$$\frac{\texttt{t}_\texttt{i} \longrightarrow \texttt{t}'_\texttt{i}}{\texttt{v}_0\texttt{.m}(\bar{\texttt{v}}, \texttt{t}_\texttt{i}, \bar{\texttt{t}}) \longrightarrow \texttt{v}'_0\texttt{.m}(\bar{\texttt{v}}, \texttt{t}'_\texttt{i}, \bar{\texttt{t}})} \qquad \text{(E-INVK-ARG)}$$

$$\frac{\texttt{t}_\texttt{i} \longrightarrow \texttt{t}'_\texttt{i}}{\texttt{new C}(\bar{\texttt{v}}, \texttt{t}_\texttt{i}, \bar{\texttt{t}}) \longrightarrow \texttt{new C}(\bar{\texttt{v}}, \texttt{t}'_\texttt{i}, \bar{\texttt{t}})} \qquad \text{(E-NEW-ARG)}$$

Casting: When casting from one type to another, we keep the cast until its subject has been reduced to an object, at which point we can check if the cast is valid or not. If the cast is valid, we simply remove it: any field or method accessed will be valid. Otherwise, the evaluation is *stuck*, leading to a run-time error.

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

```
(Object)(new Pair(new A(), new B()))
```

$$\longrightarrow$$

```
new Pair(new A(), new B()))
```

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

Here is a more interesting derivation:

`((Pair)(new Pair(new Pair(new A(), new B()), new B()).first)).setFirst(new B())`	$\longrightarrow$ (E-PROJNEW)
`((Pair)(new Pair(new A(), new B()))).setFirst(new B())`	$\longrightarrow$ (E-CASTNEW)
`(new Pair(new A(), new B())).setFirst(new B())`	$\longrightarrow$ (E-INVNEW)
`new Pair(new B(), new B())`	

12.6 Typing

The typing rules for FJ are presented below. As in previous lectures, Γ is the *environment* mapping from variables to types, where $\mathtt{t} : \mathtt{C}$ means that $\mathtt{t}$ is of type $\mathtt{C}$. We use the abbreviation $\Gamma \vdash \bar{\mathtt{t}} : \bar{\mathtt{C}}$ for the sequence $\Gamma \vdash \bar{\mathtt{t}_1} : \bar{\mathtt{C}_1}, \ldots, \Gamma \vdash \bar{\mathtt{t}_n} : \bar{\mathtt{C}_n}$.

Except for casting, the typing rules correspond to the evaluation rules. For constructors and method invocations, the rules verify that each argument has a type that is a subtype of the one declared for the formal parameter.

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

$$\frac{\Gamma \vdash \mathtt{t_0} : \mathtt{C_0} \qquad \mathit{fields}(\mathtt{C_0}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}}}{\Gamma \vdash \mathtt{t_0.f_i} : \mathtt{C_i}} \qquad \text{(T-FIELD)}$$

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

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{f}} \\ \Gamma \vdash \bar{\mathtt{t}} : \bar{\mathtt{C}} \qquad \bar{\mathtt{C}} <: \bar{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{new\ C}(\bar{\mathtt{t}}) : \mathtt{C}} \qquad \text{(T-NEW)}$$

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

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

$$\frac{\begin{array}{c}\bar{\mathtt{x}} : \bar{\mathtt{C}},\ \mathtt{this} : \mathtt{C} \vdash \mathtt{t_0} : \mathtt{E_0} \qquad \mathtt{E_0} <: \mathtt{C_0} \\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\} \\ \mathit{override}(\mathtt{m}, \mathtt{D}, \bar{\mathtt{C}} \to \mathtt{C_0})\end{array}}{\mathtt{C_0\ m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\mathtt{return\ t_0};\}\ \mathtt{OK\ in\ C}} \qquad \text{Method typing}$$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C}(\bar{\mathtt{D}}\ \bar{\mathtt{g}},\ \bar{\mathtt{C}}\ \bar{\mathtt{f}})\ \{\mathtt{super}(\bar{\mathtt{g}});\ \mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};\} \\ \mathit{fields}(\mathtt{D}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}} \qquad \bar{\mathtt{M}}\ \mathtt{OK\ in\ C}\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\}\ \mathtt{OK}} \qquad \text{Class typing}$$

12.7 Type safety

Here we prove that FJ is type safe, by stating and proving a preservation theorem and a progress theorem for FJ. When it is clear from the context, we will write $\Gamma \vdash \mathtt{t} : \mathtt{C}$ simply as $\mathtt{t} : \mathtt{C}$.

Interestingly, we will see that the type system defined thus far *does not* imply type safety of FJ. We will need to add an additional typing rule to obtain our result.

12.7.1 Preservation theorem

In previous lectures we proved preservation theorems of the following form:

$$\text{If } \mathtt{t} : \mathtt{C} \text{ and } \mathtt{t} \longrightarrow \mathtt{t}', \text{ then } \mathtt{t}' : \mathtt{C}.$$

But because of subtyping in the language such a statement is false for FJ:

$$\begin{array}{ll} (\mathtt{Object})\ \mathtt{new}\ \mathtt{B}() & \text{(E-CASTNEW)} \\ \longrightarrow\ \mathtt{new}\ \mathtt{B}() & \end{array}$$

We therefore want to want to prove the following preservation theorem:

$$\text{If } \mathtt{t} : \mathtt{C} \text{ and } \mathtt{t} \longrightarrow \mathtt{t}', \text{ then } \mathtt{t}' : \mathtt{C}' \text{ for some } \mathtt{C}' <: \mathtt{C}.$$

Let us try to prove it. As usual we use induction on the derivation of $\mathtt{t} : \mathtt{C}$ and consider the last step of the derivation. For example,

Case T-FIELD: $\mathtt{t} = \mathtt{t}_0.\mathtt{f}_\mathtt{i} : \mathtt{C}_\mathtt{i} \quad \mathtt{t}_0 : \mathtt{C}_0, \mathit{fields}(\mathtt{C}_0) = \bar{\mathtt{C}}\ \bar{\mathtt{f}}$

In this case, $\mathtt{t} \longrightarrow \mathtt{t}'$ by either E-PROJNEW or E-FIELD. If $\mathtt{t} \longrightarrow \mathtt{t}'$ by E-PROJNEW, then $\mathtt{t}_0 = \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}})$ and so $\mathtt{t}' = \mathtt{v}_\mathtt{i}$. In the derivation of $\mathtt{t}_0 : \mathtt{C}_0$ we must have an application of T-NEW for $\mathtt{t}_0$:

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{C}_0) = \bar{\mathtt{C}}\ \bar{\mathtt{f}} \\ \bar{\mathtt{v}} : \bar{\mathtt{D}} \qquad \bar{\mathtt{D}} <: \bar{\mathtt{C}}\end{array}}{\mathtt{t}_0 = \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}}) : \mathtt{C}_0}$$

implying that $\mathtt{t}' = \mathtt{v}_\mathtt{i} : \mathtt{D}_\mathtt{i} <: \mathtt{C}_\mathtt{i}$.

Otherwise, if $\mathtt{t} \longrightarrow \mathtt{t}'$ by E-FIELD then $\mathtt{t}' = \mathtt{t}_0'.\mathtt{f}$ where $\mathtt{t}_0 \longrightarrow \mathtt{t}_0'$. Applying the induction hypothesis to the derivation of $\mathtt{t}_0 : \mathtt{C}_0$ we obtain that $\mathtt{t}_0' : \mathtt{C}_0'$ for some $\mathtt{C}_0' <: \mathtt{C}_0$. By the class typing rule we have $fields(\mathtt{C}_0') = fields(\mathtt{C}_0), \bar{\mathtt{D}}\ \bar{\mathtt{g}}$. Since $\mathtt{f_i}$ is one of $\mathtt{C}_0$'s fields, we can apply T-FIELD and conclude:

$$\frac{\mathtt{t}_0' : \mathtt{C}_0' \qquad fields(\mathtt{C}_0') = fields(\mathtt{C}_0), \bar{\mathtt{D}}\ \bar{\mathtt{g}}}{\mathtt{t}_0'.\mathtt{f_i} : \mathtt{C_i}}.$$

The remaining cases that do not involve casting are similarly straightforward, so let us focus on casting.

Case T-UCAST: $\mathtt{t} = (\mathtt{C})\mathtt{t}_0 : \mathtt{C} \quad \mathtt{t}_0 : \mathtt{D}, \mathtt{D} <: \mathtt{C}$

If $\mathtt{t} \longrightarrow \mathtt{t}'$ by E-CASTNEW, then $\mathtt{t}_0 = \mathtt{new\ D}(\bar{\mathtt{v}})$, and we have:

$$\frac{\mathtt{D} <: \mathtt{C}}{(\mathtt{C})(\mathtt{new\ D}(\bar{\mathtt{v}})) \longrightarrow \mathtt{new\ D}(\bar{\mathtt{v}})}$$

Because $\mathtt{t}_0$ is well-typed, T-NEW applies to $\mathtt{t}'$ and so $\mathtt{t}' : D <: C$.

If $\mathtt{t} \longrightarrow \mathtt{t}'$ by E-CAST, then we have:

$$\frac{\mathtt{t}_0 \longrightarrow \mathtt{t}_0'}{(\mathtt{C})\mathtt{t}_0 \longrightarrow (\mathtt{C})\mathtt{t}_0'}$$

Applying the induction hypothesis to the derivation of $\mathtt{t}_0 : \mathtt{D}$ we obtain that $\mathtt{t}_0' : \mathtt{C}_0' <: \mathtt{D} <: \mathtt{C}$, so we can apply T-UCAST and infer that $\mathtt{t}' : \mathtt{C}$:

$$\frac{\mathtt{t}_0' : \mathtt{C}_0' \qquad \mathtt{C}_0' <: \mathtt{C}}{(\mathtt{C})\mathtt{t}_0 : \mathtt{C}}.$$

Case T-DCAST: $\mathtt{t} = (\mathtt{C})\mathtt{t}_0 : \mathtt{C} \quad \mathtt{t}_0 : \mathtt{D}, \mathtt{C} <: \mathtt{D}, \mathtt{C} \neq \mathtt{D}$

Notice first that it cannot be that $\mathtt{t} \longrightarrow \mathtt{t}'$ by E-CASTNEW, since then we'd have:

$$\frac{\mathtt{D} <: \mathtt{C}}{(\mathtt{C})(\mathtt{new\ D}(\bar{\mathtt{v}})) \longrightarrow \mathtt{new\ D}(\bar{\mathtt{v}})}$$

where $\mathtt{t}_0 = \mathtt{new\ D}(\bar{\mathtt{v}})$, implying that $\mathtt{D} <: \mathtt{C}$ which is impossible because we know from the induction hypothesis that $\mathtt{C} <: \mathtt{D}, \mathtt{C} \neq \mathtt{D}$ and the relation $<:$ is antisymmetric.

It must therefore be the case that $\mathtt{t} \longrightarrow \mathtt{t}'$ by E-CAST. Thus, we have:

$$\frac{\mathtt{t}_0 \longrightarrow \mathtt{t}_0'}{(\mathtt{C})\mathtt{t}_0 \longrightarrow (\mathtt{C})\mathtt{t}_0'}$$

We try to proceed as in the upcast case, applying the induction hypothesis to the derivation of $\mathtt{t}_0 : \mathtt{D}$ and obtaining that $\mathtt{t}_0' : \mathtt{C}_0' <: \mathtt{D}$.

Now, if $\texttt{C}_0' <: \texttt{C}$ we can infer (as before) using T-UCAST that $\texttt{t}' : \texttt{C}$. An example term (where $\texttt{t}_0 \longrightarrow \texttt{t}_0'$ by E-CAST) is:

$$(\underbrace{\texttt{Animal}}_{\texttt{C}})\,\underbrace{(\underbrace{\texttt{Object}}_{\texttt{D}})\ \underbrace{\texttt{new}\ \underbrace{\texttt{Cat}}_{\texttt{C}_0'}()}_{\texttt{t}_0'}}_{\texttt{t}_0}$$

If $\texttt{C} <: \texttt{C}_0'$ we can infer using T-DCAST that $\texttt{t}' : \texttt{C}$. For example,

$$(\underbrace{\texttt{PersianCat}}_{\texttt{C}})\,\underbrace{(\underbrace{\texttt{Animal}}_{\texttt{D}})\ \underbrace{\texttt{new}\ \underbrace{\texttt{Cat}}_{\texttt{C}_0'}()}_{\texttt{t}_0'}}_{\texttt{t}_0}$$

Remark: This term, while well-typed, has undefined semantics. We pay this price for supporting the notion of specializing objects via downcasts – we will only know at run-time if this term gets evaluated, at which point an error will be generated. (In full Java an exception would be thrown and the program can continue.) We will deal with this issue formally when proving the progress theorem in the next subsection.

But what happens if $\texttt{C}$ and $\texttt{C}_0'$ are unrelated by <:? Our proof gets stuck! This problem happens due to our use of small step operational semantics, which can derive programs such as:

$$(\underbrace{\texttt{Cat}}_{\texttt{C}})\,\underbrace{(\underbrace{\texttt{Object}}_{\texttt{D}})\ \underbrace{\texttt{new}\ \underbrace{\texttt{Dog}}_{\texttt{C}_0'}()}_{\texttt{t}_0'}}_{\texttt{t}_0}$$

To overcome this technical modeling problem, we introduce a new typing rule for such “stupid” casts:

$$\boxed{\frac{\Gamma \vdash \texttt{t}_0 : \texttt{D} \quad \texttt{C} \not<: \texttt{D} \quad \texttt{D} \not<: \texttt{C} \quad \text{stupid warning}}{\Gamma \vdash (\texttt{C})\texttt{t}_0 : \texttt{C}} \quad \text{(T-SCAST)}}$$

Applying the stupid cast rule in this case allows the proof to carry through. We indicate the special nature of stupid casts by including the hypothesis *stupid warning* in the T-SCAST type rule. A true Java compiler rejects such casts, and an FJ typing corresponds to a legal Java typing only if it does not contain this rule.

Having introduced stupid casts, we must consider them in the preservation proof:

Case T-SCAST: $\mathtt{t} = (\mathtt{C})\mathtt{t}_0 : \mathtt{C} \quad \mathtt{t}_0 : \mathtt{D}, \mathtt{C} \not<: \mathtt{D}, \mathtt{D} \not<: \mathtt{C}$, stupid warning

Because $\mathtt{D} \not<: \mathtt{C}$, it cannot be that $\mathtt{t} \longrightarrow \mathtt{t}'$ by E-CASTNEW. Thus E-CAST must apply and so $\mathtt{t} = (\mathtt{C})\ \mathtt{t}_0$, $\mathtt{t}' = (\mathtt{C})\ \mathtt{t}_0'$ and $\mathtt{t}_0 \longrightarrow \mathtt{t}_0'$. Applying the induction hypothesis to the derivation of $\mathtt{t}_0 : \mathtt{D}$ we find that $\mathtt{t}_0' : \mathtt{C}_0' <: \mathtt{D}$. Because a FJ class has one superclass, $\mathtt{C}_0' \not<: \mathtt{C}$. Further, $\mathtt{C} \not<: \mathtt{C}_0'$ as otherwise $\mathtt{C} <: \mathtt{D}$. Thus we can apply T-SCAST

$$\frac{\mathtt{t}_0' : \mathtt{C}_0' \quad \mathtt{C}_0' \not<: \mathtt{C} \quad \mathtt{C} \not<: \mathtt{C}_0' \\ \text{stupid warning}}{(\mathtt{C})\mathtt{t}_0' : \mathtt{C}}$$

and we are done. We have therefore proved:

Theorem 12.1 (Preservation) *If* $\mathtt{t} : \mathtt{C}$ *and* $\mathtt{t} \longrightarrow \mathtt{t}'$*, then* $\mathtt{t}' : \mathtt{C}'$ *for some* $\mathtt{C}' <: \mathtt{C}$.

12.7.2 Progress theorem

Here we show a variant of the progress theorems seen in previous lectures. We will show that the only way a well-typed FJ program can get stuck is if it reaches a point where it cannot perform a downcast.

There are several ways in which the semantics can get stuck: (1) accessing an undefined field in class, (2) calling an undefined method, (3) calling a method with the wrong number of arguments, (4) trying to cast from $\mathtt{C}$ to $\mathtt{D}$ where $\mathtt{C} \not<: \mathtt{D}$.

The last case is the more interesting case which we will focus on. Thus, we first rule out the other cases by proving they never happen for well-typed programs.

Lemma 12.2 *Suppose* $\mathtt{t}$ *is a well-typed term. Then:*

1. *If* $\mathtt{t} = \mathtt{new}\ \mathtt{C}_0(\bar{\mathtt{t}}).\mathtt{f}$*, then* $\mathit{fields}(\mathtt{C}_0) = \bar{\mathtt{C}}\ \bar{\mathtt{f}}$ *and* $\mathtt{f} \in \bar{\mathtt{f}}$.
2. *If* $\mathtt{t} = \mathtt{new}\ \mathtt{C}_0(\bar{\mathtt{t}}).\mathtt{m}(\bar{\mathtt{s}})$*, then* $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0) = (\bar{\mathtt{x}}, \mathtt{t}_0)$ *and* $|\bar{\mathtt{x}} \models |\bar{\mathtt{s}}|$.

Proof: Immediate from the typing rules. ■

Before proceeding to show that a well-typed program can only get stuck because of an impossible cast, we develop a formal mechanism to explicitly identify the failing cast when it exists. The set of *evaluation contexts* for FJ is defined as follows:

E ::=		
	$[]$	*hole*
	$E.\mathtt{f}$	*field access*
	$E.\mathtt{m}(\bar{\mathtt{t}})$	*method invocation (receiver)*
	$\mathtt{v.m}(\bar{\mathtt{v}}, E, \bar{\mathtt{t}})$	*method invocation (argument)*
	$\mathtt{new\ C}(\bar{\mathtt{v}}, E, \bar{\mathtt{t}})$	*object creation (argument)*
	$(\mathtt{C})E$	*cast*

Each evaluation context is a term with a hole (written $[]$) inside it. For example,

$$[].\mathtt{foo}$$
$$[].\mathtt{foo.bar}$$
$$\mathtt{new\ C(new\ D(),\ [].foo.bar,\ new\ E())}$$

We write $E[\mathtt{t}]$ for the ordinary term obtained by replacing the hole in E with $\mathtt{t}$.

The idea behind evaluation contexts is to capture the notion of the "next subterm to be reduced" in the following sense:

Lemma 12.3 *If* $\mathtt{t} \longrightarrow \mathtt{t}'$ *then there are unique* E*,* $\mathtt{r}$ *and* $\mathtt{r}'$ *such that (1)* $\mathtt{t} = E[\mathtt{r}]$*, (2)* $\mathtt{t}' = E[\mathtt{r}']$*, and (3)* $\mathtt{r} \longrightarrow \mathtt{r}'$ *by one of E-PROJNEW, E-INVKNEW or E-CASTNEW.*

Proof: By induction on the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$.

The inductive step is a case analysis of each possible last step in the derivation. If it is one of E-PROJNEW, E-INVKNEW or E-CASTNEW, we can take $E = []$, $\mathtt{r} = \mathtt{t}$ and $\mathtt{r}' = \mathtt{t}'$.

Now consider, for example, E-FIELD. Then $\mathtt{t} = \mathtt{t}_0.\mathtt{f}, \mathtt{t}' = \mathtt{t}'_0.\mathtt{f}$ and $\mathtt{t}_0 \longrightarrow \mathtt{t}'_0$. By the inductive hypothesis, there are unique E_0, $\mathtt{r}_0$ and $\mathtt{r}'_0$ such that (1) $\mathtt{t}_0 = E_0[\mathtt{r}_0]$, (2) $\mathtt{t}'_0 = E_0[\mathtt{r}'_0]$, and (3) $\mathtt{r}_0 \longrightarrow \mathtt{r}'_0$ by one of E-PROJNEW, E-INVKNEW or E-CASTNEW. Then $E = E_0.\mathtt{f}$ (which is a valid evaluation context), r_0 and r'_0 satisfy our requirements. The remaining cases are similarly straightforward. ■

Theorem 12.4 (Progress) *Suppose* $\mathtt{t}$ *is a well-typed term. Then either*

1. $\mathtt{t} \longrightarrow \mathtt{t}'$ *for some* $\mathtt{t}'$.
2. $\mathtt{t}$ *is a value.*
3. $\mathtt{t} = E[(\mathtt{C})(\mathtt{new\ D}(\bar{\mathtt{v}}))]$*, for some evaluation context* E*, and* $\mathtt{D} \not<: \mathtt{C}$.

Proof: We proceed using induction on the typing derivation of `t`. Let us consider the last step in the derivation:

Case T-FIELD: $\mathtt{t} = \mathtt{t_0.f_i} : \mathtt{C_i} \quad \mathtt{t_0} : \mathtt{C_0}, \mathit{fields}(\mathtt{C_0}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}}$

From the induction hypothesis on $\mathtt{t_0}$, the following are possible: (1) $\mathtt{t_0} \longrightarrow \mathtt{t_0'}$, and then E-FIELD applies to `t`, or (2) $\mathtt{t_0}$ is a value, and then E-PROJNEW applies to `t` and yields a value, or (3) $\mathtt{t_0} = E_0[(\mathtt{C_0})(\mathtt{new\ D_0}(\bar{\mathtt{v}}))]$ with $\mathtt{D_0} \not<: \mathtt{C_0}$, and then $\mathtt{t} = E[(\mathtt{C_0})(\mathtt{new\ D_0}(\bar{\mathtt{v}}))]$ for $E = E_0.\mathtt{f_i}$. Similar arguments apply to T-INVK and T-NEW.

Case T-UCAST: $\mathtt{t} = (\mathtt{C})\mathtt{t_0} : \mathtt{C} \quad \mathtt{t_0} : \mathtt{D}, \mathtt{D} <: \mathtt{C}$

Applying the induction hypothesis to $\mathtt{t_0}$, we have that either (1) $\mathtt{t_0} \longrightarrow \mathtt{t_0'}$ and then E-CAST applies, or (2) $\mathtt{t_0}$ is a value and then E-CASTNEW applies, yielding a value, or (3) $\mathtt{t_0} = E_0[(\mathtt{C_0})(\mathtt{new\ D_0}(\bar{\mathtt{v}}))]$ with $\mathtt{D_0} \not<: \mathtt{C_0}$, and then $\mathtt{t} = E[(\mathtt{C_0})(\mathtt{new\ D_0}(\bar{\mathtt{v}}))]$ for $E = (C)E_0$.

Case T-DCAST: $\mathtt{t} = (\mathtt{C})\mathtt{t_0} : \mathtt{C} \quad \mathtt{t_0} : \mathtt{D}, \mathtt{C} <: \mathtt{D}, \mathtt{C} \neq \mathtt{D}$

We apply the induction hypothesis to $\mathtt{t_0}$. As before, if $\mathtt{t_0} \longrightarrow \mathtt{t_0'}$ then E-CAST applies, and if $\mathtt{t_0} = E_0[(\mathtt{C_0})(\mathtt{new\ D_0}(\bar{\mathtt{v}}))]$ with $\mathtt{D_0} \not<: \mathtt{C_0}$, and then $\mathtt{t} = E[(\mathtt{C_0})(\mathtt{new\ D_0}(\bar{\mathtt{v}}))]$ for $E = (C)E_0$. But if $\mathtt{t_0}$ is a value then we are stuck, since E-CASTNEW does not apply. However, Lemma 12.3 applied to the last step in the evaluation derivation of `t` implies that $\mathtt{t} = E[\mathtt{r}]$, for some E and `r`, and inspecting Lemma 12.3's proof shows that `r` must be of the form $\mathtt{r} = (\mathtt{C})\mathtt{new\ D}(\bar{\mathtt{v}})$, with $\mathtt{D} \not<: \mathtt{C}$ because we know that $\mathtt{C} <: \mathtt{D}$. The remaining T-SCAST case is analogous. ■

12.8 Conclusion

In this lecture we showed semantics and typing of a language where objects and classes are primitive mechanisms, and proved it type safe. By treating objects as primitive, we can reason about their operational semantics and typing behavior directly. This approach is compatible with the way users think of their programs, and is therefore useful for language design and documentation.

Chapter 18 in the book takes a different approach, encoding objects, classes, and inheritance using features from the simply typed lambda-calculus. This lower-level approach helps in understanding the way objects are translated into lower-level languages by compilers, and in understanding the interactions between objects and other language features.

Ideally, we would like to benefit from the advantages of both approaches. For this we need to define the semantics and typing of the high level language as well as a translation from this language to a simpler lower-level language. Finally, we need to prove that the translation is *correct*, i.e., that the translation preserves the evaluation and typing properties of the high-level language. Such a formulation and proof were done for FJ by League, Trifonov, and Shao [2].

Bibliography

[1] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[2] Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 24(2):112–152, 2002.