On Inner Classes[1]

Atsushi Igarashi[2]
Department of Graphics and Computer Science
Graduate School of Arts and Sciences
University of Tokyo
3-8-1 Komaba, Meguro-ku, Tokyo 153-8902, Japan
E-mail: igarashi@graco.c.u-tokyo.ac.jp

and

Benjamin C. Pierce
Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street, Philadelphia, PA 19104, U.S.A.
E-mail: bcpierce@cis.upenn.edu

Inner classes in object-oriented languages play a role similar to nested function definitions in functional languages, allowing an object to export other objects that have direct access to its own methods and instance variables. However, the similarity is deceptive: a close look at inner classes reveals significant subtleties arising from their interactions with inheritance.

The goal of this work is a precise understanding of the essential features of inner classes; our object of study is a fragment of Java with inner classes and inheritance (and almost nothing else). We begin by giving a *direct* reduction semantics for this language. We then give an alternative semantics by *translation* into a yet smaller language with only top-level classes, closely following Java's Inner Classes Specification. We prove that the two semantics coincide, in the sense that translation commutes with reduction, and that both are type-safe.

Key Words: inner classes, Java, language design, language semantics

[1]Preliminary summaries appeared in *the informal proceedings of the 7th International Workshop on Foundations of Object-Oriented Languages (FOOL7)*, Boston, MA, January 2000 and in *the proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP2000)*, volume 1850 of Lecture Notes on Computer Science, pages 129–153, Cannes, France, June 2000, Springer-Verlag.

[2]This work was done while the author was visiting University of Pennsylvania as a research fellow of the Japan Society of the Promotion of Science.

1. INTRODUCTION

It has often been observed that the gap between object-oriented and functional programming styles is not as large as it might first appear; in essence, an object is just a record of function closures. However, there are differences as well as similarities. On the one hand, objects and classes incorporate important mechanisms not present in functions (static members, inheritance, object identity, access protection, etc.). On the other hand, functional languages usually allow *nested* definitions of functions, giving inner functions direct access to the local variables of their enclosing definitions.

In fact, the first object-oriented language, Simula [3], *did* support nested class declarations. Although most succeeding object-oriented languages left them out, a few object-oriented languages follow Simula and support various kinds of nesting. For example, Smalltalk [9] has special syntax for "block" objects, similar to anonymous functions. Beta [17] provides patterns, unifying classes and functions, that can be nested arbitrarily. More recently, *inner classes* have been popularized by their inclusion in Java 1.1 [10, 14].

Inner classes are useful when an object needs to send another object a chunk of code that can call the first object's methods and/or manipulate its instance variables. Such situations are typical in user-interface programming: for example, Java's Abstract Windowing Toolkit [13] allows a *listener object* to be registered with a user-interface component such as a button; when the button is pressed, the `actionPerformed` method of the listener is invoked. For example, suppose we want to increment a counter when a button is pressed. We begin by defining a class `Counter` with an inner class `Listener`:

```
class Counter {
  int x;
  class Listener implements ActionListener {
    public void actionPerformed(ActionEvent e) { x++; }
  }
  void listenTo(Button b) {
    b.addActionListener(new Listener());
  }
}
```

In the definition of the method `actionPerformed`, the field `x` of the enclosing `Counter` object is changed. The method `listenTo` creates a new listener object and sends it to the given `Button`. Now we can write

```
Counter c = new Counter();
Button b = new Button("Press me");
c.listenTo(b);
gui.add(b);
```

to create and display a button that increments a counter every time it is pressed.[3]

---

[3]Strictly speaking, the increment of `x` should be performed after the acquisition of the lock, by using `synchronized`, on the counter with which the listener is associated because there may be simultaneous accesses to the counter. (There may be more then one listener for one counter.)

Inner classes are a powerful abstraction mechanism, allowing programs like the one above to be expressed much more conveniently and transparently than would be possible using only top-level classes. However, this power comes at a significant cost in complexity: inner classes interact with other features of object-oriented programming—especially inheritance—in some quite subtle ways. For example, a closure in a functional language has a simple lexical environment, including all the bindings in whose scope it appears. An inner class, on the other hand, has access, via methods inherited from superclasses, to a *chain* of lexical environments—including not only the lexical environment in which it appears, but also the lexical environment of each superclass (which may in general be completely different). Conversely, the presence of inner classes complicates our intuitions about inheritance. What should it mean, for example, for an inner class to inherit from its enclosing class? What happens if a top-level class inherits from an inner class defined in a different top-level class?

The language issues arising from inner classes are not limited to their interactions with inheritance. Java's surface syntax includes a variety of abbreviated forms related to inner classes—for example, the class `Listener` above can be referred to by its simple name `Listener` inside the class `Counter`, while it should be called `Counter.Listener` (its fully qualified name), at the top-level. Such abbreviations can expanded to fully qualified forms, following scoping rules similar to those found in conventional block-structured languages. The rules are, however, somewhat more complicated in Java since the scope of a name can extend to subclasses of the class in which the name is declared. Also, the semantics of access annotations (`public`/`private`/etc.) becomes a bit more complicated (for example, what does a `public` member mean inside a `private` inner class?).

JavaSoft's Inner Classes Specification [14], later incorporated into the Java language definition [11], provides one answer to questions about inner classes by showing how to translate a program with inner classes into one using only top-level classes, adding to each inner class an extra field that points to an instance of the enclosing class. This specification gives clear basic intuitions about the behavior of inner classes, but it is not a completely satisfying account. For one thing, its style is indirect, forcing programmers to reason about their code by first passing it through a rather heavy transformation. Moreover, the document itself is somewhat imprecise, consisting only of examples and English prose. Different compilers (even different versions of Sun's JDK) have interpreted the specification differently in some significant ways (cf. Section 6).

The goal of this work is a precise understanding of the essential features of inner classes. Our contributions are threefold:

- First, we give a direct operational semantics and typing rules for a small language with inner classes and inheritance. The typing rules are shown to be sound for the operational semantics in the standard sense. To our knowledge, this direct formal semantics is the first that has been given for inner classes in any language.

We base our direct semantics on the new Java language specification [11]. To keep the model as simple as possible, we focus on the most basic form of inner classes in Java (classes that are members of other classes), omitting the related mechanisms of anonymous classes, local classes within blocks, and static nested classes. Also, we do not deal with the (important) interactions between access annotations and inner classes (cf. [14, 2, 1]).

• Next, we give a translation from our language with inner classes to an even smaller language with only top-level classes, formalizing the translation semantics of the Java Inner Classes Specification [14]. We show that the translation preserves typing.

• Finally, we prove that the two semantics coincide—that they define the same behavior for inner classes—in the sense that the translation commutes with the high-level reduction relation in the direct semantics. This property, together with the property of preservation of typing, guarantees correctness of the translation semantics with respect to the direct semantics, for the case where whole programs are being translated. (The case where some translated classes are linked with classes written directly in the target language is more subtle, and we do not handle it here. The main desired theorem in this case would be *full abstraction*, which states that translated expressions that can be distinguished by a target language context can also be distinguished in the source language. However, our present translation is *not* fully abstract, because our modeling language does not include private fields, which are used by the real translation to prevent observers from directly accessing the field of an inner class instance that holds a pointer to its containing object. The question of full abstraction for full-scale inner class translations has been considered by Abadi [1] and Pugh [2].)

Aside from the main contributions listed above, we also tackle the issue of formalizing Java's rather complex scoping rules for inner classes. To separate this problem from more basic semantic questions, we consider two different languages: an *external* language, in which abbreviated forms are allowed, and a simpler *internal* language, for which we define a formal semantics and type system. External language programs are translated to internal language programs by an *elaboration* process that records the results of the scoping rules.

The basis of our work is a core calculus called Featherweight Java, or FJ. This calculus was originally proposed in the context of a formal study [12] of GJ [4], an extension of Java with parameterized classes. FJ is designed to omit as many features of Java as possible (even assignment), while maintaining the essential flavor of the language and its type system. Its definition fits comfortably on a page, and its basic properties can be proved with no more difficulty than, say, those of the simply typed lambda-calculus with subtyping. This extreme simplicity makes it an ideal vehicle for the rigorous study of new language features such as parameterized classes and inner classes.

The omission of assignment in the present study is justified by the fact that we are only dealing here with inner classes as top-level members of other classes; the fundamental interaction of such inner classes with assignment is minimal. Other forms of inner classes, which allow classes to appear inside method bodies, interact with assignment in more interesting ways. For example, in Java, parameters and local variables referred to by an inner class within a method body are actually *copied* into instances of the inner class; these local variables must be marked `final` to ensure that this copying makes sense. To analyze the correctness of this scheme, we would need to extend our account with assignment. There is no fundamental

$$
\begin{aligned}
\texttt{L} &::= \texttt{class C extends D \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}}\\
\texttt{K} &::= \texttt{C(}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{)\{ super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}}\\
\texttt{M} &::= \texttt{C m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}}\\
\texttt{e} &::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C(}\overline{\texttt{e}}\texttt{)}
\end{aligned}
$$

FIG. 1. FJ: syntax

difficulty with doing so, but the consequent increase in notational complexity would make the formalization somewhat heavier.

The remainder of the article is organized as follows. Section 2 briefly reviews Featherweight Java. Section 3 opens with a detailed discussion of the main issues that must be dealt with to understand inner classes, then proceeds to a formal definition of the internal language FJI, an extension of FJ with inner classes, giving its syntax, typing rules, and reduction rules and developing standard type soundness results. Section 4 defines a compilation from FJI to FJ, modeling the translation semantics of the Inner Classes Specification, and proves its correctness with respect to the direct semantics in the previous section. Section 5 discusses the elaboration process from the external language to FJI. Section 6 examines some changes made from the original Inner Class Specification [14] and some behavioral differences between compilers resulting from inconsistencies in the old specification. Section 7 discusses related work, and Section 8 offers concluding remarks.

2. FEATHERWEIGHT JAVA

We begin by reviewing the basic definitions of Featherweight Java [12]. FJ is a tiny fragment of Java, including only top-level class definitions, object instantiation, field access, and method invocation. (The original version of FJ also included typecasts, which are required to model the compilation of GJ into Java. They are omitted from this article, since they do not interact with inner classes in any significant way.) Our main goal in designing FJ was to make a proof of type soundness ("well-typed programs do not get stuck") as concise as possible, while still capturing the essence of the soundness argument for the sequential part of the full Java language.

A key simplification in FJ (and also FJI) is the omission of assignment, which makes FJ a purely functional language. In essence, all fields and method parameters in FJ are implicitly marked `final`. Although, most *useful* examples of programming in Java do involve its side-effecting features, we exclude them to focus on the issues on interaction between inner classes and inheritance. In particular, member classes, the only form of inner classes we deal with here, are expected to interact with assignments less significantly than local/anonymous classes.

2.1. Syntax

The abstract syntax of FJ class declarations, constructor declarations, method declarations, and expressions is given in Figure 1. The metavariables `A`, `B`, `C`, `D`, and `E` range over class names; `f` and `g` range over field names; `m` ranges over

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

FIG. 2. FJ: subtyping Rules

method names; x ranges over parameter names; c, d and e range over expressions; L ranges over class declarations; K ranges over constructor declarations; and M ranges over method declarations. We write $\overline{\texttt{f}}$ as shorthand for a possibly empty sequence $\texttt{f}_1,\ldots,\texttt{f}_n$ (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 "$\texttt{C}_1\ \texttt{f}_1,\ldots,\texttt{C}_n\ \texttt{f}_n$" (where n is the length of both $\overline{\texttt{C}}$ and $\overline{\texttt{f}}$) and "$\overline{\texttt{C}}\ \overline{\texttt{f}};$" as shorthand for "$\texttt{C}_1\ \texttt{f}_1;\ldots\texttt{C}_n\ \texttt{f}_n;$" and "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}};$" as shorthand for "$\texttt{this.f}_1\texttt{=f}_1;\ldots\texttt{this.f}_n\texttt{=f}_n;$". Sequences of field declarations, parameter names, and method declarations are assumed to contain no duplicate names.

A class declaration has declarations of its name (class C), fields ($\overline{\texttt{C}}\ \overline{\texttt{f}}$), one constructor (K), and methods ($\overline{\texttt{M}}$); moreover, every class must explicitly declare its superclass with extends even if it is Object. Each argument of a constructor corresponds to an initial (and also final) value of each fields of the class. As in Java, fields inherited from superclasses are initialized by $\texttt{super(}\overline{\texttt{f}}\texttt{);}$ and newly declared fields by $\texttt{this.}\overline{\texttt{f}}\ \texttt{=}\ \overline{\texttt{f}}\texttt{;}$, although, as we will see, those statements do not play any role during execution of programs: they are included just so that we can say that FJ is literally a subset of full Java. A body of a method just returns an expression, which is either a variable, field access, method invocation, or object instantiation. We treat this in method bodies as an ordinary variable, and so require no special syntax for it. As we will see later, the typing rules prohibit this from appearing as a method parameter name.

A class table CT is a mapping from class names C to class declarations L. A *program* is a pair $(CT, \texttt{e})$ of a class table and an expression. Object is treated specially in every FJ program: the definition of Object class never appears in the class table and 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. (As we will see later, method lookup functions takes a pair of class and method names as arguments; the case for Object is just undefined.) To lighten the notation in what follows, we always assume a *fixed* class table CT.

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 – the reflexive and transitive closure of the immediate subclass relation given by the extends clauses in CT. Formally, it is defined in Figure 2.

The given class table is assumed to satisfy some sanity conditions: (1) $CT(\texttt{C}) =$ 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 <:

Field lookup:

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

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

Method type lookup:

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

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

Method body lookup:

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

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

FIG. 3. FJ: auxiliary definitions

relation is antisymmetric. Given these conditions, we can identify a class table with a sequence of class declarations in an obvious way.

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 3. The fields of a class C, written *fields*(C), is a sequence $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$ 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 $mtype(\mathtt{m}, \mathtt{C})$, is a pair, written $\overline{\mathtt{B}} \to \mathtt{B}$, of a sequence of argument types $\overline{\mathtt{B}}$ and a result type B. Similarly, the body of the method m in class C, written $mbody(\mathtt{m}, \mathtt{C})$, is a pair, written $(\overline{\mathtt{x}}, \mathtt{e})$, of a sequence of parameters $\overline{\mathtt{x}}$ and an expression e. (In Java proper, method body lookup is based not only on the method name but also on the static types of the actual arguments to deal with overloading, which we drop from FJ.) Note that the functions $mtype(\mathtt{m}, \mathtt{C})$ and $mbody(\mathtt{m}, \mathtt{C})$ are both partial functions: since Object is assumed to have no methods in FJ, both $mtype(\mathtt{m}, \mathtt{Object})$ and $mbody(\mathtt{m}, \mathtt{Object})$ are undefined.

2.2. 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 $\longrightarrow$, and $\longrightarrow^+$ for the transitive closure of $\longrightarrow$.

Computation:

$$\frac{\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{(\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}})).\mathtt{f}_i \longrightarrow \mathtt{e}_i} \qquad \text{(R-Field)}$$

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

Congruence:

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0.\mathtt{f} \longrightarrow \mathtt{e}_0{}'.\mathtt{f}} \qquad \text{(RC-Field)} \qquad \frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0{}'.\mathtt{m}(\overline{\mathtt{e}})} \qquad \text{(RC-InvRecv)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i{}',\ldots)} \qquad \text{(RC-InvArg)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{new}\ \mathtt{C}(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{new}\ \mathtt{C}(\ldots,\mathtt{e}_i{}',\ldots)} \qquad \text{(RC-NewArg)}$$

FIG. 4. FJ: reduction rules

The reduction rules are given in Figure 4. There are two reduction rules, one for field access and one for method invocation. A field access $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{f}_i$ looks up and obtains the field names $\overline{\mathtt{f}}$ of `C` with *fields*(`C`); then it reduces to the constructor argument $\mathtt{e}_i$ of the corresponding position. Method invocation $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})$ first looks up *mbody*(`m`, `C`) and obtains a pair of a sequence of formal arguments $\overline{\mathtt{x}}$ and the method body; then, it reduces to the method body in which $\overline{\mathtt{x}}$ are replaced with the actual arguments $\overline{\mathtt{d}}$ and `this` with the receiver $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ to stand for the result of replacing $\mathtt{x}_1$ by $\mathtt{d}_1$, ..., $\mathtt{x}_n$ by $\mathtt{d}_n$, and `y` by `e` in the 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 $\mathtt{e}.\mathtt{f} \longrightarrow \mathtt{e}'.\mathtt{f}$, and the like), which also appear in the figure.

For example, given the class definitions

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

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

Expression typing:

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

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

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} {\rightarrow} \mathtt{C} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} \mathrel{\mathtt{<:}} \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{C}} \qquad \text{(T-INVK)}$$

$$\frac{\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} \mathrel{\mathtt{<:}} \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}}) \in \mathtt{C}_0} \qquad \text{(T-NEW)}$$

Method typing:

$$\frac{\begin{array}{c} \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 \in \mathtt{E}_0 \qquad \mathtt{E}_0 \mathrel{\mathtt{<:}} \mathtt{C}_0 \\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{...\}} \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{D}} {\rightarrow} \mathtt{D}_0, \text{then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0 \end{array}}{\mathtt{C}_0\ \mathtt{m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\mathtt{return}\ \mathtt{e}_0;\}\ \mathtt{OK\ IN\ C}} \qquad \text{(T-METHOD)}$$

Class typing:

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

FIG. 5. FJ: typing rules

the expression `new Pair(new A(), new B()).setfst(new B())` reduces to `new Pair(new B(), new B())` as follows

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

where the underlined subexpressions are the ones being reduced at each step.

2.3. Typing

The typing rules for expressions, method declarations, and class declarations are in Figure 5. An environment Γ is a finite mapping from variables to types, written $\overline{\mathtt{x}}{:}\overline{\mathtt{C}}$.

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

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`. (Thus, a method with a parameter of name `this` is not allowed as the type environment is ill formed.) In case of overriding, if a method with the same name is declared in a superclass then the two methods must have the same argument and result types.

The typing judgment for class declarations has the form `L OK`, read "class declaration `L` 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.

2.4. Properties

FJ is type sound, shown by the following subject reduction and progress properties [12].

THEOREM 2.1 (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* $\mathtt{C}'$ <: `C`.

THEOREM 2.2 (Progress). *Suppose* `e` *is a well-typed expression.*

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

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

THEOREM 2.3 (FJ type soundness). *If* $\emptyset \vdash \mathtt{e} \in \mathtt{C}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$ *with* $\mathtt{e}'$ *being a normal form, then* $\mathtt{e}'$ *is a value* `v`, *given by the syntax* `v` ::= `new` $\mathtt{C}(\overline{\mathtt{v}})$, *and* $\emptyset \vdash \mathtt{v} \in \mathtt{D}$ *for some* `D` *such that* `D` <: `C`.

3. FJ WITH INNER CLASSES

We now define the language FJI by extending FJ with inner classes. Like FJ, FJI imposes some syntactic restrictions: (1) receivers of field access, method invocation, or inner class constructor invocation must be explicitly specified (no implicit `this`); (2) type names are always absolute paths to the classes they denote (no short abbreviations); and (3) an inner class instantiation expression $\mathtt{e}_0$`.new` $\mathtt{C}(\overline{\mathtt{e}})$ is annotated with the static type `T` of $\mathtt{e}_0$, written $\mathtt{e}_0$`.new` $\mathtt{T.C}(\overline{\mathtt{e}})$. Because of the conditions (2) and (3), FJI is not quite a subset of Java (whereas FJ is). Rather, FJI should be viewed as an intermediate language that we use to define a formal semantics and type system separately from dealing with the syntactic complications found in Java proper. As we mentioned in Section 1, to bridge the syntactic gap between FJI and Java, we will later define (in Section 5) a more Java-like external language and an elaboration function mapping from the external language to FJI. For the moment, we focus our attention on the intermediate language, describing the external language and the elaboration process only informally.

We begin with a brief discussion of the key idea of *enclosing instances*.

3.1. Enclosing Instances

Consider the following FJI class declaration:

```
class Outer extends Object {
  Pair p;
  Outer(Pair p) {super(); this.p = p;}
  class Inner extends Object {
    Inner() {super();}
    Object snd_p { return Outer.this.p.snd; }
  }
  Outer.Inner make_inner () { return this.new Outer.Inner(); }
}
```

Conceptually, each instance `o` of the class `Outer` contains a specialized version of the `Inner` class, which, when instantiated, yields instances of `Outer.Inner` that refer to `o`'s instance variable `p`. The object `o` is called the *enclosing instance* of these `Outer.Inner` objects.

This enclosing instance can be named explicitly by a "qualified `this`" expression (found in both Java and FJI), consisting of the simple name of the enclosing class followed by "`.this`". In general, the class $\mathtt{C}_1.\cdots.\mathtt{C}_n$ can refer to $n-1$ enclosing instances, $\mathtt{C}_1\mathtt{.this}$ to $\mathtt{C}_{n-1}\mathtt{.this}$, as well as the usual `this`, which can also be written $\mathtt{C}_n\mathtt{.this}$. To avoid ambiguity of the meaning of `C.this`, the name of an inner class must be different from any of its superclasses.

In FJI, an object of an inner class is instantiated by an expression of the form $\mathtt{e}_0\mathtt{.new\ T.C(\overline{e})}$, where $\mathtt{e}_0$ is the enclosing instance and `T` is the static type of $\mathtt{e}_0$. The result of $\mathtt{e}_0\mathtt{.new\ T.C(\overline{e})}$ is always an instance of `T.C`, regardless of the run-time type of $\mathtt{e}_0$. (Java allows only the notations $\mathtt{e}_0\mathtt{.new\ C(\overline{e})}$ (omitting the type of $\mathtt{e}_0$) and $\mathtt{new\ T.C(\overline{e})}$ (without a prefix); the latter roughly means an instantiation from the class `T.C` with an enclosing instance of the class `T`; see Section 5 for more details.) This rigidity reflects the static nature of Java's translation semantics for inner classes. The explicit annotation `T` is used in FJI to "remember" the static type of $\mathtt{e}_0$. (By contrast, inner classes in Beta can be *virtual* [16], i.e., different constructors may be invoked depending on the run-time type of the enclosing instance; for example, if `Inner` was declared virtual and there were a subclass $\mathtt{Outer}'$ of the class `Outer` that also had an inner class `Inner`, then `o.new Inner()` might build an instance of either `Outer.Inner` or $\mathtt{Outer}'\mathtt{.Inner}$, depending on the dynamic type of `o`.)

The elaboration process allows type names to be abbreviated in user programs. For example, the FJI program above can be written

```
class Outer extends Object {
  Pair p;
  Outer(Pair p) {super(); this.p = p;}
  class Inner extends Object {
    Inner() {super();}
    Object snd_p () { return p.snd; }
  }
  Inner make_inner () { return new Inner(); }
}
```

```
class A1 ... {
  T1 a1; ...
  class A2 ... { ...
    class A3 ... { ...
}}}
class B1 ... { ...
  class B2 ... { ...
    T2 b2; ...
    class B3 extends A1.A2.A3 {
    ...
}}}
class C1 ... { ...
  class C2 ... { ...
    class C3 extends B1.B2.B3 {
    ...
}}}
```

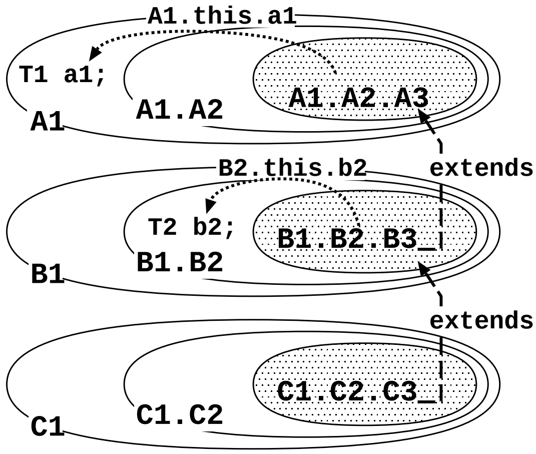

FIG. 6. A chain of environments

in the external language described in Section 5 (and in Java, which includes the external language). Here, the return type `Inner` of the `make_inner` method denotes the nearest `Inner` declaration. Also, in Java, enclosing instances can be omitted when they are `this` or a qualified `this`. Thus, `this.new Outer.Inner()` from the original example is written `new Inner()` here.

3.2. Subclassing and Inner Classes

Almost any form of inheritance involving inner classes is allowed in Java (and FJI): a top-level class can extend an inner class of another top-level class, or an inner class can extend another inner class from a completely different top-level class. An inner class can even extend its own enclosing class. (Only one case is disallowed: a class cannot extend its own inner class. We discuss the restriction later.) This liberality, however, introduces significant complexity because a method inherited from a superclass must be executed in a "lexical environment" different from the subclass's. Figure 6 shows a situation where three inner classes, `A1.A2.A3` and `B1.B2.B3` and `C1.C2.C3`, are in a subclass hierarchy. Each white oval represents an enclosing instance and the three shaded ovals indicate the regions of the program where the methods of a `C1.C2.C3` object may have been defined. A method inherited from `A1.A2.A3` is executed under the environment consisting of enclosing instances `A1.this` and `A2.this` and may access members of enclosing classes via `A1.this` and `A2.this`; similarly for `B1.B2.B3` and `C1.C2.C3`. In general, when a class has n superclasses which are inner, n different environments may be accessed by its methods. Moreover, each environment may consist of more than one enclosing instance; six enclosing instances are required for all the methods of `C1.C2.C3` to work in the example above.

From the foregoing, we see that we will have to provide, in some way, six enclosing instances to instantiate a `C1.C2.C3` object. Recall that, when an object of an inner class is instantiated, the enclosing object is provided by a prefix $\mathtt{e}_0$

of the `new` expression. For example, a `C1.C2.C3` object is instantiated by writing $\mathtt{e}_0$`.new C1.C2.C3(`$\overline{\mathtt{e}}$`)`, where $\mathtt{e}_0$ is the enclosing instance corresponding to `C2.this`. Where do the other enclosing instances come from?

First, enclosing instances from enclosing classes other than the immediately enclosing class, such as `C1.this`, do not have to be supplied to a `new` expression explicitly, because they can be reached via the direct enclosing instance—for example, the enclosing instance $\mathtt{e}_0$ in $\mathtt{e}_0$`.new C1.C2.C3(`$\overline{\mathtt{e}}$`)` has the form `new C1(`$\overline{\mathtt{c}}$`).new C1.C2(`$\overline{\mathtt{d}}$`)`, which includes the enclosing instance `new C1(`$\overline{\mathtt{c}}$`)` that corresponds to `C1.this`.

Second, the enclosing instances of superclasses are determined by the constructor of a subclass. Taking a simple example, suppose we extend the inner class `Outer.Inner`. An enclosing instance corresponding to `Outer.this` is required to make an instance of the subclass. Here is an example of a subclass of `Outer.Inner`, written in FJI:

```
class RefinedInner extends Outer.Inner {
  Object c;
  RefinedInner(Outer this$Outer$Inner, Object c) {
    this$Outer$Inner.super(); this.c=c;
}}
```

In the declaration of the constructor, the ordinary argument `this$Outer$Inner` becomes the enclosing instance prefix for the `super` constructor invocation, providing the value of `Outer.this` referred to in the inherited method `snd_p`. Similarly, in the `C1.C2.C3` example, the subclass `B1.B2.B3` is written as follows (we assume `A1.A2.A3` has a field `a3` of type `Object`):

```
class B1 extends ... { ...
  class B2 extends ... { ...
    class B3 extends A1.A2.A3 {
      Object b3;
      B3(Object a3, A1.A2 this$A1$A2$A3, Object b3) {
         this$A1$A2$A3.super(a3); this.b3 = b3; }
}}}
```

Note that, since an enclosing instance corresponding to `A1.this` is included in an enclosing instance corresponding to `A2.this`, the `B3` constructor takes only one extra argument for enclosing instances. Here is `C1.C2.C3` class:

```
class C1 extends ... { ...
  class C2 extends ... { ...
    class C3 extends B1.B2.B3 {
      Object c3;
      C3(Object a3, A1.A2 this$A1$A2$A3,
         Object b3, B1.B2 this$B1$B2$B3, Object c3) {
         this$B1$B2$B3.super(a3, this$A1$A2$A3, b3); this.c3 = c3; }
}}}
```

Since the constructor of a superclass `B1.B2.B3` initializes `A2.this`, the constructor `C3` initializes only `B2.this` by qualifying the `super` invocation; the argument `this$A1$A2$A3` is just passed to `super` as an ordinary argument.

In FJI, we restrict the qualification of `super` to be a constructor argument, whereas, in Java, which also supports qualified `super`, the qualification can be

$$
\begin{array}{lll}
\texttt{T} & ::= & \texttt{C}_1\texttt{.}\cdots\texttt{.C}_n \\[4pt]
\texttt{L} & ::= & \texttt{class C extends T \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \\[4pt]
\texttt{K} & ::= & \texttt{C(}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)\{ super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}} \\
 & \mid & \texttt{C(}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)\{ f.super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}} \\[4pt]
\texttt{M} & ::= & \texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \\[4pt]
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C(}\overline{\texttt{e}}\texttt{)} \mid \texttt{e.new T.C(}\overline{\texttt{e}}\texttt{)}
\end{array}
$$

FIG. 7. FJI: syntax

any expression (or even be omitted). This permits the same clean definition of operational semantics we saw in FJ, since all the state information (including fields and enclosing instances) of an object appears in its `new` expression. Moreover, for technical reasons connected with the name mangling involved in the translation semantics, we require that a constructor argument used for qualification of `super` be named $\texttt{this\$C}_1\texttt{\$}\cdots\texttt{\$C}_n$, where $\texttt{C}_1\texttt{.}\cdots\texttt{.C}_n$ is the (direct) superclass, as in the example above. (Note that the enclosing instance of type $\texttt{C}_1\texttt{.}\cdots\texttt{.C}_{n-1}$ is given the mangled variable name $\texttt{this\$C}_1\texttt{\$}\cdots\texttt{\$C}_n$. This naming convention avoids more than one occurrence of the same variable in constructor arguments: (possibly) more intuitive convention that gives $\texttt{this\$C}_1\texttt{\$}\cdots\texttt{\$C}_{n-1}$ may not work because one class may have, as its superclasses, two inner classes defined in the same class.)

Lastly, we can now explain why it is not allowed for a class to extend one of its (direct or indirect) inner classes. It is because there is no sensible way to make an instance of such a class. Suppose we could define the class below:

```
class Foo extends Foo.Bar {
  Foo (Foo f) { f.super(); }
  class Bar { ... }
}
```

Since `Foo` extends `Foo.Bar`, the constructor `Foo` will need an instance of `Foo` itself as an argument, making it impossible to make an instance of `Foo`. (Perhaps, in Java, one could use `null` as the enclosing instance in this case, but this would not be useful, since inner classes are usually supposed to make use of enclosing instances.)

3.3. Syntax

Now, we proceed to the formal definitions of FJI. The abstract syntax of the language is shown in Figure 7. We use the same notational conventions as in the previous section. Besides, the metavariables `S`, `T`, `U`, and `V` ranges over types, which are qualified class names (a sequence of simple names $\texttt{C}_1,\ldots,\texttt{C}_n$ concatenated by periods). For compactness in the definitions, we introduce the notation $\star$ for a "null qualification" and identify $\star.\texttt{C}$ with `C`. The metavariable `P` ranges over types (`T`) and $\star$. We write $\texttt{C} \in \texttt{P}$ if $\texttt{P} = \texttt{C}_1\texttt{.}\cdots\texttt{.C}_n$ and $\texttt{C} = \texttt{C}_i$ for some i.

A class declaration `L` includes declarations of its simple name `C`, superclass `T`, fields $\overline{\texttt{T}}\ \overline{\texttt{f}}$, constructor `K`, inner classes $\overline{\texttt{L}}$, and methods $\overline{\texttt{M}}$. There are two kinds of constructor declaration, depending on whether the superclass is inner or top-

$$\texttt{T <: T} \qquad \frac{\texttt{S <: T} \quad \texttt{T <: U}}{\texttt{S <: U}} \qquad \frac{CT(\texttt{S}) = \texttt{class C extends T \{...\}}}{\texttt{S <: T}}$$

FIG. 8. FJI: subtyping rules

level: when the superclass is inner, the subclass constructor must call the `super` constructor with a qualification "`f.`" to provide the enclosing instance visible from the superclass's methods. As we will see in typing rules, constructor arguments should be arranged in the following order: (1) the superclass's fields, initialized by `super(`$\overline{\texttt{f}}$`)` (or `f.super(`$\overline{\texttt{f}}$`)`); (2) the enclosing instance for the direct superclass (if needed); and (3) the fields of the class to be defined, initialized by `this.`$\overline{\texttt{f}}$`=`$\overline{\texttt{f}}$. Like FJ, the body of a method just returns an expression, which is a variable, field access, method invocation, or object instantiation. We assume that the set of variables includes the special variables `this` and `C.this` for every `C`; the typing rules guarantee that these variables are never used as the names of arguments to methods.

A *program* is a pair of a class table CT (a mapping from types `T` to class declarations `L`) and an expression `e`. The type `Object` is treated exactly in the same way as in FJ. From the class table, we can read off the subtype relation between classes. We write `S <: T` when `S` is a subtype of `T`—the reflexive and transitive closure of the immediate subclass relation given by the `extends` clauses in CT. This relation is defined formally in Figure 8.

We impose the following sanity conditions on the class table: (1) $CT(\texttt{P.C}) =$ `class C ...` for every $\texttt{P.C} \in dom(CT)$; (2) if $CT(\texttt{P.C})$ has an inner class declaration `L` of name `D`, then $CT(\texttt{P.C.D}) = \texttt{L}$; (3) $\texttt{Object} \notin dom(CT)$; (4) for every type `T` (except `Object`) appearing anywhere in CT, we have $\texttt{T} \in dom(CT)$; (5) for every $\texttt{e}_0$`.new T.C(`$\overline{\texttt{e}}$`)` (and `new C(`$\overline{\texttt{e}}$`)`, resp.) appearing anywhere in CT, we have $\texttt{T.C} \in dom(CT)$ (and $\texttt{C} \in dom(CT)$, resp.); (6) there are no cycles in the subtyping relation and (7) $\texttt{T} \not<: \texttt{T.U}$, for any two types `T` and `T.U`. By the conditions (1) and (2), a class table of FJI can be identified with a set of top-level classes. The condition (7) prohibits a class from extending one of its inner classes.

3.4. Auxiliary Functions

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 9. The fields of a type $\texttt{T}_0$, written $\mathit{fields}(\texttt{T}_0)$, is a sequence $\overline{\texttt{T}}\ \overline{\texttt{f}}$ pairing the type of each field with its name, for all the fields declared in class $\texttt{T}_0$ and all of its superclasses. In addition, $\mathit{fields}(\texttt{T}_0)$ collects the types of (direct) enclosing instances of all the superclasses of $\texttt{T}_0$. For example, $\mathit{fields}(\texttt{C1.C2.C3})$ returns the following sequence:

$\mathit{fields}(\texttt{C1.C2.C3}) =$

`Object`	`a3,`	(the field from `A1.A2.A3`)
`A1.A2`	`this$A1$A2$A3,`	(the enclosing instance bound to `A2.this`)
`Object`	`b3,`	(the field from `B1.B2.B3`)
`B1.B2`	`this$B1$B2$B3,`	(the enclosing instance bound to `B2.this`)
`Object`	`c3`	(the field from `C1.C2.C3`)

The third rule in the definition inserts enclosing instance information between the fields $\overline{\mathtt{S}}$ $\overline{\mathtt{g}}$ of the superclass `U.D` and the fields $\overline{\mathtt{T}}$ $\overline{\mathtt{f}}$ of the current class. In a well-typed program, *fields*($\mathtt{T}_0$) will always agree with the constructor argument list of $\mathtt{T}_0$.

The type of the method `m` in class `T`, written *mtype*(`m`, `T`), is a pair, written $\overline{\mathtt{S}}\to\mathtt{S}_0$, of a sequence of argument types $\overline{\mathtt{S}}$ and a result type $\mathtt{S}_0$. Similarly, the body of the method `m` in class `T`, written *mbody*(`m`, `T`), is a triple, written ($\overline{\mathtt{x}}$, `e`, `S`), of a sequence of parameters $\overline{\mathtt{x}}$, an expression `e`, and a class `S`; the class `S` denotes where the method is defined.

The function $\mathit{encl}_{\mathtt{T}}(\mathtt{e})$ plays a crucial role in the semantics of FJI. Intuitively, when `e` is a top-level or inner class instantiation, $\mathit{encl}_{\mathtt{T}}(\mathtt{e})$ returns the direct enclosing instance of `e` that is visible from class `T` (i.e., the enclosing instance that provides the correct lexical environment for methods inherited from `T`); thus, $\mathit{encl}_{\mathtt{T.C}}(\mathtt{e})$ is an expression of type `T` (or its subtype). The first rule is the simplest case: since the type of an expression $\mathtt{e}_0\mathtt{.new\ T.C(}\overline{\mathtt{e}}\mathtt{)}$ agrees with the subscript `T.C`, it just returns the (direct) enclosing instance $\mathtt{e}_0$. The other rules follow a common pattern; we explain the fifth rule as a representative. Since the subscripted type `T` is different from the type `S.C` of the argument $\mathtt{e}_0\mathtt{.new\ S.C(}\overline{\mathtt{d}}\mathtt{,\ d}_0\mathtt{,\ }\overline{\mathtt{e}}\mathtt{)}$, the enclosing instance $\mathtt{e}_0$ is not the correct answer. We therefore make a recursive call with an object $\mathtt{d}_0\mathtt{.new\ U.D(}\overline{\mathtt{d}}\mathtt{)}$ of the superclass obtained by dropping $\mathtt{e}_0$ and as many arguments $\overline{\mathtt{e}}$ as the fields $\overline{\mathtt{f}}$ of the class `S.C`. We keep going like this until, finally, the argument becomes an instance of `T` and we match the first rule. For example:

$$
\begin{aligned}
&\mathit{encl}_{\mathtt{A1.A2.A3}}(\mathtt{e}_0\mathtt{.new\ C1.C2.C3(a,\ e}_1\mathtt{,\ b,\ e}_2\mathtt{,\ c)})\\
&\quad= \mathit{encl}_{\mathtt{A1.A2.A3}}(\mathtt{e}_2\mathtt{.new\ B1.B2.B3(a,\ e}_1\mathtt{\ ,b)})\\
&\quad= \mathit{encl}_{\mathtt{A1.A2.A3}}(\mathtt{e}_1\mathtt{.new\ A1.A2.A3(a)})\\
&\quad= \mathtt{new\ A1().new\ A1.A2()}\\
&\qquad \text{(where } \mathtt{e}_1 = \mathtt{new\ A1().new\ A1.A2()}\\
&\qquad \text{and } \mathtt{e}_2 = \mathtt{new\ B1().new\ B1.B2()}\text{.)}
\end{aligned}
$$

Note that the *encl* function outputs only the direct enclosing instance. To obtain outer enclosing instances, such as `A1.this`, *encl* can be used repeatedly: $\mathit{encl}_{\mathtt{A1.A2}}(\mathit{encl}_{\mathtt{A1.A2.A3}}(\mathtt{e}))$.

3.5. Computation

As in FJ, the reduction relation of FJI has the form $\mathtt{e} \longrightarrow \mathtt{e}'$. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$ and $\longrightarrow^+$ for the transitive closure of $\longrightarrow$. The reduction rules are given in Figure 10. There are four reduction rules, two for field access and two for method invocation. The field access expression $\mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).f}_i$ looks up the field names $\overline{\mathtt{f}}$ of `C` using *fields*(`C`) and yields the constructor argument $\mathtt{e}_i$ in the position corresponding to $\mathtt{f}_i$ in the field list; $\mathtt{e}_0\mathtt{.new\ T.C(}\overline{\mathtt{e}}\mathtt{).f}_i$ behaves similarly. The method invocation expression $\mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).m(}\overline{\mathtt{d}}\mathtt{)}$ first calls *mbody*(`m`,`C`) to obtain a triple of the sequence of formal arguments $\overline{\mathtt{x}}$, the method body `e`, and the class $\mathtt{C}_1.\cdots.\mathtt{C}_n$ where `m` is defined; it yields a substitution instance of the method body in which the $\overline{\mathtt{x}}$ are replaced with the actual arguments $\overline{\mathtt{d}}$, the special variables `this` and $\mathtt{C}_n\mathtt{.this}$ with the receiver object $\mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)}$, and each $\mathtt{C}_i\mathtt{.this}$ (for $i < n$) with the corresponding enclosing instance $\mathtt{c}_i$, obtained from *encl*. Since the method to be invoked is defined in $\mathtt{C}_1.\cdots.\mathtt{C}_n$, the direct enclosing instance

Field lookup:

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

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

$$\frac{\begin{array}{c}CT(\texttt{T}) = \texttt{class C extends U.D \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K}\ \overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \qquad \mathit{fields}(\texttt{U.D}) = \overline{\texttt{S}}\ \overline{\texttt{g}} \\ \texttt{U} = \texttt{C}_1\texttt{.}\cdots\texttt{.C}_n \qquad \texttt{f}_0 = \texttt{this\$C}_1\texttt{\$}\cdots\texttt{\$C}_n\texttt{\$D}\end{array}}{\mathit{fields}(\texttt{T}) = \overline{\texttt{S}}\ \overline{\texttt{g}}, \texttt{U f}_0, \overline{\texttt{T}}\ \overline{\texttt{f}}}$$

Method type lookup:

$$\frac{\begin{array}{c}CT(\texttt{T}) = \texttt{class C extends S \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K}\ \overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{U}_0\ \texttt{m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \in \overline{\texttt{M}}\end{array}}{\mathit{mtype}(\texttt{m}, \texttt{T}) = \overline{\texttt{U}} \to \texttt{U}_0}$$

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

Method body lookup:

$$\frac{\begin{array}{c}CT(\texttt{T}) = \texttt{class C extends S \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K}\ \overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{U}_0\ \texttt{m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \in \overline{\texttt{M}}\end{array}}{\mathit{mbody}(\texttt{m}, \texttt{T}) = (\overline{\texttt{x}}, \texttt{e}, \texttt{T})}$$

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

Enclosing instance lookup:

$$\mathit{encl}_{\texttt{T.C}}(\texttt{e}_0\texttt{.new T.C(}\overline{\texttt{e}}\texttt{)}) = \texttt{e}_0$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;...\}} \qquad \#(\overline{\texttt{f}}) = \#(\overline{\texttt{e}})}{\mathit{encl}_{\texttt{T}}(\texttt{new C(}\overline{\texttt{d}}\texttt{, }\overline{\texttt{e}}\texttt{)}) = \mathit{encl}_{\texttt{T}}(\texttt{new D(}\overline{\texttt{d}}\texttt{)})}$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends U.D \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;...\}} \qquad \#(\overline{\texttt{f}}) = \#(\overline{\texttt{e}})}{\mathit{encl}_{\texttt{T}}(\texttt{new C(}\overline{\texttt{d}}\texttt{, d}_0\texttt{, }\overline{\texttt{e}}\texttt{)}) = \mathit{encl}_{\texttt{T}}(\texttt{d}_0\texttt{.new U.D(}\overline{\texttt{d}}\texttt{)})}$$

$$\frac{\texttt{T} \neq \texttt{S.C} \qquad CT(\texttt{S.C}) = \texttt{class C extends D \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;...\}} \qquad \#(\overline{\texttt{f}}) = \#(\overline{\texttt{e}})}{\mathit{encl}_{\texttt{T}}(\texttt{e}_0\texttt{.new S.C(}\overline{\texttt{d}}\texttt{, }\overline{\texttt{e}}\texttt{)}) = \mathit{encl}_{\texttt{T}}(\texttt{new D(}\overline{\texttt{d}}\texttt{)})}$$

$$\frac{\texttt{T} \neq \texttt{S.C} \qquad CT(\texttt{S.C}) = \texttt{class C extends U.D \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;...\}} \qquad \#(\overline{\texttt{f}}) = \#(\overline{\texttt{e}})}{\mathit{encl}_{\texttt{T}}(\texttt{e}_0\texttt{.new S.C(}\overline{\texttt{d}}\texttt{, d}_0\texttt{, }\overline{\texttt{e}}\texttt{)}) = \mathit{encl}_{\texttt{T}}(\texttt{d}_0\texttt{.new U.D(}\overline{\texttt{d}}\texttt{)})}$$

FIG. 9. FJI: auxiliary definitions

Computation:

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).f}_i \longrightarrow \mathtt{e}_i} \quad (\text{Ri-FieldT}) \qquad \frac{\mathit{fields}(\mathtt{T.C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{e}_0\mathtt{.new\ T.C(}\overline{\mathtt{e}}\mathtt{).f}_i \longrightarrow \mathtt{e}_i} \quad (\text{Ri-FieldI})$$

$$\frac{\begin{array}{c} \mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{d}_0, \mathtt{C}_1 . \cdots . \mathtt{C}_n) \\ \mathtt{c}_n \stackrel{\text{def}}{=} \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \qquad \mathtt{c}_i \stackrel{\text{def}}{=} \mathit{encl}_{\mathtt{C}_1 . \cdots . \mathtt{C}_{i+1}}(\mathtt{c}_{i+1})^{\ i \in 1 \ldots n-1} \end{array}}{\mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).m(}\overline{\mathtt{d}}\mathtt{)} \longrightarrow \left[\begin{array}{l} \overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{c}_n/\mathtt{this}, \\ \mathtt{c}_i/\mathtt{C}_i\mathtt{.this}^{\ i \in 1 \ldots n} \end{array}\right] \mathtt{d}_0} \quad (\text{Ri-InvkT})$$

$$\frac{\begin{array}{c} \mathit{mbody}(\mathtt{m}, \mathtt{T.C}) = (\overline{\mathtt{x}}, \mathtt{d}_0, \mathtt{C}_1 . \cdots . \mathtt{C}_n) \\ \mathtt{c}_n \stackrel{\text{def}}{=} \mathtt{e}_0\mathtt{.new\ T.C(}\overline{\mathtt{e}}\mathtt{)} \qquad \mathtt{c}_i \stackrel{\text{def}}{=} \mathit{encl}_{\mathtt{C}_1 . \cdots . \mathtt{C}_{i+1}}(\mathtt{c}_{i+1})^{\ i \in 1 \ldots n-1} \end{array}}{\mathtt{e}_0\mathtt{.new\ T.C(}\overline{\mathtt{e}}\mathtt{).m(}\overline{\mathtt{d}}\mathtt{)} \longrightarrow \left[\begin{array}{l} \overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{c}_n/\mathtt{this}, \\ \mathtt{c}_i/\mathtt{C}_i\mathtt{.this}^{\ i \in 1 \ldots n} \end{array}\right] \mathtt{d}_0} \quad (\text{Ri-InvkI})$$

Congruence:

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0\mathtt{.f} \longrightarrow \mathtt{e}_0'\mathtt{.f}} \quad (\text{RCi-Field}) \qquad \frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \longrightarrow \mathtt{e}_0'\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}} \quad (\text{RCi-InvRecv})$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{e}_0\mathtt{.m(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{e}_0\mathtt{.m(}\ldots\mathtt{,e}_i'\mathtt{,}\ldots\mathtt{)}} \quad (\text{RCi-InvArg})$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ C(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{new\ C(}\ldots\mathtt{,e}_i'\mathtt{,}\ldots\mathtt{)}} \quad (\text{RCi-TopArg})$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0\mathtt{.new\ T.C(}\overline{\mathtt{e}}\mathtt{)} \longrightarrow \mathtt{e}_0'\mathtt{.new\ T.C(}\overline{\mathtt{e}}\mathtt{)}} \quad (\text{RCi-InnerEnc})$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{e}_0\mathtt{.new\ T.C(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{e}_0\mathtt{.new\ T.C(}\ldots\mathtt{,e}_i'\mathtt{,}\ldots\mathtt{)}} \quad (\text{RCi-InnerArg})$$

FIG. 10. FJI: reduction rules

$\mathtt{C}_{n-1}$`.this` is obtained by $encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e})$, where `e` is the receiver object; similarly, $\mathtt{C}_{n-2}$`.this` is obtained by $encl_{\mathtt{C}_1.\cdots.\mathtt{C}_{n-1}}(encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e}))$, and so on. 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 also appear in the figure.

For example, if the class table includes `Outer`, `RefinedInner`, `Pair`, `A`, and `B`, then

```
new RefinedInner(
 new Outer(new Pair(new A(), new B())), new Object()).snd_p()
```

reduces to `new B()` as follows:

```
    new RefinedInner(
     new Outer(new Pair(new A(), new B())), new Object()).snd_p()
⟶  new Outer(new Pair(new A(), new B())).p.snd
⟶  new Pair(new A(), new B()).snd
⟶  new B()
```

3.6. Typing

The typing rules for expressions, method declarations, and class declarations are given in Figure 11. An environment Γ is a finite mapping from variables to types, written $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}$. The typing judgment for expressions has the form $\Gamma \vdash \mathtt{e} \in \mathtt{T}$, read "in the environment Γ, expression `e` has type `T`." The typing rules are syntax directed, with one rule for each form of expression. The typing rules for object instantiations and method invocations check that each actual parameter has a type which is a subtype of the corresponding formal parameter type obtained by *fields* or *mtype*; the enclosing object must have a type which is a subtype of the annotated type `T` in `new T.C(`$\overline{\mathtt{e}}$`)`. We abbreviate sequences of typing or subtyping judgments in the obvious way, writing $\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{T}}$ as shorthand for $\Gamma \vdash \mathtt{e}_1 \in \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{e}_n \in \mathtt{T}_n$ and $\overline{\mathtt{S}} <: \overline{\mathtt{T}}$ as shorthand for $\mathtt{S}_1 <: \mathtt{T}_1, \ldots, \mathtt{S}_n <: \mathtt{T}_n$.

The typing judgment for method declarations has the form $\mathtt{M}$ `OK IN` $\mathtt{C}_1.\cdots.\mathtt{C}_n$, read "method declaration `M` is ok if it is declared in class $\mathtt{C}_1.\cdots.\mathtt{C}_n$." The body of the method is typed under the context in which the formal parameters of the method have their declared types and each $\mathtt{C}_i$`.this` has the type $\mathtt{C}_1.\cdots.\mathtt{C}_i$. (Thus, as in FJ, a method with a parameter of name `this` or `C.this` is rejected as the type environment is ill formed.) If a method with the same name is declared in the superclass then it must have the same type in the subclass.

The typing judgment for class declarations has the form `L OK IN P`, read "class declaration `L` is ok if it is declared in `P`." If `P` is a type `T`, the class declaration `L` is an inner class; otherwise, `L` is a top-level class. The typing rules check that the constructor applies `super` to the fields of the superclass and initializes the fields declared in this class, and that each method declaration and inner class declaration in the class is ok. The condition $\mathtt{C} \notin \mathtt{P}$ ensures that the (simple) class name to be defined is not also a simple name of one of the enclosing classes, so as to avoid ambiguity of the meaning of `C.this`.

Expression typing:

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

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

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{U}} \to \mathtt{U}_0 \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{U}}}{\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{U}_0} \qquad \text{(TI-INVK)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}} \qquad \text{(TI-NEWTOP)}$$

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{T.C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Gamma \vdash \mathtt{e}_0 \in \mathtt{S} \qquad \mathtt{S} \mathrel{<:} \mathtt{T} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}}\end{array}}{\Gamma \vdash \mathtt{e}_0\mathtt{.new\ T.C(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T.C}} \qquad \text{(TI-NEWINNER)}$$

Method typing:

$$\frac{\begin{array}{c} \begin{array}{l}\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}_1. \cdots .\mathtt{C}_n, \\ \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1. \cdots .\mathtt{C}_i \ {}^{i \in 1 \ldots n}\end{array} \vdash \mathtt{e}_0 \in \mathtt{S}_0 \qquad \mathtt{S}_0 \mathrel{<:} \mathtt{T}_0 \\ \mathit{CT}(\mathtt{C}_1. \cdots .\mathtt{C}_n) = \mathtt{class\ C}_n\ \mathtt{extends\ S\ \{...\}} \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{U}} \to \mathtt{U}_0, \text{then } \overline{\mathtt{U}} = \overline{\mathtt{T}} \text{ and } \mathtt{U}_0 = \mathtt{T}_0 \end{array}}{\mathtt{T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{return\ e}_0\mathtt{;\}\ OK\ IN\ C}_1. \cdots .\mathtt{C}_n} \qquad \text{(TI-METHOD)}$$

Class typing:

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C(}\overline{\mathtt{S}}\ \overline{\mathtt{g}}\mathtt{,\ }\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{\ super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}}\mathtt{;\ \}} \\ \mathtt{C} \notin \mathtt{P} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{S}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ in\ P.C} \qquad \overline{\mathtt{L}}\ \mathtt{OK\ in\ P.C}\end{array}}{\mathtt{class\ C\ extends\ D\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{L}}\ \overline{\mathtt{M}}\mathtt{\}\ OK\ IN\ P}} \qquad \text{(TI-EXTTOP)}$$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C(}\overline{\mathtt{S}}\ \overline{\mathtt{g}}\mathtt{,\ T\ g}_0\mathtt{,\ }\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{\ g}_0\mathtt{.super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}}\mathtt{;\ \}} \\ \mathtt{C} \notin \mathtt{P} \qquad \mathit{fields}(\mathtt{T.D}) = \overline{\mathtt{S}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ in\ P.C} \qquad \overline{\mathtt{L}}\ \mathtt{OK\ in\ P.C}\end{array}}{\mathtt{class\ C\ extends\ T.D\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{L}}\ \overline{\mathtt{M}}\mathtt{\}\ OK\ IN\ P}} \qquad \text{(TI-EXTINNER)}$$

FIG. 11. FJI: typing Rules

3.7. Properties

As well as FJ programs, FJI programs also enjoy standard subject reduction (Theorem 3.1) and progress properties (Theorem 3.2), which together guarantee that a well-typed (closed) program never gets stuck on field accesses or method invocations and yields a fully-evaluated value of an appropriate type (Theorem 3.3.

THEOREM 3.1 (Subject Reduction). *If* $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$, *then* $\Gamma \vdash \mathtt{e}' \in \mathtt{T}'$ *for some* $\mathtt{T}'$ *such that* $\mathtt{T}' <: \mathtt{T}$.

THEOREM 3.2 (Progress). *Suppose* $\mathtt{e}$ *is a well-typed expression.*

1.*If* $\mathtt{e}$ *includes* $\mathtt{new\ C_0(\overline{e}).f}$ *as a subexpression, then fields*$(\mathtt{C_0}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *and* $\mathtt{f} \in \overline{\mathtt{f}}$. *Similarly, if* $\mathtt{e}$ *includes* $\mathtt{e_0.new\ T_0.C(\overline{e}).f}$ *as a subexpression, then fields*$(\mathtt{T_0.C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *and* $\mathtt{f} \in \overline{\mathtt{f}}$.

2.*If* $\mathtt{e}$ *includes* $\mathtt{new\ C_0(\overline{e}).m(\overline{d})}$ *as a subexpression, then mbody*$(\mathtt{m}, \mathtt{C_0}) = (\overline{\mathtt{x}}, \mathtt{e_0}, \mathtt{C_1.\cdots.C}_n)$ *and* $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$, *and* $\mathtt{c}_1, \ldots, \mathtt{c}_n$ *appearing in the rule* RI-INVKT *are well defined. Similarly, if* $\mathtt{e}$ *includes* $\mathtt{e_0.new\ T_0.C(\overline{e}).m(\overline{d})}$ *as a subexpression, then mbody*$(\mathtt{m}, \mathtt{T_0.C}) = (\overline{\mathtt{x}}, \mathtt{d_0}, \mathtt{C_1.\cdots.C}_n)$ *and* $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$ *and* $\mathtt{c}_1, \ldots, \mathtt{c}_n$ *appearing in the rule* RI-INVKI *are well defined.*

THEOREM 3.3 (FJI type soundness). *If* $\emptyset \vdash \mathtt{e} \in \mathtt{T}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$ *with* $\mathtt{e}'$ *being a normal form, then* $\mathtt{e}'$ *is a value* $\mathtt{v}$, *given by the syntax* $\mathtt{v} ::= \mathtt{new\ C(\overline{v})} \mid \mathtt{v.new\ T(\overline{v})}$, *such that* $\emptyset \vdash \mathtt{v} \in \mathtt{S}$ *for some type* $\mathtt{S}$ *with* $\mathtt{S} <: \mathtt{T}$.

Proof. Immediate from Theorems 3.1 and 3.2. ∎

We develop proofs of Theorems 3.1 and 3.2 below. In what follows, the underlying class table is assumed to be ok.

LEMMA 3.1. *If* $\Gamma \vdash \mathtt{e} \in \mathtt{T}$, *then* $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} \in \mathtt{T}$.

Proof. By straightforward induction on the derivation of $\Gamma \vdash \mathtt{e} \in \mathtt{T}$. ∎

LEMMA 3.2. *If mtype*$(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{U}} \rightarrow \mathtt{U_0}$, *then mtype*$(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{U}} \rightarrow \mathtt{U_0}$ *for all* $\mathtt{S} <: \mathtt{T}$.

Proof. Straightforward induction on the derivation of $\mathtt{S} <: \mathtt{T}$. ∎

LEMMA 3.3. *If* $\Gamma \vdash \mathtt{new\ C(\overline{e})} \in \mathtt{T}$ *and* $\mathtt{T} <: \mathtt{U.D}$, *then* $\Gamma \vdash \mathit{encl}_{\mathtt{U.D}}(\mathtt{new\ C(\overline{e})}) \in \mathtt{S}$ *for some* $\mathtt{S}$ *such that* $\mathtt{S} <: \mathtt{U}$. *Similarly, if* $\Gamma \vdash \mathtt{e_0.new\ T_0.C(\overline{e})} \in \mathtt{T}$ *and* $\mathtt{T} <: \mathtt{U.D}$, *then* $\Gamma \vdash \mathit{encl}_{\mathtt{U.D}}(\mathtt{e_0.new\ T_0.C(\overline{e})}) \in \mathtt{S}$ *for some* $\mathtt{S}$ *such that* $\mathtt{S} <: \mathtt{U}$.

Proof. Both parts are proved simultaneously by induction on the derivation of $\mathtt{T} <: \mathtt{U.D}$.

Case. $\mathtt{T} = \mathtt{U.D}$

The assumption $\Gamma \vdash \mathtt{new\ C(\overline{e})} \in \mathtt{T}$ of the first part never holds. As for the second part, by the rule TI-NEWINNER, we have

$$\begin{array}{lll} \mathtt{T_0} = \mathtt{U} & \mathtt{C} = \mathtt{D} & \\ \Gamma \vdash \mathtt{e_0} \in \mathtt{S_0} & \mathtt{S_0} <: \mathtt{T_0} & \mathit{fields}(\mathtt{T_0}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} & \overline{\mathtt{S}} <: \overline{\mathtt{T}} & \end{array}$$

Since $\mathit{encl}_{\mathtt{U.D}}(\mathtt{e_0.new\ T_0.C(\overline{e})}) = \mathtt{e_0}$, letting $\mathtt{S} = \mathtt{S_0}$ finishes the case.

Case. $\quad CT(\texttt{T}) = \texttt{class C extends U.D \{}\overline{\texttt{U}}\ \overline{\texttt{g}}\texttt{; ...\}}$
We have two subcases depending on whether T is a simple name or not. We show only the subcase where $\texttt{T} = \texttt{C}$ as a representative case since the other case is similar. It suffices to show the first part where we have $\Gamma \vdash \texttt{new C(}\overline{\texttt{e}}\texttt{)} \in \texttt{C}$. (We never have $\Gamma \vdash \texttt{e}_0\texttt{.new T}_0\texttt{.C(}\overline{\texttt{e}}\texttt{)} \in \texttt{C}$.) By the rule TI-NEWTOP, we have

$$\begin{array}{l} \mathit{fields}(\texttt{C}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \\ \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \\ \overline{\texttt{S}} <: \overline{\texttt{T}} \end{array}$$

Also, by the third rule in the definition of *fields*, we have

$$\overline{\texttt{T}}\ \overline{\texttt{f}} = \mathit{fields}(\texttt{U.D}), \texttt{U this\$C}_1\texttt{\$}\cdots\texttt{\$C}_n\texttt{\$D}, \overline{\texttt{U}}\ \overline{\texttt{g}}$$

where $\texttt{U} = \texttt{C}_1\texttt{.}\cdots\texttt{.C}_n$. Let $\overline{\texttt{e}} = \overline{\texttt{d}}\texttt{, d}_0\texttt{, }\overline{\texttt{c}}$ where $\#(\overline{\texttt{c}}) = \#(\overline{\texttt{g}})$. Then, $\Gamma \vdash \texttt{d}_0 \in \texttt{S}$ for some S and $\texttt{S} <: \texttt{U}$. On the other hand, by definition,

$$\mathit{encl}_{\texttt{U.D}}(\texttt{new C(}\overline{\texttt{d}}\texttt{, d}_0\texttt{, }\overline{\texttt{c}}\texttt{)}) = \mathit{encl}_{\texttt{U.D}}(\texttt{d}_0\texttt{.new U.D(}\overline{\texttt{d}}\texttt{)}) = \texttt{d}_0.$$

finishing the case.

Case. $\quad \texttt{T} <: \texttt{S} \qquad \texttt{S} <: \texttt{U.D}$
We have four subcases depending on whether S is a simple name or not and whether T is a simple name or not. We show the case where $\texttt{T} = \texttt{C}$ and $\texttt{S} = \texttt{E}$ for some C and E; the other cases are similar.

Since $\Gamma \vdash_{\mathrm{FJI}} \texttt{new C(}\overline{\texttt{e}}\texttt{)} \in \texttt{C}$, it is easy to show that $\Gamma \vdash_{\mathrm{FJI}} \texttt{new E(}\overline{\texttt{d}}\texttt{)} \in \texttt{E}$ for some $\overline{\texttt{d}}$ such that $\overline{\texttt{e}} = \overline{\texttt{d}}, \ldots$. By the induction hypothesis, $\Gamma \vdash \mathit{encl}_{\texttt{U.D}}(\texttt{new E(}\overline{\texttt{d}}\texttt{)}) \in \texttt{V}$ for some V such that $\texttt{V} <: \texttt{U}$. By induction on the derivation of $\texttt{C} <: \texttt{E}$, it is easy to show that $\mathit{encl}_{\texttt{U.D}}(\texttt{new E(}\overline{\texttt{d}}\texttt{)}) = \mathit{encl}_{\texttt{U.D}}(\texttt{new C(}\overline{\texttt{e}}\texttt{)})$, finishing the subcase.
■

LEMMA 3.4 (Term Substitution). *If* $\Gamma, \overline{\texttt{x}} : \overline{\texttt{U}} \vdash \texttt{e} \in \texttt{T}$ *and* $\Gamma \vdash \overline{\texttt{d}} \in \overline{\texttt{T}}$ *where* $\overline{\texttt{T}} <: \overline{\texttt{U}}$, *then* $\Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{e} \in \texttt{S}$ *and* $\texttt{S} <: \texttt{T}$.

Proof. By induction on the derivation of $\Gamma, \overline{\texttt{x}} : \overline{\texttt{U}} \vdash \texttt{e} \in \texttt{T}$.

Case TI-VAR. $\quad \texttt{e} = \texttt{y} \qquad \texttt{T} = \Gamma(\texttt{y})$
The subcase $\texttt{y} \notin \overline{\texttt{x}}$ is trivial since $[\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{y} = \texttt{y}$. On the other hand, if $\texttt{y} = \texttt{x}_i$ and $\texttt{T} = \texttt{U}_i$, then, since $[\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{y} = \texttt{d}_i$, letting $\texttt{S} = \texttt{T}_i$ finishes the case.

Case TI-FIELD. $\quad \texttt{e} = \texttt{e}_0\texttt{.f}_i \qquad \texttt{T} = \texttt{S}_i \qquad \Gamma, \overline{\texttt{x}} : \overline{\texttt{U}} \vdash \texttt{e}_0 \in \texttt{T}_0$
$\qquad \mathit{fields}(\texttt{T}_0) = \overline{\texttt{S}}\ \overline{\texttt{f}}$
By the induction hypothesis, we have some $\texttt{S}_0$ such that $\Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{e}_0 \in \texttt{S}_0$ and $\texttt{S}_0 <: \texttt{T}_0$. It is easy to show that

$$\mathit{fields}(\texttt{S}_0) = \mathit{fields}(\texttt{T}_0), \overline{\texttt{D}}\ \overline{\texttt{g}}$$

for some $\overline{\texttt{D}}\ \overline{\texttt{g}}$. Therefore, by the rule TI-FIELD, $\Gamma \vdash ([\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{e}_0)\texttt{.f}_i \in \texttt{S}_i$.

Case TI-INVK. $\quad \mathtt{e} = \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \qquad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{U}} \vdash \mathtt{e}_0 \in \mathtt{T}_0$
$\quad mtype(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{V}} \rightarrow \mathtt{T}$
$\quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{U}} \vdash \overline{\mathtt{e}} \in \overline{\mathtt{T}} \qquad \overline{\mathtt{T}} <: \overline{\mathtt{V}}$

By the induction hypothesis, we have some $\mathtt{S}_0$ and $\overline{\mathtt{S}}$ such that

$$\begin{array}{ll} \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 \in \mathtt{S}_0 & \mathtt{S}_0 <: \mathtt{T}_0 \\ \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} \in \overline{\mathtt{S}} & \overline{\mathtt{S}} <: \overline{\mathtt{T}} \end{array}$$

By Lemma 3.2, $mtype(\mathtt{m}, \mathtt{S}_0) = \overline{\mathtt{V}} \rightarrow \mathtt{T}$. Moreover, $\overline{\mathtt{S}} <: \overline{\mathtt{V}}$ by transitivity of <:. Therefore, by the rule TI-INVK, $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\mathtt{.m(}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T}$.

Case TI-NEWTOP. $\quad \mathtt{e} = \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \qquad fields(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{U}} \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}}$
$\quad \overline{\mathtt{S}} <: \overline{\mathtt{T}}$

By the induction hypothesis, we have $\overline{\mathtt{V}}$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} \in \overline{\mathtt{V}}$ and $\overline{\mathtt{V}} <: \overline{\mathtt{S}}$. Moreover $\overline{\mathtt{V}} <: \overline{\mathtt{T}}$, by transitivity of <:. Therefore, by the rule TI-NEWTOP, $\Gamma \vdash \mathtt{new\ C(}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}$.

The case for TI-NEWINNER is similar. ■

LEMMA 3.5. *If* $mtype(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{U}} \rightarrow \mathtt{U}_0$ *and* $mbody(\mathtt{m}, \mathtt{T}) = (\overline{\mathtt{x}}, \mathtt{e}_0, \mathtt{C}_1 . \cdots . \mathtt{C}_n)$*, then* $\mathtt{T} <: \mathtt{C}_1 . \cdots . \mathtt{C}_n$ *and* $\overline{\mathtt{x}} : \overline{\mathtt{U}}, \mathtt{this} : \mathtt{C}_1 . \cdots . \mathtt{C}_n, \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1 . \cdots . \mathtt{C}_i\ ^{i \in 1 \ldots n} \vdash \mathtt{e}_0 \in \mathtt{T}_0$ *for some* $\mathtt{T}_0$ *such that* $\mathtt{T}_0 <: \mathtt{U}_0$.

Proof. By induction on the derivation of $mbody(\mathtt{m}, \mathtt{T})$. The base case (where m is defined in T and $\mathtt{T} = \mathtt{C}_1 . \cdots . \mathtt{C}_n$) is easy since $\overline{\mathtt{x}} : \overline{\mathtt{U}}, \mathtt{this} : \mathtt{C}_1 . \cdots . \mathtt{C}_n, \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1 . \cdots . \mathtt{C}_i\ ^{i \in 1 \ldots n} \vdash \mathtt{e} \in \mathtt{T}_0$ for some $\mathtt{T}_0$ such that $\mathtt{T}_0 <: \mathtt{U}_0$ by TI-METHOD. The case for induction step is also straightforward. ■

Proof (Proof of Theorem 3.1). By induction on a derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$, with a case analysis on the reduction rule used.

Case RI-FIELDT. $\quad \mathtt{e} = \mathtt{(new\ C_0(}\overline{\mathtt{e}}\mathtt{)).f}_i \qquad fields(\mathtt{C}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathtt{e}' = \mathtt{e}_i$

By the rule TI-FIELD, we have

$$\Gamma \vdash \mathtt{new\ C_0(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T}_0 \qquad \mathtt{C} = \mathtt{T}_i$$

for some $\mathtt{T}_0$. Then, by the rule TI-NEWTOP,

$$\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} <: \overline{\mathtt{T}} \qquad \mathtt{T}_0 = \mathtt{C}_0$$

In particular, $\Gamma \vdash \mathtt{e}_i \in \mathtt{S}_i$, finishing the case since $\mathtt{S}_i <: \mathtt{T}_i$.

Case RI-FIELDI. Similar to the case for RI-FIELDT.

Case RI-INVKT. $\quad \mathtt{e} = \mathtt{(new\ C_0(}\overline{\mathtt{e}}\mathtt{)).m(}\overline{\mathtt{d}}\mathtt{)}$
$\quad mbody(\mathtt{m}, \mathtt{C}_0) = (\overline{\mathtt{x}}, \mathtt{e}_0, \mathtt{C}_1 . \cdots . \mathtt{C}_n)$
$$\mathtt{c}_i = \begin{cases} \mathtt{new\ C_0(}\overline{\mathtt{e}}\mathtt{)} & (i = n) \\ encl_{\mathtt{C}_1 . \cdots . \mathtt{C}_{i+1}}(\mathtt{c}_{i+1}) & (i \in 1 \ldots n-1) \end{cases}$$
$\quad \mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{c}_n/\mathtt{this}, \mathtt{c}_i/\mathtt{C}_i\mathtt{.this}\ ^{i \in 1 \ldots n}]\mathtt{e}_0$

By the rules TI-INVK and TI-NEWTOP, we have

$$\Gamma \vdash \mathtt{new\ C_0(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}_0 \qquad \Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} <: \overline{\mathtt{T}} \qquad mtype(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{T}} \rightarrow \mathtt{T}$$

By Lemma 3.5,

$$\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{C}_1.\cdots.\mathtt{C}_n, \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1.\cdots.\mathtt{C}_i \ ^{i\in 1\ldots n} \vdash \mathtt{e}_0 \in \mathtt{S}_0$$

where $\mathtt{S}_0$ <: $\mathtt{T}$ and $\mathtt{C}$ <: $\mathtt{C}_1.\cdots.\mathtt{C}_n$. By Lemma 3.1,

$$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{C}_1.\cdots.\mathtt{C}_n, \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1.\cdots.\mathtt{C}_i \ ^{i\in 1\ldots n} \vdash \mathtt{e}_0 \in \mathtt{S}_0.$$

By using the fact that $\mathtt{C}$ <: $\mathtt{C}_1.\cdots.\mathtt{C}_n$ and Lemma 3.3 repeatedly, we have $\overline{\mathtt{U}}$ such that $\Gamma \vdash \mathtt{c}_i \in \mathtt{U}_i$ and $\mathtt{U}_i$ <: $\mathtt{C}_1.\cdots.\mathtt{C}_i$ for $i \in 1\ldots n$. Then, by Lemma 3.4,

$$\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{c}_n/\mathtt{this},\ \mathtt{c}_i/\mathtt{C}_i\mathtt{.this}\ ^{i\in 1\ldots n}]\mathtt{e}_0 \in \mathtt{U}_0$$

for some $\mathtt{U}_0$ <: $\mathtt{S}_0$. Finally, letting $\mathtt{T}' = \mathtt{U}_0$ finishes this case.

Case RI-INVKI. Similar to the case RI-INVKT.

Cases for congruence rules (RCI-$\cdots$) are straightforward. ■

Proof (Proof of Theorem 3.2).

1. If $\mathtt{e}$ has $\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}})\mathtt{.f}$ (or $\mathtt{e}_0\mathtt{.new}\ \mathtt{T}_0\mathtt{.C}(\overline{\mathtt{e}})\mathtt{.f}$) as a subexpression, then, by well-typedness of the subexpression, it's easy to check that *fields*$(\mathtt{C}_0)$ (or *fields*$(\mathtt{T}_0\mathtt{.C})$) is well-defined and $\mathtt{f}$ appears in it.

2. If $\mathtt{e}$ has $\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}})\mathtt{.m}(\overline{\mathtt{d}})$ as a subexpression, then, it's also easy to show $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0, \mathtt{C}_0.\cdots.\mathtt{C}_n)$ and $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$ from the fact that $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{D}}\to\mathtt{D}$ where $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{D}})$. Finally, by Lemma 3.3, $\mathit{encl}_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}}))$, $\ldots$, $\mathit{encl}_{\mathtt{C}_1}(\cdots\mathit{encl}_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}})))$ are well defined. Similarly for a subexpression of the form $\mathtt{e}_0\mathtt{.new}\ \mathtt{T}_0\mathtt{.C}(\overline{\mathtt{e}})\mathtt{.m}(\overline{\mathtt{d}})$.

■

4. TRANSLATION SEMANTICS

In this section we consider the other style of semantics: translation from FJI to FJ. Every inner class is compiled to a top-level class with one additional field holding a reference to the direct enclosing instance; occurrences of qualified `this` are translated into accesses to this field. For example, the `Outer` and `RefinedInner` classes in the previous section are compiled to the following three FJ classes.

```
class Outer extends Object {
  Pair p;
  Outer(Pair p) { super(); this.p = p; }
  Outer$Inner make_inner () { return new Outer$Inner(this); }
}

class Outer$Inner extends Object {
  Outer this$Outer$Inner;
  Outer$Inner(Outer this$Outer$Inner) {
    super(); this.this$Outer$Inner = this$Outer$Inner; }
  Object snd_p { return this.this$Outer$Inner.p.snd; }
}
```

```
class RefinedInner extends Outer$Inner {
  Object c;
  RefinedInner(Outer this$Outer$Inner, Object c) {
    super(this$Outer$Inner); this.c = c;
  }
}
```

The inner class `Outer.Inner` is compiled to the top-level class `Outer$Inner`; the field `this$Outer$Inner` holds an `Outer` object, which corresponds to the direct enclosing instance `Outer.this` in the original FJI program; thus, `Outer.this` is compiled to the field access expression `this.this$Outer$Inner`.

We give a compilation function $|\cdot|$ for each syntactic category. Except for types, the compilation functions take as their second argument the FJI class name (or, sometimes, $\star$) where the entity being translated is defined, written $|\cdot|_{\mathtt{T}}$ (or $|\cdot|_{\star}$).

4.1. Compilation of Types, Expressions and Methods

The compilation of types, written $|\mathtt{T}|$, compilation of expressions, written $|\mathtt{e}|_{\mathtt{T}}$, compilation of methods, written $|\mathtt{M}|_{\mathtt{T}}$, are given in Figure 12. We write $|\overline{\mathtt{e}}|_{\mathtt{T}}$ as shorthand for $|\mathtt{e}_1|_{\mathtt{T}}, \ldots, |\mathtt{e}_n|_{\mathtt{T}}$ (and similarly for $|\overline{\mathtt{T}}|$, $|\overline{\mathtt{M}}|_{\mathtt{T}}$ and $|\overline{\mathtt{L}}|_{\mathtt{P}}$).

First of all, every qualified class name is translated to a simple name obtained by syntactic replacement of `.` with `$`.

The compilation of expressions is fairly straightforward except for constructs relevant to inner classes. As we saw above, a compiled inner class has one additional field, called $\mathtt{this\$}|\mathtt{T}|$, where `T` is the original class name. Then, an enclosing instance $\mathtt{e}_0$ of $\mathtt{e}_0\mathtt{.new\ T.C}(\overline{\mathtt{e}})$ will become the last argument of the compiled constructor invocation, while $\mathtt{C}_i\mathtt{.this}$ in the class $\mathtt{C}_1.\cdots.\mathtt{C}_n$ becomes an expression that follows references to the direct enclosing instance in sequence until it reaches the desired one.

The compilation of methods is also straightforward; each type and the method body is compiled. We use the notation $|\overline{\mathtt{T}}|\ \ \overline{\mathtt{x}}$ for $|\mathtt{T}_1|\ \ \mathtt{x}_1, \ldots, |\mathtt{T}_n|\ \ \mathtt{x}_n$.

4.2. Compilation of Constructors and Classes

Compilation of constructors, written $|\mathtt{K}|_{\mathtt{T}}$, is given in Figure 13. It has four cases, depending on whether the constructor is for a top-level class or an inner class and whether its direct superclass is a top-level class or an inner class. When the constructor is for an inner class, one more argument corresponding to the enclosing instance is added to the argument list; the name of the constructor becomes $|\mathtt{T.C}|$, the translation of the qualified name of the class. When the direct superclass is inner (the second and fourth cases), the argument used for the qualification of $\mathtt{f.super}(\overline{\mathtt{f}})$ becomes the last argument of the `super()` invocation.

Finally, the compilation of classes, written $|\mathtt{L}|_{\mathtt{P}}$, is given also in Figure 13. The constructor, inner classes, and methods of class `C` defined in `P` are compiled with the auxiliary argument `P.C`. Inner classes $\overline{\mathtt{L}}$ become top-level classes. As in constructor compilation, when the class being compiled is inner, its name changes to $|\mathtt{T.C}|$ and the field $\mathtt{this\$}|\mathtt{T.C}|$, holding an enclosing instance, is added. The compilation of the class table, written $|CT|$, is achieved by compiling all top-level classes $\overline{\mathtt{L}}$ in CT (i.e., $|\overline{\mathtt{L}}|_{\star}$).

Compilation of types:

$$|\mathtt{C}_1.\cdots.\mathtt{C}_n| = \mathtt{C}_1\$\cdots\$\mathtt{C}_n$$

Compilation of expressions:

$$\begin{array}{lcl}
|\mathtt{x}|_{\mathtt{T}} & = & \mathtt{x} \\
|\mathtt{e}_0.\mathtt{f}|_{\mathtt{T}} & = & |\mathtt{e}_0|_{\mathtt{T}}.\mathtt{f} \\
|\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})|_{\mathtt{T}} & = & |\mathtt{e}_0|_{\mathtt{T}}.\mathtt{m}(\,|\overline{\mathtt{e}}|_{\mathtt{T}}\,) \\
|\mathtt{new\ D}(\overline{\mathtt{e}})|_{\mathtt{T}} & = & \mathtt{new\ D}(\,|\overline{\mathtt{e}}|_{\mathtt{T}}\,) \\
|\mathtt{e}_0.\mathtt{new\ T.D}(\overline{\mathtt{e}})|_{\mathtt{T}} & = & \mathtt{new}\ |\mathtt{T.D}|\,(\,|\overline{\mathtt{e}}|_{\mathtt{T}}\,,\,|\mathtt{e}_0|_{\mathtt{T}}\,) \\
|\mathtt{this}|_{\mathtt{T}} & = & \mathtt{this} \\
|\mathtt{C}_n.\mathtt{this}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n} & = & \mathtt{this} \\
|\mathtt{C}_i.\mathtt{this}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n} & = & |\mathtt{C}_{i+1}.\mathtt{this}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n}.\mathtt{this}\$\mathtt{C}_1\$\cdots\$\mathtt{C}_{i+1} \\
 & & (1 \le i \le n-1)
\end{array}$$

Compilation of methods:

$$\big|\mathtt{T}_0\ \mathtt{m}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return\ e;}\ \}\big|_{\mathtt{T}} = |\mathtt{T}_0|\ \ \mathtt{m}(\,|\overline{\mathtt{T}}|\ \ \overline{\mathtt{x}})\{\ \mathtt{return}\ \ |\mathtt{e}|_{\mathtt{T}}\,;\ \}$$

FIG. 12. Compilation of expressions and methods

4.3. Properties of Translation Semantics

We develop three theorems here. First, the translation semantics preserves typing, in the sense that a well-typed FJI program is compiled to a well-typed FJ program (Theorem 4.1). Second, we show that the behavior of a compiled program exactly reflects the behavior of the original program in FJI: for every step of reduction of a well-typed FJI program, the compiled program takes one or more steps and reaches a corresponding state (Theorem 4.2) and vice versa (Theorem 4.3). In what follows, we use subscripts FJ and FJI to show which set of rules is used.

THEOREM 4.1 (Compilation preserves typing). *When* $\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{T}}$, *we write* $|\Gamma|$ *for* $\overline{\mathtt{x}} : |\overline{\mathtt{T}}|$. *If an FJI class table* CT *is ok and* $\overline{\mathtt{x}} : \overline{\mathtt{T}}$, $\mathtt{this} : \mathtt{C}_1.\cdots.\mathtt{C}_n$, $\mathtt{C}_i.\mathtt{this} : \mathtt{C}_1.\cdots.\mathtt{C}_i\ {}^{i\in 1\ldots n} \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{T}$ *with respect to* CT, *then* $|CT|$ *is ok and* $\overline{\mathtt{x}} : |\overline{\mathtt{T}}|, \mathtt{this} : |\mathtt{C}_1.\cdots.\mathtt{C}_n| \vdash_{\mathrm{FJ}} |\mathtt{e}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n} \in |\mathtt{T}|$ *with respect to* $|CT|$.

THEOREM 4.2 (Compilation preserves execution). *If* $\Gamma \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{T}$ *where* $dom(\Gamma)$ *includes neither* `this` *nor* `C.this` *for any* `C`, *and* $\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{e}'$, *then* $|\mathtt{e}|_\star \longrightarrow_{\mathrm{FJ}}{}^{+} |\mathtt{e}'|_\star$.

THEOREM 4.3 (Compilation preserves termination). *If* $\Gamma \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{T}$ *where* $dom(\Gamma)$ *includes neither* `this` *nor* `C.this` *for any* `C`, *and* $|\mathtt{e}|_\star \longrightarrow_{\mathrm{FJ}} \mathtt{e}'$, *then* $\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{e}''$ *and* $\mathtt{e}' \longrightarrow_{\mathrm{FJ}}{}^{*} |\mathtt{e}''|_\star$ *for some* $\mathtt{e}''$.

Unfortunately, Theorems 4.2 and 4.3 would not hold for a call-by-value version of FJI, since their properties depend on our non-deterministic reduction strategy. An intuitive reason is as follows. In FJI, after method invocation, `C.this` is directly replaced with the corresponding enclosing instance. On the other hand, in the compiled FJ program, `C.this` is translated to an expression $\mathtt{this.f}_1.\mathtt{f}_2.\cdots.\mathtt{f}_n$, where each $\mathtt{f}_i$ is a mangled field name, and its evaluation may be blocked by the

Compilation of constructors:

$$\left| \begin{array}{l} \mathtt{C}(\overline{\mathtt{S}}\ \overline{\mathtt{g}},\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}) \\ \quad \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}};\} \end{array} \right|_{\mathtt{C}} = \begin{array}{l} \mathtt{C}(\left|\overline{\mathtt{S}}\right|\ \overline{\mathtt{g}},\ \left|\overline{\mathtt{T}}\right|\ \overline{\mathtt{f}}) \\ \quad \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}};\} \end{array}$$

$$\left| \begin{array}{l} \mathtt{C}(\overline{\mathtt{S}}\ \overline{\mathtt{g}},\ \mathrm{S}_0\ \mathrm{g}_0,\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}) \\ \quad \{\mathrm{g}_0.\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}};\} \end{array} \right|_{\mathtt{C}} = \begin{array}{l} \mathtt{C}(\left|\overline{\mathtt{S}}\right|\ \overline{\mathtt{g}},\ \left|\mathrm{S}_0\right|\ \mathrm{g}_0,\ \left|\overline{\mathtt{T}}\right|\ \overline{\mathtt{f}}) \\ \quad \{\mathtt{super}(\overline{\mathtt{g}},\ \mathrm{g}_0);\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}};\} \end{array}$$

$$\left| \begin{array}{l} \mathtt{C}(\overline{\mathtt{S}}\ \overline{\mathtt{g}},\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}) \\ \quad \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}};\} \end{array} \right|_{\mathtt{T.C}} = \begin{array}{l} \left|\mathtt{T.C}\right|(\left|\overline{\mathtt{S}}\right|\ \overline{\mathtt{g}},\ \left|\overline{\mathtt{T}}\right|\ \overline{\mathtt{f}}, \\ \qquad \left|\mathtt{T}\right|\ \mathtt{this\$}\left|\mathtt{T.C}\right|) \\ \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}}; \\ \quad \mathtt{this.this\$}\left|\mathtt{T.C}\right|\ \mathtt{=}\ \mathtt{this\$}\left|\mathtt{T.C}\right|;\} \end{array}$$

$$\left| \begin{array}{l} \mathtt{C}(\overline{\mathtt{S}}\ \overline{\mathtt{g}},\ \mathrm{S}_0\ \mathrm{g}_0,\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}) \\ \quad \{\mathrm{g}_0.\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}};\} \end{array} \right|_{\mathtt{T.C}} = \begin{array}{l} \left|\mathtt{T.C}\right|(\left|\overline{\mathtt{S}}\right|\ \overline{\mathtt{g}},\ \left|\mathrm{S}_0\right|\ \mathrm{g}_0, \\ \qquad \left|\overline{\mathtt{T}}\right|\ \overline{\mathtt{f}},\ \left|\mathtt{T}\right|\ \mathtt{this\$}\left|\mathtt{T.C}\right|) \\ \{\mathtt{super}(\overline{\mathtt{g}},\ \mathrm{g}_0);\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}}; \\ \quad \mathtt{this.this\$}\left|\mathtt{T.C}\right|\ \mathtt{=}\ \mathtt{this\$}\left|\mathtt{T.C}\right|;\} \end{array}$$

Compilation of classes:

$$\left|\mathtt{class\ C\ extends\ S\ }\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{L}}\ \overline{\mathtt{M}}\}\right|_{\star}$$
$$= \mathtt{class\ C\ extends}\ \left|\mathtt{S}\right|\ \{\left|\overline{\mathtt{T}}\right|\ \overline{\mathtt{f}};\ \left|\mathtt{K}\right|_{\mathtt{C}}\ \left|\overline{\mathtt{M}}\right|_{\mathtt{C}}\}\ \left|\overline{\mathtt{L}}\right|_{\mathtt{C}}$$

$$\left|\mathtt{class\ C\ extends\ S\ }\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{L}}\ \overline{\mathtt{M}}\}\right|_{\mathtt{T}}$$
$$= \mathtt{class}\ \left|\mathtt{T.C}\right|\ \mathtt{extends}\ \left|\mathtt{S}\right|\ \{\left|\overline{\mathtt{T}}\right|\ \overline{\mathtt{f}};\ \left|\mathtt{T}\right|\ \mathtt{this\$}\left|\mathtt{T.C}\right|;\ \left|\mathtt{K}\right|_{\mathtt{T.C}}\ \left|\overline{\mathtt{M}}\right|_{\mathtt{T.C}}\}\ \left|\overline{\mathtt{L}}\right|_{\mathtt{T.C}}$$

FIG. 13. Compilation of constructors and classes

context in which it appears. Therefore, reduction steps do not commute with compilation straightforwardly. Nevertheless, it should be possible to show correctness by using another technique, such as contextual equivalence [20], as Glew proved a similar result in the context of object closure conversion for a call-by-value object calculus [8].

In the rest of this section, we develop proofs of the theorems above. We begin with developing a number of required lemmas.

LEMMA 4.1. *Suppose* $|CT|$ *is well defined.*

1. *If* $\mathit{fields}_{\mathrm{FJI}}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, *then* $\mathit{fields}_{\mathrm{FJ}}(|\mathtt{C}|) = |\overline{\mathtt{T}}|\ \overline{\mathtt{f}}$.
2. *If* $\mathit{fields}_{\mathrm{FJI}}(\mathtt{T.C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *then* $\mathit{fields}_{\mathrm{FJ}}(|\mathtt{T.C}|) = |\overline{\mathtt{T}}|\ \overline{\mathtt{f}},\ |\mathtt{T}|\ \mathtt{this\$}|\mathtt{T.C}|$.

Proof. Both parts are simultaneously proved by induction on the derivation of $\mathit{fields}_{\mathrm{FJI}}(\mathtt{T})$ with an inspection of the compilation rule for classes. ∎

LEMMA 4.2. *If the compiled FJ class table* $|CT|$ *is well defined and* $\mathit{mtype}_{\mathrm{FJI}}(\mathtt{m},\ \mathtt{T}) = \overline{\mathtt{T}}\rightarrow\mathtt{U}$, *then* $\mathit{mtype}_{\mathrm{FJ}}(\mathtt{m},\ |\mathtt{T}|) = |\overline{\mathtt{T}}| \rightarrow |\mathtt{U}|$.

Proof. By induction on the derivation of $\mathit{mtype}_{\mathrm{FJI}}(\mathtt{m},\ \mathtt{T})$ with an analysis of the compilation rules for classes and methods. ∎

LEMMA 4.3. *If* $|CT|$ *is well defined and* $\mathit{mbody}_{\mathrm{FJI}}(\mathtt{m},\ \mathtt{T}) = (\overline{\mathtt{x}},\ \mathtt{e}_0,\ \mathtt{S})$, *then* $\mathit{mbody}_{\mathrm{FJ}}(\mathtt{m},\ |\mathtt{T}|) = (\overline{\mathtt{x}},\ |\mathtt{e}_0|_{\mathtt{S}})$.

Proof. By induction on the derivation of $\mathit{mbody}_{\mathrm{FJI}}(\mathtt{m},\ \mathtt{T})$ with an analysis of compilation rules for classes and methods. ∎

LEMMA 4.4. $\mathtt{S} <:_{\mathrm{FJI}} \mathtt{T}$ *if and only if* $|\mathtt{S}| <:_{\mathrm{FJ}} |\mathtt{T}|$.

Proof. Straightforward induction on the derivation of $\mathtt{S} <:_{\mathrm{FJI}} \mathtt{T}$. ∎

LEMMA 4.5. *Suppose* $\mathtt{T} <:_{\mathrm{FJI}} \mathtt{C}_1.\cdots.\mathtt{C}_n$ *where* $n \geq 2$. *If* $\mathtt{T} = \mathtt{C}$ *and* $\Gamma \vdash_{\mathrm{FJI}} \mathtt{new\ C}(\overline{\mathtt{e}}) \in \mathtt{C}$ *where* $\mathit{dom}(\Gamma)$ *does not include* `this` *or* `D.this` *for any* `D`, *then*

$$|\mathtt{new\ C}(\overline{\mathtt{e}})|_\star.\mathtt{this\$C}_1\mathtt{\$}\cdots\mathtt{\$C}_n \longrightarrow_{\mathrm{FJ}} |\mathit{encl}_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{new\ C}(\overline{\mathtt{e}}))|_\star.$$

Similarly, if $\mathtt{T} = \mathtt{S.C}$ *and* $\Gamma \vdash \mathtt{e}_0.\mathtt{new\ S.C}(\overline{\mathtt{e}}) \in \mathtt{S.C}$, *then*

$$|\mathtt{e}_0.\mathtt{new\ S.C}(\overline{\mathtt{e}})|_\star.\mathtt{this\$C}_1\mathtt{\$}\cdots\mathtt{\$C}_n \longrightarrow_{\mathrm{FJ}} |\mathit{encl}_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e}_0.\mathtt{new\ S.C}(\overline{\mathtt{e}}))|_\star.$$

Proof. By induction on the derivation of $\mathtt{T} <:_{\mathrm{FJI}} \mathtt{C}_1.\cdots.\mathtt{C}_n$.

Case. $\mathtt{T} = \mathtt{C}_1.\cdots.\mathtt{C}_n$ $\quad \mathtt{S} = \mathtt{C}_1.\cdots.\mathtt{C}_{n-1}$ $\quad \mathtt{C} = \mathtt{C}_n$
$\Gamma \vdash_{\mathrm{FJI}} \mathtt{e}_0.\mathtt{new\ S.C}(\overline{\mathtt{e}}) \in \mathtt{S.C}$

Since $|\mathtt{e}_0.\mathtt{new\ S.C}(\overline{\mathtt{e}})|_\star = \mathtt{new}\ \ |\mathtt{S.C}|\ (\,|\overline{\mathtt{e}}|_\star\,,\ |\mathtt{e}_0|_\star\,)$ and, by Lemma 4.1,

$$\mathit{fields}_{\mathrm{FJ}}(|\mathtt{S.C}|) = |\overline{\mathtt{T}}|\ \ \overline{\mathtt{f}},\ |\mathtt{S}|\ \ \mathtt{this\$C}_1\mathtt{\$}\cdots\mathtt{\$C}_n$$

for some $\overline{\mathtt{T}}$ and $\overline{\mathtt{f}}$,

$$|\mathtt{e}_0.\mathtt{new\ S.C}(\overline{\mathtt{e}})|_\star.\mathtt{this\$C}_1\mathtt{\$}\cdots\mathtt{\$C}_n \longrightarrow_{\mathrm{FJ}} |\mathtt{e}_0|_\star.$$

Finally, by definition, $encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e}_0\mathtt{.new\ S.C}(\overline{\mathtt{e}})) = \mathtt{e}_0$, finishing the case.

Case. $CT(\mathtt{T}) = \mathtt{class\ C\ extends\ C}_1\mathtt{.}\cdots\mathtt{.C}_n\ \{\overline{\mathtt{U}}\ \overline{\mathtt{f}};\ \ldots\}$
We have two cases depending on whether T is a top-level class or not. We show the case where $\mathtt{T} = \mathtt{C}$ as a representative since the other case is similar.

Let $\overline{\mathtt{e}} = \overline{\mathtt{d}}, \mathtt{e}_0, \overline{\mathtt{c}}$ such that $\#(\overline{\mathtt{c}}) = \#(\overline{\mathtt{f}})$. Using Lemma 4.1, we have

$$|\mathtt{new\ C}(\overline{\mathtt{e}})|_\star\mathtt{.this\$C}_1\mathtt{\$}\cdots\mathtt{\$C}_n \longrightarrow_{\mathrm{FJ}} |\mathtt{e}_0|_\star .$$

By definition,

$$encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{new\ C}(\overline{\mathtt{e}})) = encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e}_0\mathtt{.new\ C}_1\mathtt{.}\cdots\mathtt{.C}_n(\overline{\mathtt{d}})) = \mathtt{e}_0$$

finishing the case.

Case. $\mathtt{T} <:_{\mathrm{FJI}} \mathtt{U} \qquad \mathtt{U} <:_{\mathrm{FJI}} \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n$
We have four subcases depending on whether T is a simple name or not and whether U is a simple name or not. We show the case where $\mathtt{T} = \mathtt{C}$ and $\mathtt{U} = \mathtt{D}$ for some C and D; the other cases are similar.

Since $\Gamma \vdash_{\mathrm{FJI}} \mathtt{new\ C}(\overline{\mathtt{e}}) \in \mathtt{C}$, it is easy to show that $\Gamma \vdash_{\mathrm{FJI}} \mathtt{new\ D}(\overline{\mathtt{d}}) \in \mathtt{D}$ for some $\overline{\mathtt{d}}$ such that $\overline{\mathtt{e}} = \overline{\mathtt{d}}, \ldots$. By the induction hypothesis,

$$\left|\mathtt{new\ D}(\overline{\mathtt{d}})\right|_\star\mathtt{.this\$C}_1\mathtt{\$}\cdots\mathtt{\$C}_n \longrightarrow_{\mathrm{FJ}} \left|encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{new\ D}(\overline{\mathtt{d}}))\right|_\star .$$

By induction on the derivation of $\mathtt{C} <:_{\mathrm{FJI}} \mathtt{D}$, it is easy to show that

$$|\mathtt{new\ C}(\overline{\mathtt{e}})|_\star\mathtt{.this\$C}_1\mathtt{\$}\cdots\mathtt{\$C}_n \longrightarrow_{\mathrm{FJ}} \left|encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{new\ D}(\overline{\mathtt{d}}))\right|_\star$$

and that

$$encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{new\ D}(\overline{\mathtt{d}})) = encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{new\ C}(\overline{\mathtt{e}})),$$

finishing the subcase.
■

Proof (Proof of Theorem 4.1). We prove the theorem in three steps: first, we show $|CT|$ is well defined; second, it is shown that, if $\Gamma, \mathtt{this} : \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n, \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_i\ ^{i\in 1\ldots n} \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{T}$ with respect to CT, then $|\Gamma|, \mathtt{this} : |\mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n| \vdash_{\mathrm{FJ}} |\mathtt{e}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n} \in |\mathtt{T}|$ with respect to $|CT|$; and third, we show $|CT|$ is ok.

The first step is easy since each method body is well typed and so there are no such C.this that $\mathtt{C} \notin \mathtt{S}$ where S is the class to which the method belongs. Note that well-definedness of $|CT|$ implies well-definedness of the auxiliary functions.

The second step is proved by induction on the derivation of Γ, $\mathtt{this} : \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n$, $\mathtt{C}_i\mathtt{.this} : \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_i\ ^{i\in 1\ldots n} \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{T}$ with a case analysis on the last rule used. We show a few main cases.

Case Ti-Field. By the assumption on the rule used, we have

$$\begin{array}{l}
\mathtt{e} = \mathtt{e}_0\mathtt{.f}_i \\
\Gamma, \mathtt{this} : \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n, \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_i\ ^{i\in 1\ldots n} \vdash_{\mathrm{FJI}} \mathtt{e}_0 \in \mathtt{T}_0 \\
\mathit{fields}_{\mathrm{FJI}}(\mathtt{T}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\
\mathtt{T} = \mathtt{T}_i
\end{array}$$

By the induction hypothesis, $|\Gamma|, \mathtt{this} : |\mathtt{C}_1 . \cdots . \mathtt{C}_n| \vdash_{\mathrm{FJ}} |\mathtt{e}_0|_{\mathtt{C}_1 . \cdots . \mathtt{C}_n} \in |\mathtt{T}_0|$. By Lemma 4.1, $\mathit{fields}_{\mathrm{FJ}}(|\mathtt{T}_0|) = \ldots, |\mathtt{T}_i|\ \ \mathtt{f}_i, \ldots$. Then, the rule T-Field finishes the case.

Case Ti-NewInner. By the assumption on the rule used, we have

$$\begin{array}{l}
\mathtt{e} = \mathtt{e}_0\mathtt{.new\ T}_0\mathtt{.C(}\overline{\mathtt{e}}\mathtt{)} \\
\Gamma, \mathtt{this} : \mathtt{C}_1 . \cdots . \mathtt{C}_n, \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1 . \cdots . \mathtt{C}_i\ ^{i \in 1 \ldots n} \vdash_{\mathrm{FJI}} \mathtt{e}_0 \in \mathtt{S}_0 \\
\mathtt{S}_0 <:_{\mathrm{FJI}} \mathtt{T}_0 \\
\mathit{fields}_{\mathrm{FJI}}(\mathtt{T}_0\mathtt{.C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\
\Gamma, \mathtt{this} : \mathtt{C}_1 . \cdots . \mathtt{C}_n, \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1 . \cdots . \mathtt{C}_i\ ^{i \in 1 \ldots n} \vdash_{\mathrm{FJI}} \overline{\mathtt{e}} \in \overline{\mathtt{S}} \\
\overline{\mathtt{S}} <:_{\mathrm{FJI}} \overline{\mathtt{T}} \\
\mathtt{T} = \mathtt{T}_0\mathtt{.C}
\end{array}$$

We must show $|\Gamma|, \mathtt{this} : |\mathtt{C}_1 . \cdots . \mathtt{C}_n| \vdash_{\mathrm{FJ}} \mathtt{new}\ \ |\mathtt{T}_0\mathtt{.C}|\mathtt{(}\,|\overline{\mathtt{e}}|_{\mathtt{C}_1 . \cdots . \mathtt{C}_n}\mathtt{,}\ \ |\mathtt{e}_0|_{\mathtt{C}_1 . \cdots . \mathtt{C}_n}\mathtt{)} \in |\mathtt{T}_0\mathtt{.C}|$. By the induction hypothesis,

$$|\Gamma|, \mathtt{this} : |\mathtt{C}_1 . \cdots . \mathtt{C}_n| \vdash_{\mathrm{FJ}} |\mathtt{e}_0|_{\mathtt{C}_1 . \cdots . \mathtt{C}_n} \in |\mathtt{S}_0|$$

and

$$|\Gamma|, \mathtt{this} : |\mathtt{C}_1 . \cdots . \mathtt{C}_n| \vdash_{\mathrm{FJ}} |\overline{\mathtt{e}}|_{\mathtt{C}_1 . \cdots . \mathtt{C}_n} \in \left|\overline{\mathtt{S}}\right|.$$

By Lemma 4.1, $\mathit{fields}_{\mathrm{FJ}}(|\mathtt{T}_0\mathtt{.C}|) = \left|\overline{\mathtt{T}}\right|\ \ \overline{\mathtt{f}}', |\mathtt{T}_0|\ \ \mathtt{this\$}|\mathtt{T}_0\mathtt{.C}|$. Since $|\mathtt{S}_0| <:_{\mathrm{FJ}} |\mathtt{T}_0|$ and $\left|\overline{\mathtt{S}}\right| <:_{\mathrm{FJ}} \left|\overline{\mathtt{T}}\right|$ by Lemma 4.4, the rule T-New finishes the case.

Finally, the third step is proved by showing that `L OK IN P` implies $|\mathtt{P}|_\star\ \ \mathtt{OK}$; it is proved by the induction on the derivation of `L OK IN P` using the result of the second step. ■

Proof (Proof of Theorem 4.2). By induction on the derivation of $\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{e}'$ with a case analysis on the last rule used. We show only the cases for Ri-FieldT and Ri-InvkT since the other base cases are similar to either of them. The cases for congruence rules are straightforward.

Case Ri-FieldT. $\quad \mathtt{e} = \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).f}_i \qquad \mathtt{e}' = \mathtt{e}_i \qquad \mathit{fields}_{\mathrm{FJI}}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$
By Lemma 4.1, $\mathit{fields}_{\mathrm{FJ}}(|\mathtt{C}|) = \ldots, |\mathtt{T}_i|\ \ \mathtt{f}_i, \ldots$, and thus,

$$|\mathtt{e}|_\star = \mathtt{new\ C(}\,|\overline{\mathtt{e}}|_\star\mathtt{).f}_i \longrightarrow |\mathtt{e}_i|_\star\,.$$

Case Ri-InvokeT.

$$\begin{array}{l}
\mathtt{e} = \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).m(}\overline{\mathtt{d}}\mathtt{)} \\
\mathit{mbody}_{\mathrm{FJI}}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0, \mathtt{C}_1 . \cdots . \mathtt{C}_n) \\
\mathtt{c}_n = \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \\
\mathtt{c}_i = \mathit{encl}_{\mathtt{c}_{i+1}}(\mathtt{C}_1 . \cdots . \mathtt{C}_{i+1}) \quad (i \in 1 \ldots n-1) \\
\mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{c}_n/\mathtt{this}, \mathtt{c}_i/\mathtt{C}_i\mathtt{.this}\ ^{i \in 1 \ldots n}]\mathtt{e}_0.
\end{array}$$

By Lemma 4.3, $\mathit{mbody}_{\mathrm{FJ}}(\mathtt{m}, |\mathtt{C}|) = (\overline{\mathtt{x}}, |\mathtt{e}_0|_{\mathtt{C}_1 . \cdots . \mathtt{C}_n})$. Thus,

$$|\mathtt{e}|_\star = \mathtt{new\ C(}\,|\overline{\mathtt{e}}|_\star\mathtt{).m(}\left|\overline{\mathtt{d}}\right|_\star\mathtt{)} \longrightarrow [\left|\overline{\mathtt{d}}\right|_\star/\overline{\mathtt{x}}, \mathtt{new\ C(}\,|\overline{\mathtt{e}}|_\star\mathtt{)/this}]\,|\mathtt{e}_0|_{\mathtt{C}_1 . \cdots . \mathtt{C}_n}\,.$$

Since $|\mathtt{C}_i\mathtt{.this}|_{\mathtt{C}_1 . \cdots . \mathtt{C}_n} = \mathtt{this.this\$C}_1\mathtt{\$} \cdots \mathtt{\$C}_n . \cdots . \mathtt{this\$C}_1\mathtt{\$} \cdots \mathtt{\$C}_i$, by Lemma 4.5,

$$[\left|\overline{\mathtt{d}}\right|_\star/\overline{\mathtt{x}}, \mathtt{new\ C(}\,|\overline{\mathtt{e}}|_\star\mathtt{)/this}]\,|\mathtt{C}_i\mathtt{.this}|_{\mathtt{C}_1 . \cdots . \mathtt{C}_n} \longrightarrow_{\mathrm{FJ}}{}^* |\mathtt{c}_i|_\star$$

Then, by using congruence rules,

$$\begin{array}{ll} & [|\overline{\mathtt{d}}|_\star/\overline{\mathtt{x}}, \mathtt{new\ C(}|\overline{\mathtt{e}}|_\star\mathtt{)/this}]\,|\mathtt{e}_0|_{\mathtt{C}_1.\cdots.\mathtt{C}_n} \\ \longrightarrow_{\mathrm{FJ}}{}^* & |[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{c}_i/\mathtt{C}_i\mathtt{.this}\ {}^{i\in 1\ldots n}]\mathtt{e}_0|_\star \\ = & |\mathtt{e}'|_\star \end{array}$$

finishing the case.

■

Proof (Proof of Theorem 4.3). By induction on the derivation of $|\mathtt{e}|_\star \longrightarrow_{\mathrm{FJ}} \mathtt{e}'$ with a case analysis on the last rule used.

Case R-Field. $|\mathtt{e}|_\star = \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).f}_i$ $\quad$ $\mathit{fields}_{\mathrm{FJ}}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ $\quad$ $\mathtt{e}' = \mathtt{e}_i$
By inspecting compilation rules, e must be field access to a top-level object or an inner class object; moreover, e is well typed. By Theorem 3.2 and Lemma 4.1, we have $\mathtt{e}\longrightarrow_{\mathrm{FJI}}\mathtt{e}_i{}'$ where $|\mathtt{e}_i{}'|_\star = \mathtt{e}_i$, finishing the case.

Case R-Invk. $|\mathtt{e}|_\star = \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).m(}\overline{\mathtt{d}}\mathtt{)}$
$\mathit{mbody}_{\mathrm{FJ}}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$
$\mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)/this}]\mathtt{e}_0$
By inspecting compilation rules, e must be method invocation on a top-level object or inner class object constructor; moreover, e is well typed. By Theorem 3.2, we have $\mathtt{e}\longrightarrow_{\mathrm{FJI}}\mathtt{e}''$ by using Ri-InvokeI or Ri-InvokeT. In either case, by Theorem 4.2, we have $|\mathtt{e}|_\star \longrightarrow_{\mathrm{FJ}}{}^* |\mathtt{e}''|_\star$. Then, it is easy to check that

$$|\mathtt{e}|_\star \longrightarrow_{\mathrm{FJ}}\mathtt{e}'\longrightarrow_{\mathrm{FJ}}{}^* |\mathtt{e}''|_\star$$

by Lemma 4.3. (Refer to the case for Ri-InvokeT in the proof of Theorem 4.2.)

Case RC-New-Arg. $|\mathtt{e}|_\star = \mathtt{new\ C(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)}$ $\quad$ $\mathtt{e}_i\longrightarrow_{\mathrm{FJ}}\mathtt{e}_i{}'$
$\mathtt{e}' = \mathtt{new\ C(}\ldots\mathtt{,e}_i{}'\mathtt{,}\ldots\mathtt{)}$
By inspecting compilation rules, e must be an object constructor. We have three subcases according to the form of e.

Subcase. $\mathtt{e} = \mathtt{d}_0\mathtt{.new\ T}_0\mathtt{.C}_0\mathtt{(}\ldots\mathtt{,d}_i\mathtt{,}\ldots\mathtt{)}$
$|\mathtt{d}_j|_\star = \mathtt{e}_j \quad (j \in 1\ldots n-1)$ $\qquad$ $|\mathtt{d}_0|_\star = \mathtt{e}_n$
By the induction hypothesis, $\mathtt{d}_i \longrightarrow_{\mathrm{FJI}} \mathtt{d}_i{}'$ for some $\mathtt{d}_i{}'$. By RCi-Inner-Arg,

$$\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{d}_0\mathtt{.new\ T}_0\mathtt{.C}_0\mathtt{(}\ldots\mathtt{,d}_i{}'\mathtt{,}\ldots\mathtt{)}.$$

The other subcases (where e is a top-level class constructor and where e is an inner class constructor and $\mathtt{e}_i$ is the last argument) are similar.

Case RC-Field, RC-Invk-Recv, RC-Invk-Arg. Easy.

■

5. ELABORATION

In this section we formalize the elaboration of external language programs (i.e., programs in the language the user actually sees) to FJI. In user programs, the receivers of field access or method invocation, the enclosing instances of inner class instantiation, and the qualifications of type names may be omitted. For example, a simple name C means an inner class T.C when it is used in the direct enclosing class T. A basic job of elaboration is to find where a name f, m, or C is bound and to recover its receiver information or "absolute path" form.

In the conventional scoping rules of simple block structured languages, simple names are bound to their syntactically nearest declaration. In Java, however, they can be bound to declarations in superclasses, or even in superclasses of enclosing classes. For example, in the class below, f in the method m is bound to the field f of the enclosing class C *unless* D has a field f.

```
class C extends Object {
  Object f; ...
  class D extends Object { ...
    Object m () { return f; }
  }
}
```

Similarly, f in the method m is bound to the field f of its superclass B[4] in the following classes.

```
class B extends Object { Object f; ... }
class C extends Object { ...
  class D extends B { ...
    Object m () { return f; }
  }
}
```

In general, beginning with the class where a field/method name is used, the search algorithm looks for the definition in superclasses; if there is no definition in any superclass, it looks in the direct enclosing class and its superclasses, and then in the second direct enclosing class and its super classes, and so on. Once the declaration where a name is bound is known, it is easy to construct the appropriate qualification. In the examples above, f becomes C.this.f and D.this.f, respectively.

Simple type names obey similar elaboration rules. For example, D occurring in C is elaborated to C.D. However, unlike field names and method names, pre-elaboration type names themselves can be qualified. In such a case the head name is elaborated first, then the definitions of the following names, in a manner similar to field lookup. For example, consider the following class declarations:

```
class A extends Object { ...
  class B extends Object { ... }
}
class C extends Object { ...
  class D extends A { ... }
}
class E extends C { D.B f; ... }
```

[4] even when C has a field f—this rule, following the new specification [11], is different from the old specification [14]; also see Section 6.

The type D.B of f is elaborated to A.B as follows:

1. The first name D is elaborated to C.D.

2. We check whether C.D.B makes sense; in this case, it does not, since the inner class D does not have a declaration of B. The elaborator replaces C.D with its superclass A and elaborates A.B in the context of C.

3. Since A is not declared in C, it denotes the top-level class A.

4. Finally, since B is declared in the top-level class A, A.B is the elaborated type for D.B in the context of E.

Last, we describe how a constructor invocation new $\mathtt{T}(\overline{\mathtt{e}})$ is elaborated. This is slightly more involved than the other elaboration steps, since it requires both elaborating the type and recovering an enclosing instance (when it turns out to be instantiation of an inner class). First of all, the pre-elaboration type name T is elaborated to $\mathtt{T}'$. If $\mathtt{T}'$ is a simple name C, then the constructor invocation does not need an enclosing instance. On the other hand, if $\mathtt{T}'$ is U.C, then we have to make up an enclosing instance D.this, whose type is subtype of U, by checking which enclosing class is a subclass of U. Finally, among such enclosing classes, the innermost one is chosen and new $\mathtt{T}(\overline{\mathtt{e}})$ is elaborated to D.this.new U.C(...). The annotation U has to be recovered now to specify which inner class is instantiated, since there might be more than one inner class C defined in classes between D and U. Consider the following classes and the expression new A.B() inside the class D.E:

```
class A extends Object { ...
  class B extends Object { ... }
}
class C extends A { ...
  class B extends Object { ... }
}
class D extends C { ...
  class E extends C { ...
    Object m () { ... new A.B() ...}
  }
}
```

First, A.B is elaborated to itself. Now, we need to find out which enclosing class (including the current class) is a subclass of A. In this case, both D and D.E are; then, the innermost one, D.E, is chosen, and new A.B() is elaborated to E.this.new A.B(). The qualified name A.B is important since we have to remember that the class A.B is to be instantiated (not C.B).

In the rest of this section, we give the formal rules of elaboration. We use the metavariables X, Y, and Z for pre-elaboration type names, which are non-empty strings obtained by concatenating simple names by ".". The notation P.C always denotes an (elaborated) type.

5.1. Syntax of External Language

In external language programs, pre-elaboration names X are used where types are required; it is allowed to write a field access f and a method invocation $\mathtt{m}(\overline{\mathtt{e}})$ without a receiver, and constructor invocation new $\mathtt{X}(\overline{\mathtt{e}})$ of pre-elaboration type name without an enclosing instance. (In Java, when enclosing instance is explicit the class name must be simple.) We assume only the sanity conditions (1)–(3) from in Section 3; (4) will be automatically satisfied when elaboration of types succeeds (see Theorem 5.1 (1)) and (5)–(7) can be checked after elaboration.

$$
\begin{array}{lcll}
\mathtt{L} & ::= & \mathtt{class\ C\ extends\ X\ \{\overline{X}\ \overline{f};\ K\ \overline{L}\ \overline{M}\}} & \\
\mathtt{K} & ::= & \mathtt{C(\overline{X}\ \overline{f})\ \{\ super(\overline{f});\ this.\overline{f}\ =\ \overline{f};\ \}} & \\
 & | & \mathtt{C(\overline{X}\ \overline{f})\ \{\ f.super(\overline{f});\ this.\overline{f}\ =\ \overline{f};\ \}} & \\
\mathtt{M} & ::= & \mathtt{X\ m\ (\overline{X}\ \overline{x})\{\ return\ e;\ \}} & \text{method declarations} \\
\mathtt{e} & ::= & \mathtt{x} & \text{variable or field access} \\
 & | & \mathtt{e.f} & \text{field access} \\
 & | & \mathtt{m(\overline{e})} & \text{method invocation} \\
 & | & \mathtt{e.m(\overline{e})} & \text{method invocation} \\
 & | & \mathtt{new\ X(\overline{e})} & \text{constructor invocation} \\
 & | & \mathtt{e.new\ C(\overline{e})} & \text{inner class constructor}
\end{array}
$$

We do not deal with omitted qualifications of `super` constructor invocations, permitted in Java: an explicit qualification using a constructor argument is needed.

5.2. Elaboration Rules

Elaboration is performed in two steps: (1) elaboration of types (except the ones that occur in method bodies); and (2) elaboration of expressions. Step (2) must be performed after step (1) since elaboration of expressions require type names (in particular `X` after `extends`) to be elaborated.

5.2.1. Elaboration of Types

The elaboration relation for types is written $\mathtt{P} \vdash \mathtt{X} \Rightarrow \mathtt{T}$, read "`X` is elaborated to `T` in `P`." The elaboration rules are given in Figure 14. We write $\mathtt{P} \vdash \mathtt{X} \Uparrow$, which means there is no `T` such that $\mathtt{P} \vdash \mathtt{X} \Rightarrow \mathtt{T}$. The key rules are ET-SimpEncl and ET-SimpSup. The rule ET-SimpEncl is used when the ambiguous name `D` is defined in neither the current class `P.C` nor its superclasses ($\mathtt{P} \vdash \mathtt{X.D} \Uparrow$); `D` is resolved in the context of the direct enclosing class `P`. On the other hand, the rule ET-SimpSup is used when the ambiguous name `D` is defined in a superclass ($\mathtt{P} \vdash \mathtt{X.D} \Rightarrow \mathtt{T}$).

Remark:. A straightforward elaboration algorithm obtained by reading the rules in a bottom-up manner might diverge. For example, consider the following class declaration.

```
class A extends A.B {
  A () { super(); }
}
```

Since there is no class `A.B`, elaboration of `A.B` must fail. However, using ET-LongSup, it tries to find `T` such that $\star \vdash \mathtt{A.B.B} \Rightarrow \mathtt{T}$ since `A` does not have `B` and `A.B` is specified as a superclass of `A`; it then tries to find $\mathtt{T}'$ such that $\star \vdash \mathtt{A.B.B.B} \Rightarrow \mathtt{T}'$ and so on. To prevent divergence, an elaboration algorithm should detect circularity by

$$\texttt{P} \vdash \texttt{Object} \Rightarrow \texttt{Object} \qquad \text{(ET-Object)}$$

$$\frac{\texttt{P.C} \in dom(CT)}{\texttt{P} \vdash \texttt{C} \Rightarrow \texttt{P.C}} \qquad \text{(ET-InCT)}$$

$$\frac{\begin{array}{c}\texttt{P.C.D} \notin dom(CT) \qquad \texttt{P} \vdash \texttt{D} \Rightarrow \texttt{T} \\ CT(\texttt{P.C}) = \texttt{class C extends X \{...\}} \qquad \texttt{P} \vdash \texttt{X.D} \Uparrow\end{array}}{\texttt{P.C} \vdash \texttt{D} \Rightarrow \texttt{T}} \qquad \text{(ET-SimpEncl)}$$

$$\frac{\begin{array}{c}\texttt{P.C.D} \notin dom(CT) \\ CT(\texttt{P.C}) = \texttt{class C extends X \{...\}} \qquad \texttt{P} \vdash \texttt{X.D} \Rightarrow \texttt{T}\end{array}}{\texttt{P.C} \vdash \texttt{D} \Rightarrow \texttt{T}} \qquad \text{(ET-SimpSup)}$$

$$\frac{\texttt{P} \vdash \texttt{X} \Rightarrow \texttt{T} \qquad \texttt{T.C} \in dom(CT)}{\texttt{P} \vdash \texttt{X.C} \Rightarrow \texttt{T.C}} \qquad \text{(ET-Long)}$$

$$\frac{\begin{array}{c}\texttt{P} \vdash \texttt{X} \Rightarrow \texttt{P}'\texttt{.D} \qquad \texttt{P}'\texttt{.D.C} \notin dom(CT) \\ CT(\texttt{P}'\texttt{.D}) = \texttt{class D extends Y \{...\}} \qquad \texttt{P}' \vdash \texttt{Y.C} \Rightarrow \texttt{U}\end{array}}{\texttt{P} \vdash \texttt{X.C} \Rightarrow \texttt{U}} \qquad \text{(ET-LongSup)}$$

FIG. 14. Elaboration of types

keeping previous inputs for recursive calls: in this example, the algorithm will try to find $\mathtt{T}$ such that $\star \vdash \mathtt{A.B} \Rightarrow \mathtt{T}$ twice.

5.2.2. Elaboration of Expressions, Methods, and Classes

After elaboration of types, we can check all the sanity conditions except (5). Then, elaboration can proceed to the next step—that is, elaboration of expressions, methods, and classes.

We need auxiliary functions to lookup a field/method definition in enclosing classes and their superclasses. The functions *field-encl*(f, T) and *meth-encl*(m, T) defined below return the simple name of the enclosing class that has (or inherits) the declaration of the simple name f or m, when it is mentioned in T. The function *subty-encl*(U, $\mathtt{C}_1.\cdots.\mathtt{C}_n$), used in elaboration of constructor invocations, returns the simple name $\mathtt{C}_i$ of the innermost enclosing class such that $\mathtt{C}_1.\cdots.\mathtt{C}_i$ <: U.

$$\frac{\mathit{fields}(\mathtt{P.C}) = \ldots, \mathtt{S\ f}, \ldots}{\mathit{field\text{-}encl}(\mathtt{f}, \mathtt{P.C}) = \mathtt{C}} \qquad \frac{\mathtt{f}\text{ does not appear in }\mathit{fields}(\mathtt{T.C})}{\mathit{field\text{-}encl}(\mathtt{f}, \mathtt{T.C}) = \mathit{field\text{-}encl}(\mathtt{f}, \mathtt{T})}$$

$$\frac{\mathit{mtype}(\mathtt{m}, \mathtt{P.C}) = \overline{\mathtt{S}} \to \mathtt{S}_0}{\mathit{meth\text{-}encl}(\mathtt{m}, \mathtt{P.C}) = \mathtt{C}} \qquad \frac{\mathit{mtype}(\mathtt{m}, \mathtt{T.C})\text{ is undefined}}{\mathit{meth\text{-}encl}(\mathtt{m}, \mathtt{T.C}) = \mathit{meth\text{-}encl}(\mathtt{m}, \mathtt{T})}$$

$$\frac{\mathtt{P.C} <: \mathtt{U}}{\mathit{subty\text{-}encl}(\mathtt{U}, \mathtt{P.C}) = \mathtt{C}} \qquad \frac{\mathtt{T.C} \not<: \mathtt{U}}{\mathit{subty\text{-}encl}(\mathtt{U}, \mathtt{T.C}) = \mathit{subty\text{-}encl}(\mathtt{U}, \mathtt{T})}$$

The elaboration relation for expressions (method bodies) $\mathtt{T}; \overline{\mathtt{x}} \vdash \mathtt{e} \Rightarrow \mathtt{e}'$ is read "e is elaborated to e′ in the class T when $\overline{\mathtt{x}}$ are formal arguments of the method." The elaboration rules are given in Figure 15. Thanks to the auxiliary functions, most rules are straightforward. Note that the elaborated expression is not an FJI expression yet, since the static types of the enclosing instances in inner class constructors $\mathtt{e}_0\mathtt{.new\ C}(\overline{\mathtt{e}})$ are still omitted; they are recovered during typechecking.

Elaboration of methods, written $\mathtt{T} \vdash \mathtt{M} \Rightarrow \mathtt{M}'$ and read "method M in class T is elaborated to M′, just replaces the method body since the return type and argument types are already elaborated. Elaboration of classes, written $\mathtt{P} \vdash \mathtt{L} \Rightarrow \mathtt{L}'$ read "class L declared in P is elaborated to L′, is also straightforward; methods $\overline{\mathtt{M}}$ and inner classes $\overline{\mathtt{L}}$ are elaborated, recursively. The formal elaboration rules also appear in Figure 15.

5.3. Properties of Elaboration

A minimal requirement for elaboration is that the guessed information is reasonable in the sense that elaborated types are really defined in the class table and recovered receivers `this` or `C.this` really offer the field or the method to be used. This property looks rather weak but, actually, it is all we can expect: elaboration just provides programmers handy abbreviations to make their programs more concise and does not have very deep "semantic" significance. In this sense, the definition itself is the only interesting part.

THEOREM 5.1.

1.*If* $\mathtt{P} \vdash \mathtt{X} \Rightarrow \mathtt{T}$, *then* $\mathtt{T} \in \mathit{dom}(CT)$.

2.*If* $\mathtt{T}; \overline{\mathtt{x}} \vdash \mathtt{f} \Rightarrow \mathtt{C.this.f}$, *then* $\mathtt{T} = \mathtt{P.C.S}$ *for some* P *and* S *with* $\mathit{fields}(\mathtt{P.C}) = \ldots, \mathtt{U\ f}, \ldots$ *for some* U.

Elaboration of expressions:

$$\frac{\mathtt{x} \in \overline{\mathtt{x}}}{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{x} \;\Rightarrow\; \mathtt{x}} \qquad \text{(E-Var)}$$

$$\frac{\mathtt{f} \notin \overline{\mathtt{x}} \qquad \textit{field-encl}(\mathtt{f},\mathtt{T}) = \mathtt{C}}{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{f} \;\Rightarrow\; \mathtt{C.this.f}} \qquad \text{(E-FieldSimp)}$$

$$\frac{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{e}_0 \;\Rightarrow\; \mathtt{e}_0{'}}{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{e}_0\mathtt{.f} \;\Rightarrow\; \mathtt{e}_0{'}\mathtt{.f}} \qquad \text{(E-Field)}$$

$$\frac{\textit{meth-encl}(\mathtt{m},\mathtt{T}) = \mathtt{C} \qquad \mathtt{T};\overline{\mathtt{x}} \vdash \overline{\mathtt{e}} \;\Rightarrow\; \overline{\mathtt{e}}'}{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{m}(\overline{\mathtt{e}}) \;\Rightarrow\; \mathtt{C.this.m}(\overline{\mathtt{e}}')} \qquad \text{(E-InvkSimp)}$$

$$\frac{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{e}_0 \;\Rightarrow\; \mathtt{e}_0{'} \qquad \mathtt{T};\overline{\mathtt{x}} \vdash \overline{\mathtt{e}} \;\Rightarrow\; \overline{\mathtt{e}}'}{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}}) \;\Rightarrow\; \mathtt{e}_0{'}\mathtt{.m}(\overline{\mathtt{e}}')} \qquad \text{(E-Invk)}$$

$$\frac{\mathtt{T} \vdash \mathtt{X} \;\Rightarrow\; \mathtt{C} \qquad \mathtt{T};\overline{\mathtt{x}} \vdash \overline{\mathtt{e}} \;\Rightarrow\; \overline{\mathtt{e}}'}{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{new\ X}(\overline{\mathtt{e}}) \;\Rightarrow\; \mathtt{new\ C}(\overline{\mathtt{e}}')} \qquad \text{(E-NewTop)}$$

$$\frac{\mathtt{T} \vdash \mathtt{X} \;\Rightarrow\; \mathtt{U.D} \qquad \textit{subty-encl}(\mathtt{U},\mathtt{T}) = \mathtt{C} \qquad \mathtt{T};\overline{\mathtt{x}} \vdash \overline{\mathtt{e}} \;\Rightarrow\; \overline{\mathtt{e}}'}{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{new\ X}(\overline{\mathtt{e}}) \;\Rightarrow\; \mathtt{C.this.new\ U.D}(\overline{\mathtt{e}}')} \qquad \text{(E-NewInner)}$$

$$\frac{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{e}_0 \;\Rightarrow\; \mathtt{e}_0{'} \qquad \mathtt{T};\overline{\mathtt{x}} \vdash \overline{\mathtt{e}} \;\Rightarrow\; \overline{\mathtt{e}}'}{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{e}_0\mathtt{.new\ C}(\overline{\mathtt{e}}) \;\Rightarrow\; \mathtt{e}_0{'}\mathtt{.new\ C}(\overline{\mathtt{e}}')} \qquad \text{(E-New)}$$

$$\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{this} \;\Rightarrow\; \mathtt{this} \qquad \text{(E-This)}$$

$$\frac{\mathtt{C} \in \mathtt{T}}{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{C.this} \;\Rightarrow\; \mathtt{C.this}} \qquad \text{(E-QlThis)}$$

Elaboration of methods:

$$\frac{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{e} \;\Rightarrow\; \mathtt{e}'}{\begin{array}{l}\mathtt{T} \vdash \mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}\\ \quad \Rightarrow \mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e';}\ \}\end{array}} \qquad \text{(E-Method)}$$

Elaboration of classes:

$$\frac{\mathtt{P.C} \vdash \overline{\mathtt{L}} \;\Rightarrow\; \overline{\mathtt{L}}' \qquad \mathtt{P.C} \vdash \overline{\mathtt{M}} \;\Rightarrow\; \overline{\mathtt{M}}'}{\begin{array}{l}\mathtt{P} \vdash \mathtt{class\ C\ extends\ T}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{L}}\ \overline{\mathtt{M}}\ \}\\ \quad \Rightarrow \mathtt{class\ C\ extends\ T}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{L}}'\ \overline{\mathtt{M}}'\}\end{array}} \qquad \text{(E-Class)}$$

FIG. 15. Elaboration of expressions, methods, and classes

3.*If* $\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{m}(\overline{\mathtt{e}}) \Rightarrow \mathtt{C.this.m}(\overline{\mathtt{e}}')$, *then* $\mathtt{T} = \mathtt{P.C.S}$ *for some* P *and* S *with* *mtype*(P.C) *being well defined.*

4.*If* $\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{new\ X}(\overline{\mathtt{e}}) \Rightarrow \mathtt{C.this.new\ U.D}(\overline{\mathtt{e}}')$, *then* $\mathtt{T} = \mathtt{P.C.S}$ *for some* S *with* $\mathtt{U.D} \in \mathit{dom}(\mathit{CT})$ *and* $\mathtt{P.C} <: \mathtt{U}$.

Proof. *Each clause is proved by induction of the derivation of the condition.* ■

6. FROM THE OLD INNER CLASSES SPECIFICATION TO THE SECOND EDITION OF THE JAVA LANGUAGE SPECIFICATION

The formalization we have presented in this article is based on the new language specification [11] (hereafter called JLS2). In this section, we note two substantial changes from the original specification [14] (hereafter called ICS) and show some experiments with several versions of Sun's JDK compiler. In the course of the experiment, we have found several bugs and inconsistencies in several versions of the compiler; among them, we show the most interesting ones related to the changes, as most of them are trivial and already known to the developers. We think, without our effort of formalization, it would have been hard to pinpoint those bugs and inconsistencies. In fact, bugs were often found when we tried to supplement a vacuum in the documentation by observing the behavior of the compiler.

6.1. Name Conflicts in Elaboration

The first interesting change is in the elaboration process. Although the scoping rules described in ICS are basically the same, there is one additional rule concerning interaction between names inherited from a superclass and ones declared in enclosing classes.

Suppose a field called f is referred to in the class $\mathtt{C}_1.\cdots.\mathtt{C}_n$. The elaboration looks for the declaration of f in the class $\mathtt{C}_1.\cdots.\mathtt{C}_n$ and its superclasses, followed by the direct enclosing class $\mathtt{C}_1.\cdots.\mathtt{C}_{n-1}$ and its superclasses, and so on. Now, suppose the algorithm above finds the definition of the field/method in one of the (proper) superclasses of the class $\mathtt{C}_1.\cdots.\mathtt{C}_i$. Then, a field of the same name must not be defined in any of its enclosing classes $\mathtt{C}_1$, $\mathtt{C}_1.\mathtt{C}_2$, ..., $\mathtt{C}_1.\cdots.\mathtt{C}_{i-1}$. Similarly for methods and types (i.e., member classes). For example, the Java program below

```
class B { Object f; }
class C { Object f;
  class D extends B {
    Object m () { return f; }
  }
}
```

should be rejected according to ICS. For example, JDK compiler version 1.1.7 yields the following compiler error:

```
test.java:7: Method 'm' is inherited in inner class C. D, and hides
a method of the same name in class C. An explicit 'this' qualifier must
be used to select the desired instance.
        m();
         ^
```

The user must write `C.this.f` or `D.this.f`, specifying the enclosing instance explicitly.

Although this ambiguity rule has been abandoned in JLS2, it is still present in recent implementations of JDK compilers (versions 1.2.2 and 1.3beta2 for linux). Moreover, old implementations (before 1.2) that assumed the ambiguity rule seem to have one exception, which is not mentioned in ICS: it is *not* considered ambiguous if the definition found in a superclass is *also* the syntactically nearest definition in enclosing classes. This situation occurs when an inner class extends one of its enclosing classes. For example, suppose `E` does not declare the field `f` in the class definition below.

```
class C {
  Object f; ...
  class D { ...
    class E extends C { ...
      Object m () { return f; }
}}}
```

The reference to `f` in `m` is not conflicting unless `D` declares a field `f`.[5] (The algorithm finds the definition `f` declared in a *superclass* of `E`.)

6.2. Semantics of Qualified `this`

The second interesting change is the semantics of qualified `this` expressions. As we discussed, in the class $\mathtt{C}_1.\cdots.\mathtt{C}_n$, the expression $\mathtt{C}_i$`.this` denotes the $(n-i)$th direct enclosing instance. In ICS, however, it was not made clear what happens if $\mathtt{C}_i$ is also a superclass of $\mathtt{C}_n$. The result was some significant variations in the interpretation of qualified `this` expressions. Consider the following Java program:

```
class C {
  void who () {
    System.out.println("I'm a C object");
  }

  class D extends C {
    void m () { C.this.who(); }
    void who () {
      System.out.println("I'm a C.D object");
    }
  }

  public static void main (String[] args) {
    new C().new D().m();
  }
}
```

Surprisingly, this program prints out `I'm a C.D object` when compiled with JDK 1.1.7, but `I'm a C object` under JDK 1.2.2. In the old JDK, the meaning of

[5]It looks like a bug of the old compilers that this exceptional rule does not apply to resolution of method names; even if two conflicting method definitions are in fact the same, a compiler error is yielded.

`C.this` is exactly the same as `D.this` or `this` when `C` is a superclass of the inner class `C.D`; thus, `C.this` is bound to the receiver `new C().new D ()`. In JDK 1.2.2 (and later), on the other hand, `C.this` is always bound to the enclosing object of the receiver regardless of superclass. This point has been clarified in favor of the latter choice (i.e., $\mathtt{C.this} \neq \mathtt{D.this}$) in JLS2 (Section 15.8.4).

7. RELATED WORK

Nested classes in object-oriented languages. As we mentioned in the introduction, the idea of nested classes dates back to Simula [3]. In Simula, an object can be thought of as a procedure's activation record that can remain alive after the execution; a class is just a generator for such an activation record. Thus, like ordinary procedures, class declarations can be arbitrarily nested.

Compared to Java, however, the allowed forms of inheritance in Simula are rather restricted: a class whose qualified name would be `P.C` can extend another class $\mathtt{P}'\mathtt{.D}$ if and only if either (1) both `P.C` and $\mathtt{P}'\mathtt{.D}$ are top-level classes (i.e., $\mathtt{P} = \mathtt{P}' = \star$) or (2) $\mathtt{P} <: \mathtt{P}'$. This rule implies that any two classes in a subtyping relation are defined at the same depth of the nested program structure. For example, if $\mathtt{C}_1.\cdots.\mathtt{C}_n <: \mathtt{D}_1.\cdots.\mathtt{D}_m$, then it must be the case that $n = m$ and $\mathtt{C}_i <: \mathtt{D}_i$ for all i. This restriction seems to stem from Simula's implementation scheme for objects. According to the analogy between objects and activation records, an object is given just one static link to the activation record (we could call it an enclosing instance) of the enclosing block in which the class is defined. Thus, the direct enclosing instance must be shared among the methods, resulting in the above restriction; on the other hand, this scheme makes the notation of qualified `super`, found in Java, unnecessary.

Beta [17], a successor of Simula, also allows nested class definitions (as an instance of nested *patterns*, the only abstraction mechanism in Beta, unifying classes and procedures). Their basic behavior is the same as Java's inner classes in the sense that each method can have its own environment derived from the enclosing classes of the method definition. There are, however, two significant differences from Java's inner classes.

First, an enclosing instance of an inner class is a *part* of the type generated by the inner class. Suppose the objects `a1` and `a2`, instances of the class `A`, which has an inner class `B`. In Beta, `a1` and `a2` are considered to have their own *distinct* inner class of the name `B`; thus, the expressions `a1.B` and `a2.B`, by themselves, form distinct classes (or types)—they can be considered a kind of dependent types. Moreover, Beta dispenses with qualified `super`: superclasses are specified by expressions like `a1.B` rather than `A.B`, making `a1` as the enclosing instance that methods in `B` access.

Second, inner classes in Beta can be declared *virtual* [16]. Usually, if a subclass declares an inner class of the same name as its superclass' inner class, it just *hides* the declaration in the super class, just like shadowing of field declarations in Java. On the other hand, if the superclass `C` declares an inner class `E` virtual, `E` can be extended and *overridden* in the subclass `D` of `C`. Then, a reference to `E` is virtual in the sense that it depends on the *run-time type* of the enclosing instance which constructor is invoked. A constructor invocation $\mathtt{new\ e_0.E(\overline{e})}$ (remember $\mathtt{e_0.E}$ itself forms a class in Beta) instantiates an object using `E` inside `C` when the run-time type of $\mathtt{e}_0$ is `C` while it instantiates an object of class `E` inside `D` when that of $\mathtt{e}_0$ is `D`.

Madsen has recently described the algorithm for elaboration (he calls it "semantic analysis") used in the Mjølner Beta compiler [15]. The algorithm is very close to the rules presented in Section 5, in the sense that the search order is the same as ours, although the presence of virtual classes significantly complicates the algorithm in Beta.

Nested class declarations are also permitted in C++ [21], though their use is rather restricted: an inside class cannot access members of the enclosing classes and, thus, there is no notion corresponding to enclosing instances. Rather than inner classes discussed so far, they would correspond to Java's static nested classes. Such restricted nested classes are used mainly for hierarchical organization of classes: nested class declarations realize a non-flat name space (qualified names can be used for classes) and, combined with access annotations (`public`, `private`, and so on), they provide better control on the name space.

Bruce, Odersky, and Wadler [5] used nested classes as a device to group mutually dependent classes together and enforce on programmers the subclassing protocol that mutually dependent classes should be simultaneously extended. For example, if a programmer think that mutually dependent classes `C` and `D` should simultaneously extended, he or she declares them as members of a (top-level) class `A`; the classes `C` and `D` can be extended only by extending `A` and declaring subclasses of both `C` and `D` inside the subclass of `A`. Their use of nested classes is also for better organization of classes, rather than for obtaining access to enclosing classes.

Specification of inner classes. In the original Inner Classes Specification [14], the semantics of inner classes is given as a translation from inner classes to top-level classes. The document also explains how inner classes affect other language features, such as synchronization, access restriction and binary compatibility. However, the description is rather informal and sometimes vague, yielding different implementations with different semantics, as we described in the previous section. The updated language specification [11] clarifies and amends some flaws in the old one, but is still informal.

Object closure conversion. Recently, Glew [8] has studied closure conversion in the context of a call-by-value object calculus (without classes) and showed correctness of conversion based on contextual equivalence. Our translation semantics can also be viewed as closure conversion of class definitions. Since his calculus does not have classes, no semantic account of interaction between inheritance and nested classes is given.

Microsoft's delegates. Microsoft has proposed *delegates* [18] as an alternative to inner classes. The basic idea of delegates resembles the function pointers found in C and C++. Programmers can create a delegate with an expression of the form `e.m` (without parameters) and pass it elsewhere; later, the method `m` can be invoked through the delegate. We believe it would be possible to model delegates in an extension of FJ, as we have done here for inner classes. On one hand, the formalization would be simpler than inner classes due to the absence of interaction with inheritance. On the other hand, it seems difficult to model the implementation scheme of delegates as a translation into a language like pure Java.

Other core calculi for Java. Several calculi [6, 22, 19, 7] have been proposed as foundations for studying formal properties and extensions of Java; none of them treat inner classes, but we do not see any inherent difficulty with integrating inner classes into these calculi, following the lines of the account given here.

8. CONCLUSIONS AND FUTURE WORK

We have formalized two styles of semantics for inner classes: a direct style and a translation style, where semantics is given by compilation to a low-level language without inner classes, following Java's Inner Classes Specification. We have proved that the two styles correspond, in the sense that the translation commutes with the high-level reduction relation in the direct semantics.

Although our results have shown that inner classes in the style of Java are reasonable (in the sense of type soundness and correctness of compilation), we feel that one particular design decision in Java made the semantics disproportionately complicated: namely, allowing a subclass of an inner class to be defined in another unrelated class (and introducing the corresponding qualified `super` constructor invocation). As we have seen, this trick of qualified `super` is needed for methods of one object to be executed under various environments (constructed by enclosing instances). However, we can abuse it so that the same notation `C.this` in one scope denotes different enclosing instances. For example, consider the inner classes `A.B` and `A.C` below

```
class A {
  Object f;
  class B { ... A.this.f ...}
  class C extends B {
    C(A a) { a.super(); }
    ... A.this.f ...
}
```

and an instance of `A.C`. Now, an instance of `C` is of the form `a1.new C(a2)`; `a1` is bound to `A.this` in `C` and `a2` to `A.this` in `B`. The occurrences of `A.this` in `B` and `C` may or may not be the same object![6] It might be reasonable to leave out the prefixed `super` completely from the language and to force the rule that a subclass of an inner class must be defined in the same scope (which may include subclasses of the enclosing class of the superclass). Then, the semantics of FJI, in particular the definition of *encl*, would be much simpler because only prefixes of `new` (rather than arguments) can be bound to `C.this`. (In Java, a subclass may be defined in a package different from the one of the superclass; it looks like a very similar situation in the sense that the same name may denote different things. However, there is one significant difference that those references are resolved *statically* while qualified `this` is inherently dynamic.)

[6] As we have mentioned, in Java proper, the invocation `a.super()` can be omitted; in this case the prefix `a` of the constructor invocation `a.new C()` is automatically bound to `A.this` in both `B` and `C` (and shared by methods of `B` and `C`.)

In the course of this work, we first formalized the old specification [14] as found in the older versions of this article, retargeted to the new language specification [11], and tried to make our formalization as close to the new one as possible. Besides deepening our own understanding of inner classes, this work has uncovered a significant underspecification in the official specification and several bugs in the compiler implementation (most of these are known to the developers). Despite of all the effort we made, it is possible that our formalization might have some discrepancies. One lesson we have learned is that it is quite challenging to formalize an informal idea, especially when they are *not* the ones developing it. In particular, when the documentation the developers provide is silent about a certain aspect, all we could do was to observe the behavior of an artifact realizing the idea (i.e., compiler). Then, it is often hard to obtain intuitions behind the behavior.

For future work, the interaction between inner classes and access restrictions in Java is clearly worth investigating formally. We would also hope to be able to model, in a variant of FJI, Java's other forms of inner classes: anonymous classes and local classes in method bodies. On one hand, we have to deal with the complexity due to the fact that method arguments (not just fields) can occur in them as free variables. On the other hand, they are simpler than member classes with respect to interaction with inheritance. First of all, anonymous classes are inherently final and there can be no subclass of an anonymous class. Concerning local classes, a direct subclass of a local class is always defined in the scope where the local class is declared. It implies that all the enclosing instances accessible from the local class are also enclosing instances of the subclass. Thus, unlike inheritance involving member classes, all the methods from one class can be executed under one environment consisting of the enclosing instances of that class.

ACKNOWLEDGMENT

This work was supported by the University of Pennsylvania and the National Science Foundation under grant CCR-9701826, *Principled Foundations for Programming with Objects*. We would like to thank Robert Harper for his valuable comments and the bug parade in Java Developer Connection (`http://developer.java.sun.com/developer/bugParade/index.html`) for providing useful information. Comments from the anonymous referees of FOOL7, ECOOP2000, and Information and Computation helped us improve the final presentation.

REFERENCES

1. Martín Abadi. Protection in programming-language translations. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 868–883, Aalborg, Denmark, July 1998. Springer-Verlag. also appeared as DEC SRC Research Report 154 (April 1998).
2. Anasua Bhowmik and William Pugh. A secure implementation of Java inner classes. Handout from PLDI '99 Poster Session. Available through `http://www.cs.umd.edu/~pugh/java`.
3. Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Studentlitteratur, Lund, Sweden, 1973.
4. 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, *ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.
5. Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In Eric Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented*

Programming, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549, Brussels, Belgium, July 1998. Springer-Verlag.

6. 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.
7. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 171–183, San Diego, CA, January 1998. ACM.
8. Neal Glew. Object closure conversion. In Andrew Gordon and Andrew Pitts, editors, *Proceedings of the 3rd International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'99)*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, France, September 1999. Elsevier. Available through `http://www.elsevier.nl/locate/entcs/volume26.html`.
9. Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.
10. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, Reading, MA, first edition, August 1996.
11. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, Reading, MA, second edition, June 2000.
12. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Linda M. Northrop, editor, *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOSPLA'99)*, ACM SIGPLAN Notices, volume 34, number 10, pages 132–146, Denver, CO, October 1999. ACM Press.
13. JavaSoft. Java API documentation. Available through `http://java.sun.com/docs/index.html`.
14. JavaSoft. Inner classes specification, February 1997. Available through `http://java.sun.com/products/JDK/1.1/`.
15. Ole Lehrmann Madsen. Semantic analysis of virtual classes and nested classes. In Linda M. Northrop, editor, *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices, volume 34, number 10, pages 114–131, Denver, CO, October 1999. ACM Press.
16. Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*, ACM SIGPLAN Notices, volume 24, number 10, pages 397–406, New Orleans, LA, October 1989.
17. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley, 1993.
18. Microsoft. Microsoft Java SDK 3.2 documentation. Available online through `http://www.microsoft.com/Java/sdk/32/`, 1999.
19. Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe — definitely. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 161–170, San Diego, CA, January 1998. ACM.
20. G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
21. Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley Longman, Reading, MA, third edition, 1997.
22. Don Syme. Proving Java type soundness. Technical Report 427, Computer Laboratory, University of Cambridge, June 1997.