Computer Science 15-312
Featherweight Java

Robert Harper

Spring, 2000

1 Introduction

Two central ideas of object-oriented programming are the closely related concepts of *subtyping* and *inheritance*.

Subtyping is a binary relation between types that validates the *subsumption* principle — if a type τ is a subtype of a type τ', then a value of type τ may be provided whenever a value of type τ' is required. In other words, it is always valid to "weaken" the type of an expression to any supertype.

Inheritance is the ability to define a class as a "modification" of another class obtained, either by *extending* that class with new fields or methods, or by *overriding* a method in that class with a new definition. The new class is said to be a *subclass of*, or to *inherit from*, the old class. Since the methods of a class can call one another via the pseudo-variable `this`, standing for the object itself, overriding a method has the effect of changing the behavior of all other methods that may call it. This is sometimes called *method specialization*, or *dynamic dispatch*, or *late binding* — whether to call the new or the old "version" of the method is determined *at run time* by the class of the invoking method.

We will consider a tiny subset of the Java language, called *Featherweight Java*, or FJ, that models subtyping and inheritance in Java. We will then discuss design alternatives in the context of FJ. For example, in FJ, as in Java, the subtype relation is tightly coupled to the subclass relation. Is this necessary? Is it desirable? We will also use FJ as a framework for discussing other aspects of Java, including interfaces, privacy, and arrays.

2 Abstract Syntax

The abstract syntax of FJ is given by the following grammar:

$$
\begin{array}{llcl}
\textit{Classes} & C & ::= & \texttt{class}\, c\, \texttt{extends}\, c\, \{\, \overline{c}\,\overline{f}\texttt{;}\; k\,\overline{d}\,\} \\
\textit{Constructors} & k & ::= & c\texttt{(}\overline{c}\,\overline{x}\texttt{)}\,\{\,\texttt{super(}\overline{x}\texttt{);}\,\texttt{this.}\overline{f}\texttt{=}\overline{x}\texttt{;}\,\} \\
\textit{Methods} & d & ::= & c\, m\texttt{(}\overline{c}\,\overline{x}\texttt{)}\,\{\,\texttt{return}\, e\texttt{;}\,\} \\
\textit{Types} & \tau & ::= & c \\
\textit{Expressions} & e & ::= & x \mid e\texttt{.}f \mid e\texttt{.}m\texttt{(}\overline{e}\texttt{)} \mid \texttt{new}\, c\texttt{(}\overline{e}\texttt{)} \mid \texttt{(}c\texttt{)}\; e
\end{array}
$$

The variable f ranges over a set of *field names*, c over a set of *class names*, m over a set of *method names*, and x over a set of *variable names*. We assume that these sets are countably infinite and pairwise disjoint. We assume that there is a distinguished class name, `Object`, standing for the root of the class hierarchy. It's role will become clear below. We assume that there is a distinguished variable `this` that cannot otherwise be declared in a program.

As a notational convenience we use "overbarring" to stand for sequences of phrases. For example, $\overline{d}$ stands for a sequence of d's, whose individual elements we designate $d_1, \ldots, d_k$, where k is the length of the sequence. We write $\overline{c}\,\overline{f}$ for the sequence $c_1\, f_1, \ldots, c_k\, f_k$, where k is the length of the sequences $\overline{c}$ and $\overline{f}$. Similar conventions govern the other uses of sequence notation.

The class expression

$$\texttt{class}\, c\, \texttt{extends}\, c'\, \{\, \overline{c}\,\overline{f}\texttt{;}\; k\,\overline{d}\,\}$$

declares the class c to be a subclass of the class c'. The subclass has additional fields $\overline{c}\,\overline{f}$, single constructor k, and method suite $\overline{d}$. The methods of the subclass may override those of the superclass, or may be new methods specific to the subclass.

The constructor expression

$$c\texttt{(}\overline{c'}\,\overline{x'}\texttt{,}\overline{c}\,\overline{x}\texttt{)}\,\{\,\texttt{super(}\overline{x'}\texttt{);}\,\texttt{this.}\overline{f}\texttt{=}\overline{x}\texttt{;}\,\}$$

declares the constructor for class c with arguments $\overline{c'}\,\overline{x'}, \overline{c}\,\overline{x}$, corresponding to the fields of the superclass followed by those of the subclass. The variables $\overline{x'}$ and $\overline{x}$ are bound in the body of the constructor. The body of the constructor indicates the initialization of the superclass with the arguments $\overline{x'}$ and of the subclass with arguments $\overline{x}$.

The method expression

$$c\, m\texttt{(}\overline{c}\,\overline{x}\texttt{)}\,\{\,\texttt{return}\, e\texttt{;}\,\}$$

declares a method m yielding a value of class c, with arguments $\overline{x}$ of class $\overline{c}$ and body returning the value of the expression e. The variables $\overline{x}$ and `this` are bound in e.

The set of types is, for the time being, limited to the set of class names. That is, the only types are those declared by a class. In Java there are more types than just these, including the primitive types `integer` and `boolean` and the array types.

The set of expressions is the minimal "interesting" set sufficient to illustrate subtyping and inheritance. The expression $e.f$ selects the contents of field f from instance e. The expression $e.m(\overline{e})$ invokes the method m of instance e with arguments $\overline{e}$. The expression $\texttt{new}\, c(\overline{e})$ creates a new instance of class c, passing arguments $\overline{e}$ to the constructor for c. The expression $(c)\ e$ casts the value of e to class c.

The methods of a class may invoke one another by sending messages to `this`, standing for the instance itself. We may think of `this` as a bound variable of the instance, but we will arrange things so that renaming of `this` is never necessary to avoid conflicts.

A *class table* T is a finite function assigning classes to class names. The classes declared in the class table are bound within the table so that all classes may refer to one another via the class table.

A *program* is a pair (T, e) consisting of a class table T and an expression e. We generally suppress explicit mention of the class table, and consider programs to be expressions.

3 Static Semantics

The static semantics of FJ is defined by a collection of judgments of the following forms:

$\tau \texttt{ <: } \tau'$	*subtyping*
$\Gamma \vdash e : \tau$	*expression typing*
$d\ \mathsf{ok\ in}\ c$	*well-formed method*
$C\ \mathsf{ok}$	*well-formed class*
$T\ \mathsf{ok}$	*well-formed class table*
$\mathsf{fields}(c) = \overline{c}\,\overline{f}$	*field lookup*
$\mathsf{type}(m, c) = \overline{c} \to c$	*method type*

The typing rules for expressions are as follows:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_0 : c_0 \quad \mathsf{fields}(c_0) = \overline{c}\,\overline{f}}{\Gamma \vdash e_0\,.\,f_i : c_i}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : c_0 \quad \Gamma \vdash \overline{e} : \overline{c} \\ \mathsf{type}(m, c_0) = \overline{c'} \to c \quad \overline{c} \mathrel{\texttt{<:}} \overline{c'}\end{array}}{\Gamma \vdash e_0\,.\,m\,(\overline{e}) : c}$$

$$\frac{\Gamma \vdash \overline{e} : \overline{c} \quad \overline{c} \mathrel{\texttt{<:}} \overline{c'} \quad \mathsf{fields}(c) = \overline{c'}\,\overline{f}}{\Gamma \vdash \texttt{new}\, c\,(\overline{e})}$$

$$\frac{\Gamma \vdash e_0 : d}{\Gamma \vdash \texttt{(}c\texttt{)}\; e_0 : c}$$

The subtyping relation is easily read off of the class table. Subtyping is the smallest reflexive, transitive relation containing the subclass relation:

$$\frac{}{\tau \mathrel{\texttt{<:}} \tau} \qquad \frac{\tau \mathrel{\texttt{<:}} \tau' \quad \tau' \mathrel{\texttt{<:}} \tau''}{\tau \mathrel{\texttt{<:}} \tau''}$$

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\; \ldots\, ;\; \ldots \;\}}{c \mathrel{\texttt{<:}} c'}$$

Well-formed classes are defined as follows:

$$\frac{\begin{array}{c} k = c\,(\overline{c'}\,\overline{x'}, \overline{c}\,\overline{x})\;\{\; \texttt{super}\,(\overline{x'})\,\texttt{;}\; \texttt{this.}\overline{f}\texttt{=}\overline{x}\texttt{;}\; \} \\ \mathsf{fields}(c') = \overline{c'}\,\overline{f'} \quad \overline{c''}\;\mathsf{ok\;in}\; c \end{array}}{\texttt{class}\, c\, \texttt{extends}\, c' \;\{\; \overline{c}\,\overline{f}\texttt{;}\; k\, \overline{c''} \;\}\; \mathsf{ok}}$$

Methods are well-formed relative to a class, to take account of overriding:

$$\frac{\begin{array}{c} T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\; \ldots\, ;\; \ldots \;\} \\ \mathsf{type}(m, c') = \overline{c} \to c_0 \quad \overline{x}{:}\overline{c}, \texttt{this}{:}c \vdash e_0 : c_0 \end{array}}{c_0\; m\,(\overline{c}\,\overline{x})\;\{\; \texttt{return}\, e_0\,\texttt{;}\; \}\; \mathsf{ok\;in}\; c}$$

A method table is well-formed iff all of its classes are well-formed:

$$\frac{\forall c \in \mathrm{dom}(T)\;\; T(c)\; \mathsf{ok}}{T\; \mathsf{ok}}$$

Note that well-formedness of a class is relative to the method table!

A program is well-formed iff its method table is well-formed and the expression is well-formed:

$$\frac{T\; \mathsf{ok} \quad \emptyset \vdash e : \tau}{(T, e)\; \mathsf{ok}}$$

Finally, the auxiliary lookup judgments determine the types of fields and methods of an object. The types of the fields of an object are determined by the following rules:

$$\frac{}{\mathsf{fields}(\texttt{Object}) = \bullet} \qquad \frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\, \overline{c}\,\overline{f}\texttt{;}\ \ldots\ \} \quad \mathsf{fields}(c') = \overline{c'}\,\overline{f'}}{\mathsf{fields}(c) = \overline{c'}\,\overline{f'}, \overline{c}\,\overline{f}}$$

The type of a method is determined by the following rules:

$$\frac{\begin{array}{c} T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\ \ldots\texttt{;}\ \ldots\ \overline{c''}\ \} \\ c''_i = c_i\, m\, \texttt{(}\overline{c_i}\,\overline{x}\texttt{)}\, \{\, \texttt{return}\, e\texttt{;}\, \} \end{array}}{\mathsf{type}(m_i, c) = \overline{c_i} \to c_i}$$

$$\frac{\begin{array}{c} T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\ \ldots\texttt{;}\ \ldots\ \overline{c''}\ \} \\ m \notin \overline{c''} \quad \mathsf{type}(m, c') = \overline{c_i} \to c_i \end{array}}{\mathsf{type}(m, c) = \overline{c_i} \to c_i}$$

4 Dynamic Semantics

The dynamic semantics of FJ may be specified using SOS rules similar to those for MinML. We'll specify a call-by-value interpretation here.

The instruction transition rules are as follows:

$$\frac{\mathsf{fields}(c) = \overline{c'}\,\overline{f'}, \overline{c}\,\overline{f} \quad \overline{e'}\ \mathsf{value} \quad \overline{e}\ \mathsf{value}}{\texttt{new}\, c\texttt{(}\overline{e'}, \overline{e}\texttt{)}\,\texttt{.}\, f'_i \mapsto e'_i}$$

$$\frac{\mathsf{fields}(c) = \overline{c'}\,\overline{f'}, \overline{c}\,\overline{f} \quad \overline{e'}\ \mathsf{value} \quad \overline{e}\ \mathsf{value}}{\texttt{new}\, c\texttt{(}\overline{e'}, \overline{e}\texttt{)}\,\texttt{.}\, f_i \mapsto e_i}$$

$$\frac{\mathsf{body}(m, c) = \overline{x} \to e_0 \quad \overline{e}\ \mathsf{value} \quad \overline{e'}\ \mathsf{value}}{\texttt{new}\, c\texttt{(}\overline{e}\texttt{)}\,\texttt{.}\, m\texttt{(}\overline{e'}\texttt{)} \mapsto [\overline{e'}, \texttt{new}\, c\texttt{(}\overline{e}\texttt{)}/\overline{x}, \texttt{this}]e_0}$$

$$\frac{c\ \texttt{<:}\ c' \quad \overline{e}\ \mathsf{value}}{\texttt{(}c'\texttt{)}\ \texttt{new}\, c\texttt{(}\overline{e}\texttt{)} \mapsto \texttt{new}\, c\texttt{(}\overline{e}\texttt{)}}$$

The search rules are as follows:

$$\frac{e_0 \mapsto e'_0}{e_0\,\texttt{.}\, f \mapsto e'_0\,\texttt{.}\, f}$$

$$\frac{e_0 \mapsto e'_0}{e_0\,\texttt{.}\, m\texttt{(}\overline{e}\texttt{)} \mapsto e'_0\,\texttt{.}\, m\texttt{(}\overline{e}\texttt{)}}$$

$$\frac{e_0\ \mathsf{value} \quad \overline{e} \mapsto \overline{e'}}{e_0\,.\,m\,(\overline{e}) \mapsto e_0\,.\,m\,(\overline{e'})}$$

$$\frac{\overline{e} \mapsto \overline{e'}}{\mathtt{new}\,c\,(\overline{e}) \mapsto \mathtt{new}\,c\,(\overline{e'})}$$

$$\frac{e_0 \mapsto e_0'}{(c)\ e_0 \mapsto (c)\ e_0'}$$

This rules make use of the following auxiliary relation for determining the body of a method:

$$\frac{\begin{array}{c}T(c) = \mathtt{class}\,c\,\mathtt{extends}\,c'\,\{\ \ldots\,;\ \ldots\,\overline{d}\,\}\\ d_i = c_i\,m\,(\overline{c_i}\,\overline{x})\,\{\,\mathtt{return}\,e\,;\,\}\end{array}}{\mathsf{body}(m_i, c) = \overline{x} \to e}$$

$$\frac{\begin{array}{c}T(c) = \mathtt{class}\,c\,\mathtt{extends}\,c'\,\{\ \ldots\,;\ \ldots\,\overline{d}\,\}\\ m \notin \overline{d} \quad \mathsf{type}(m, d) = \overline{x} \to e\end{array}}{\mathsf{body}(m, c) = \overline{x} \to e}$$

The rules defining values are as follows:

$$\frac{\overline{e}\ \mathsf{value}}{\mathtt{new}\,c\,(\overline{e})\ \mathsf{value}}$$

Finally, we require rules for evaluating sequences of expressions from left to right, and correspondingly defining when a sequence is a value (*i.e.*, consists only of values).

$$\frac{e_1\ \mathsf{value} \quad \ldots \quad e_{i-1}\ \mathsf{value} \quad e_i \mapsto e_i'}{e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n \mapsto e_1, \ldots, e_{i-1}, e_i', e_{i+1}, \ldots, e_n}$$

$$\frac{e_1\ \mathsf{value} \quad \ldots \quad e_n\ \mathsf{value}}{\overline{e}\ \mathsf{value}}$$

This completes the dynamic semantics of FJ.

5 Type Safety

The safety of FJ is stated in the usual manner by the Preservation and Progress Theorems.

Since the dynamic semantics of casts preserves the "true" type of an instance, the type of an expression may become "smaller" in the subtype ordering during execution.

Theorem 1 (Preservation)
If $e : \tau$ *and* $e \mapsto e'$*, then* $e' : \tau'$ *for some* τ' *such that* τ' `<:` τ.

The statement of Progress must take account of the possibility that a cast may fail at execution time. Note, however, that field selection or message send can never fail — the required field or method will always be present.

Theorem 2 (Progress)
If $e : \tau$ *then either*

1. v value*, or*

2. e *contains an instruction of the form* $(c)\,\texttt{new}\,d(e_0)$ *with* e_0 value *and* $d \not<: c$*, or*

3. *there exists* e' *such that* $e \mapsto e'$.

It follows that if no casts occur in the source program, then the second case cannot arise. This can be sharpened somewhat to admit source-level casts for which it is known statically that the type of casted expression is a subtype of the target of the cast. However, we cannot predict, in general, statically whether a given cast will succeed or fail dynamically.

Lemma 3 (Canonical Forms)
If $e : c$ *and* e value*, then* e *has the form* $\texttt{new}\,d(e_0)$ *with* e_0 value *and* d `<:` c.

6 Acknowledgement

This chapter is based on, and follows closely, the paper "Featherweight Java: A Minimal Core Calculus for Java and GJ" by Atsushi Igarashi, Benjamin Pierce, and Philip Wadler.