On Inner Classes

Atsushi Igarashi[1]* and Benjamin C. Pierce[2]

[1] Department of Information Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan
igarashi@is.s.u-tokyo.ac.jp
[2] Department of Computer & Information Science, University of Pennsylvania
200 South 33rd St., Philadelphia, PA 19104, USA
bcpierce@cis.upenn.edu

Abstract. Inner classes in object-oriented languages play a role similar to nested function definitions in functional languages, allowing an object to export other objects with 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.

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.

A few object-oriented languages do support this sort of nesting. For example, Smalltalk [8] has special syntax for "block" objects, similar to anonymous functions. Beta [15] 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 [9, 12].

Inner classes are useful when an object needs to send another object a chunk of code that can manipulate the first object's methods and/or instance variables.

* This work was done while the author was visiting University of Pennsylvania.

Such situations are typical in user-interface programming: for example, Java's Abstract Window Toolkit [4] 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 ActionListner {
    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("Increment");
c.listenTo(b);
gui.add(b);
```

to create and display a button that increments a counter every time it is pressed.[1]

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 environments—including not only the lexical environment in which it appears, but also the lexical environment of each superclass. 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?

JavaSoft's Inner Classes Specification [12] provides one answer to these questions 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

[1] Strictly speaking, the increment of `x` should be `synchronized` with the listener's own counter, written `Counter.this`: listener methods are generally triggered in a thread different from the constructor thread of the current object.

about the behavior of inner classes, but it falls short of a completely satisfying account. First, the style is indirect: it forces programmers to reason about their code by first passing it through a rather heavy transformation. Second, 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 main 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 style of semantics is formalized for the first time.
  To keep the model as simple as possible, we focus on the most basic form of inner classes in Java, 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 (`public`/`private`/etc.) and inner classes (cf. [12, 2, 1]).
- Next, we give a translation from the language with inner classes to an even smaller language with only top-level classes, formalizing the translation semantics of the Java Inner Classes Specification. We show that the translation preserves typing.
- Finally, we prove that the two semantics 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. Unfortunately, our 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].)

Recently, Glew [7] has studied closure conversion in the context of an object calculus without classes; our translation semantics can be viewed as closure conversion of class definitions. However, since his calculus does not have classes, a semantic account of interaction between inheritance and nested classes has not been given.

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 [10] of GJ [3], an extension of Java with parameterized classes. It was 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 remainder of the paper is organized as follows. Section 2 briefly reviews Featherweight Java. Section 3 defines FJI, an extension of FJ with inner classes, giving its syntax, typing rules, and reduction rules, and stating 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 user programs to FJI, which is considered an intermediate language to define semantics. Section 6 examines some behavioral differences between compilers resulting from inconsistencies in the existing specification, Section 7 discusses related work, and Section 8 offers concluding remarks.

For brevity, proofs of theorems are omitted; they appear in a companion technical report [11].

2 Featherweight Java

We begin by reviewing the basic definitions of Featherweight Java [10]. 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 paper, 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 don't get stuck") as concise as possible, while still capturing the essence of the soundness argument for the full Java language. Any language feature that made the soundness proof *longer* without making it significantly *different* was a candidate for omission. Even assignments are omitted from FJ, as well as advanced features such as reflection and concurrency. Since FJ is a sublanguage of the extension defined in Section 3, we just show its syntax and an example of program execution here. The rest of the definition can be found in Figure 4.

The abstract syntax of FJ class declarations, constructor declarations, method declarations, and expressions is given as follows:

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

The metavariables `A`, `B`, `C`, `D`, and `E` range over class names; `f` and `g` range over field names; `m` ranges over method names; `x` ranges over parameter names;

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 $\texttt{f}_1,\ldots,\texttt{f}_n$ (and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, etc.) and write $\overline{\texttt{M}}$ as shorthand for $\texttt{M}_1\ldots\texttt{M}_n$ (with no commas). We write the empty sequence as $\bullet$ and denote concatenation of sequences using a comma. The length of a sequence $\overline{\texttt{x}}$ is written $\#(\overline{\texttt{x}})$. We abbreviate operations on pairs of sequences in the obvious way, writing "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" as shorthand for "$\texttt{C}_1\ \texttt{f}_1,\ldots,\texttt{C}_n\ \texttt{f}_n$" and "$\overline{\texttt{C}}\ \overline{\texttt{f}}$;" as shorthand for "$\texttt{C}_1\ \texttt{f}_1;\ldots\texttt{C}_n\ \texttt{f}_n;$" and "this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;" as shorthand for "this.$\texttt{f}_1$=$\texttt{f}_1$;...this.$\texttt{f}_n$=$\texttt{f}_n$;". For the sake of conciseness, we often abbreviate the keyword extends to the symbol extends and the keyword return to the symbol return. Sequences of field declarations, parameter names, and method declarations are assumed to contain no duplicate names.

A key simplification in FJ is the omission of assignment, making FJ purely functional. It is realized by assuming that all fields and method parameters are implicitly marked final. (Of course, most *useful* examples of programming in Java do involve its side-effecting features, and inner classes do interact with assignment: in particular, if inner classes may appear inside method definitions, then local variables of the enclosing method must be marked final if they are mentioned in an inner class. To handle this feature, our model would need to be extended with assignment. However, we do not need it for the present modeling task, and, by omitting assignment from FJ and FJI, we obtain a much simpler model that offers just as much insight into inner classes.) An object's fields are initialized by its constructor and never changed afterwards. Moreover, a constructor has a stylized syntax such that there is one parameter for each field, with the same name as the field; the super constructor is invoked on the fields of the supertype; and the remaining fields are initialized to the corresponding parameters. (These constraints are enforced by the typing rules.) This stylized syntax makes the operational semantics simple: a field access expression new C($\overline{\texttt{e}}$).$\texttt{f}_i$ just reduces to the corresponding constructor argument $\texttt{e}_i$. Also, since FJ does not have assignment statements, a method body always consists of a single return statement: all the computation in the language goes on in the expressions following these returns. A method invocation expression new C($\overline{\texttt{e}}$).m($\overline{\texttt{d}}$) is reduced by looking up the expression $\texttt{e}_0$ following the return of method m in class C in the class table, and reducing to the instance of $\texttt{e}_0$ in which $\overline{\texttt{d}}$ and the receiver object (new C($\overline{\texttt{e}}$)) are substituted for formal arguments and the special variable this, respectively. Figure 4 states these reduction rules precisely.

A program in FJ is a pair of a *class table* (a set of class definitions) and an expression (corresponding to the main method in a Java program). The reduction relation is of the form $\texttt{e} \longrightarrow \texttt{e}'$, read "expression e reduces to expression e′ in one step."

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);
  }
}
```

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

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

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

3 FJ with Inner Classes

We now define the language FJI by extending FJ with inner classes. Like FJ, FJI imposes some syntactic restrictions to simplify its operational semantics: (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 $\texttt{e}_0\texttt{.new C(}\overline{\texttt{e}}\texttt{)}$ is annotated with the static type `T` of $\texttt{e}_0$, written $\texttt{e}_0\texttt{.new<T> C(}\overline{\texttt{e}}\texttt{)}$.

Because of conditions (2) and (3), FJI is not quite a subset of Java (whereas FJ is); instead, we view FJI as an intermediate language, to which the user's programs are translated by a process of *elaboration*. We describe the elaboration process only informally in this paper (in Section 5), since it is rather complex but not especially deep, consisting mainly of a large number of rules for abbreviating long qualified names; a detailed treatment is given in the companion technical report [11]. 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, $\mathtt{C}_1$`.this` to $\mathtt{C}_{n-1}$`.this`, as well as the usual `this`, which can also be written $\mathtt{C}_n$`.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 $\mathtt{e}_0$`.new<T> C(`$\overline{\mathtt{e}}$`)`, where $\mathtt{e}_0$ is the enclosing instance and `T` is the static type of $\mathtt{e}_0$. The result of $\mathtt{e}_0$`.new<T> C(`$\overline{\mathtt{e}}$`)` is always an instance of `T.C`, regardless of the run-time type of $\mathtt{e}_0$. (We avoid a notation like $\mathtt{e}_0$`.new T.C(`$\overline{\mathtt{e}}$`)` because it is *not* in the Java syntax. Java allows only the notation `new T.C(`$\overline{\mathtt{e}}$`)` (without a prefix), which roughly means an instantiation from the class `T.C` with an enclosing instance `T.this`; 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 are *virtual* [14], i.e., different constructors may be invoked depending on the run-time type of the enclosing instance; for example, if there were a subclass `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 `Outer`$'$`.Inner`, depending on the dynamic type of `o`.)

The elaboration process allows type names to be abbreviated in Java 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(); }
}
```

in Java. 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: 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 1 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.

```
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 {
    ...
}}}
```

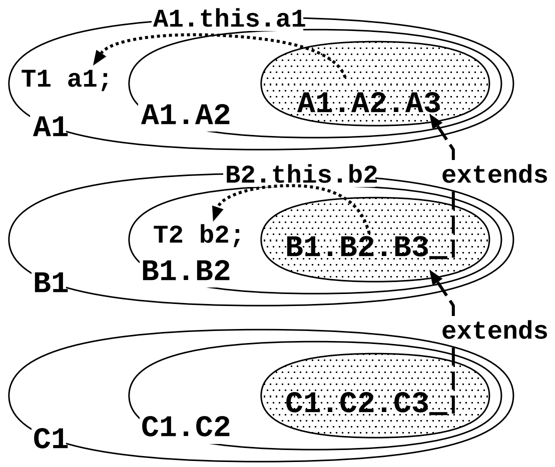

Fig. 1. A chain of environments

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`:

```
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 `RefinedInner` 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 Java allows any expression for the qualification. 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 $\mathtt{this\$C_1\$\cdots\$C}_n$, where $\mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n$ is the (direct) superclass, as in the example above.

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` as an argument, making it impossible to make an instance of `Foo`. (Perhaps 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 at the top left of Figure 2. We use the same notational conventions as in the previous section. The metavariables `S`, `T`, and `U` ranges over types, which are qualified class names (a sequence of simple names $\mathtt{C}_1,\ldots,\mathtt{C}_n$ concatenated by periods). For compactness in the definitions, we introduce the notation $\star$ for a "null qualification" and identify $\star$.`C` with `C`. The metavariable `P` ranges over types (`T`) and $\star$. We write $\mathtt{C} \in \mathtt{P}$ if $\mathtt{P} = \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n$ and $\mathtt{C} = \mathtt{C}_i$ for some i.

A class declaration `L` includes declarations of its simple name `C`, superclass `T`, fields $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$, constructor `K`, inner classes $\overline{\mathtt{L}}$, and methods $\overline{\mathtt{M}}$. There are two kinds of constructor declaration, depending on whether the superclass is inner or top-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 $\mathtt{super(}\overline{\mathtt{f}}\mathtt{)}$ (or $\mathtt{f.super(}\overline{\mathtt{f}}\mathtt{)}$); (2) the enclosing instance of the superclass (if needed); and (3) the fields of the class to be defined, initialized by $\mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}$. 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`, and that these variables are never used as the names of arguments to methods.

Syntax:

$$\begin{array}{lll}
\mathtt{T} & ::= & \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n \\
\mathtt{L} & ::= & \mathtt{class\ C} \lhd \mathtt{T\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{L}}\ \overline{\mathtt{M}}\mathtt{\}} \\
\mathtt{K} & ::= & \mathtt{C(}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\ \{} \\
 & & \quad \mathtt{super(}\overline{\mathtt{f}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{\ =\ }\overline{\mathtt{f}}\mathtt{;\}} \\
 & | & \mathtt{C(}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\ \{} \\
 & & \quad \mathtt{f.super(}\overline{\mathtt{f}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{\ =\ }\overline{\mathtt{f}}\mathtt{;\}} \\
\mathtt{M} & ::= & \mathtt{T\ m\ (}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)} \\
 & | & \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{e.new{<}T{>}\ C(}\overline{\mathtt{e}}\mathtt{)}
\end{array}$$

Computation:

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

$$\frac{\mathit{fields}(\mathtt{T.C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{e}_0\mathtt{.new{<}T{>}\ C(}\overline{\mathtt{e}}\mathtt{).f}_i \longrightarrow \mathtt{e}_i}$$

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

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

Subtyping:

$$\mathtt{T} <: \mathtt{T} \qquad \frac{\mathtt{S} <: \mathtt{T} \qquad \mathtt{T} <: \mathtt{U}}{\mathtt{S} <: \mathtt{U}}$$

$$\frac{\mathit{CT}(\mathtt{S}) = \mathtt{class\ C} \lhd \mathtt{T\ \{...\}}}{\mathtt{S} <: \mathtt{T}}$$

Expression typing:

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} \in \mathtt{T}}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m},\mathtt{T}_0) = \overline{\mathtt{U}} \rightarrow \mathtt{U}_0 \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} <: \overline{\mathtt{U}}\end{array}}{\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{U}_0}$$

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

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

Method typing:

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

Class typing:

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

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

Fig. 2. FJI: Main Definitions

A *program* is a pair of a class table CT (a mapping from types `T` to class declarations `L`) and an expression `e`. `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 at the bottom left Figure 2.

We impose the following sanity conditions on the class table: (1) $CT(\texttt{P.C}) = \texttt{class C ...}$ 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 $\texttt{e}_0\texttt{.new<T> C(}\overline{\texttt{e}}\texttt{)}$ (and $\texttt{new C(}\overline{\texttt{e}}\texttt{)}$, 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. (7) $\texttt{T} \not<: \texttt{T.U}$, for any two types `T` and `T.U`. By conditions (1) and (2), a class table of FJI can be identified with a set of top-level classes. 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 3. The fields of a class `T`, written *fields*(`T`), is a sequence $\overline{\texttt{T}}\ \overline{\texttt{f}}$ pairing the class of each field with its name, for all the fields declared in class `T` and all of its superclasses. In addition, *fields*(`T`) collects the types of (direct) enclosing instances of all the superclasses of `T`. For example, *fields*(`C1.C2.C3`) returns the following sequence:

$$\begin{array}{rlll}
\textit{fields}(\texttt{C1.C2.C3}) = & \texttt{Object} & \texttt{a3,} & \text{(field from } \texttt{A1.A2.A3}\text{)} \\
& \texttt{A1.A2} & \texttt{this\$A1\$A2\$A3,} & \text{(enclosing instance } \texttt{A2.this}\text{)} \\
& \texttt{Object} & \texttt{b3,} & \text{(field from } \texttt{B1.B2.B3}\text{)} \\
& \texttt{B1.B2} & \texttt{this\$B1\$B2\$B2,} & \text{(enclosing instance } \texttt{B2.this}\text{)} \\
& \texttt{Object} & \texttt{c3} & \text{(field from } \texttt{C1.C2.C3}\text{)}
\end{array}$$

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

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

The function $encl_{\texttt{T}}(\texttt{e})$ plays a crucial role in the semantics of FJI. Intuitively, when `e` is a top-level or inner class instantiation, $encl_{\texttt{T}}(\texttt{e})$ 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`). The first rule is the simplest case: since the type of an expression $\texttt{e}_0\texttt{.new<T> C(}\overline{\texttt{e}}\texttt{)}$ agrees with the subscript `T.C`, it just returns the (direct) enclosing instance

Field lookup:

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

$$\frac{\begin{array}{c} CT(\mathtt{T}) = \mathtt{class\ C \lhd D\ \{\overline{T}\ \overline{f};\ K\ \overline{L}\ \overline{M}\}} \\ fields(\mathtt{D}) = \overline{\mathtt{S}}\ \overline{\mathtt{g}} \end{array}}{fields(\mathtt{T}) = \overline{\mathtt{S}}\ \overline{\mathtt{g}}, \overline{\mathtt{T}}\ \overline{\mathtt{f}}}$$

$$\frac{\begin{array}{c} CT(\mathtt{T}) = \mathtt{class\ C \lhd U.D\ \{\overline{T}\ \overline{f};\ K\ \overline{L}\ \overline{M}\}} \\ fields(\mathtt{U.D}) = \overline{\mathtt{S}}\ \overline{\mathtt{g}} \\ \mathtt{U} = \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n \\ \mathtt{f}_0 = \mathtt{this\$C_1\$}\cdots\mathtt{\$C}_n\mathtt{\$D} \end{array}}{fields(\mathtt{T}) = \overline{\mathtt{S}}\ \overline{\mathtt{g}}, \mathtt{U}\ \mathtt{f}_0, \overline{\mathtt{T}}\ \overline{\mathtt{f}}}$$

Method type lookup:

$$\frac{\begin{array}{c} CT(\mathtt{T}) = \mathtt{class\ C \lhd S\ \{\overline{S}\ \overline{f};\ K\ \overline{L}\ \overline{M}\}} \\ \mathtt{U_0\ m\ (\overline{U}\ \overline{x})\ \{\uparrow e;\}} \in \overline{\mathtt{M}} \end{array}}{mtype(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{U}} \to \mathtt{U}_0}$$

$$\frac{\begin{array}{c} CT(\mathtt{T}) = \mathtt{class\ C \lhd S\ \{\overline{S}\ \overline{f};\ K\ \overline{L}\ \overline{M}\}} \\ \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}} \end{array}}{mtype(\mathtt{m}, \mathtt{T}) = mtype(\mathtt{m}, \mathtt{S})}$$

Method body lookup:

$$\frac{\begin{array}{c} CT(\mathtt{T}) = \mathtt{class\ C \lhd S\ \{\overline{S}\ \overline{f};\ K\ \overline{L}\ \overline{M}\}} \\ \mathtt{U_0\ m\ (\overline{U}\ \overline{x})\ \{\uparrow e;\}} \in \overline{\mathtt{M}} \end{array}}{mbody(\mathtt{m}, \mathtt{T}) = (\overline{\mathtt{x}}, \mathtt{e}, \mathtt{T})}$$

$$\frac{\begin{array}{c} CT(\mathtt{T}) = \mathtt{class\ C \lhd S\ \{\overline{S}\ \overline{f};\ K\ \overline{L}\ \overline{M}\}} \\ \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}} \end{array}}{mbody(\mathtt{m}, \mathtt{T}) = mbody(\mathtt{m}, \mathtt{S})}$$

Enclosing instance lookup:

$$encl_{\mathtt{T.C}}(\mathtt{e_0.new{<}T{>}\ C(\overline{e})}) = \mathtt{e}_0$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C \lhd D\ \{\overline{S}\ \overline{f};\ldots\}} \\ \#(\overline{\mathtt{f}}) = \#(\overline{\mathtt{e}}) \end{array}}{\begin{array}{c} encl_{\mathtt{T}}(\mathtt{new\ C(\overline{d},\ \overline{e})}) \\ = encl_{\mathtt{T}}(\mathtt{new\ D(\overline{d})}) \end{array}}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C \lhd U.D\ \{\overline{S}\ \overline{f};\ldots\}} \\ \#(\overline{\mathtt{f}}) = \#(\overline{\mathtt{e}}) \end{array}}{\begin{array}{c} encl_{\mathtt{T}}(\mathtt{new\ C(\overline{d},\ d_0,\ \overline{e})}) \\ = encl_{\mathtt{T}}(\mathtt{d_0.new{<}U{>}\ D(\overline{d})}) \end{array}}$$

$$\frac{\begin{array}{c} CT(\mathtt{S.C}) = \mathtt{class\ C \lhd D\ \{\overline{S}\ \overline{f};\ldots\}} \\ \#(\overline{\mathtt{f}}) = \#(\overline{\mathtt{e}}) \qquad \mathtt{T} \neq \mathtt{S.C} \end{array}}{\begin{array}{c} encl_{\mathtt{T}}(\mathtt{e_0.new{<}S{>}\ C(\overline{d},\ \overline{e})}) \\ = encl_{\mathtt{T}}(\mathtt{new\ D(\overline{d})}) \end{array}}$$

$$\frac{\begin{array}{c} CT(\mathtt{S.C}) = \mathtt{class\ C \lhd U.D\ \{\overline{S}\ \overline{f};\ldots\}} \\ \#(\overline{\mathtt{f}}) = \#(\overline{\mathtt{e}}) \qquad \mathtt{T} \neq \mathtt{S.C} \end{array}}{\begin{array}{c} encl_{\mathtt{T}}(\mathtt{e_0.new{<}S{>}\ C(\overline{d},\ d_0,\ \overline{e})}) \\ = encl_{\mathtt{T}}(\mathtt{d_0.new{<}U{>}\ D(\overline{d})}) \end{array}}$$

Fig. 3. FJI: Auxiliary definitions

$\mathtt{e}_0$. 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 of the argument $\mathtt{e_0.new{<}S{>}\ C(\overline{d},\ d_0,\ \overline{e})}$, the enclosing instance $\mathtt{e}_0$ is not the correct answer. We therefore make a recursive call with an object $\mathtt{d_0.new{<}U{>}\ D(\overline{d})}$ 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:

$$\begin{array}{l} \quad encl_{\mathtt{A1.A2.A3}}(\mathtt{e_0.new{<}C1.C2{>}\ C3(a,\ e_1,\ b,\ e_2,\ c)}) \\ = encl_{\mathtt{A1.A2.A3}}(\mathtt{e_2.new{<}B1.B2{>}\ B3(a,\ e_1\ ,b)}) \\ = encl_{\mathtt{A1.A2.A3}}(\mathtt{e_1.new{<}A1.A2{>}\ A3(a)}) \\ = \mathtt{new\ A1().new{<}A1{>}\ A2()} \\ \quad \text{where } \mathtt{e}_1 = \mathtt{new\ A1().new{<}A1{>}\ A2()} \text{ and } \mathtt{e}_2 = \mathtt{new\ B1().new{<}B1{>}\ B2()}. \end{array}$$

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: $\mathit{encl}_{\texttt{A1.A2}}(\mathit{encl}_{\texttt{A1.A2.A3}}(\texttt{e}))$.

3.5 Computation

As in FJ, the reduction relation of FJI has the form $\texttt{e} \longrightarrow \texttt{e}'$. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. The reduction rules are given in the middle of the left column of Figure 2. There are four reduction rules, two for field access and two for method invocation. The field access expression $\texttt{new C(}\overline{\texttt{e}}\texttt{).f}_i$ looks up the field names $\overline{\texttt{f}}$ of `C` using *fields*(`C`) and yields the constructor argument $\texttt{e}_i$ in the position corresponding to $\texttt{f}_i$ in the field list; $\texttt{e}_0\texttt{.new<T> C(}\overline{\texttt{e}}\texttt{).f}_i$ behaves similarly. The method invocation expression $\texttt{new C(}\overline{\texttt{e}}\texttt{).m(}\overline{\texttt{d}}\texttt{)}$ first calls *mbody*(`m`, `C`) to obtain a triple of the sequence of formal arguments $\overline{\texttt{x}}$, the method body `e`, and the class $\texttt{C}_1\texttt{.}\cdots\texttt{.C}_n$ where `m` is defined; it yields a substitution instance of the method body in which the $\overline{\texttt{x}}$ are replaced with the actual arguments $\overline{\texttt{d}}$, the special variables `this` and $\texttt{C}_n\texttt{.this}$ with the receiver object $\texttt{new C(}\overline{\texttt{e}}\texttt{)}$, and each $\texttt{C}_i\texttt{.this}$ (for $i < n$) with the corresponding enclosing instance $\texttt{c}_i$, obtained from *encl*. Since the method to be invoked is defined in $\texttt{C}_1\texttt{.}\cdots\texttt{.C}_n$, the direct enclosing instance $\texttt{C}_{n-1}\texttt{.this}$ is obtained by $\mathit{encl}_{\texttt{C}_1.\cdots.\texttt{C}_n}(\texttt{e})$, where `e` is the receiver object; similarly, $\texttt{C}_{n-2}\texttt{.this}$ is obtained by $\mathit{encl}_{\texttt{C}_1.\cdots.\texttt{C}_{n-1}}(\mathit{encl}_{\texttt{C}_1.\cdots.\texttt{C}_n}(\texttt{e}))$, and so on. The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules (if $\texttt{e} \longrightarrow \texttt{e}'$ then $\texttt{e.f} \longrightarrow \texttt{e}'\texttt{.f}$, and the like), which we omit here.

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 Rules

The typing rules for expressions, method declarations, and class declarations are given in the right column of Figure 2. An environment Γ is a finite mapping from variables to types, written $\overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}$. The typing judgment for expressions has the form $\Gamma \vdash \texttt{e} \in \texttt{T}$, 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>.

The typing judgment for method declarations has the form $\mathtt{M\ OK\ IN\ C_1.\cdots.C}_n$, read "method declaration M is ok if it is declared in class $\mathtt{C_1.\cdots.C}_n$." The body of the method is typed under the context in which the formal parameters of the method have their declared types and each $\mathtt{C}_i\mathtt{.this}$ has the type $\mathtt{C_1.\cdots.C}_i$. 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.

3.7 Properties

As well as FJ programs, FJI programs also enjoy standard subject reduction and progress properties, which together guarantee that well-typed programs never get stuck on field accesses or method invocations.

Theorem 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* $\mathtt{T}' \mathrel{<:} \mathtt{T}$.

Theorem 2 (Progress). *Suppose* e *is a well-typed expression.*

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

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

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, $\star$) where the entity being translated is defined, written $|\cdot|_{\mathtt{T}}$ (or $|\cdot|_{\star}$).

4.1 Types, Expressions and Methods

Every qualified class name is translated to a simple name obtained by syntactic replacement of `.` with `$`.

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

The compilation of expressions, written $|\mathtt{e}|_{\mathtt{T}}$, is given below. 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 $|\overline{\mathtt{L}}|_{\mathtt{P}}$).

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

As we saw above, a compiled inner class has one additional field, called $\mathtt{this\$}\,|\mathtt{T}|$, where `T` is the original class name. $\mathtt{C}_i\mathtt{.this}$ 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. An enclosing instance $\mathtt{e}_0$ of $\mathtt{e}_0\mathtt{.new<T>\ C(}\overline{\mathtt{e}}\mathtt{)}$ will become the last argument of the compiled constructor invocation.

Compilation of methods, written $|\mathtt{M}|_{\mathtt{T}}$, is straightforward. We use the notation $|\overline{\mathtt{T}}|\ \overline{\mathtt{x}}$ for $|\mathtt{T}_1|\ \mathtt{x}_1, \ldots, |\mathtt{T}_n|\ \mathtt{x}_n$.

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

4.2 Constructors and Classes

Compilation of constructors, written $|\mathtt{K}|_{\mathtt{T}}$, is given below.

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

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

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

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

It has four cases, depending on whether the current class is a top-level class or an inner class and whether its superclass is a top-level class or an inner class. When the current class is 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 superclass is inner (the third and fourth cases), the argument used for the qualification of $\mathtt{f.super(}\overline{\mathtt{f}}\mathtt{)}$ becomes the last argument of the `super()` invocation.

Finally, the compilation of classes, written $|\mathtt{L}|_{\mathtt{P}}$, is as follows:

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

$$\left|\mathtt{class\ C}\triangleleft\mathtt{S}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{L}}\ \overline{\mathtt{M}}\}\right|_{\mathtt{T}} = \begin{array}{l}\mathtt{class}\ \ |\mathtt{T.C}|\triangleleft|\mathtt{S}|\ \{\\ \quad |\overline{\mathtt{T}}|\ \overline{\mathtt{f}};\ |\mathtt{T}|\ \mathtt{this\$}|\mathtt{T.C}|\,;\ |\mathtt{K}|_{\mathtt{T.C}}\ \ |\overline{\mathtt{M}}|_{\mathtt{T.C}}\}\\ |\overline{\mathtt{L}}|_{\mathtt{T.C}}\end{array}$$

The constructor, inner classes, and methods of class `C` defined in `P` are compiled with auxiliary argument `P.C`. Inner classes $\overline{\mathtt{L}}$ become top-level classes. As in constructor compilation, when the compiled class 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., $|\overline{\mathtt{L}}|_{\star}$).

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 3). 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) and vice versa (Theorem 5).

Theorem 3 (Compilation preserves typing). *When* $\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{T}}$*, we write* $|\Gamma|$ *for* $\overline{\mathtt{x}} : \left|\overline{\mathtt{T}}\right|$*. 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* $\overline{\mathtt{x}} : \left|\overline{\mathtt{T}}\right|, \mathtt{this} : |\mathtt{C}_1 . \cdots . \mathtt{C}_n| \vdash_{\mathrm{FJ}} |\mathtt{e}|_{\mathtt{C}_1 . \cdots . \mathtt{C}_n} \in |\mathtt{T}|$ *with respect to* $|CT|$*.*

Theorem 4 (Compilation commutes with reduction). *If* $\Gamma \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{T}$ *where* $dom(\Gamma)$ *does not include* `this` *or* `C.this` *for any* `C`*, and* $\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{e}'$*, then* $|\mathtt{e}|_\star \longrightarrow_{\mathrm{FJ}}{}^{+} |\mathtt{e}'|_\star$*.*

Theorem 5 (Compilation preserves termination). *If* $\Gamma \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{T}$ *where* $dom(\Gamma)$ *does not include* `this` *or* `C.this`*, 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 and 5 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 guarded by its context. Therefore, reduction steps do not commute with compilation straightforwardly. Nevertheless, it should be possible to show correctness pby using another technique, such as contextual equivalence [18], as Glew proved a similar result in the context of object closure conversion for a call-by-value object calculus [7].

5 Elaboration

In this section we formalize the elaboration of user programs. 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.

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` (when neither `C` nor `D` has field `f`) 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 current class where the 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 recover the appropriate qualification. In the examples above, `f` becomes `C.this.f` and `D.this.f`, respectively.

Suppose the algorithm above finds the definition of the field/method in one of the superclasses of the current class. Then, a field/method of the same name must not be defined in any of the enclosing classes. Similarly, if the field/method definition is found in a superclass of an enclosing class `C`, a field/method of the same name must not be defined in any of `C`'s enclosing classes. In the example above, if both `B` and `C` declared a field `f` (and `D` did not), then elaboration would fail as `f` in `m` is *ambiguous*; the user must write `C.this.f` or `D.this.f`, specifying the enclosing instance explicitly. This rule also has one significant exception: 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 extends Object {
  Object f; ...
  class D extends Object { ...
    class E extends C { ...
      Object m () { return f; }
    }
  }
}
```

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

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-elaborated type names themselves can be qualified. In such a case the head simple name is elaborated first, then it looks up 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; ... }
```

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

1. The first name `D` is elaborated to `C.D`.
2. It is checked whether `C.D.B` makes sense; in this case, it does not, since the inner class `D` does not have the 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 T(`$\overline{\texttt{e}}$`)` is elaborated. Actually, it is slightly more involved than others since it requires both elaboration of the type and recovering of an enclosing instance (when it turns out to be instantiation of an inner class). First of all, the pre-elaborated type name `T` is elaborated to $\texttt{T}'$. If $\texttt{T}'$ is a simple name `C`, then the constructor invocation does not need an enclosing instance. On the other hand, if $\texttt{T}'$ 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 T(`$\overline{\texttt{e}}$`)` is elaborated to `D.this.new<U> C(...)`. The annotation `<U>` is important 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 annotation `<A>` is important since we have to remember that the class `A.B` is to be instantiated (not `C.B`).

For brevity, we omit the formal rules of elaboration, which closely follow the algorithm described above; interested readers are referred to a companion technical report [11].

6 Interpretations of the Inner Class Specification

Through this work, we have experimented a few Java compilers, including Sun's JDK (for Solaris), JDK for linux, and `guavac`. Besides finding a few bugs related to inner classes (mostly already known to the developers), we observed some interesting variations in behavior corresponding to an underspecification in the currently available Inner Classes Specification [12], concerning the meaning of the `C.this` expression. 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 1.1.7a, but `I'm a C object` under JDK 1.2. In the old JDK, the meaning of `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, on the other hand, `C.this` is always bound to the enclosing object of the receiver regardless of superclass.

7 Related Work

Nested classes in Beta. Beta [15] also allows nested class definitions (as an instance of nested *patterns*, the only abstraction mechanism in Beta, which unifies classes and procedures). There are two significant differences from inner classes. First, inner classes are covariantly specialized in a subclass: for example, if `C <: D` and both `C` and `D` have the declaration of an inner class of name `E`, then `C.E` must extend `D.E`. Second, nested classes are *virtual* [14], in the sense that it depends on *run-time type* of the enclosing instance which constructor is invoked. A constructor invocation `e.new E(ē)` instantiates an object of class `C.E` when the run-time type of `e` is `C` while it instantiates an object of class `D.E` when that of `e` is `D`.

Madsen has recently described the algorithm of elaboration (they call semantic analysis) used in the Mjølner Beta compiler [13]. The algorithm is very close to the rules presented in Section 5, in a sense that the search order is the same as ours, although the presence of virtual classes complicates the algorithm.

Specification of inner classes. In the currently available Inner Classes Specification [12], semantics of inner classes is given as a translation from inner classes to top-level classes. It also explains how inner classes affect other language aspects, such as synchronization, access restriction and binary compatibility. However, description is rather informal and sometimes vague, resulting in different implementations with different semantics, as explained in the previous section.

Object closure conversion. Recently, Glew [7] has studied closure conversion in the context of a call-by-value object calculus (without classes) and shown 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, semantic account of interaction between inheritance and nested classes is not given.

Microsoft's delegates. Microsoft has proposed *delegates* [16] 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 the one hand, the formalization would be simpler than inner classes due to the absence of interaction with inheritance. On the other hand, it would be hard to model the implementation scheme of delegates, since it depends on Java's reflection features.

Other core calculi for Java. There have been proposed several calculi [5, 19, 17, 6] to study formal properties and extensions of Java; none of them, however, treats inner classes, although we don't see any inherent difficulty to integrate inner classes into their calculi.

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. Besides deepening our own understanding of inner classes, this work has uncovered a significant underspecification in the official specification.

For future work, the interaction between inner classes and access restrictions in Java is clearly worth investigating. We also hope to be able to model Java's other forms of inner classes: anonymous classes and local classes, which can be declared in method bodies; these are slightly more complicated, since method arguments (not just fields) can occur in them as free variables, but we expect they can be captured by a variant of FJI.

Acknowledgments

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

We would like to thank 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 POPL'99, FOOL7, and ECOOP2000 helped us improve the final presentation.

References

[1] Martín Abadi. Protection in programming-language translations. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, pages 868–883. Springer-Verlag, July 1998. 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] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.

[4] Patrick Chan and Rosanna Lee. *The Java Class Libraries*, volume 2. Addison-Wesley, Reading, MA, second edition, October 1997.

[5] 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.

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

[7] 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`.

[8] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[10] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Linda M. Northrop, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, volume 34, number 10, pages 132–146. ACM Press, October 1999.

[11] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. Technical Report MS-CIS-99-23, University of Pennsylvania, Philadelphia, PA, November 1999. Available through `http://web.yl.is.s.u-tokyo.ac.jp/~igarashi/papers.html`.

[12] JavaSoft. Inner classes specification, February 1997. Available through `http://java.sun.com/products/JDK/1.1/`.

[13] Ole Lehrmann Madsen. Semantic analysis of virtual classes and nested classes. In Linda M. Northrop, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, volume 34, number 10, pages 114–131, Denver, CO, October 1999. ACM Press.

[14] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1989.

[15] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[16] Microsoft. Microsoft Java SDK 3.2 documentation. Available online through `http://www.microsoft.com/Java/sdk/32/`, 1999.

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

[18] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

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

Syntax:

$$\mathtt{L} ::= \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{C}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}$$

$$\begin{aligned}\mathtt{K} ::= {} & \mathtt{C}(\overline{\mathtt{C}}\ \overline{\mathtt{f}})\\ & \quad \{\mathtt{super}(\overline{\mathtt{f}});\ \mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{f}};\}\end{aligned}$$

$$\mathtt{M} ::= \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}$$

$$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$$

Computation:

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})).\mathtt{f}_i \longrightarrow \mathtt{e}_i}$$

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

Subtyping:

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

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}}$$

Expression typing:

$$\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})$$

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

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

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) \in \mathtt{C}}$$

Method typing:

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 \in \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0\\ \mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}\ \{\ldots\}\\ \text{if } \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{D}} \to \mathtt{D}_0,\\ \text{then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e}_0;\}\ \mathtt{OK\ IN\ C}}$$

Class typing:

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

Field lookup:

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

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}\end{array}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$

Method type lookup:

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\\ \mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}}$$

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

Method body lookup:

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\\ \mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{e})}$$

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

Fig. 4. FJ Definitions