Featherweight Java

A Minimal Core Calculus for Java and GJ

Atsushi Igarashi
Dept. of Comp. & Info. Sci.
University of Pennsylvania
igarasha@saul.cis.upenn.edu

Benjamin Pierce
Dept. of Comp. & Info. Sci.
University of Pennsylvania
bcpierce@cis.upenn.edu

Philip Wadler
Bell Laboratories
Lucent Technologies
wadler@research.bell-labs.com

Abstract

Several recent studies have introduced lightweight versions of Java: reduced languages in which complex features like threads and reflection are dropped to enable rigorous arguments about key properties such as type safety. We carry this process a step further, omitting almost all features of the full language (including interfaces and even assignment) to obtain a small calculus, Featherweight Java, for which rigorous proofs are not only possible but easy.

Featherweight Java bears a similar relation to full Java as the lambda-calculus does to languages such as ML and Haskell. It offers a similar computational "feel," providing classes, methods, fields, inheritance, and dynamic typecasts, with a semantics closely following Java's. A proof of type safety for Featherweight Java thus illustrates many of the interesting features of a safety proof for the full language, while remaining pleasingly compact. The syntax, type rules, and operational semantics of Featherweight Java fit on one page, making it easier to understand the consequences of extensions and variations.

As an illustration of its utility in this regard, we extend Featherweight Java with *generic classes* in the style of GJ (Bracha, Odersky, Stoutamire, and Wadler) and sketch a proof of type safety. The extended system formalizes for the first time some of the key features of GJ.

Subject areas: theoretical foundations, language design and implementation.

1 Introduction

"Inside every large language is a small language struggling to get out..."

Formal modeling can offer a significant boost to the design of complex real-world artifacts such as programming languages. A formal model may be used to describe some aspect of a design precisely, to state and prove its properties, and to direct attention to issues that might otherwise be overlooked. In formulating a model, however, there is a tension between completeness and compactness: the more aspects the model addresses at the same time, the more unwieldy it becomes. Often it is sensible to choose a model that is less complete but more compact, offering maximum insight for minimum investment. This strategy may be seen in a flurry of recent papers on the formal properties of Java, which omit advanced features such as concurrency and reflection and concentrate on fragments of the full language to which well-understood theory can be applied.

We propose Featherweight Java, or FJ, as a new contender for a *minimal* core calculus for modeling Java's type system. The design of FJ favors compactness over completeness almost obsessively, having just five forms of expression: object creation, method invocation, field access, casting, and variables. Its syntax, typing rules, and operational semantics fit comfortably on a single page. Indeed, our aim has been to omit as many features as possible – even assignment – while retaining the core features of Java typing. There is a direct correspondence between FJ and a purely functional core of Java, in the sense that every FJ program is literally an executable Java program.

FJ is only a little larger than Church's lambda calculus [3] or Abadi and Cardelli's object calculus [1], and is significantly smaller than previous formal models of class-based languages like Java, including those put forth by Drossopoulou, Eisenbach, and Khurshid [11], Syme [21], Nipkow and Oheimb [18], and Flatt, Krishnamurthi, and Felleisen [14, 15]. Being smaller, FJ lets us focus on just a few key issues. For example, we have discovered that capturing the behavior of Java's cast construct in a traditional "small-step" operational semantics is trickier than we would have expected, a point that has been overlooked or underemphasized in other models.

One use of FJ is as a starting point for modeling languages that extend Java. Because FJ is so compact, we can focus attention on essential aspects of the extension. Moreover, because the proof of soundness for pure FJ is very simple, a rigorous soundness proof for even a significant extension may remain manageable. The second part of the paper illustrates this utility by enriching FJ with generic classes and methods *à la* GJ [7]. The model omits some important aspects of GJ (such as "raw types" and type argument inference for generic method calls). Nonetheless, it has been a useful tool in clarifying our thought, and led to the discovery and fix of at least one bug in the GJ compiler. Because the model is small, it is easy to contemplate further extensions, and we have begun the work of adding raw types to the model; so far, this has revealed at least one corner of the design that was underspecified.

Our main goal in designing FJ was to make a proof of type soundness ("well-typed programs don't get stuck") as concise as possible, while still capturing the essence of the soundness argument for the full Java language. Any language feature that made the soundness proof *longer* without making it significantly *different* was a candidate for omission. As in previous studies of type soundness in Java, we don't treat advanced features such as concurrency, inner classes, and reflection. Other Java features omitted from FJ include assignment, interfaces, overloading, messages to `super`, `null` pointers, base types (`int`, `bool`, etc.), abstract method declarations, shadowing of superclass fields by subclass fields, access control (`public`, `private`, etc.), and exceptions. The features of Java that we *do* model include mutually recursive class definitions, object creation, field access, method invocation, method override, method recursion through `this`, subtyping, and casting.

One key simplification in FJ is the omission of assignment. We assume that an object's fields are initialized by its constructor and never changed afterwards. This restricts FJ to a "functional" fragment of Java, in which many common Java idioms, such as use of enumerations, cannot be represented. Nonetheless, this fragment is computationally complete (it is easy to encode the lambda calculus into it), and is large enough to include many useful programs (many of the programs in Felleisen and Friedman's Java text [12] use a purely functional style). Moreover, most of the tricky typing issues in both Java and GJ are independent of assignment. An important exception is that the type inference algorithm for generic method invocation in GJ has some twists imposed on it by the need to maintain soundness in the presence of assignment. This paper treats a simplified version of GJ without type inference.

The remainder of this paper is organized as follows. Section 2 introduces the main ideas of Featherweight Java, presents its syntax, type rules, and reduction rules, and sketches a type soundness proof. Section 3 extends Featherweight Java to Featherweight GJ, which includes generic classes and methods. Section 4 presents an erasure map from FGJ to FJ, modeling the techniques used to compile GJ into Java. Section 5 discusses related work, and Section 6 concludes.

2 Featherweight Java

In FJ, a program consists of a collection of class definitions plus an expression to be evaluated. (This expression corresponds to the body of the `main` method in Java.) Here are some typical class definitions in FJ.

```
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);
  }
}

class A extends Object {
  A() { super(); }
}

class B extends Object {
  B() { super(); }
}
```

For the sake of syntactic regularity, we always include the supertype (even when it is `Object`), we always write out the constructor (even for the trivial classes `A` and `B`), and we always write the receiver for a field access (as in `this.snd`) or a method invocation. Constructors always take the same stylized form: there is one parameter for each field, with the same name as the field; the `super` constructor is invoked on the fields of the supertype; and the remaining fields are initialized to the corresponding parameters. Here the supertype is always `Object`, which has no fields, so the invocations of `super` have no arguments. Constructors are the only place where `super` or `=` appears in an FJ program. Since FJ provides no side-effecting operations, a method body always consists of `return` followed by an expression, as in the body of `setfst()`.

In the context of the above definitions, the expression

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

evaluates to the expression

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

There are five forms of expression in FJ. Here, `new A()`, `new B()`, and `new Pair(e1,e2)` are *object constructors*, and `e3.setfst(e4)` is a *method invocation*. In the body of `setfst`, the expression `this.snd` is a *field access*, and the occurrences of `newfst` and `this` are *variables*. FJ differs from Java in that `this` is an ordinary variable rather than a special keyword.

The remaining form of expression is a *cast*. The expression

```
((Pair)new Pair(new Pair(new A(), new B()),
               new A()).fst).snd
```

evaluates to the expression

```
new B().
```

Here, `((Pair)e7)`, where `e7` is `new Pair(...).fst`, is a cast. The cast is required, because `e7` is a field access to `fst`, which is declared to contain an `Object`, whereas the next field access, to `snd`, is only valid on a `Pair`. At run time, it is checked whether the `Object` stored in the `fst` field is a `Pair` (and in this case the check succeeds).

In Java, one may prefix a field or parameter declaration with the keyword `final` to indicate that it may not be assigned to, and all parameters accessed from an inner class must be declared `final`. Since FJ contains no assignment and no inner classes, it matters little whether or not `final` appears, so we omit it for brevity.

Dropping side effects has a pleasant side effect: evaluation can be easily formalized entirely within the syntax of FJ, with no additional mechanisms for modeling the heap. Moreover, in the absence of side effects, the order in which expressions are evaluated does not affect the final outcome, so we can define the operational semantics of FJ straightforwardly using a nondeterministic small-step reduction relation, following long-standing tradition in the lambda calculus. Of course, Java's call-by-value evaluation strategy is subsumed by this more general relation, so the soundness properties we prove for reduction will hold for Java's evaluation strategy as a special case.

There are three basic computation rules: one for field access, one for method invocation, and one for casts. Recall that, in the lambda calculus, the beta-reduction rule for applications assumes that the function is first simplified to a lambda abstraction. Similarly, in FJ the reduction rules assume the object operated upon is first simplified to a `new` expression. Thus, just as the slogan for the lambda calculus is "everything is a function," here the slogan is "everything is an object."

Here is the rule for field access in action:

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

Because of the stylized form for object constructors, we know that the constructor has one parameter for each field, in the same order that the fields are declared. Here the fields are `fst` and `snd`, and an access to the `snd` field selects the second parameter.

Here is the rule for method invocation in action (/ denotes substitution):

$$\begin{array}{ll} & \texttt{new Pair(new A(), new B()).setfst(new B())} \\ \longrightarrow & \left[\begin{array}{l}\texttt{new B()/newfst,} \\ \texttt{new Pair(new A(),new B())/this}\end{array}\right] \\ & \quad \texttt{new Pair(newfst, this.snd)} \\ \text{i.e.,} & \quad \texttt{new Pair(new B(),} \\ & \quad\quad \texttt{new Pair(new A(), new B()).snd)} \end{array}$$

The receiver of the invocation is the object `new Pair(new A(), new B())`, so we look up the `setfst` method in the `Pair` class, where we find that it has formal parameter `newfst` and body `new Pair(newfst, this.snd)`. The invocation reduces to the body with the formal parameter replaced by the actual, and the special variable `this` replaced by the receiver object. This is similar to the beta rule of the lambda calculus, $(\lambda\texttt{x.e0})\texttt{e1} \longrightarrow [\texttt{e1/x}]\texttt{e0}$. The key differences are the fact that the class of the receiver determines where to look for the body (supporting method override), and the substitution of the receiver for `this` (supporting "recursion through self"). Readers familiar with Abadi and Cardelli's Object Calculus will see a strong similarity to their ς reduction rule [1]. In FJ, as in the lambda calculus and the pure Abadi-Cardelli calculus, if a formal parameter appears more than once in the body this may lead duplication of the actual, but since there are no side effects this causes no problems.

Here is the rule for a cast in action:

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

Once the subject of the cast is reduced to an object, it is easy to check that the class of the constructor is a subclass of the target of the cast. If so, as is the case here, then the reduction removes the cast. If not, as in the expression `(A)new B()`, then no rule applies and the computation is *stuck*, denoting a run-time error.

There are three ways in which a computation may get stuck: an attempt to access a field not declared for the class, an attempt to invoke a method not declared for the class ("message not understood"), or an attempt to cast to something other than a superclass of the class. We will prove that the first two of these never happen in well-typed programs, and the third never happens in well-typed programs that contain no downcasts (or "stupid casts"—a technicality explained below).

As usual, we allow reductions to apply to any subexpression of an expression. Here is a computation for the second example expression, where the next subexpression to be reduced is underlined at each step.

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

We will prove a type soundness result for FJ: if an expression `e` reduces to expression $\texttt{e}'$, and if `e` is well typed, then $\texttt{e}'$ is also well typed and its type is a subtype of the type of `e`.

With this informal introduction in mind, we may now proceed to a formal definition of FJ.

2.1 Syntax

The syntax, typing rules, and computation rules for FJ are given in Figure 1, with a few auxiliary functions in Figure 2.

The metavariables `A`, `B`, `C`, `D`, and `E` range over class names; `f` and `g` range over field names; `m` ranges over method names; `x` ranges over parameter names; `d` and `e` range over expressions; `CL` ranges over class declarations; `K` ranges over constructor declarations; and `M` ranges over method declarations. We write $\overline{\texttt{f}}$ as shorthand for $\texttt{f}_1,\ldots,\texttt{f}_n$ (and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, etc.) and write $\overline{\texttt{M}}$ as shorthand for $\texttt{M}_1\ldots\texttt{M}_n$ (with no commas). We write the empty sequence as $\bullet$ and denote concatenation of sequences using a comma. The length of a sequence $\overline{\texttt{x}}$ is written $\#(\overline{\texttt{x}})$. We abbreviate operations on pairs of sequences in the obvious way, writing "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" as shorthand for "$\texttt{C}_1\ \texttt{f}_1,\ldots,\texttt{C}_n\ \texttt{f}_n$", and similarly "$\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}$" as shorthand for "$\texttt{C}_1\ \texttt{f}_1\texttt{;}\ldots\texttt{C}_n\ \texttt{f}_n\texttt{;}$", and "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$" as shorthand for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;}\ldots\texttt{;this.f}_n\texttt{=f}_n\texttt{;}$". Sequences of field declarations, parameter names, and method declarations are assumed to contain no duplicate names.

A class table CT is a mapping from class names `C` to class declarations `CL`. A program is a pair $(CT, \texttt{e})$ of a class table and an expression. To lighten the notation in what follows, we always assume a *fixed* class table CT.

The abstract syntax of FJ class declarations, constructor declarations, method declarations, and expressions is given at the top left of Figure 1. As in Java, we assume that casts bind less tightly than other forms of

Syntax:

$$
\begin{array}{lcl}
\mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \\
\mathtt{K} & ::= & \mathtt{C(\overline{C}\ \overline{f})\ \{super(\overline{f});\ this.\overline{f}\ =\ \overline{f};\}} \\
\mathtt{M} & ::= & \mathtt{C\ m(\overline{C}\ \overline{x})\ \{return\ e;\}} \\
\mathtt{e} & ::= & \mathtt{x} \\
 & | & \mathtt{e.f} \\
 & | & \mathtt{e.m(\overline{e})} \\
 & | & \mathtt{new\ C(\overline{e})} \\
 & | & \mathtt{(C)e}
\end{array}
$$

Subtyping:

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

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

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

Computation:

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathtt{(new\ C(\overline{e})).f}_i \longrightarrow \mathtt{e}_i} \quad \text{(R-Field)}$$

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{e}_0)}{\begin{array}{c}\mathtt{(new\ C(\overline{e})).m(\overline{d})} \\ \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{new\ C(\overline{e})/this}]\mathtt{e}_0\end{array}} \quad \text{(R-Invk)}$$

$$\frac{\mathtt{C} <: \mathtt{D}}{\mathtt{(D)(new\ C(\overline{e}))} \longrightarrow \mathtt{new\ C(\overline{e})}} \quad \text{(R-Cast)}$$

Expression typing:

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

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

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

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) \in \mathtt{C}} \quad \text{(T-New)}$$

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

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

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \\ \mathit{stupid\ warning}\end{array}}{\Gamma \vdash \mathtt{(C)e}_0 \in \mathtt{C}} \quad \text{(T-SCast)}$$

Method typing:

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

Class typing:

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

Figure 1: FJ: Main definitions

Field lookup:

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

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

Method type lookup:

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

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

Method body lookup:

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

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

Valid method overriding:

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

Figure 2: FJ: Auxiliary definitions

expression. We assume that the set of variables includes the special variable `this`, but that `this` is never used as the name of an argument to a method.

Every class has a superclass, declared with `extends`. This raises a question: what is the superclass of the `Object` class? There are various ways to deal with this issue; the simplest one that we have found is to take `Object` as a distinguished class name whose definition does *not* appear in the class table. The auxiliary functions that look up fields and method declarations in the class table are equipped with special cases for `Object` that return the empty sequence of fields and the empty set of methods. (In full Java, the class `Object` does have several methods. We ignore these in FJ.)

By looking at the class table, we can read off the subtype relation between classes. We write `C <: D` when `C` is a subtype of `D` – i.e., subtyping is the reflexive and transitive closure of the immediate subclass relation given by the `extends` clauses in CT. Formally, it is defined in the middle of the left column of Figure 1.

The given class table is assumed to satisfy some sanity conditions: (1) $CT(\texttt{C}) = \texttt{class C...}$ for every $\texttt{C} \in \mathit{dom}(CT)$; (2) $\texttt{Object} \notin \mathit{dom}(CT)$; (3) for every class name `C` (except `Object`) appearing anywhere in CT, we have $\texttt{C} \in \mathit{dom}(CT)$; and (4) there are no cycles in the subtype relation induced by CT – that is, the `<:` relation is antisymmetric.

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 2. The fields of a class `C`, written $\mathit{fields}(\texttt{C})$, is a sequence $\overline{\texttt{C}}\ \overline{\texttt{f}}$ pairing the class of a field with its name, for all the fields declared in class `C` and all of its superclasses. The type of the method `m` in class `C`, written $\mathit{mtype}(\texttt{m}, \texttt{C})$, is a pair, written $\overline{\texttt{B}} \rightarrow \texttt{B}$, of a sequence of argument types $\overline{\texttt{B}}$ and a result type `B`. Similarly, the body of the method `m` in class `C`, written $\mathit{mbody}(\texttt{m}, \texttt{C})$, is a pair, written $(\overline{\texttt{x}}\texttt{,e})$, of a sequence of parameters $\overline{\texttt{x}}$ and an expression `e`. The predicate $\mathit{override}(\texttt{C}_0 \rightarrow \overline{\texttt{C}}, \texttt{m}, \texttt{D})$ judges if a method `m` with argument types $\overline{\texttt{C}}$ and a result type $\texttt{C}_0$ may be defined in a subclass of `D`. In case of overriding, if a method with the same name is declared in the superclass then it must have the same type.

2.2 Typing

The typing rules for expressions, method declarations, and class declarations are in the right column of Figure 1. An environment Γ is a finite mapping from variables to types, written $\overline{\texttt{x}}\texttt{:}\overline{\texttt{C}}$.

The typing judgment for expressions has the form $\Gamma \vdash \texttt{e} \in \texttt{C}$, read "in the environment Γ, expression `e` has type `C`." The typing rules are syntax directed, with one rule for each form of expression, save that there are three rules for casts. The typing rules for constructors and method invocations check that each actual parameter has a type that is a subtype of the corresponding formal. We abbreviate typing judgments on sequences in the obvious way, writing $\Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{C}}$ as shorthand for $\Gamma \vdash \texttt{e}_1 \in \texttt{C}_1, \ldots, \Gamma \vdash \texttt{e}_n \in \texttt{C}_n$ and writing $\overline{\texttt{C}}$ `<:` $\overline{\texttt{D}}$ as shorthand for $\texttt{C}_1$ `<:` $\texttt{D}_1, \ldots, \texttt{C}_n$ `<:` $\texttt{D}_n$.

One technical innovation in FJ is the introduction of "stupid" casts. There are three rules for type casts: in an *upcast* the subject is a subclass of the target, in a *downcast* the target is a subclass of the subject, and in a *stupid* cast the target is unrelated to the subject. The Java compiler rejects as ill typed an expression containing a stupid cast, but we must allow stupid casts in FJ if we are to formulate type soundness as a subject reduction theorem for a small-step semantics. This is because a sensible expression may be reduced to one containing a stupid cast. For example, consider the following, which uses classes `A` and `B` as defined as in the previous section:

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

We indicate the special nature of stupid casts by including the hypothesis *stupid warning* in the type rule for stupid casts (T-SCAST); an FJ typing corresponds to a legal Java typing only if it does not contain this rule.

(Stupid casts were omitted from Classic Java [14], causing its published proof of type soundness to be incorrect; this error was discovered independently by ourselves and the Classic Java authors.)

The typing judgment for method declarations has the form `M OK IN C`, read "method declaration `M` is ok if it occurs in class `C`." It uses the expression typing judgment on the body of the method, where the free variables are the parameters of the method with their declared types, plus the special variable `this` with type `C`.

The typing judgment for class declarations has the form `CL OK`, read "class declaration `CL` is ok." It checks that the constructor applies `super` to the fields of the superclass and initializes the fields declared in this class, and that each method declaration in the class is ok.

The type of an expression may depend on the type of any methods it invokes, and the type of a method depends on the type of an expression (its body), so it behooves us to check that there is no ill-defined circularity here. Indeed there is none: the circle is broken because the type of each method is explicitly declared. It is possible to load and use the class table before all the classes in it are checked, so long as each class is eventually checked.

2.3 Computation

The reduction relation is of the form $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression `e` reduces to expression $\mathtt{e}'$ in one step." We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

The reduction rules are given in the bottom left column of Figure 1. There are three reduction rules, one for field access, one for method invocation, and one for casting. These were already explained in the introduction to this section. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ for the result of replacing $\mathtt{x}_1$ by $\mathtt{d}_1$, ..., $\mathtt{x}_n$ by $\mathtt{d}_n$, and `y` by `e` in expression $\mathtt{e}_0$.

The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules (if $\mathtt{e} \longrightarrow \mathtt{e}'$ then $\mathtt{e.f} \longrightarrow \mathtt{e}'\mathtt{.f}$, and the like), which we omit here.

2.4 Properties

Formal definitions are fun, but the proof of the pudding is in... well, the proof. If our definitions are sensible, we should be able to prove a type soundness result, which relates typing to computation. Indeed we can prove such a result: if a term is well typed and it reduces to a second term, then the second term is well typed, and furthermore its type is a subtype of the type of the first term.

2.4.1 Theorem [Subject Reduction]: If $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ and $\mathtt{e} \longrightarrow \mathtt{e}'$, then $\Gamma \vdash \mathtt{e}' \in \mathtt{C}'$ for some $\mathtt{C}' <: \mathtt{C}$.

Proof sketch: The main property required in the proof is the following term-substitution lemma:

> If $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{E}} \vdash \mathtt{e} \in \mathtt{D}$, and $\Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{D}}$ where $\overline{\mathtt{D}} <: \overline{\mathtt{E}}$, then $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{C}$ for some $\mathtt{C} <: \mathtt{D}$.

This is proved by induction on the derivation of $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{E}} \vdash \mathtt{e} \in \mathtt{D}$. An interesting case is when $\mathtt{e} = (\mathtt{C})\mathtt{e}_0$, where the final rule used in the derivation is T-DCast. Suppose the type of $\mathtt{e}_0$ is $\mathtt{C}_0$ and $\mathtt{C} <: \mathtt{C}_0$. By the induction hypothesis, $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{D}_0$ for some $\mathtt{D}_0 <: \mathtt{C}_0$. But, since $\mathtt{D}_0$ and `C` may or may not be in the subtype relation, the derivation of $\Gamma \vdash (\mathtt{C})[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{C}$ may involve a *stupid warning*. On the other hand, if $(\mathtt{C})\mathtt{e}_0$ is derived using T-UCast, then $(\mathtt{C})[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}$ will also be an upcast.

The theorem itself is now proved by induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$, with a case analysis on the last rule used. The case for R-Invk is easy, using the lemma above. Other base cases are also straightforward, as are most of the induction steps. The only interesting case is the congruence rule for casting—that is, the case where $(\mathtt{C})\mathtt{e} \longrightarrow (\mathtt{C})\mathtt{e}'$ is derived using $\mathtt{e} \longrightarrow \mathtt{e}'$. Using an argument similar to the term substitution lemma above, we see that a downcast expression may be reduced to a stupid cast and an upcast expression will be always reduced to an upcast. ∎

We can also show that if a program is well typed, then the only way it can get stuck is if it reaches a point where it cannot perform a downcast.

2.4.2 Theorem [Progress]: Suppose `e` is a well-typed expression.

(1) If `e` includes $\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}})\mathtt{.f}$ as a subexpression, then $\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ and $\mathtt{f} \in \overline{\mathtt{f}}$.

(2) If `e` includes $\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}})\mathtt{.m}(\overline{\mathtt{d}})$ as a subexpression, then $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0) = (\overline{\mathtt{x}}, \mathtt{e}_0)$ and $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$.

To state a similar property for casts, we say that an expression `e` is *safe* in Γ if the type derivations of the underlying CT and $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ contain no downcasts or stupid casts (uses of rules T-DCast or T-SCast). In other words, a safe program includes only upcasts. Then we see that a safe expression always reduces to another safe expression, and, moreover, typecasts in a safe expression will never fail, as shown in the following pair of theorems.

2.4.3 Theorem: [Reduction preserves safety] If `e` is safe in Γ and $\mathtt{e} \longrightarrow \mathtt{e}'$, then $\mathtt{e}'$ is safe in Γ.

2.4.4 Theorem [Progress of safe programs]: Suppose `e` is safe in Γ. If `e` has $(\mathtt{C})\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}})$ as a subexpression, then $\mathtt{C}_0 <: \mathtt{C}$.

3 Featherweight GJ

Just as GJ adds generic types to Java, Featherweight GJ (or FGJ, for short) adds generic types to FJ. Here is the class definition for pairs in FJ, rewritten with generic type parameters in FGJ.

```
class Pair<X extends Object, Y extends Object>
        extends Object {
  X fst;
  Y snd;
  Pair(X fst, Y snd) {
    super(); this.fst=fst; this.snd=snd;
  }
  <Z extends Object>
        Pair<Z,Y> setfst(Z newfst) {
    return new Pair<Z,Y>(newfst, this.snd);
  }
```

```
}

class A extends Object {
  A { super(); }
}

class B extends Object {
  B { super(); }
}
```

Both classes and methods may have generic type parameters. Here X and Y are parameters of the class, and Z is a parameter of the `setfst` method. Each type parameter has a *bound*; here X, Y, and Z are each bounded by `Object`.

In the context of the above definitions, the expression

```
new Pair<A,B>(new A(), new B()).setfst<B>(new B())
```

evaluates to the expression

```
new Pair<B,B>(new B(), new B())
```

If we were being extraordinarily pedantic, we would write `A<>` and `B<>` instead of `A` and `B`, but we allow the latter as an abbreviation for the former in order that FJ is a proper subset of FGJ.

In GJ, type parameters to generic method invocations are inferred. Thus, in GJ the expression above would be written

```
new Pair<A,B>(new A(), new B()).setfst(new B())
```

with no `<B>` in the invocation of `setfst`. So while FJ is a subset of Java, FGJ is not quite a subset of GJ. We regard FGJ as an intermediate language – the form that would result after type parameters have been inferred. While parameter inference is an important aspect of GJ, we chose in FGJ to concentrate on modeling other aspects of GJ.

The bound of a type variable may not be a type variable, but may be a type expression involving type variables, and may be recursive (or even, if there are several bounds, mutually recursive). For example, if `C<X>` and `D<Y>` are classes with one parameter each, one may have bounds such as `<X extends C<X>>` or even `<X extends C<Y>, Y extends D<X>>`. For more on bounds, including examples of the utility of recursive bounds, see the GJ paper [7].

GJ and FGJ are intended to support either of two implementation styles. They may be implemented *directly*, augmenting the run-time system to carry information about type parameters, or they may be implemented by *erasure*, removing all information about type parameters at run-time. This section explores the first style, giving a direct semantics for FGJ that maintains type parameters, and proving a type soundness theorem. Section 4 explores the second style, giving an erasure mapping from FGJ into FJ and showing a correspondence between reductions on FGJ expressions and reductions on FJ expressions. The second style corresponds to the current implementation of GJ, which compiles GJ into the Java Virtual Machine (JVM), which of course maintains no information about type parameters at run-time; the first style would correspond to using an augmented JVM that maintains information about type parameters.

3.1 Syntax

In what follows, for the sake of conciseness we abbreviate the keyword `extends` to the symbol $\triangleleft$ and the keywork `return` to the symbol $\uparrow$.

The syntax, typing rules, and computation rules for FGJ are given in Figure 3, with a few auxiliary functions in Figure 4. The metavariables X, Y, and Z range over type variables; T, U, and V range over types; and N and O range over nonvariable types (types other than type variables). We write $\overline{\mathtt{X}}$ as shorthand for $\mathtt{X}_1,\ldots,\mathtt{X}_n$ (and similarly for $\overline{\mathtt{T}}$, $\overline{\mathtt{N}}$, etc.), and assume sequences of type variables contain no duplicate names.

The abstract syntax of FGJ is given at the top left of Figure 3. We allow `C<>` and `m<>` to be abbreviated as C and m, respectively.

As before, we assume a fixed class table CT, which is a mapping from class names C to class declarations CL, obeying the same sanity conditions as given previously.

3.2 Typing

A type environment Δ is a finite mapping from type variables to nonvariable types, written $\overline{\mathtt{X}}$ <: $\overline{\mathtt{N}}$, that takes each type variable to its bound.

Bounds of types

We write $bound_\Delta(\mathtt{T})$ for the upper bound of T in Δ, as defined in Figure 4. Unlike calculi such as $F_\leq$ [9], this promotion relation does not need to be defined recursively: the bound of a type variable is always a nonvariable type.

Subtyping

The subtyping relation is defined in the left column of Figure 3. As before, subtyping is the reflexive and transitive closure of the $\triangleleft$ relation. Type parameters are *invariant* with regard to subtyping (for reasons explained in the GJ paper), so $\overline{\mathtt{T}}$ <: $\overline{\mathtt{U}}$ does *not* imply $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$ <: $\mathtt{C}\langle\overline{\mathtt{U}}\rangle$.

Well-formed types

If the declaration of a class C begins `class` $\mathtt{C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle$, then a type like $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$ is well formed only if substituting $\overline{\mathtt{T}}$ for $\overline{\mathtt{X}}$ respects the bounds $\overline{\mathtt{N}}$, that is if $\overline{\mathtt{T}}$ <: $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$. We write $\Delta \vdash \mathtt{T}$ ok if type T is well-formed in context Δ. The rules for well-formed types appear in Figure 3. Note that we perform a simultaneous substitution, so any variable in $\overline{\mathtt{X}}$ may appear in $\overline{\mathtt{N}}$, permitting recursion and mutual recursion between variables and bounds.

A type environment Δ is well formed if $\Delta \vdash \Delta(\mathtt{X})$ ok for all X in $dom(\Delta)$. We also say that an environment Γ is well formed with respect to Δ, written $\Delta \vdash \Gamma$ ok, if $\Delta \vdash \Gamma(\mathtt{x})$ ok for all x in $dom(\Gamma)$.

Field and method lookup

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 4; these are fairly straightforward adaptations of the lookup rules given previously. The fields of a nonvariable type N, written $\mathit{fields}(\mathtt{N})$, are a sequence of corresponding types and field names, $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$. The type of the method invocation m

Syntax:

$$\begin{array}{lcl}
\texttt{CL} & ::= & \texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\triangleleft\texttt{N \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\
\texttt{K} & ::= & \texttt{C(}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{) \{super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{;\}} \\
\texttt{M} & ::= & \texttt{<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{> T m (}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e;\}} \\
\texttt{e} & ::= & \texttt{x} \\
 & | & \texttt{e.f} \\
 & | & \texttt{e.m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \\
 & | & \texttt{new N(}\overline{\texttt{e}}\texttt{)} \\
 & | & \texttt{(N)e} \\
\texttt{T} & ::= & \texttt{X} \\
 & | & \texttt{N} \\
\texttt{N} & ::= & \texttt{C<}\overline{\texttt{T}}\texttt{>}
\end{array}$$

Subtyping:

$$\Delta \vdash \texttt{T} <: \texttt{T}$$

$$\frac{\Delta \vdash \texttt{S} <: \texttt{T} \qquad \Delta \vdash \texttt{T} <: \texttt{U}}{\Delta \vdash \texttt{S} <: \texttt{U}}$$

$$\Delta \vdash \texttt{X} <: \Delta(\texttt{X})$$

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\triangleleft\texttt{N \{...\}}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}}$$

Well-formed types:

$$\Delta \vdash \texttt{Object}\ \text{ok}$$

$$\frac{\texttt{X} \in dom(\Delta)}{\Delta \vdash \texttt{X}\ \text{ok}}$$

$$\frac{\begin{array}{c}CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\triangleleft\texttt{N \{...\}} \\ \Delta \vdash \overline{\texttt{T}}\ \text{ok} \qquad \Delta \vdash \overline{\texttt{T}} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}}\end{array}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ \text{ok}}$$

Computation:

$$\frac{\mathit{fields}(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\texttt{(new N(}\overline{\texttt{e}}\texttt{)).f}_i \longrightarrow \texttt{e}_i}$$

$$\frac{\mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{N}) = (\overline{\texttt{x}}, \texttt{e}_0)}{\begin{array}{l}\texttt{(new N(}\overline{\texttt{e}}\texttt{)).m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)} \\ \longrightarrow [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new N(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}_0\end{array}}$$

$$\frac{\emptyset \vdash \texttt{N} <: \texttt{O}}{\texttt{(O)(new N(}\overline{\texttt{e}}\texttt{))} \longrightarrow \texttt{new N(}\overline{\texttt{e}}\texttt{)}}$$

Expression typing:

$$\Delta; \Gamma \vdash \texttt{x} \in \Gamma(\texttt{x})$$

$$\frac{\Delta; \Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\texttt{T}_0)) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\Delta; \Gamma \vdash \texttt{e}_0\texttt{.f}_i \in \texttt{T}_i}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \\ \mathit{mtype}(\texttt{m}, \mathit{bound}_\Delta(\texttt{T}_0)) = \texttt{<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{O}}\texttt{>}\overline{\texttt{U}}\texttt{→U} \\ \Delta \vdash \overline{\texttt{V}}\ \text{ok} \qquad \Delta \vdash \overline{\texttt{V}} <: [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{O}} \\ \Delta; \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \qquad \Delta \vdash \overline{\texttt{S}} <: [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}}\end{array}}{\Delta; \Gamma \vdash \texttt{e}_0\texttt{.m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}}$$

$$\frac{\begin{array}{c}\Delta \vdash \texttt{N}\ \text{ok} \qquad \mathit{fields}(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \\ \Delta; \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \qquad \Delta \vdash \overline{\texttt{S}} <: \overline{\texttt{T}}\end{array}}{\Delta; \Gamma \vdash \texttt{new N(}\overline{\texttt{e}}\texttt{)} \in \texttt{N}}$$

$$\frac{\Delta; \Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \qquad \Delta \vdash \texttt{T}_0 <: \texttt{N}}{\Delta; \Gamma \vdash \texttt{(N)e}_0 \in \texttt{N}}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \qquad \Delta \vdash \texttt{N}\ \text{ok} \\ \Delta \vdash \texttt{N} <: \mathit{bound}_\Delta(\texttt{T}_0) \qquad \texttt{N} \neq \mathit{bound}_\Delta(\texttt{T}_0) \\ \mathit{downcast}(\texttt{N}, \mathit{bound}_\Delta(\texttt{T}_0))\end{array}}{\Delta; \Gamma \vdash \texttt{(N)e}_0 \in \texttt{N}}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \qquad \Delta \vdash \texttt{N}\ \text{ok} \\ \Delta \vdash \mathit{bound}_\Delta(\texttt{T}_0) \not<: \texttt{N} \qquad \Delta \vdash \texttt{N} \not<: \mathit{bound}_\Delta(\texttt{T}_0) \\ \mathit{stupid\ warning}\end{array}}{\Delta; \Gamma \vdash \texttt{(N)e}_0 \in \texttt{N}}$$

Method typing:

$$\frac{\begin{array}{c}\Delta = \overline{\texttt{X}} <: \overline{\texttt{N}}, \overline{\texttt{Y}} <: \overline{\texttt{O}} \\ \Delta \vdash \overline{\texttt{T}}\ \text{ok} \qquad \Delta \vdash \texttt{T}\ \text{ok} \qquad \Delta \vdash \overline{\texttt{O}}\ \text{ok} \\ \Delta; \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C<}\overline{\texttt{X}}\texttt{>} \vdash \texttt{e}_0 \in \texttt{S} \qquad \Delta \vdash \texttt{S} <: \texttt{T} \\ CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\triangleleft\texttt{N \{...\}} \\ \mathit{override}(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Z}}\triangleleft\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\texttt{→U})\end{array}}{\texttt{<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{O}}\texttt{> T m (}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e}_0\texttt{;\} OK IN C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}}$$

Class typing:

$$\frac{\begin{array}{c}\overline{\texttt{X}} <: \overline{\texttt{N}} \vdash \overline{\texttt{N}}\ \text{ok} \qquad \overline{\texttt{X}} <: \overline{\texttt{N}} \vdash \texttt{N}\ \text{ok} \qquad \overline{\texttt{X}} <: \overline{\texttt{N}} \vdash \overline{\texttt{T}}\ \text{ok} \\ \mathit{fields}(\texttt{N}) = \overline{\texttt{U}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\ \texttt{OK IN C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>} \\ \texttt{K = C(}\overline{\texttt{U}}\ \overline{\texttt{g}}\texttt{, }\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{) \{super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{;\}}\end{array}}{\texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\triangleleft\texttt{N \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\} OK}}$$

Figure 3: FGJ: Main definitions

Bound of type:

$$\begin{aligned} bound_\Delta(\mathtt{X}) &= \Delta(\mathtt{X}) \\ bound_\Delta(\mathtt{N}) &= \mathtt{N}. \end{aligned}$$

Field lookup:

$$fields(\mathtt{Object}) = \bullet$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad fields([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{fields(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}}$$

Method type lookup:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{O}}\mathtt{>\ U\ m\ (}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\ \{\uparrow e;\}} \in \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{O}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = mtype(\mathtt{m}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})}$$

Method body lookup:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{O}}\mathtt{>\ U\ m\ (}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\ \{\uparrow e_0;\}} \in \overline{\mathtt{M}}}{mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = (\overline{\mathtt{x}}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{e}_0)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}}{mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})}$$

Valid method overriding:

$$\frac{mtype(\mathtt{m}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U} \text{ implies } \overline{\mathtt{O}}, \overline{\mathtt{T}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}](\overline{\mathtt{P}}, \overline{\mathtt{U}}) \text{ and } \Delta \vdash \mathtt{T} <: [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}]\mathtt{U}}{override(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U})}$$

Valid downcast:

$$\frac{\Delta \vdash \mathtt{C<}\overline{\mathtt{S}}\mathtt{>} <: \mathtt{T} \text{ and } \Delta \vdash \mathtt{C<}\overline{\mathtt{S}}\mathtt{>} \text{ ok implies } \overline{\mathtt{S}} = \overline{\mathtt{T}} \text{ for all } \overline{\mathtt{S}}}{downcast(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}, \mathtt{T})}$$

Figure 4: FGJ: Auxiliary definitions

at nonvariable type N, written $mtype(\mathtt{m},\mathtt{N})$, is a type of the form $\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U}$. Similarly, the body of the method invocation m at nonvariable type N with type parameters $\overline{\mathtt{V}}$, written $mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},\mathtt{N})$, is a pair, written $(\overline{\mathtt{x}},\mathtt{e})$, of a sequence of parameters $\overline{\mathtt{x}}$ and an expression e.

Typing rules

Typing rules for expressions, methods, and classes appear in Figure 3.

The typing judgment for expressions is of form $\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}$, read as "in the type environment Δ and the environment Γ, e has type T." Most of the subtleties are in the field and method lookup relations that we have already seen; the typing rules themselves are straightforward.

In the rule GT-DCast, the last premise ensures that the result of the cast will be the same at run time, no matter whether we use the high-level (type-passing) reduction rules defined later in this section or the erasure semantics considered in Section 4. For example, suppose we have defined:

```
class List<X ◁ Object> ◁ Object { ... }
class LinkedList<X ◁ Object> ◁ List<X> { ... }
```

Now, if o has type Object, then the cast (List<C>)o is not permitted. (If, at run time, o is bound to new List<D>(), then the cast would fail in the type-passing semantics but succeed in the erasure semantics, since (List<C>)o erases to (List)o while both new List<C>() and new List<D>() erase to new List().) On the other hand, if cl has type List<C>, then the cast (LinkedList<C>)cl is permitted, since the type-passing and erased versions of the cast are guaranteed to either both succeed or both fail.

The typing rule for methods contains one additional subtlety. In FGJ (and GJ), unlike in FJ (and Java), covariant subtyping of method results is allowed. That is, the result type of a method may be a subtype of the result type of the corresponding method in the superclass, although the bounds of type variables and the argument types must be identical (modulo renaming of type variables).

As before, a class table is ok if all its class definitions are ok.

3.3 Reduction

The operational semantics of FGJ programs is only a little more complicated than what we had in FJ. The rules appear in Figure 3.

3.4 Properties

FGJ programs enjoy subject reduction and progress properties exactly like programs in FJ (2.4.1 and 2.4.2). The basic structures of the proofs are similar to those of Theorem 2.4.1 and 2.4.2. For subject reduction, however, since we now have parametric polymorphism combined with subtyping, we need a few more lemmas. The main lemmas required are a term substitution lemma as before, plus similar lemmas about the preservation of subtyping and typing under *type* substitution. (Readers familiar with proofs of subject reduction for typed lambda-calculi like $F_{\leq}$ [9] will notice many similarities). We begin with the three substitution lemmas, which are proved by straightforward induction on a derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$ or $\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}$.

3.4.1 Lemma: [Type substitution preserves subtyping] If $\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{S} \mathtt{<:} \mathtt{T}$ and $\Delta_1 \vdash \overline{\mathtt{U}} \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ with $\Delta_1 \vdash \overline{\mathtt{U}}$ ok, and none of $\overline{\mathtt{X}}$ appearing in Δ_1, then $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S} \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.

3.4.2 Lemma: [Type substitution preserves typing] If Δ_1, $\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}$, $\Delta_2; \Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\Delta_1 \vdash \overline{\mathtt{U}} \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ where $\Delta_2 \vdash \overline{\mathtt{U}}$ ok and none of $\overline{\mathtt{X}}$ appears in Δ_1, then Δ_1, $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e} \in \mathtt{S}$ for some $\mathtt{S}$ such that Δ_1, $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S} \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.

3.4.3 Lemma: [Term substitution preserves typing] If $\Delta; \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} \in \mathtt{T}$ and, $\Delta; \Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{S}}$ where $\Delta \vdash \overline{\mathtt{S}} \mathtt{<:} \overline{\mathtt{T}}$, then $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{S}$ for some $\mathtt{S}$ such that $\Delta \vdash \mathtt{S} \mathtt{<:} \mathtt{T}$.

3.4.4 Theorem [Subject reduction]: If $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\mathtt{e} \longrightarrow \mathtt{e}'$, then $\Delta; \Gamma \vdash \mathtt{e}' \in \mathtt{T}'$, for some $\mathtt{T}'$ such that $\Delta \vdash \mathtt{T}' \mathtt{<:} \mathtt{T}$.

Proof sketch: By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$ with a case analysis on the reduction rule used. We show in detail just the base case where $\mathtt{e}$ is a method invocation. From the premises of the rule GR-Invk, we have

$$\begin{array}{l} \mathtt{e} = \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{d}}\mathtt{)} \\ \mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e}_0) \\ \mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)/this}]\mathtt{e}_0. \end{array}$$

By the rule GT-Invk and GT-New, we also have

$$\begin{array}{l} \Delta; \Gamma \vdash \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{N} \\ \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{N})) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{O}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to U} \\ \Delta \vdash \overline{\mathtt{V}} \mathtt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{O}} \\ \Delta \vdash \overline{\mathtt{V}} \text{ ok} \\ \Delta; \Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{S}} \\ \Delta \vdash \overline{\mathtt{S}} \mathtt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \\ \mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}. \end{array}$$

By examining the derivation of $\mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{N}))$, we can find a supertype $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ of $\mathtt{N}$ where

$$\begin{array}{l} \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{O}}; \overline{\mathtt{x}} : \overline{\mathtt{U}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \vdash \mathtt{e}_0 \in \mathtt{S} \\ \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{O}} \vdash \mathtt{S} \mathtt{<:} \mathtt{U} \end{array}$$

and none of the $\overline{\mathtt{Y}}$ appear in $\overline{\mathtt{T}}$. Now, by Lemma 3.4.2,

$$\emptyset; \overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \vdash \mathtt{e}_0 \in [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{S}.$$

From this, a straightforward weakening lemma (not shown here), plus Lemma 3.4.3 and Lemma 3.4.1, gives

$$\begin{array}{l} \Delta; \Gamma \vdash \mathtt{e}' \in \mathtt{S}' \\ \Delta \vdash \mathtt{S}' \mathtt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{S} \\ \Delta \vdash [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{S} \mathtt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}. \end{array}$$

Letting $\mathtt{T}' = \mathtt{S}'$ finishes the case, since $\Delta \vdash \mathtt{S}' \mathtt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$ by S-Trans. ∎

3.4.5 Theorem [Progress]: Suppose $\mathtt{e}$ is a well-typed expression.

(1) If $\mathtt{e}$ includes $\mathtt{new\ N_0(}\overline{\mathtt{e}}\mathtt{).f}$ as a subexpression, then $\mathit{fields}(\mathtt{N}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ and $\mathtt{f} \in \overline{\mathtt{f}}$.

(2) If $\mathtt{e}$ includes $\mathtt{new\ N_0(}\overline{\mathtt{e}}\mathtt{).m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{d}}\mathtt{)}$ as a subexpression, then $\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{N}_0) = (\overline{\mathtt{x}}, \mathtt{e}_0)$ and $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$.

FGJ is backward compatible with FJ. Intuitively, this means that an implementation of FGJ can be used to typecheck and execute FJ programs without changing their meaning. We can show that a well-typed FJ program is always a well-typed FGJ program and that FJ and FGJ reduction correspond. (Note that it isn't quite the case that the well-typedness of an FJ program under the FGJ rules implies its well-typedness in FJ, because FGJ allows covariant overriding and FJ does not.) In the statement of the theorem, we use $\longrightarrow_{\text{FJ}}$ and $\longrightarrow_{\text{FGJ}}$ to show which set of reduction rules is used.

3.4.6 Theorem [Backward compatibility]: If an FJ program ($\mathtt{e}$, CT) is well typed under the typing rules of FJ, then it is also well-typed under the rules of FGJ. Moreover, for all FJ programs $\mathtt{e}$ and $\mathtt{e}'$ (whether well typed or not), $\mathtt{e} \longrightarrow_{\text{FJ}} \mathtt{e}'$ iff $\mathtt{e} \longrightarrow_{\text{FGJ}} \mathtt{e}'$.

Proof: The first half is shown by straightforward induction on the derivation of $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ (using FJ typing rules), followed by an analysis of the rules GT-Method and GT-Class. In the second half, both directions are shown by induction on a derivation of the reduction relation, with a case analysis on the last rule used. ∎

4 Compiling FGJ to FJ

We now explore the second implementation style for GJ and FGJ. The current GJ compiler works by translation into the standard JVM, which maintains no information about type parameters at run-time. We model this compilation in our framework by an *erasure* translation from FGJ into FJ. We show that this translation maps well-typed FGJ programs into well-typed FJ programs, and that the behavior of a program in FGJ matches (in a suitable sense) the behavior of its erasure under the FJ reduction rules.

A program is erased by replacing types with their erasures, inserting downcasts where required. A type is erased by removing type parameters, and replacing type variables with the erasure of their bounds. For example, the class `Pair<X,Y>` in the previous section erases to the following:

```
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);
  }
}
```

Similarly, the field selection

```
new Pair<A,B>(new A(), new B()).snd
```

erases to

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

where the added downcast `(B)` recovers type information of the original program. We call such downcasts inserted by erasure *synthetic*.

4.1 Erasure of Types

To erase a type, we remove any type parameters and replace type variables with the erasure of their bounds. Write $|\mathtt{T}|_\Delta$ for the erasure of type `T` with respect to type environment Δ

$$|\mathtt{T}|_\Delta = \mathtt{C}$$

where $\mathit{bound}_\Delta(\mathtt{T}) = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$.

4.2 Field and Method Lookup

In FGJ (and GJ), a subclass may extend an instantiated superclass. This means that, unlike in FJ (and Java), the types of the fields and the methods in the subclass may not be identical to the types in the superclass. In order to specify a type-preserving erasure from FGJ to FJ, it is necessary to define additional auxiliary functions that look up the type of a field or method in the *highest* superclass in which it is defined.

For example, we previously defined the generic class `Pair<X,Y>`. We may declare a specialized subclass `PairOfA` as a subclass of the instantiation `Pair<A,A>`, which instantiates both `X` and `Y` to a given class `A`.

```
class PairOfA extends Pair<A,A> {
  PairOfA(A fst, A snd) {
    super(fst, snd);
  }
  PairOfA setfst(A newfst) {
    return new PairOfA(newfst, this.fst);
  }
}
```

Note that, in the `setfst` method, the argument type `A` matches the argument type of `setfst` in `Pair<A,A>`, while the result type `PairOfA` is a subtype of the result type in `Pair<A,A>`; this is permitted by FGJ's covariant subtyping, as discussed in the previous section. Erasing the class `PairOfA` yields the following:

```
class PairOfA extends Pair {
  PairOfA(Object fst, Object snd) {
    super(fst, snd);
  }
  Pair setfst(Object newfst) {
    return new PairOfA(newfst, this.fst);
  }
}
```

Here arguments to the constructor and the method are given type `Object`, even though the erasure of `A` is itself; and the result of the method is given type `Pair`, even though the erasure of `PairOfA` is itself. In both cases, the types are chosen to correspond to types in `Pair`, the highest superclass in which the fields and method are defined.

We define variants of the auxiliary functions that find the types of fields and methods in the highest superclass in which they are defined. The maximum field types of a class `C`, written $\mathit{fieldsmax}(\mathtt{C})$, is the sequence of pairs of a type and a field name defined as follows:

$$\mathit{fieldsmax}(\mathtt{Object}) = \bullet$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{U}}\mathtt{>}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \ldots\ \} \\ \Delta = \overline{\mathtt{X}} \mathtt{<:} \overline{\mathtt{N}} \qquad \overline{\mathtt{C}}\ \overline{\mathtt{g}} = \mathit{fieldsmax}(\mathtt{D}) \end{array}}{\mathit{fieldsmax}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{g}},\ |\overline{\mathtt{T}}|_\Delta\ \overline{\mathtt{f}}}$$

The maximum method type of `m` in `C`, written $\mathit{mtypemax}(\mathtt{m}, \mathtt{C})$, is defined as follows:

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{U}}\mathtt{>}\ \{\ldots\} \\ \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{O}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T} = \mathit{mtype}(\mathtt{m}, \mathtt{D<}\overline{\mathtt{U}}\mathtt{>}) \end{array}}{\mathit{mtypemax}(\mathtt{m}, \mathtt{C}) = \mathit{mtypemax}(\mathtt{m}, \mathtt{D})}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{U}}\mathtt{>}\ \{\ldots\} \\ \mathit{mtype}(\mathtt{m}, \mathtt{D<}\overline{\mathtt{U}}\mathtt{>})\ \text{undefined} \\ \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{O}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T} = \mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{X}}\mathtt{>}) \qquad \Delta = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}},\ \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{O}} \end{array}}{\mathit{mtypemax}(\mathtt{m}, \mathtt{C}) = |\overline{\mathtt{T}}|_\Delta \rightarrow |\mathtt{T}|_\Delta}$$

We also need a way to look up the maximum type of a given field. If $\mathit{fieldsmax}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$ then we set $\mathit{fieldsmax}(\mathtt{C})(\mathtt{f}_i) = \mathtt{D}_i$.

4.3 Erasure of Expressions

The erasure of an expression depends on the typing of that expression, since the types are used to determine which downcasts to insert. The erasure rules are optimized to omit casts when it is trivially safe to do so; this happens when the maximum type is equal to the erased type.

Write $|\mathtt{e}|_{\Delta,\Gamma}$ for the erasure of a well-typed expression `e` with respect to environment Γ and type environment Δ:

$$|\mathtt{x}|_{\Delta,\Gamma} = \mathtt{x}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e}_0\mathtt{.f} \in \mathtt{T} \qquad \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{fieldsmax}(|\mathtt{T}_0|_\Delta)(\mathtt{f}) = |\mathtt{T}|_\Delta \end{array}}{|\mathtt{e}_0\mathtt{.f}|_{\Delta,\Gamma} = |\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.f}}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e}_0\mathtt{.f} \in \mathtt{T} \qquad \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{fieldsmax}(|\mathtt{T}_0|_\Delta)(\mathtt{f}) \neq |\mathtt{T}|_\Delta \end{array}}{|\mathtt{e}_0\mathtt{.f}|_{\Delta,\Gamma} = (|\mathtt{T}|_\Delta)\ |\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.f}}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T} \qquad \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{mtypemax}(\mathtt{m}, |\mathtt{T}_0|_\Delta) = \overline{\mathtt{C}}\rightarrow\mathtt{D} \qquad \mathtt{D} = |\mathtt{T}|_\Delta \end{array}}{|\mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma} = |\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.m(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T} \qquad \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{mtypemax}(\mathtt{m}, |\mathtt{T}_0|_\Delta) = \overline{\mathtt{C}}\rightarrow\mathtt{D} \qquad \mathtt{D} \neq |\mathtt{T}|_\Delta \end{array}}{|\mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma} = (|\mathtt{T}|_\Delta)\ |\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.m(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}}$$

$$|\mathtt{new}\ \mathtt{N(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma} = \mathtt{new}\ |\mathtt{N}|_\Delta\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}$$

$$|\mathtt{(N)}\mathtt{e}_0|_{\Delta,\Gamma} = (|\mathtt{N}|_\Delta)\ |\mathtt{e}_0|_{\Delta,\Gamma}$$

(Strictly speaking, one should think of the erasure operation as acting on typing derivations rather than expressions. Since well-typed expressions are in 1-1 correspondence with their typing derivations, the abuse of notation creates no confusion.)

4.4 Erasure of Methods and Classes

The erasure of a method m with respect to type environment Δ in class C, written $|\mathtt{M}|_{\Delta,\mathtt{C}}$, is defined as follows:

$$\frac{\begin{array}{c}\Gamma = \overline{\mathtt{x}}:\overline{\mathtt{T}} \qquad \Delta' = \Delta, \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{O}} \\ \mathtt{e}' = [(|\overline{\mathtt{T}}|_{\Delta'})\overline{\mathtt{x}}'/\overline{\mathtt{x}}]|\mathtt{e}|_{\Delta',\Gamma} \\ \mathit{mtypemax}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{D}}{\rightarrow}\mathtt{D}\end{array}}{|\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{O}}\mathtt{>}\ \mathtt{T}\ \mathtt{m}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}|_{\Delta,\mathtt{C}} = \mathtt{D}\ \mathtt{m}\ (\overline{\mathtt{D}}\ \overline{\mathtt{x}}')\ \{\uparrow\mathtt{e}';\}}$$

(In GJ, the actual erasure is somewhat more complex, involving the introduction of bridge methods, so that one ends up with two overloaded methods: one with the maximum type, and one with the instantiated type. We don't model that extra complexity here, because it depends on overloading of method names, which is not modeled in FJ.)

The erasure of constructors and classes is:

$$\begin{array}{l}|\mathtt{C}(\overline{\mathtt{U}}\ \overline{\mathtt{g}},\ \overline{\mathtt{T}}\ \overline{\mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}};\}|_{\Delta,\mathtt{C}} \\ = \mathtt{C}(\mathit{fieldsmax}(\mathtt{C}))\ \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}};\}\end{array}$$

$$\frac{\Delta = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}}{\begin{array}{c}|\mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}| \\ = \mathtt{class}\ \mathtt{C}\triangleleft|\mathtt{N}|_\Delta\{|\overline{\mathtt{T}}|_\Delta\ \overline{\mathtt{f}};\ |\mathtt{K}|_\Delta\ |\overline{\mathtt{M}}|_{\Delta,\mathtt{C}}\}\end{array}}$$

4.5 Properties of Erasure

Having defined erasure, we may investigate some of its properties. First, a well-typed FGJ program erases to a well-typed FJ program, as expected:

4.5.1 Theorem [Erasure preserves typing]: If an FGJ class table CT is ok and $\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}$, then $|\Gamma|_\Delta \vdash |\mathtt{e}|_{\Delta,\Gamma} \in |\mathtt{T}|_\Delta$ and $|CT|$ is ok using FJ rules.

Proof sketch: By induction on the derivation of $\Delta;\Gamma \vdash_{\mathrm{FGJ}} \mathtt{e} \in \mathtt{T}$, using the following lemmas: (1) if $\Delta \vdash \mathtt{N}$ ok then $|\mathit{fields}_{\mathrm{FGJ}}(\mathtt{N})|_\Delta$ <: $\mathit{fieldsmax}(|\mathtt{N}|_\Delta)$; (2) if $\Delta \vdash \mathtt{N}$ ok and $\mathit{methodtype}_{\mathrm{FGJ}}(\mathtt{m},\ \mathtt{N}) = \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{O}}\mathtt{>}\overline{\mathtt{U}}{\rightarrow}\mathtt{U}$, then $\mathit{mtypemax}(\mathtt{m}, |\mathtt{N}|_\Delta) = \overline{\mathtt{C}}{\rightarrow}\mathtt{D}$ and $|[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}|_\Delta$ <: $\overline{\mathtt{C}}$ and $|[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}|_\Delta$ <: D; and (3) if $\Delta \vdash \Gamma$ ok and $\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}$ for some well-formed type environment Δ, then $\Delta \vdash \mathtt{T}$ ok. ∎

More interestingly, we would intuitively expect that erasure from FGJ to FJ should also preserve the reduction behavior of FGJ programs:

$$\begin{tikzcd} \mathtt{e} \arrow[r, "\text{reduce (FGJ)}"] \arrow[d, "\text{erase}"'] & \mathtt{e}' \arrow[d, dotted, "\text{erase}"] \\ |\mathtt{e}| \arrow[r, dotted, "\text{reduce (FJ)}"'] & |\mathtt{e}'| \end{tikzcd}$$

Unfortunately, this is not quite true. For example, consider the FGJ expression

$\mathtt{e} = \mathtt{new\ Pair<A,B>(a,b).fst}$,

where a and b are expressions of type A and B, respectively, and its erasure:

$|\mathtt{e}|_{\Delta,\Gamma} = \mathtt{(A)new\ Pair(}|\mathtt{a}|_{\Delta,\Gamma}\mathtt{,}|\mathtt{b}|_{\Delta,\Gamma}\mathtt{).fst}$

In FGJ, e reduces to $|\mathtt{a}|_{\Delta,\Gamma}$, while the erasure $|\mathtt{e}|_{\Delta,\Gamma}$ reduces to $\mathtt{(A)}|\mathtt{a}|_{\Delta,\Gamma}$ in FJ; it does not reduce to $|\mathtt{a}|_{\Delta,\Gamma}$ when a is not a new expression. (Note that it is not an artifact of our nondeterministic reduction strategy: it happens even if we adopt a call-by-value reduction strategy, since, after method invocation, we may obtain an expression like (A)e where e is not a new expression.) Thus, the above diagram does not commute even if one-step reduction ($\longrightarrow$) at the bottom is replaced with many-step reduction ($\longrightarrow^*$). In general, synthetic casts can persist for a while in the FJ expression, although we expect those casts will eventually turn out to be upcasts when a reduces to a new expression.

In the example above, an FJ expression d reduced from $|\mathtt{e}|_{\Delta,\Gamma}$ had *more* synthetic casts than $|\mathtt{e}'|_{\Delta,\Gamma}$. However, this is not always the case: d may have *less* casts than $|\mathtt{e}'|_{\Delta,\Gamma}$ when the reduction step involves method invocation. Consider the following class and its erasure:

```
class C<X extends Object> extends Object {
  X f;
  C(X f) { this.f = f; }
  C<X> m() { return new C<X>(this.f); }
}

class C extends Object {
  Object f;
  C(Object f) { this.f = f; }
  C m() { return new C(this.f); }
}
```

Now consider the FGJ expression

$\mathtt{e} = \mathtt{new\ C<A>(new\ A()).m()}$

and its erasure

$|\mathtt{e}|_{\Delta,\Gamma} = \mathtt{new\ C(new\ A()).m()}$.

In FGJ,

$\mathtt{e} \longrightarrow_{\mathrm{FGJ}} \mathtt{new\ C<A>(new\ C<A>(new\ A()).f)}$.

In FJ, on the other hand, $|\mathtt{e}|_{\Delta,\Gamma}$ reduces to new C(new C(new A()).f), which has fewer synthetic casts than new C((A)new C(new A()).f), which is the erasure of the reduced expression in FGJ. The subtlety we observe here is that, when the erased term is reduced, synthetic casts may become "coarser" than the casts inserted when the reduced term is erased, or may be removed entirely as in this example. (Removal of downcasts can be considered as a combination of two operations: replacement of (A) with the coarser cast (Object) and removal of the upcast (Object), which does not affect the result of computation.)

To formalize both of these observations, we define an auxiliary relation that relates FJ expressions differing only by the addition and replacement of some synthetic casts. Let us call a well-typed expression d an *expansion* of a well-typed expression e, written $\mathtt{e} \Longrightarrow \mathtt{d}$, if d is obtained from e by some combination of (1) addition of zero or more synthetic upcasts, (2) replacement of some synthetic casts (D) with (C), where C is a supertype of D, or (3) removal of some synthetic casts.

4.5.2 Theorem: [Erasure preserves reduction modulo expansion] If $\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\mathtt{e} \longrightarrow_{\mathrm{FGJ}}{}^* \mathtt{e}'$, then there exists some FJ expression $\mathtt{d}'$ such that

$|\texttt{e}'|_{\Delta,\Gamma} \Longrightarrow \texttt{d}'$ and $|\texttt{e}|_{\Delta,\Gamma} \longrightarrow_{\text{FJ}} \texttt{d}'$. In other words, the following diagram commutes.

$$\begin{tikzcd}
\texttt{e} \arrow[rr, "\text{reduce (FGJ)}_*"] \arrow[dd, "\text{erase}"'] & & \texttt{e}' \arrow[d, dotted, "\text{erase}"] \\
& & |\texttt{e}'| \arrow[d, Rightarrow, dotted] \\
|\texttt{e}| \arrow[rr, dotted, "*", "\text{reduce (FJ)}"'] & & \texttt{d}'
\end{tikzcd}$$

As easy corollaries of this theorem, it can be shown that, if an FGJ expression `e` reduces to a "fully-evaluated expression," then the erasure of `e` reduces to exactly its erasure, and that if FGJ reduction gets stuck at a stupid cast, then FJ reduction also gets stuck because of the same typecast. We use the metavariable `v` for fully evaluated expressions, defined as follows:

$\texttt{v} ::= \texttt{new N(}\overline{\texttt{v}}\texttt{)}.$

4.5.3 Corollary: [Erasure preserves execution results] If $\Delta;\Gamma \vdash \texttt{e} \in \texttt{T}$ and $\texttt{e} \longrightarrow_{\text{FGJ}}{}^{*}\ \texttt{v}$, then $|\texttt{e}|_{\Delta,\Gamma} \longrightarrow_{\text{FJ}}{}^{*}\ |\texttt{v}|_{\Delta,\Gamma}$.

4.5.4 Corollary: [Erasure preserves typecast errors] If $\Delta;\Gamma \vdash \texttt{e} \in \texttt{T}$ and $\texttt{e} \longrightarrow_{\text{FGJ}}{}^{*}\ \texttt{e}'$, where $\texttt{e}'$ has a stuck subexpression $\texttt{(C<}\overline{\texttt{S}}\texttt{>)new D<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$, then $|\texttt{e}|_{\Delta,\Gamma} \longrightarrow_{\text{FJ}}{}^{*}\ \texttt{d}'$ such that $\texttt{d}'$ has a stuck subexpression $\texttt{(C)new D(}\overline{\texttt{d}}\texttt{)}$, where $\overline{\texttt{d}}$ are expansions of the erasures of $\overline{\texttt{e}}$, in the same position (modulo synthetic casts) as the erasure of $\texttt{e}'$.

5 Related Work

Core calculi for Java. There are several known proofs in the literature of type soundness for subsets of Java. In the earliest, Drossopoulou and Eisenbach [11] (using a technique later mechanically checked by Syme [21]) prove soundness for a fairly large subset of sequential Java. Like us, they use a small-step operational semantics, but they avoid the subtleties of "stupid casts" by omitting casting entirely. Nipkow and Oheimb [18] give a mechanically checked proof of soundness for a somewhat larger core language. Their language does include casts, but it is formulated using a "big-step" operational semantics, which sidesteps the stupid cast problem. Flatt, Krishnamurthi, and Felleisen [14, 15] use a small-step semantics and formalize a language with both assignment and casting. Their system is somewhat larger than ours (the syntax, typing, and operational semantics rules take perhaps three times the space), and the soundness proof, though correspondingly longer, is of similar complexity. Their published proof of subject reduction in the earlier version is slightly flawed — the case that motivated our introduction of stupid casts is not handled properly — but the problem can be repaired by applying the same refinement we have used here.

Of these three studies, that of Flatt, Krishnamurthi, and Felleisen is closest to ours in an important sense: the goal there, as here, is to choose a core calculus that is as *small* as possible, capturing just the features of Java that are relevant to some particular task. In their case, the task is analyzing an extension of Java with Common Lisp style mixins – in ours, extensions of the core type system. The goal of the other two systems, on the other hand, is to include as *large* a subset of Java as possible, since their primary interest is proving the soundness of Java itself.

Other class-based object calculi. The literature on foundations of object-oriented languages contains many papers formalizing class-based object-oriented languages, either taking classes as primitive (e.g., [22, 8, 6, 5]) or translating classes into lower-level mechanisms (e.g., [13, 4, 1, 20]. Some of these systems (e.g. [20, 8]) include generic classes and methods, but only in fairly simple forms.

Generic extensions of Java. A number of extensions of Java with generic classes and methods have been proposed by various groups, including the language of Agesen, Freund, and Mitchell [2]; PolyJ, by Myers, Bank, and Liskov [17]; Pizza, by Odersky and Wadler [19]; GJ, by Bracha, Oderksy, Stoutamire, and Wadler [7]; and NextGen, by Cartwright and Steele [10]. While all these languages are believed to be typesafe, our study of FGJ is the first to give rigorous proof of soundness for a generic extension of Java. We have used GJ as the basis for our generic extension, but similar techniques should apply to the forms of genericity found in the rest of these languages.

6 Discussion

We have presented Featherweight Java, a core language for Java modeled closely on the lambda-calculus and embodying many of the key features of Java's type system. FJ's definition and proof of soundness are both concise and straightforward, making it a suitable arena for the study of ambitious extensions to the type system, such as the generic types of GJ. We have developed this extension in detail, stated some of its fundamental properties, and sketched their proofs.

FJ itself is not quite complete enough to model some of the interesting subtleties found in GJ. In particular, the full GJ language allows some parameters to be instantiated by a special "bottom type" `*`, using a slightly delicate rule to avoid unsoundness in the presence of assignment. Capturing the relevant issues in FGJ requires extending it with assignment and `null` values (both of these extensions seem straightforward, but cost us some of the pleasing compactness of FJ as it stands). The other somewhat subtle aspect of GJ that is not accurately modeled in FGJ is the use of bridge methods in the compilation from GJ to JVM bytecodes. To treat this compilation exactly as GJ does, we would need to extend FJ with overloading.

Our formalization of GJ also does not include *raw types*, a unique aspect of the GJ design that supports compatibility between old, unparameterized code and new, parameterized code. We are currently experimenting with an extension of FGJ with raw types.

Formalizing generics has proven to be a useful application domain for FJ, but there are other areas where its extreme simplicity may yield significant leverage. For example, work is under way on formalizing Java 1.1's *inner classes* using FJ [16].

Acknowledgments

This work was supported by the University of Pennsylvania and the National Science Foundation under grant CCR-9701826, *Principled Foundations for Programming with Objects*. Igarashi is a research fellow of the Japan Society for the Promotion of Science.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, October 1997.

[3] H. P. Barendregt. *The Lambda Calculus*. North Holland, revised edition, 1984.

[4] Viviana Bono and Kathleen Fisher. An imperative first-order calculus with object extension. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.

[5] Viviana Bono, Amit J. Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, June 1999.

[6] Viviana Bono, Amit J. Patel, Vitaly Shmatikov, and John C. Mitchell. A core calculus of classes and objects. In *Fifteenth Confertence on the Mathematical Foundations of Programming Semantics*, April 1999.

[7] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.

[8] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. Preliminary version in POPL 1993, under the title "Safe type checking in a statically typed object-oriented programming language".

[9] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. Preliminary version in TACS '91 (Sendai, Japan, pp. 750–770).

[10] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, SIGPLAN Notices 33(10), pages 201–215, Vancouver, BC, Octover 1998. ACM.

[11] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java Type System Sound? *Theory and Practice of Object Systems*, 7(1):3–24, 1999. Preliminary version in ECOOP '97.

[12] Matthias Felleisen and Daniel P. Friedman. *A little Java, A few Patterns*. The MIT Press, Cambridge, Massachusetts, 1998.

[13] Kathleen Fisher and John C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–25, 1998.

[14] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages (POPL)*, San Diego, January 1998. ACM.

[15] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR97-293, Computer Science Department, Rice University, February 1998. Corrected version appeared in June, 1999.

[16] Atsushi Igarashi and Benjamin C. Pierce. On inner classes, July 1999. Submitted for publication.

[17] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 1997.

[18] Tobias Nipkow and David von Oheimb. Java_{light} is type-safe — definitely. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 161–170, San Diego, January 1998. ACM.

[19] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[20] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. Preliminary version in Principles of Programming Languages (POPL), 1993.

[21] Don Syme. Proving Java type soundness. Technical Report 427, Computer Laboratory, University of Cambridge, June 1997.

[22] Mitchell Wand. Type inference for objects with instance variables and inheritance. Technical Report NU-CCS-89-2, College of Computer Science, Northeastern University, February 1989. Also

in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).