Formalizing Advanced Class Mechanisms

by

Atsushi Igarashi

A Dissertation

Submitted to

the Graduate School of

the University of Tokyo

on December 21, 1999

in Partial Fulfillment of the Requirements
for the Degree of Doctor of Science

Thesis Supervisor: Naoki Kobayashi
Lecturer of Information Science

Abstract

Class-based languages, such as C++ and Java, form the mainstream of object-oriented programming. The basic function of classes is to describe objects with similar behavior concisely. Recent languages have added to the basic class mechanism a variety of advanced features, such as inner classes, found in Beta and in Java 1.1, and type parameterization of classes, found in C++ as templates and in several extensions for Java, including GJ [Bracha, Odersky, Stoutamire, and Wadler]. Inner classes enable a programming style similar to nested functions/procedures, while type parameterization makes programming of polymorphic data structures, such as lists, more convenient. However, this power comes at a significant cost in complexity, which makes it difficult to understand the behavior of programs.

The main goal of this work is to clarify the essence of the type systems and compilation schemes of inner classes and GJ-style type parameterization. We approach this goal by building their formal models based on a small core calculus, called *Featherweight Java*, for class-based object-oriented languages, and by proving their properties, such as type soundness. Our contributions are as follows.

1. Design and formalization of Featherweight Java. It is intended to be a smallest possible language to capture the essential parts of type systems of class-based object-oriented languages, so that proofs for the complex extensions are tractable.

2. Formalization of the core of inner classes and proof of its type safety. Featherweight Java is extended with inner classes; the definition of its semantics illuminates complexity related to interaction between inner classes and inheritance.

3. Formalization of the core of GJ and proof of its type safety. We extend Featherweight Java with type parameterization and raw types, one of GJ's distinctive features. Raw types are designed to maintain compatibility between polymorphic classes and their legacy clients that still expect the old monomorphic version of those classes. The proof of type soundness of raw types uncovers some underspecifications and flaws in the current design of raw types.

4. Formalization and proof of correctness of the compilation schemes of inner classes and GJ. We model the current compilers by giving translation from the extended languages into Featherweight Java, and prove that the translation preserves well-typedness and semantics.

Contents

List of Figures

Acknowledgments

I have many people to thank for helping me complete my thesis work. First of all, I cordially thank Naoki Kobayashi, my thesis supervisor, who has led me to the interesting research area of programming language theory. His advice has helped me learn how to think scientifically. I also owe a large debt to him for suggesting me to go to study in the United States of America.

I express my profound gratitude to Benjamin Pierce, the advisor during my stay at Indiana University and University of Pennsylvania from June 1998 until now. Working with him was not only a wonderful experience but also a great fun. I will miss discussions with him. Aside from work, he greatly helped me settle down in the U.S. when I was not accustomed to the life in a foreign country. His preprocessor for TeX source files has contributed for typesetting of this thesis.

I also express my appreciation to Philip Wadler, who, as one of the designers of GJ, gave me many useful comments to clarify intuition behind the GJ design. His punning always amused me; the namings of Featherweight Java and "cooked types" as opposed to "raw types" are attributed to him.

Akinori Yonezawa deserves special mention for being my advisor from my senior student year and first year at the graduate school and for continuous encouragements and insightful comments.

It is important for a graduate student to have good colleagues to stimulate each other; luckily enough, I was blessed with such great colleagues. Thanks go to the members of Programming Language Club at University of Pennsylvania, including my officemates, Haruo Hosoya and Michael Levin, who spent their precious time with me for both fruitful discussions and easy conversations. Also, I am grateful to the members of Yonezawa Laboratory and Kobayashi Laboratory for discussions and comments on my work. Especially, Kenjiro Taura gave me many invaluable comments from the practical point of view. I thank Noriko Matsumoto, Toshihiro Shimizu, Eijiro Sumii, and Yoko Taura for helping my paperwork.

I express my gratitude to the Japan Society of the Promotion of Science for funding me for the three years of my PhD program via a fellowship.

My friends, including Motoki Nakade and Junji Tamatsukuri, have been keeping me sane during my graduate student years, which can easily be discouraging and stressful.

Last, but not least, I thank my parents for everything they have done for me over the years. Without their generous support, I could not finish my thesis.

Atsushi Igarashi,
December 21, 1999

Chapter 1

Introduction

1.1 Class-Based Object-Oriented Programming

In the past few decades, object-oriented programming languages have been widely spread and extensively used in various application areas. The central notion in object-oriented programming is an *object*, which is, roughly speaking, a data structure consisting of some (encapsulated) information and a set of operations on it. For example, we can think of a (two-dimensional) point object, consisting of x- and y-coordinate information and a move operation to change the coordinates. A large software is usually built by combining various small objects into a large object, thus making reusability one of the main issues in object-oriented design and programming.

Most object-oriented programming languages proposed so far fall into one of the following three families of languages: (1) prototype-based languages, such as Self [US91] and Obliq [Car95], where programmers derive a new object from an existing object by adding or modifying functionalities that the old one has; (2) class-based languages, such as C++ [Str97], Eiffel [Mey92], Java [GJS96], and Smalltalk [GR83], where an object is instantiated from a *class*, serving as a template of objects; (3) multi-method-based languages, such as CLOS [DG87] and Cecil [Cha97], where one operation can be associated with more than one object, which is not the case in the first two families.

Among them, class-based languages has become most popular; main reasons seem that the abstraction mechanism of classes provides programmers a simple and intuitive view of programming, and that it is also convenient to write reusable and structured software components. Although different languages have different features about classes, most class-based languages share several basic notions, briefly reviewed below.

Concise description of objects. As mentioned above, the most fundamental role of classes is a template of objects: programmers write a class to describe objects of similar behavior and instantiate as many objects as he or she uses, from one class. This is an example of a class declaration for point objects, written in Java:

```
class Point {
  int x; int y;
  Point(int x', int y') { x = x'; y = y'; }
  void move(int ox, int oy) { x = x + ox; y = y + oy; }
}
```

The class is given name `Point`, and, inside curly brackets, it has declarations of *members*, consisting of two *fields*, one *constructor*, and one *method*. Fields are considered the internal state of an object; in this example, every point object has two integers of the names `x` and `y`. The third line is a declaration of a constructor, which performs the initialization of the object state when an object is instantiated from the class. The constructor above takes two integers `x'` and `y'` as arguments and assign them to the fields. In Java, a constructor is invoked with a `new` expression—for example, `new Pair(0,0)` instantiates a point object that represents the origin. The fourth line is a declaration of the method `move`; it takes two integers `ox` and `oy` and increments each coordinate by the corresponding argument (and it returns nothing, as indicated by `void`). To invoke the method, an expression of the form $\mathtt{e}_0$`.move(1,2)` is used in Java, where $\mathtt{e}_0$ is expected to evaluate to a point object. Most languages also provide a way to access the value of a field, written $\mathtt{e}_0$`.f` (in Java) where `f` is the field name.

As such, the `Point` class gives us a concise description of what point objects are and how they behave.

Remark. Actually, in Java (and most object-oriented languages), an access to a field defined in the current class is an abbreviation of the standard field access expression $\mathtt{e}_0$`.f`; in Java, the special keyword `this` is used for $\mathtt{e}_0$ to represent the object of the current class. Thus, the class definition above is, in fact, an abbreviation of the following:

```
class Point {
  int x; int y;
  Point(int x', int y') { this.x = x'; this.y = y'; }
  void move(int ox, int oy) { this.x = this.x + ox; this.y = this.y + oy; }
}
```

Similarly for method invocations: an expression of the form `m(...)` is an abbreviation of `this.m(...)`.

Inheritance for code reuse. Now suppose we want to use colored points, which has a field for their color as well as ones for their coordinates. One might copy the `Point` class and modify it to the `CPoint` class. However, it is no good to have two similar but distinct definitions; even worse, such a program is hard to maintain since future changes should be made in both classes if the programmer want to keep the behavior of both kinds of points consistent.

Class-based languages provide the mechanism of *inheritance* to reuse a class definition to define a new class. For example, the `CPoint` class is defined as follows:

```
class CPoint extends Point {
  Color c;
  CPoint(int x', int y', Color c') { x = x'; y = y'; c = c'; }
  void move(int ox, int oy) { x = x + ox; y = y + oy; c = White; }
}
```

(where we assume the type `Color` and the constant `White`.) The class `CPoint` also has declarations of a field, a constructor, and a method. Moreover, it specifies its *superclass* `Point` after the keyword `extends`. The class `CPoint`, called a *subclass* of `Point`, inherits the definitions of the fields and methods of `Point`; as a result, colored objects also have the fields `x` and `y`. Not only can programmers add new methods, they can also *override* an inherited method definition: in the `CPoint` class, the method `move` is overridden with the new definition. Constructors are not considered inherited since they are used for instantiation of an object of the class in which they are declared. However, as in this example, it is often the case that initialization codes are similar; hence, several languages (including Java) allows to access constructors (and methods) of the superclass using the keyword `super`. The declaration above can be rewritten as follows:

```
class CPoint extends Point {
  Color c;
  CPoint(int x', int y', Color c') { super(x',y'); c = c'; }
  void move(int ox, int oy) { super.move(ox,oy); c = White; }
}
```

where `super(x′,y′)` invokes the constructor in `Pair` and `super.move(ox,oy)` invokes the method `move` defined in `Pair`.

Using inheritance, programmer can not only save the effort to maintain two similar definitions but also make changes in a superclass automatically propagate to all the subclasses.

Types and subtyping In popular statically-typed languages, such as C++, Eiffel, and Java, each class declaration introduces a new type of the same name as the class—for example, objects instantiated from the class `Pair` belong to the type `Pair`. Moreover, the subclass relation induces a *subtyping* relation—for example, the type `CPoint` is said to be *subtype* of `Point` and, conversely, the type `Point` is said to be *supertype* of `CPoint`. The notion of subtyping is used to guarantee safe substitutivity: if `A` is subtype of `B`, then any expression of type `A` may be used without type errors in any context that requires an expression of type `B`. For example, consider an expression `x.move(2,2)` where the variable `x` is given type `Point`. We can replace `x` with not only a `Point` object but also a `CPoint` object; in fact, in both cases, the method invocation can be executed without any problems. Note that, however, it depends on the *run-time* type, `Point` or `CPoint`, of the object which `move` method is invoked even if `x` is given type `Point`. This mechanism is called *dynamic dispatch* (or dynamic binding), considered one of the important features of object-oriented languages.

Java has the class `Object` and every class definition without an `extends` clause is considered to extend the class `Object` implicitly. As a result, `Object` serves as the top type, which is supertype of all types.

Although this subclass-subtyping correspondence reflects the intuition that, for example, a color point is a kind of point, the connection between inheritance and subtyping is not as strong as it first appeared. For example, even if a rectangle object has the method `move`, it is not allowed to substitute the object for `x` in the example above, unless its class is a subclass of `Point`, thus limiting flexibility. Moreover, in some languages that has the notion of the so-called `MyType`, the subclass relation should not be considered the subtyping relation [CHC90]. Hence, several languages [Bru94, BSvG95] separate subclass and subtyping relations. Nevertheless, for its simplicity, even a new language like Java still adopts the subclass-subtype correspondence, discarding flexibility.[1]

Access control. Data encapsulation is an important aspect of object-oriented programming. Most class-based languages let programmers control accessibility of fields, constructors, and methods with annotations. Common annotations include *private*, which means that access to the member is allowed only in the current class in which it is defined, *protected*, which means that access is allowed only in the current class and its subclasses, and *public*, which means that the member may be accessed everywhere. For example, programmers might write:

```
class Point {
  private int x;
  private int y;
  public Point(int x', int y') { x = x'; y = y'; }
```

[1]Java has introduced *interfaces* for a partial remedy for the lack of the flexibility.

```
    public void move(int ox, int oy) { x = x + ox; y = y + oy; }
  }
```

to protect `x` and `y` from direct access while the constructor and the method `move` are open to public. Thus, an expression like `new Point(0,0).x` is not allowed (outside the definition of `Point`). By using access control, programmers can hide the internal structure of objects and export only relevant operations to outside.

1.2 Advanced Class Mechanisms

Recently, several advanced class mechanisms have been proposed to strengthen the abstraction power of classes. We briefly summarize some of them below:

- *Parametric classes*, found in C++ (known as templates), Eiffel, several extensions to Java [AFM97, MBL97, OW97, BOSW98b, CS98], OCaml [RV98]. The basic idea of parametric classes is closely based on parametric polymorphism, found in functional languages such as ML [MTHM97] and Haskell [HJW$^+$92]: type information in classes can be abstracted out as parameters so that different concrete types for the parameters can be specified later. By using parametric classes, generic data structure, such as lists and trees, can be described more concisely—that is, a parametric class for lists is defined so that it takes a type parameter that represents the type of elements; this definition is used for integer lists or string lists by applying it to the integer or string type, respectively. Some proposals even allow each method to take its own type parameters. Such polymorphic methods are useful to define operations such as "map" operation on lists.

- *Virtual types*, found in Beta [MMPN93, MMP89], and an extension to Java [Tho97]. This mechanism allows types to be members of a class, as well as fields and methods; moreover, such type member definitions can be overridden in subclasses. Virtual types can also be used to define generic data structures [MMP89, Tho97].

- *Nested classes* (a.k.a. *inner classes*), found in Beta, C++, and Java. Programmers can declare a class as a member of another class. Since the inner class has direct access to members of its enclosing class, they 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.

- *Mixins* [BC90]. In most class-based languages, the subclass relation is hard-wired in the class definition. On the other hand, mixin-based inheritance enables a more flexible style of inheritance by allowing the superclass information to be abstracted out as a parameter; thus, it is easy to add (or sometimes override) the same set of functionality to different classes.

These advanced mechanisms are useful and, in fact, some of them has become popular—for example, the Standard Template Library [SL94] is a C++ library using templates, and the Java Class Libraries [CL97] use inner classes extensively.

This additional power, however, comes at a significant cost in complexity, which makes it difficult to understand the behavior of programs. For example, inner classes show subtle interaction with inheritance. First, the language semantics is not straightforward since any form of inheritance is allowed to inner classes: for example, it is not very clear what a top-level subclass of an inner class should mean. Second, scoping rules become complicated: the use of a member name may be bound to not only a declaration in enclosing classes (as in conventional languages with block structure) but also a declaration in a superclass.

1.3 Subject of the Thesis

This thesis is concerned with clarification of the essential mechanisms behind such advanced class mechanisms. In particular, we pick up Java-style inner classes [Jav97] and GJ-style parametric classes [BOSW98b].

One common problem lies in the way how their semantics is described. Their specification documents [Jav97, BOSW98a] provide answers to questions about their semantics, but they fall short of a completely satisfying account. First, their styles are indirect: the semantics are defined as translation into Java Virtual Machine Language (JVML) [LY99], which is intended to be a portable low-level language for class-based object-oriented languages. This indirect style of semantics makes it hard to reason about program behavior, since programs first have to be passed through a rather heavy transformation. Second, the documents themselves are somewhat imprecise consisting only of examples and English prose and sometimes include underspecifications as we will see later. Hence, it is desirable to give a rigorous definitions of a *direct* style of semantics of inner classes and parametric classes.

Aside from the problems of the official specifications, it is worth investigating consistency between direct semantics and translation-based semantics. The target language JVML of compilation is very close to Java, but neither inner classes nor parametric classes are supported. Thus, it is interesting to clarify how top-level monomorphic classes can simulate inner classes and parametric classes and to show that the current compilation scheme is correct with respect to a certain direct semantics.

GJ is one of the leading proposals of extensions to Java with parametric classes, introducing a few novel features including type inference for polymorphic method invocation. Among those features, the mechanism of *raw types*, is especially of interest. They allow parametric classes to be used without type parameters so that upgrading monomorphic library classes to a polymorphic version does not spoil client programs that expect the old version. For example, even if a class `List` is revised to a polymorphic class `List<X>` (where `X` is the type parameter for the element type), the old client can still refer to the new class via the raw type `List`. As such, raw type are designed to smooth *program evolution*, by which we mean upgrading of programs from a monomorphic class to a polymorphic version. However, type safety argument of raw types is very subtle.

This thesis will address the issues above in a rigorous manner. In fact, they *must* be discussed rigorously, since compiler writers or, sometimes, even application programmers want to know elaborate details of what they are dealing with.

1.4 Our Approach

Our approach to tackle the problems is (1) to introduce a small core language for class-based object-oriented languages, and (2) to build formal models of inner classes and parametric classes on top of that language.

Our core language, *Featherweight Java* (or *FJ* for short), is intended to be a smallest possible language to capture the essential parts of class-based object-oriented languages. Although, its syntax, semantics and type system closely follow Java's, it models only top-level class declarations, inheritance, subtyping, method override, object instantiation, field access, method invocation, and method recursion through `this`. It omits even some of the basic features such as access control and assignments as well as Java's advanced features such as reflection and threads. This extreme simplicity not only make proofs of useful properties such as type soundness concise but also let us focus on the essential issues of complex extensions and make proofs of properties of extensions tractable. Although some of the omitted features are expected to interact significantly with inner classes or GJ-style parametric classes, we have decided not to include them and chosen simplicity for the first step of the work, and left the study of such features for future work. In particular, access control is not discussed at all, even though its interaction with inner classes seems interesting.

In the literature of theoretical foundations of object-oriented languages [Wan89, Bru94, FM98, BF98, AC96, PT94], classes are often encoded into more primitive object calculi without classes, or even into languages without objects, such as System $F^{\omega}_{\leq}$ (the omega-order polymorphic lambda-calculus with subtyping [Car90, CL91, PS94, Com94]), while FJ takes classes as a primitive. Our approach has several advantages over them to investigate advanced class mechanisms.

1. We need not be bothered by the complexity of the encoding. Since even encoding of the basic mechanisms of classes is complicated, encoding of advanced features can easily be too complex to be handled. By taking classes as a primitive, we can concentrate on complexity of extensions. Moreover, it is suitable for one of our purposes: clarification of the essence of compilation, which can be modeled by translation from the extensions to FJ.

2. We need not be bothered by the fundamental difference of the type systems. Most work on type systems for objects have separated object types from classes and adopted *structural subtyping*, based on the shape of objects themselves, while FJ (and most popular languages) uses *name-based subtyping*, based on the name of the class and the subclass relation. Since the connection between these two styles of subtyping is not well-understood yet, we stick to the name-based subtyping here. Above all, it seems challenging to express Java's *typecasts*, which checks the dynamic type of an expression, in a language with structural subtyping. Typecasts play an important role in the model of GJ.

In short, we design our base language so that it does not introduce extra complexity that is not very essential to understand the subject of our study.

1.5 Our Contributions

The contributions of this thesis are summarized as follows:

- The design of Featherweight Java. As mentioned above, we design Featherweight Java as a vehicle for our study of inner classes and GJ. We present its formal syntax, semantics and type system and develop a proof of type soundness using a standard technique, subject reduction and progress [WF94].

- Formal model of the core of Java-style inner classes. We extend FJ with inner classes and define *FJI*. For FJI, two styles of semantics are defined: direct style, defined by a reduction relation between FJI expressions, and translation style, defined by a translation from FJI to FJ. The former style provides a direct view of execution while the latter style serves as a model of the current specification. As we will see, the direct semantics is not so straightforward as one might expect, due to the interaction between inner classes and inheritance. We prove type soundness of the direct semantics and equivalence of the direct and translation semantics: it is proved that the translation preserves typing and behavior of the program in an appropriate sense.

- Formal model of the core of GJ-style parametric classes. As we do for inner classes, we define *Featherweight GJ* (or *FGJ* for short) by extending FJ with parametric classes. First, we define type-passing semantics, defined by a reduction relation where type parameter information is carried along, and prove type soundness. Second, we give an erasure translation from FGJ to FJ, where type parameter information is removed; it models the current specification and compilation scheme of GJ. We prove not only that the translation preserves typing but also that the translation preserves execution results. As we will see, the argument for preservation of behavior is much trickier than one would expect.

- Formalization of raw types. Finally, we extend FGJ with raw types, one of the distinctive features of GJ. The main result here is a proof of type soundness. Actually, the current type system, given by the official specification [BOSW98a], is found flawed. We prove subject reduction for a fixed type system. In addition, we discuss the notion of program evolution based on our model. We conjecture desired properties of raw types about program evolution and discuss the current GJ design.

Direct semantics discussed in both extensions is important since semantics depending on translation makes reasoning about program behavior hard. For both inner classes and GJ (and other proposals of parametric classes for Java), this work is the first (at least to our knowledge) formal presentation of their direct semantics. On the other hand, translation-based semantics is useful to understand the essence of compilation. Proving equivalence of the two styles corresponds to showing correctness of the compiler.

Through this work, we have tested a few Java compilers and the GJ compiler, and found several small bugs in them (though they have been fixed). Moreover, we have found a few underspecifications in the current specifications and even some design bugs (of GJ).

1.6 Overview of the Thesis

The rest of this thesis is organized as follows. Chapter 2 introduces Featherweight Java. After its syntax, semantics and type system are presented, type soundness is proved; then, we discuss a small extension of FJ, which serves as a base language for Featherweight GJ. In Chapter 3, we extend Featherweight Java with inner classes. For the extended language FJI, we present the two styles of semantics and prove their type soundness and equivalence in an appropriate sense. We also discuss scoping issues in a language with inner classes. Chapters 4 and 5 are concerned with GJ-style parametric classes. In Chapter 4, we deal with the core of GJ without raw types. Like the previous chapter, extending Featherweight Java with parametric classes, we introduce Featherweight GJ with the definitions of both direct and translation semantics; we prove their type soundness and equivalence of the two styles. Then, we further augment Featherweight GJ with raw types in Chapter 5, in which the main theoretical result is proof of type soundness. We also discuss relationship between raw types and program evolution, and conjecture the desired properties. Finally, after discussing related work in Chapter 6, we conclude the thesis with discussion on future work in Chapter 7.

Chapter 2

Featherweight Java

In this chapter, we introduce the vehicle of our study, Featherweight Java (or FJ), on which we will build models of inner classes and GJ in the later chapters. One main goal in designing FJ is to make proofs for complex extensions tractable. Although the language closely follows Java's syntax, typing rules, and semantics (in fact, every FJ program is literally an executable Java program), any language feature that made those proofs *longer* without making it significantly *different* was a candidate for omission. As in previous studies of core languages for Java, we don't treat advanced features such as concurrency, reflection, and packages. Other Java features omitted from FJ include assignment, interfaces, overloading, messages to `super`, `null` pointers, base types (`int`, `boolean`, etc.), abstract method declarations, shadowing of superclass fields by subclass fields, access control (`public`, `private`, etc.), and exceptions. The features of Java that we *do* model include mutually recursive class definitions, inheritance, subtyping, method override, object instantiation, field access, method invocation, method recursion through the special variable `this`. This extreme simplicity lets us focus on the essential issues of complex extensions.

One key simplification in FJ is the omission of assignment. We assume that an object's fields are initialized by its constructor and never changed afterwards. This restricts FJ to a "functional" fragment of Java, in which many common Java idioms, such as use of enumerations, cannot be represented. Nonetheless, this fragment is computationally complete (it is easy to encode the lambda calculus into it), and is large enough to include many useful programs (many of the programs in Felleisen and Friedman's Java text [FF98] use a purely functional style). Moreover, most of the tricky typing issues in any of Java, inner classes, and GJ are independent of assignment.

The remainder of this chapter is organized as follows. Section 2.1 introduces the main ideas of FJ. The following three sections (Sections 2.2, 2.3, and 2.4) present the syntax, semantics, and typing rules of FJ. Section 2.5 develops a type soundness proof. Section 2.6 extends FJ with typecasts, which play an important role in the modeling of GJ. Finally, Section 2.7 summarizes this chapter.

2.1 Overview of Featherweight Java

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

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

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

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

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

In the context of the above definitions, the expression

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

evaluates to the expression

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

There are five forms of expression in FJ. Here, `new A()`, `new B()`, and `new Pair(`e_1`,`e_2`)` are *object constructors*, and e_3`.setfst(`e_4`)` is a *method invocation*. In the body of `setfst`, the expression `this.snd` is a *field access*, and the occurrences of `newfst` and `this` are *variables*. FJ treats `this` as a special variable, which can be a subject of substitution but may not be used as, for example, a method parameter.

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

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

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

Here is the rule for field access in action:

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

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

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

$$\begin{array}{ll} \multicolumn{2}{l}{\texttt{new Pair(new A(), new B()).setfst(new B())}} \\ \longrightarrow & \left[\begin{array}{l}\texttt{new B()/newfst,}\\ \texttt{new Pair(new A(),new B())/this}\end{array}\right] \texttt{new Pair(newfst, this.snd)} \\ \text{i.e.,} & \texttt{new Pair(new B(), new Pair(new A(), new B()).snd)}\end{array}$$

The receiver of the invocation is the object `new Pair(new A(), new B())`, so we look up the `setfst` method in the `Pair` class, where we find that it has formal parameter `newfst` and body `new Pair(newfst, this.snd)`. The invocation reduces to the body with the formal parameter replaced by the actual, and the special variable `this` replaced by the receiver object. This is similar to the beta rule of the lambda calculus, $(\lambda\texttt{x}.\texttt{e}_0)\texttt{e}_1 \longrightarrow [\texttt{e}_1/\texttt{x}]\texttt{e}_0$. The key differences are the fact that the class of the receiver determines where to look for the body (supporting method override), and the substitution of the receiver for `this` (supporting "recursion through self"). Readers familiar with Abadi and Cardelli's Object Calculus will see a strong similarity to their ς-reduction rule [AC96]. In FJ, as in the lambda calculus and the pure Abadi-Cardelli calculus, if a formal parameter appears more than once in the body this may lead duplication of the actual, but since there are no side effects this causes no problems.

There are two ways in which a computation may get stuck: an attempt to access a field not declared for the class (corresponding to the Java exception `NoSuchFieldError`), or an attempt to invoke a method not declared for the class (`NoSuchMethodError`). We will prove a type soundness result, which guarantees that these two errors never happen in well-typed programs, by using a standard technique [WF94].

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

2.2 Syntax and Auxiliary Definitions

We give the syntax for FJ programs and a few auxiliary functions to look up from a class table information used for reduction and typing rules. The metavariables `A`, `B`, `C`, `D`, and `E` range over class names; `f` and `g` range over field names; `m` ranges over method names; `x` ranges over parameter names; `d` and `e` range over expressions; `L` ranges over class declarations; `K` ranges over constructor declarations; and $\texttt{M}$ ranges over method declarations. The syntax of FJ classes is given below:

$$\begin{array}{lcll}
\texttt{L} & ::= & \texttt{class C extends D \{} & \\
 & & \quad \texttt{C}_1\ \texttt{f}_1\texttt{;}\ \cdots\ \texttt{C}_n\ \texttt{f}_n\texttt{;} & \\
 & & \quad \texttt{K} & \\
 & & \quad \texttt{M}_1\ \cdots\ \texttt{M}_k & \\
 & & \texttt{\}} & \\
\texttt{K} & ::= & \texttt{C(C}_1\ \texttt{f}_1\texttt{,...,}\ \texttt{C}_n\ \texttt{f}_n\texttt{) \{} & \\
 & & \quad \texttt{super(f}_1\texttt{,...,f}_k\texttt{);} & \\
 & & \quad \texttt{this.f}_{k+1}\ \texttt{=}\ \texttt{f}_{k+1}\texttt{;}\ \cdots\ \texttt{this.f}_n\ \texttt{=}\ \texttt{f}_n\texttt{; \}} & \\
\texttt{M} & ::= & \texttt{C m(C}_1\ \texttt{f}_1\texttt{,...,}\ \texttt{C}_n\ \texttt{f}_n\texttt{) \{ return e; \}} & \\
\texttt{e} & ::= & \texttt{x} & \text{variables} \\
 & | & \texttt{e.f} & \text{field access} \\
 & | & \texttt{e.m(e}_1\texttt{,...,e}_n\texttt{)} & \text{method invocation} \\
 & | & \texttt{new C(e}_1\texttt{,...,e}_n\texttt{)} & \text{object constructor}
\end{array}$$

We write $\overline{\texttt{f}}$ as shorthand for $\texttt{f}_1\texttt{,...,f}_n$ (and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, etc.) and write $\overline{\texttt{M}}$ as shorthand for $\texttt{M}_1\texttt{...M}_n$ (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 "$\texttt{C}_1\ \texttt{f}_1\texttt{,...,C}_n\ \texttt{f}_n$", and similarly "$\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}$" as shorthand for "$\texttt{C}_1\ \texttt{f}_1\texttt{;...C}_n\ \texttt{f}_n\texttt{;}$", and "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$" as shorthand for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;...this.f}_n\texttt{=f}_n\texttt{;}$". We assume that the set of variables includes the special variable `this`, but that `this` is never used as either the name of a formal parameter of a method, or the name of a field. We write $\texttt{X}_i \in \overline{\texttt{X}}$ when $\texttt{X}_i$ appear in the sequence $\overline{\texttt{X}}$, where `X` stands for a syntactic entry such as a field name or a method definition.

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 `super();` and newly declared fields by $\texttt{this.}\overline{\texttt{f}}\ \texttt{=}\ \overline{\texttt{f}}\texttt{;}$, although, as we will see, those statements do not have significance during execution of programs. A body of a method just returns an expression, which is either a variable, field access, method invocation, or object instantiation.

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. To lighten the notation in what follows, we always assume a *fixed* class table CT.

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

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

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

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

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

The given class table is assumed to satisfy some sanity conditions: (1) sequences of field declarations, parameter names, and method declarations are assumed to contain no duplicate names; (2) $CT(\texttt{C}) = \texttt{class C...}$ for every $\texttt{C} \in dom(CT)$; (3) $\texttt{Object} \notin dom(CT)$; (4) for every class name C (except Object) appearing anywhere in CT, we have $\texttt{C} \in dom(CT)$; and (5) there are no cycles in the subtype relation induced by CT—that is, the <: relation is antisymmetric. By the condition (1), we can identify a class table with a corresponding sequence of class declarations in an obvious way.

For the typing and reduction rules, we need a few auxiliary definitions to look up information on fields or methods from a class table. The fields of a class C, written *fields*(C), is a sequence $\overline{\texttt{C}}\ \overline{\texttt{f}}$ pairing the class of a field with its name, for all the fields declared in class C and all of its superclasses.

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

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

It first looks up the fields $\overline{\texttt{D}}\ \overline{\texttt{g}}$ of the superclass and append the fields defined in the current class $\overline{\texttt{C}}\ \overline{\texttt{f}}$ after it.

2.2.1 Example: With a class table including class Pair,

$$\mathit{fields}(\texttt{Pair}) = \texttt{Object fst, Object snd}$$

holds.

The body of the method m in class C, written *mbody*(m, C), is a pair, written $(\overline{\texttt{x}}\texttt{,e})$, of a sequence of formal parameters $\overline{\texttt{x}}$ and an expression e.

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

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

In case of overriding, from (possibly) several definitions of m, it will find one in the nearest superclass to C.

Similarly, the type of the method m in class C, written *mtype*(m, C), is a pair, written $\overline{\texttt{B}}\to\texttt{B}$, of a sequence of argument types $\overline{\texttt{B}}$ and a result type B.

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

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

2.2.2 Example: With a class table including `Pair` class, we have

$$\mathit{mbody}(\texttt{setfst},\texttt{Pair}) = (\texttt{newfst}, \texttt{new Pair(newfst, this.snd)})$$

and

$$\mathit{mtype}(\texttt{setfst},\texttt{Pair}) = \texttt{Object}{\to}\texttt{Pair}.$$

2.3 Semantics

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

The reduction rules for basic computation are given as follows.

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

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

There are two reduction rules, one for field access, and one for method invocation. These were already explained in the introduction to this chapter. We write $[\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{e}/\texttt{y}]\texttt{e}_0$ for the result of replacing $\texttt{x}_1$ by $\texttt{d}_1$, ..., $\texttt{x}_n$ by $\texttt{d}_n$, and `y` by `e` in expression $\texttt{e}_0$. Note that, since there are no bound variables in expressions, $[\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{e}/\texttt{y}]$ performs just syntactic replacements.

The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules below.

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0'}{\texttt{e}_0\texttt{.f} \longrightarrow \texttt{e}_0'\texttt{.f}} \tag{RC-Field}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0'}{\texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{)} \longrightarrow \texttt{e}_0'\texttt{.m(}\overline{\texttt{e}}\texttt{)}} \tag{RC-Invk-Recv}$$

$$\frac{\texttt{e}_i \longrightarrow \texttt{e}_i'}{\texttt{e}_0\texttt{.m(}\ldots\texttt{,e}_i\texttt{,}\ldots\texttt{)} \longrightarrow \texttt{e}_0\texttt{.m(}\ldots\texttt{,e}_i'\texttt{,}\ldots\texttt{)}} \tag{RC-Invk-Arg}$$

$$\frac{\texttt{e}_i \longrightarrow \texttt{e}_i'}{\texttt{new C(}\ldots\texttt{,e}_i\texttt{,}\ldots\texttt{)} \longrightarrow \texttt{new C(}\ldots\texttt{,e}_i'\texttt{,}\ldots\texttt{)}} \tag{RC-New-Arg}$$

2.4 Typing

An environment, ranged over by Γ, is a finite mapping from variables to types. We write $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{C}}$ for an environment Γ where its domain is $\overline{\mathtt{x}}$ and $\Gamma(\mathtt{x}_i) = \mathtt{C}_i$ for each $\mathtt{x}_i$. The typing judgment for expressions has the form $\Gamma \vdash \mathtt{e} \in \mathtt{C}$, read "in the environment Γ, expression $\mathtt{e}$ has type $\mathtt{C}$."

The typing rules, given below, are syntax directed, with one rule for each form of expression.

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

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

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

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

The typing rules for constructors and method invocations check that each actual parameter has a type that is a subtype of the corresponding formal. Note that, following Java's typing rules, we don't allow subsumption (if $\mathtt{e}$ is given type $\mathtt{C}$, which is a subtype of $\mathtt{D}$, then $\mathtt{e}$ is given type $\mathtt{D}$) in an arbitrary place. 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 $\overline{\mathtt{C}}\ \mathtt{<:}\ \overline{\mathtt{D}}$ as shorthand for $\mathtt{C}_1\ \mathtt{<:}\ \mathtt{D}_1, \ldots, \mathtt{C}_n\ \mathtt{<:}\ \mathtt{D}_n$.

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`," and is derived by the following rule:

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

The method body $\mathtt{e}_0$ must be well typed under the environment where the parameters of the method are given their declared types, plus the special variable `this` with type `C`, in which the method declaration should occur. In case of overriding, if a method with the same name is declared in the superclass then it 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," and is derived by the following rule:

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

It checks that the constructor applies `super` to the fields of the superclass and initializes the fields declared in this class, and that each method declaration in the class is ok.

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

2.5 Properties

FJ typing rules are proved to be sound with respect to reduction. Type soundness consists of two theorems: subject reduction and progress. The first theorem ensures that, if $\mathtt{e}$ is well-typed and reduces to $\mathtt{e}'$, then $\mathtt{e}'$ is also well-typed and given a subtype of $\mathtt{e}$'s type. The second one ensures that a well-typed expression does not include expressions that try to access (or invoke) a non-existing field (or method). Two theorems together guarantee that a well-typed program will never cause errors due to access (or invocation) of a non-existing field (or method).

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

Proof: See below. ■

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

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

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

Proof: Straightforward induction on the derivation of $\Gamma \vdash \mathtt{e} \in \mathtt{C}$. We only show two main cases:

Case T-FIELD: $\quad \mathtt{e} = \mathtt{e}_0.\mathtt{f}_i \quad \mathtt{C} = \mathtt{C}_i \quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e}_0 \in \mathtt{D}_0 \quad \mathit{fields}(\mathtt{D}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$

If $\mathtt{e}_0$ is an object constructor, it must be of the form $\mathtt{new}\ \mathtt{D}_0(\overline{\mathtt{e}})$ because $\Gamma \vdash \mathtt{e}_0 \in \mathtt{D}_0$; the conclusion follows from $\mathit{fields}(\mathtt{D}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$. The case where $\mathtt{e}_0$ is not an object constructor is trivial by using the induction hypothesis.

Case T-METHOD: $\quad \mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \quad \Gamma \vdash \mathtt{e}_0 \in \mathtt{D}_0 \quad \mathit{mtype}(\mathtt{m}, \mathtt{D}_0) = \overline{\mathtt{E}} \to \mathtt{C}$
$\quad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{D}} \quad \overline{\mathtt{D}} <: \overline{\mathtt{E}}$

If $\mathtt{e}_0$ is an object constructor, it must be of the form $\mathtt{new}\ \mathtt{D}_0(\overline{\mathtt{d}})$; it is easy to show $\mathit{mbody}(\mathtt{m}, \mathtt{D}_0) = (\overline{\mathtt{x}}, \mathtt{e}_0)$ and $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{e}})$ from the fact that $\mathit{mtype}(\mathtt{m}, \mathtt{D}_0) = \overline{\mathtt{E}} \to \mathtt{C}$ and $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{E}})$. The case where $\mathtt{e}_0$ is not an object constructor is trivial. ■

Before giving a proof of Theorem 2.5.1, we develop a number of required lemmas.

2.5.3 Lemma: If $\mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{C}} \to \mathtt{C}_0$, then $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{C}} \to \mathtt{C}_0$ for all $\mathtt{C} <: \mathtt{D}$.

Proof: Straightforward induction on the derivation of $\mathtt{C} <: \mathtt{D}$. Note that whether $\mathtt{m}$ is not defined in $CT(\mathtt{C})$ or not, $\mathit{mtype}(\mathtt{m}, \mathtt{C})$ should be the same as $\mathit{mtype}(\mathtt{m}, \mathtt{E})$ where $CT(\mathtt{C}) = \mathtt{class\ C\ extends\ E\ \{...\}}$. ■

2.5.4 Lemma [Term substitution preserves typing]: If $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e} \in \mathtt{D}$, and $\Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{A}}$ where $\overline{\mathtt{A}} <: \overline{\mathtt{B}}$, then $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{C}$ for some $\mathtt{C} <: \mathtt{D}$.

Proof: By induction on the derivation of $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e} \in \mathtt{D}$.

Case T-VAR: $\quad \mathtt{e} = \mathtt{x} \quad \mathtt{D} = \Gamma(\mathtt{x})$

If $\mathtt{x} \notin \overline{\mathtt{x}}$, then it's trivial since $[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x} = \mathtt{x}$. On the other hand, if $\mathtt{x} = \mathtt{x}_i$ and $\mathtt{D} = \mathtt{B}_i$, then, since $[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x} = \mathtt{d}_i$, letting $\mathtt{C} = \mathtt{A}_i$ finishes the case.

Case T-FIELD: $\quad \mathtt{e} = \mathtt{e}_0\mathtt{.f}_i \quad \mathtt{D} = \mathtt{C}_i \quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e}_0 \in \mathtt{D}_0 \quad \mathit{fields}(\mathtt{D}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$

By the induction hypothesis, we have some $\mathtt{C}_0$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 \in \mathtt{C}_0$ and $\mathtt{C}_0$ <: $\mathtt{D}_0$. Then, it is easy to show that

$$\mathit{fields}(\mathtt{C}_0) = \mathit{fields}(\mathtt{D}_0), \overline{\mathtt{D}}\ \overline{\mathtt{g}}$$

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

Case T-INVK: $\quad \mathtt{e} = \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}}) \quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e}_0 \in \mathtt{D}_0 \quad \mathit{mtype}(\mathtt{m}, \mathtt{D}_0) = \overline{\mathtt{E}} \to \mathtt{D}$
$\quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \overline{\mathtt{e}} \in \overline{\mathtt{D}} \quad \overline{\mathtt{D}} <: \overline{\mathtt{E}}$

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

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

By Lemma 2.5.3, $\mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{E}} \to \mathtt{D}$. Moreover, $\overline{\mathtt{C}} <: \overline{\mathtt{E}}$ by transitivity of <:. Therefore, by the rule T-INVK, $\Gamma \vdash ([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0)\mathtt{.m}([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}) \in \mathtt{D}$.

Case T-NEW: $\quad \mathtt{e} = \mathtt{new\ C}(\overline{\mathtt{e}}) \quad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \quad \overline{\mathtt{C}} <: \overline{\mathtt{D}}$

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

■

2.5.5 Lemma [Weakening]: If $\Gamma \vdash \mathtt{e} \in \mathtt{C}$, then $\Gamma, \mathtt{x} : \mathtt{D} \vdash \mathtt{e} \in \mathtt{C}$.

Proof: By straightforward induction on the derivation of $\Gamma \vdash \mathtt{e} \in \mathtt{C}$. ■

2.5.6 Lemma: If $\mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{D}$, and $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0) = (\overline{\mathtt{x}}, \mathtt{e})$, then there exist some $\mathtt{C}$ and $\mathtt{D}_0$ such that $\mathtt{C} <: \mathtt{D}$ and $\mathtt{C}_0 <: \mathtt{D}_0$ and $\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{D}_0 \vdash \mathtt{e} \in \mathtt{C}$.

Proof: By induction on the derivation of $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0)$. The base case, where $\mathtt{m}$ is defined in $CT(\mathtt{C}_0)$, is easy since it must be the case that $\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{C}_0 \vdash \mathtt{e} \in \mathtt{C}$. by T-CLASS and T-METHOD. The induction step is also straightforward. ■

Proof of Theorem 2.5.1: By induction on a derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$, with a case analysis on the reduction rule used.

Case R-FIELD: $\quad \mathtt{e} = (\mathtt{new\ C}_0(\overline{\mathtt{e}}))\mathtt{.f}_i \quad \mathtt{e}' = \mathtt{e}_i \quad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$

By the rule T-FIELD, we have

$$\begin{array}{l} \Gamma \vdash \mathtt{new\ C}_0(\overline{\mathtt{e}}) \in \mathtt{D}_0 \\ \mathtt{C} = \mathtt{D}_i. \end{array}$$

for some $\mathtt{D}_0$. Again, by the rule T-NEW,

$$\begin{array}{l} \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \\ \overline{\mathtt{C}} <: \overline{\mathtt{D}} \\ \mathtt{D}_0 = \mathtt{C}_0 \end{array}$$

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

Case R-Invk: $\texttt{e} = \texttt{(new C}_0\texttt{(}\overline{\texttt{e}}\texttt{)).m(}\overline{\texttt{d}}\texttt{)}$
$\texttt{e}' = [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new C}_0\texttt{(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}_0$
$\mathit{mbody}(\texttt{m}, \texttt{C}_0) = (\overline{\texttt{x}}, \texttt{e}_0)$

By the rules T-Invk and T-New, we have

$\Gamma \vdash \texttt{new C}_0\texttt{(}\overline{\texttt{e}}\texttt{)} \in \texttt{C}_0$
$\mathit{mtype}(\texttt{m}, \texttt{C}_0) = \overline{\texttt{D}}\rightarrow\texttt{C}$
$\Gamma \vdash \overline{\texttt{d}} \in \overline{\texttt{C}}$
$\overline{\texttt{C}} \mathrel{<:} \overline{\texttt{D}}.$

By Lemma 2.5.6, $\overline{\texttt{x}} : \overline{\texttt{D}}, \texttt{this} : \texttt{D}_0 \vdash \texttt{e}_0 \in \texttt{B}$ for some $\texttt{D}_0$ and $\texttt{B}$ where $\texttt{C}_0 \mathrel{<:} \texttt{D}_0$ and $\texttt{B} \mathrel{<:} \texttt{C}$. By Lemma 2.5.5, $\Gamma, \overline{\texttt{x}} : \overline{\texttt{D}}, \texttt{this} : \texttt{D}_0 \vdash \texttt{e}_0 \in \texttt{B}$. Then, by Lemma 2.5.4, $\Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new C}_0\texttt{(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}_0 \in \texttt{E}$ for some $\texttt{E} \mathrel{<:} \texttt{B}$. By transitivity of <:, $\texttt{E} \mathrel{<:} \texttt{C}$. Finally, letting $\texttt{C}' = \texttt{E}$ finishes this case.

Cases for congruence rules are easy. ∎

2.6 Extension with Typecasts

In this section, we discuss a small but interesting extension for FJ—that is, FJ with typecasts, which will serve as a base language when a model of GJ is built in Chapter 4. As we will see later, unlike FJ, this extension raises an interesting technical problem in proving type soundness.

The class `Pair` shown in Section 2.1 is not so useful as one might expect: since the types of `fst` and `snd` fields are `Object`, we cannot perform any interesting operations on a result of field access. For example, the expression

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

is ill typed because the expression `new Pair(...).fst` is given type `Object`, which has no fields, even though it can successfully reduce to `new B()`.

One solution for this problem is to declare different classes for each type of values to be stored, for example, `PairOfAA` for pairs of `A` objects, `PairOfAB` for pairs of an `A` object and a `B` object, and so on. Another solution is use of typecasts, which can change the static type of an expression. In Java, an expression `(C)e` is given type `C`, and, at run-time, it checks whether `e` evaluates to an object of type `C` (or subtype of `C`). Thus, using typecasts, we can rewrite the expression above to the expression below:

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

Now that `(Pair)new Pair(new Pair(new A(), new B()), new A()).fst` is given type `Pair`, the whole expression is well-typed. Here is the rule for a cast in action:

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

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

In what follows, we give the additional syntax, semantics, and typing rules for FJ with typecasts. As we will see later, typecasts introduce a subtlety into type soundness proof.

Syntax: First, we add typecast expressions to the syntax:

$$
\begin{array}{lll}
\texttt{e} & ::= & \\
 & \vdots & \\
 & | & \texttt{(C)e}
\end{array}
$$

As in Java, we assume that casts bind less tightly than other forms of expression. For example, `(C)e.f` means `(C)(e.f)`, not `((C)e).f`.

Reduction: As mentioned above, typecasts will succeed when the class of the constructor is a subclass of the target of the cast. We also require a congruence rule for typecasts.

$$\frac{\texttt{C <: D}}{\texttt{(D)(new C(}\overline{\texttt{e}}\texttt{))} \longrightarrow \texttt{new C(}\overline{\texttt{e}}\texttt{)}} \qquad \text{(R-CAST)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0{}'}{\texttt{(C)e}_0 \longrightarrow \texttt{(C)e}_0{}'} \qquad \text{(RC-CAST)}$$

Typing: There are three rules, shown below, for type casts: in an *upcast* the subject is a subclass of the target (T-UCAST), in a *downcast* the target is a subclass of the subject (T-DCAST), and in a *stupid* cast the target is unrelated to the subject. In Java, the target type of a typecast must be comparable with the (static) type of the subject by the subtyping relation. For example, both `(Object)new A()` and `(A)new Object()` are allowed but `(A)new B()` is not. Thus, the Java compiler rejects as ill typed an expression containing a stupid cast, but we must allow stupid casts in FJ if we are to formulate type soundness as a subject reduction theorem for a small-step semantics. This is because a sensible expression may be reduced to one containing a stupid cast. For example, consider the following reduction step:

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

We indicate the special nature of stupid casts by including the hypothesis *stupid warning* in the type rule for stupid casts (T-SCAST); an expression signaling stupid warning must fail when the typecast is to be performed. An FJ typing corresponds to a legal Java typing only if it does not contain this rule. (Stupid casts were omitted from the first published version of Classic Java [FKF98a], causing its published proof of type soundness to be incorrect; this error was discovered independently by ourselves and the Classic Java authors [FKF98b].)

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

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

$$\frac{\Gamma \vdash \texttt{e}_0 \in \texttt{D} \qquad \texttt{C} \not<: \texttt{D} \qquad \texttt{D} \not<: \texttt{C} \qquad \textit{stupid warning}}{\Gamma \vdash \texttt{(C)e}_0 \in \texttt{C}} \qquad \text{(T-SCAST)}$$

Properties: This extension also enjoys the same properties, including subject reduction and progress, as stated in Section 2.5. Since the proof method is also the same, we show only interesting parts that involve stupid casts.

2.6.1 Lemma [Term substitution preserves typing]: If $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e} \in \mathtt{D}$, and $\Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{A}}$ where $\overline{\mathtt{A}} <: \overline{\mathtt{B}}$, then $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{C}$ for some $\mathtt{C} <: \mathtt{D}$.

Proof: By induction on the derivation of $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e} \in \mathtt{D}$ with a case analysis on the last rule used. We show only the cases additional to the proof of Lemma 2.5.4.

Case T-UCast: $\mathtt{e} = \mathtt{(D)e_0} \quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e_0} \in \mathtt{C} \quad \mathtt{C} <: \mathtt{D}$

By the induction hypothesis, we have some $\mathtt{E}$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e_0} \in \mathtt{E}$ and $\mathtt{E} <: \mathtt{C}$. Moreover, $\mathtt{E} <: \mathtt{D}$ by transitivity of $<:$; finally, $\Gamma \vdash \mathtt{(D)}([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e_0}) \in \mathtt{D}$ is derived by the rule T-UCast.

Case T-DCast: $\mathtt{e} = \mathtt{(D)e_0} \quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e_0} \in \mathtt{C} \quad \mathtt{D} <: \mathtt{C} \quad \mathtt{D} \neq \mathtt{C}$

By the induction hypothesis, we have some $\mathtt{E}$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e_0} \in \mathtt{E}$ and $\mathtt{E} <: \mathtt{C}$. If $\mathtt{E} <: \mathtt{D}$ or $\mathtt{D} <: \mathtt{E}$, then $\Gamma \vdash \mathtt{(D)}([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e_0}) \in \mathtt{D}$ by the rule T-UCast or T-DCast, respectively. On the other hand, if both $\mathtt{D} \not<: \mathtt{E}$ and $\mathtt{E} \not<: \mathtt{D}$ hold, then $\Gamma \vdash \mathtt{(D)}([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e_0}) \in \mathtt{D}$ with *stupid warning* by the rule T-SCast.

Case T-SCast: $\mathtt{e} = \mathtt{(D)e_0} \quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e_0} \in \mathtt{C} \quad \mathtt{D} \not<: \mathtt{C} \quad \mathtt{C} \not<: \mathtt{D}$

By the induction hypothesis, we have some $\mathtt{E}$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e_0} \in \mathtt{E}$ and $\mathtt{E} <: \mathtt{C}$.

Now we show $\mathtt{D} \not<: \mathtt{E}$ and $\mathtt{E} \not<: \mathtt{D}$ by contradiction. Suppose $\mathtt{E} <: \mathtt{D}$. By using that fact that, for every class $\mathtt{F}$, there is only one class $\mathtt{G}$ such that $CT(\mathtt{F}) =$ `class F extends G`, we can show that either proof of $\mathtt{E} <: \mathtt{C}$ or $\mathtt{E} <: \mathtt{D}$ is a part of the other proof; it means $\mathtt{C} <: \mathtt{D}$ or $\mathtt{D} <: \mathtt{C}$, which contradicts the assumption. Therefore, $\mathtt{E} \not<: \mathtt{D}$. Furthermore, since $\mathtt{D} <: \mathtt{E}$ leading to $\mathtt{D} <: \mathtt{C}$ contradicts the assumption, $\mathtt{D} \not<: \mathtt{E}$.

Finally, $\Gamma \vdash \mathtt{(D)}([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e_0}) \in \mathtt{D}$ with *stupid warning*, by the rule T-SCast. ∎

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

Proof: By induction on a derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$, with a case analysis on the reduction rule used. We show the cases R-Cast and RC-Cast here; otherwise the proof is the same as before (Theorem 2.5.1).

Case R-Cast: $\mathtt{e} = \mathtt{(D)(new\ C_0(\overline{e}))} \quad \mathtt{e}' = \mathtt{new\ C_0(\overline{e})} \quad \mathtt{C_0} <: \mathtt{D}$

The proof of $\Gamma \vdash \mathtt{(D)(new\ C_0(\overline{e}))} \in \mathtt{C}$ must end with the rule T-UCast since the derivation ending with T-SCast or T-DCast contradicts the assumption $\mathtt{C_0} <: \mathtt{D}$. By the rule T-UCast, we have $\Gamma \vdash \mathtt{new\ C_0(\overline{e})} \in \mathtt{C_0}$ and $\mathtt{D} = \mathtt{C}$, which finishes the case.

Case RC-Cast: $\mathtt{e} = \mathtt{(D)e_0} \quad \mathtt{e}' = \mathtt{(D)e_0}' \quad \mathtt{e_0} \longrightarrow \mathtt{e_0}'$

We have three subcases according to the last type rule used.

Subcase T-UCast: $\Gamma \vdash \mathtt{e_0} \in \mathtt{C_0} \quad \mathtt{C_0} <: \mathtt{D} \quad \mathtt{D} = \mathtt{C}$

By the induction hypothesis, $\Gamma \vdash \mathtt{e_0}' \in \mathtt{C_0}'$ for some $\mathtt{C_0}' <: \mathtt{C_0}$. By transitivity of $<:$, $\mathtt{C_0}' <: \mathtt{C}$. Therefore, by the rule T-UCast, $\Gamma \vdash \mathtt{(C)e_0}' \in \mathtt{C}$ (without any additional *stupid warning*).

Subcase T-DCast: $\Gamma \vdash \mathtt{e_0} \in \mathtt{C_0} \quad \mathtt{D} <: \mathtt{C_0} \quad \mathtt{D} = \mathtt{C}$

By the induction hypothesis, $\Gamma \vdash \mathtt{e_0}' \in \mathtt{C_0}'$ for some $\mathtt{C_0}' <: \mathtt{C_0}$. If $\mathtt{C_0}' <: \mathtt{C}$ or $\mathtt{C} <: \mathtt{C_0}'$, then $\Gamma \vdash \mathtt{(C)e_0}' \in \mathtt{C}$ by the rule T-UCast or T-DCast (without any additional *stupid warning*). On the other hand, if both $\mathtt{C_0}' \not<: \mathtt{C}$ and $\mathtt{C} \not<: \mathtt{C_0}'$, then, $\Gamma \vdash \mathtt{(C)e_0}' \in \mathtt{C}$ with *stupid warning* by the rule T-SCast.

Subcase T-SCast: $\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0$ $\quad \mathtt{D} \not<: \mathtt{C}_0$ $\quad \mathtt{C}_0 \not<: \mathtt{D}$ $\quad \mathtt{D} = \mathtt{C}$
By the induction hypothesis, $\Gamma \vdash \mathtt{e}_0{}' \in \mathtt{C}_0{}'$ for some $\mathtt{C}_0{}' <: \mathtt{C}_0$. Then, both $\mathtt{C}_0{}' \not<: \mathtt{C}$ and $\mathtt{C} \not<: \mathtt{C}_0{}'$ also hold. Therefore $\Gamma \vdash \mathtt{(C)e}_0{}' \in \mathtt{C}$ with *stupid warning.* ■

As mentioned before, these properties above guarantee that a well-typed program never cause `NoSuchFieldError` or `NoSuchMethodError` during execution, but they do not say anything about typecasts; in fact, the proofs of Lemma 2.6.1 and Theorem 2.6.2 show that a downcast expression, which is typed by T-DCast, may reduce to a stupid cast expression, which is typed by T-SCast.

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

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

Proof: By induction on a derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$, with a case analysis on the reduction rule used. Note that, the derivation of $\mathtt{e}'$ will have additional *stupid warning* only if the derivation of `e` (and *CT*) uses the rules T-DCast and/or T-SCast. ■

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

Proof: Trivial from the fact that the subexpression $\mathtt{(C)new\ C}_0\mathtt{(\overline{e})}$ is given type `C` by the rule T-UCast. ■

2.7 Summary

We have presented Featherweight Java, a core language for Java. Its design strongly favors compactness instead of completeness; we have dropped as many features as possible while still capturing flavors of computation and the type system in Java. As a result, the proof of soundness is both concise and straightforward, making it a suitable arena for the study of ambitious extensions to the type system, discussed in the following chapters.

As a small extension, we have added typecasts, which are required to model compilation of GJ to JVML, and proved type soundness of the extended language. In order to type soundness for semantics based on a small-step reduction relation, the type system requires the stupid cast rule, which is not a part of Java's typing rules. This small but interesting technicality has not been pointed out in the previous work of core languages for Java.

Chapter 3

FJI: Featherweight Java with Inner Classes

In this chapter, we discuss an extension of FJ with inner classes in the style of Java 1.1. Just like nested definitions of functions, found in functional languages, nested structure of classes allows inner classes to access directly to the members of their enclosing definitions. Such nested structures can be found in a few object-oriented languages. For example, Smalltalk [GR83] has special syntax for "block" objects, similar to anonymous functions, and Beta [MMPN93] provides patterns, unifying classes and functions, which can be nested arbitrarily.

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. Such situations are typical in user-interface programming: for example, Java's Abstract Window Toolkit [CL97] 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 Java 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.

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 [Jav97] 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 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 details.

We will clarify the essential features of inner classes as follows:

- First, we define an extension of FJ with inner classes, called FJI, and give a direct operational semantics and typing rules. The reduction rules show the inherent complexity of inner classes. 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 control annotations (`public`/`private`/etc.) and inner classes (cf. [Jav97, BP]).

- Next, we give a translation from FJI to FJ, 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 [Aba98] and Bhowmik and Pugh [BP].)

FJI omits assignments as well as FJ; of course, most *useful* examples of programming with inner classes do involve the side-effecting features of Java. In fact, 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. But, 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.

The remainder of the chapter is organized as follows. We begin with an informal overview of FJI and a discussion how tricky inner classes can be in the presence of inheritance in Section 3.1. Then, we proceed to a formal definition of FJI, consisting of its syntax (3.2), two styles of semantics (3.4 and 3.5), and typing rules (3.6). In Section 3.7, we prove standard type soundness for direct semantics, type preservation of translation semantics, and equivalence of two semantics. Like FJ, FJI imposes some syntactic restriction, not found in Java, to make the definition of semantics easier. For example, a field access to the current object must be written `this.f` (as in FJ), instead of just `f`, and types for inner classes must always be written in a full path form, such as `Counter.Listener`, instead of an abbreviation `Listener` even in the definition of the class `Counter`. Section 3.8 discusses elaboration process from an abbreviated program to an FJI program. We also summarize an underspecification, which caused to have different meanings in different versions of JDK, in Section 3.9.

3.1 Overview of FJI

3.1.1 Enclosing Instances

Before proceeding to formal definitions of FJI, we begin with an informal discussion of FJI and its the key idea of *enclosing instances*, and argue that interactions between inner classes and subclassing are trickier than one would expect.

This is a sample 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 field `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 $\texttt{C}_1.\cdots.\texttt{C}_n$ can refer to $n-1$ enclosing instances, $\texttt{C}_1$`.this` to $\texttt{C}_{n-1}$`.this`, as well as the usual `this`, which can also be written $\texttt{C}_n$`.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 `e.new<T>` $\texttt{C}(\overline{\texttt{e}})$, where `e` is the enclosing instance and `T` is the static type of `e`. The result of `e.new<T>` $\texttt{C}(\overline{\texttt{e}})$ is always an instance of `T.C`, regardless of the run-time type of `e`. 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 `e`. (By contrast, inner classes in Beta are *virtual* [MMP89], 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`.)

Like FJ, FJI imposes some syntactic restrictions, which is not found in Java, 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 `e.new C(ē)` is annotated with the static type `T` of `e`, written `e.new<T> C(ē)`. 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*. 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. Section 3.8 describes how to recover omitted information.

3.1.2 Inner Classes and Inheritance

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 3.1 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.

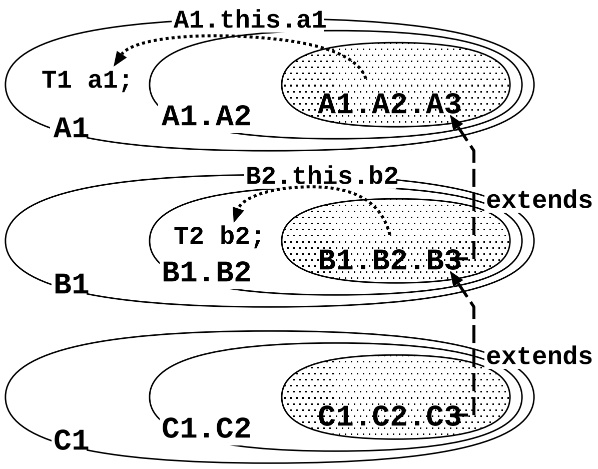

Figure 3.1: A chain of environments

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 `e` of the `new` expression. For example, a `C1.C2.C3` object is instantiated by writing $\texttt{e.new<C1.C2> C3(}\overline{\texttt{e}}\texttt{)}$, where `e` 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 `e` in $\texttt{e.new<C1.C2> C3(}\overline{\texttt{e}}\texttt{)}$ has the form $\texttt{new C1(}\overline{\texttt{c}}\texttt{).new<C1> C2(}\overline{\texttt{d}}\texttt{)}$, which includes the enclosing instance $\texttt{new C1(}\overline{\texttt{c}}\texttt{)}$ 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. (Thus, in Java, extra arguments are not mandatory; moreover, such qualifications can be even omitted when they are `C.this`.) 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 $\mathtt{C_1.\cdots.C_n}$ 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.)

3.2 Syntax

Now, we proceed to the formal definitions of FJI. 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 $\mathtt{C_1},\ldots,\mathtt{C_n}$ concatenated by periods). For compactness in the definitions, we introduce the notation $\star$ for a "null qualification" and identify $\star.\mathtt{C}$ with `C`. The metavariable `P` ranges over types (`T`) and $\star$. We write $\mathtt{C} \in \mathtt{P}$ if $\mathtt{P} = \mathtt{C_1.\cdots.C_n}$ and $\mathtt{C} = \mathtt{C_i}$ for some i.

The abstract syntax of the language is as follows:

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

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 $\mathtt{super(\overline{f})}$ (or $\mathtt{f.super(\overline{f})}$); (2) the enclosing instance of the superclass (if needed); and (3) the fields of the class to be defined, initialized by $\mathtt{this.\overline{f}=\overline{f}}$. 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.

In FJI, a class table CT is a mapping from types T to class declarations L. From the class table, we can read off the subtype relation between classes, as in FJ. 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. The formal rules are given below:

$$\mathtt{T} <: \mathtt{T}$$

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

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

We impose the following sanity conditions on the class table: (1) $CT(\mathtt{P.C}) = \mathtt{class\ C\ ...}$ for every $\mathtt{P.C} \in dom(CT)$. (2) If $CT(\mathtt{P.C})$ has an inner class declaration L of name D, then $CT(\mathtt{P.C.D}) = \mathtt{L}$. (3) $\mathtt{Object} \notin dom(CT)$. (4) For every type T (except Object) appearing anywhere in CT, we have $\mathtt{T} \in dom(CT)$. (5) For every $\mathtt{e_0.new{<}T{>}\ C(\overline{e})}$ (and $\mathtt{new\ C(\overline{e})}$, respectively) appearing anywhere in CT, we have $\mathtt{T.C} \in dom(CT)$ (and $\mathtt{C} \in dom(CT)$, respectively). (6) There are no cycles in the subtyping relation. (7) $\mathtt{T} \not<: \mathtt{T.U}$, 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.

3.3 Auxiliary Definitions

We need to extend the auxiliary definitions such as *fields* to deal with inner classes. Two functions *mtype* and *mbody* are essentially the same as before, except that *mbody*(`m`, `C`) returns the qualified class name where the definition of `m` is found, as well as the formal arguments and the method body:

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

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

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

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

The fields of a class `T`, written *fields*(`T`), are given by the following rules:

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

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

$$\frac{\begin{array}{c} CT(\texttt{T}) = \texttt{class C extends C}_1\texttt{.}\cdots\texttt{.C}_n\ \texttt{\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \\ fields(\texttt{C}_1\texttt{.}\cdots\texttt{.C}_n) = \overline{\texttt{S}}\ \overline{\texttt{g}} \\ \texttt{f}_0 = \texttt{this\$C}_1\texttt{\$}\cdots\texttt{\$C}_n \end{array}}{fields(\texttt{T}) = \overline{\texttt{S}}\ \overline{\texttt{g}}, \texttt{C}_1\texttt{.}\cdots\texttt{.C}_{n-1}\ \texttt{f}_0, \overline{\texttt{T}}\ \overline{\texttt{f}}}$$

It collects not only the type of each field with its name but also the types of (direct) enclosing instances of all the superclasses of `T`. For example, *fields*(`C1.C2.C3`) returns the following sequence:

fields(`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$B2$B2,`	(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 $\overline{\texttt{S}}\ \overline{\texttt{g}}$ of the superclass $\texttt{C}_1\texttt{.}\cdots\texttt{.C}_n$ 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`.

Finally, the function $\mathit{encl}_{\mathtt{T}}(\mathtt{e})$, which plays a crucial role in the semantics of FJI, is defined below:

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

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

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ U.D\ \{\overline{S}\ \overline{f};\ \ldots\}} \qquad \#(\overline{\mathtt{f}}) = \#(\overline{\mathtt{e}})}{\mathit{encl}_{\mathtt{T}}(\mathtt{new\ C(\overline{d},\ d_0,\ \overline{e})}) = \mathit{encl}_{\mathtt{T}}(\mathtt{d}_0\mathtt{.new<U>\ D(\overline{d})})}$$

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

$$\frac{\mathit{CT}(\mathtt{S.C}) = \mathtt{class\ C\ extends\ U.D\ \{\overline{S}\ \overline{f};\ \ldots\}} \qquad \#(\overline{\mathtt{f}}) = \#(\overline{\mathtt{e}}) \qquad \mathtt{T} \neq \mathtt{S.C}}{\mathit{encl}_{\mathtt{T}}(\mathtt{e}_0\mathtt{.new<S>\ C(\overline{d},\ d_0,\ \overline{e})}) = \mathit{encl}_{\mathtt{T}}(\mathtt{d}_0\mathtt{.new<U>\ D(\overline{d})})}$$

Intuitively, when `e` is an object 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`). The first rule is the simplest case: since the type of an expression $\mathtt{e}_0\mathtt{.new<T>\ C(\overline{e})}$ 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 of the argument $\mathtt{e}_0\mathtt{.new<S>\ C(\overline{d},\ d_0,\ \overline{e})}$, the enclosing instance $\mathtt{e}_0$ is not the correct answer. We therefore make a recursive call with an object $\mathtt{d}_0\mathtt{.new<U>\ D(\overline{d})}$ 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:

$$\begin{array}{ll} & \mathit{encl}_{\mathtt{A1.A2.A3}}(\mathtt{e.new<C1.C2>\ C3(a,\ new\ A1().new<A1>\ A2(),} \\ & \qquad\qquad \mathtt{b,\ new\ B1().new<B1>\ B2(),\ c)} \\ = & \mathit{encl}_{\mathtt{A1.A2.A3}}(\mathtt{new\ B1().new<B1>\ B2.new<B1.B2>\ B3(a,\ new\ A1().new<A1>\ A2(),\ b)}) \\ = & \mathit{encl}_{\mathtt{A1.A2.A3}}(\mathtt{new\ A1().new<A1>\ A2().new<A1.A2>\ A3(a)}) \\ = & \mathtt{new\ A1().new<A1>\ A2()} \end{array}$$

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

3.4 Direct Semantics

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 $\longrightarrow$. First, the rules for computation are as follows:

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

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

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

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

There are four computation 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; $\mathtt{e.new{<}T{>}\ C}(\overline{\mathtt{e}}).\mathtt{f}_i$ behaves similarly. The method invocation expression $\mathtt{new\ C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})$ 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 $\mathtt{new\ C}(\overline{\mathtt{e}})$, 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 $\mathtt{C}_{n-1}.\mathtt{this}$ is obtained by $\mathit{encl}_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e})$, where e is the receiver object; similarly, $\mathtt{C}_{n-2}.\mathtt{this}$ is obtained by $\mathit{encl}_{\mathtt{C}_1.\cdots.\mathtt{C}_{n-1}}(\mathit{encl}_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e}))$, and so on. Similarly for $\mathtt{e.new{<}T{>}\ C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})$. The context rules to apply a computation rule at any point in an expression are given below:

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0.\mathtt{f} \longrightarrow \mathtt{e}_0{}'.\mathtt{f}} \qquad \text{(IRC-Field)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0{}'.\mathtt{m}(\overline{\mathtt{e}})} \qquad \text{(IRC-Invk-Recv)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{e}_0.\mathtt{m}(\dots,\mathtt{e}_i,\dots) \longrightarrow \mathtt{e}_0.\mathtt{m}(\dots,\mathtt{e}_i{}',\dots)} \qquad \text{(IRC-Invk-Arg)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{new\ C}(\dots,\mathtt{e}_i,\dots) \longrightarrow \mathtt{new\ C}(\dots,\mathtt{e}_i{}',\dots)} \qquad \text{(IRC-Top-Arg)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0.\mathtt{new{<}T{>}\ C}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0{}'.\mathtt{new{<}T{>}\ C}(\overline{\mathtt{e}})} \qquad \text{(IRC-Inner-Enc)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{e}_0.\mathtt{new{<}T{>}\ C}(\dots,\mathtt{e}_i,\dots) \longrightarrow \mathtt{e}_0.\mathtt{new{<}T{>}\ C}(\dots,\mathtt{e}_i{}',\dots)} \qquad \text{(IRC-Inner-Arg)}$$

For example, if the class table includes Outer and RefinedInner of this chapter and Pair, A, and B from the previous chapter, then

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

reduces to `new B()` as follows,

$$\begin{array}{ll} & \underline{\texttt{new RefinedInner(new Outer(new Pair(new A(), new B())), x).snd\_p()}} \\ \longrightarrow & \underline{\texttt{new Outer(new Pair(new A(), new B())).p}}\texttt{.snd} \\ \longrightarrow & \underline{\texttt{new Pair(new A(), new B()).snd}} \\ \longrightarrow & \texttt{new B()} \end{array}$$

where the underlined subexpressions are the redices at each reduction step.

3.5 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 $|\cdot|_{\star}$).

3.5.1 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 $|\overline{\mathtt{L}}|_{\mathtt{P}}$).

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

As we saw above, a compiled inner class has one additional field, called `this$`$|\mathtt{T}|$, where `T` is the original class name. $\mathtt{C}_i$`.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 $\mathtt{e}_0$`.new<T> C(`$\overline{\mathtt{e}}$`)` 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 $|\mathtt{T}_1|\ \mathtt{x}_1,\ldots,|\mathtt{T}_n|\ \mathtt{x}_n$.

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

3.5.2 Constructors and Classes

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

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

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

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

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

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 argument used for the qualification of `f.super(f̄)` becomes the last argument of the `super()` invocation.

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

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

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

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 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., $\left|\overline{\mathtt{L}}\right|_{\star}$).

3.6 Typing

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`." We abbreviate sequences of typing or subtyping judgments as before, 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 rules are given as follows:

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

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

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

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

$$\frac{\mathit{fields}(\mathtt{T.C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \mathtt{e}_0 \in \mathtt{S} \qquad \mathtt{S} <: \mathtt{T} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} <: \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{e}_0\mathtt{.new<T>\ C(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T.C}} \qquad \text{(IT-NewInner)}$$

The typing rules of FJI are a straightforward extension of those of FJ. The typing rules for object instantiations and method invocations check that each actual parameter has a type which is 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 $\mathtt{M\ OK\ IN\ C}_1\mathtt{.}\cdots\mathtt{.C}_n$, read "method declaration `M` is ok if it is declared in class $\mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n$," and given by the following rule, which is also a straightforward extension of the rule T-Method:

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

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`,” and given by the following rules:

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

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

We have two cases according to whether the class extends a top-level class or an inner class; if `P` is a type `T`, the class declaration `L` is an inner class, and 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

3.7.1 Properties of Direct Semantics

It is easy to show that FJI programs enjoy standard subject reduction and progress properties, exactly like FJ programs do.

3.7.1.1 Theorem [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 $\mathtt{T}' <: \mathtt{T}$.

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

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

(2) If `e` includes $\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})$ as a subexpression, then $\mathit{mbody}(\mathtt{m},\mathtt{C}_0) = (\overline{\mathtt{x}},\mathtt{e}_0,\mathtt{C}_1.\cdots.\mathtt{C}_n)$ and $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$ and $\mathtt{c}_1,\ldots,\mathtt{c}_n$ appearing in the rule IR-InvkT are well defined.

    Similarly, if `e` includes $\mathtt{e}_0.\mathtt{new}\texttt{<}\mathtt{T}_0\texttt{>}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})$ as a subexpression, then $\mathit{mbody}(\mathtt{m},\mathtt{T}_0.\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{d}_0,\mathtt{C}_1.\cdots.\mathtt{C}_n)$ and $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$ and $\mathtt{c}_1,\ldots,\mathtt{c}_n$ appearing in the rule IR-InvkI are well defined.

Moreover, FJI is a conservative extension of FJ. In the statement of the theorem, We use subscripts FJ and FJI to show which set of rules is used.

3.7.1.3 Theorem [FJI is a conservative extension of FJ]: Let $\mathtt{e}$ be an FJ expression and the underlyingCT be an FJ class table. Then, $\Gamma \vdash_{\mathrm{FJ}} \mathtt{e} \in \mathtt{C}$ if and only if $\Gamma \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{C}$. Moreover, $\mathtt{e} \longrightarrow_{\mathrm{FJ}} \mathtt{e}'$ if and only if $\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{e}'$.

Proof: In the first half, both directions are shown by straightforward induction on the derivation of $\Gamma \vdash_{\mathrm{FJ}} \mathtt{e} \in \mathtt{C}$ or $\Gamma \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{C}$. In the second half, both directions are shown by induction on a derivation of the reduction relation. ∎

In what follows, we develop proofs of Theorem 3.7.1.1 and Theorem 3.7.1.2 after proofs of required lemmas.

3.7.1.4 Lemma: 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 $\Gamma \vdash \mathtt{e} \in \mathtt{T}$. ∎

3.7.1.5 Lemma: 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 $\mathtt{S} <: \mathtt{T}$. Note that whether $\mathtt{m}$ is not defined in $CT(\mathtt{S})$ or not, $mtype(\mathtt{m}, \mathtt{S})$ should be the same as $mtype(\mathtt{m}, \mathtt{U})$ where $CT(\mathtt{S}) =$ `class C extends U ...`. ∎

3.7.1.6 Lemma: If $\Gamma \vdash \mathtt{new\ C(\overline{e})} \in \mathtt{T}$ and $\mathtt{T} <: \mathtt{U.D}$, then $\Gamma \vdash encl_{\mathtt{U.D}}(\mathtt{new\ C(\overline{e})}) \in \mathtt{S}$ for some $\mathtt{S}$ such that $\mathtt{S} <: \mathtt{U}$. Similarly, if $\Gamma \vdash \mathtt{e}_0\mathtt{.new<T_0>\ C(\overline{e})} \in \mathtt{T}$ and $\mathtt{T} <: \mathtt{U.D}$, then $\Gamma \vdash encl_{\mathtt{U.D}}(\mathtt{e}_0\mathtt{.new<T_0>\ C(\overline{e})}) \in \mathtt{S}$ 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 IT-NewInner, we have

$$\begin{array}{ll} \mathtt{T}_0 = \mathtt{U} & \mathtt{C} = \mathtt{D} \\ \Gamma \vdash \mathtt{e}_0 \in \mathtt{S}_0 & \mathtt{S}_0 <: \mathtt{T}_0 \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} & \overline{\mathtt{S}} <: fields(\mathtt{T}_0\mathtt{.C}) \end{array}$$

Since $encl_{\mathtt{U.D}}(\mathtt{e}_0\mathtt{.new<T_0>\ C(\overline{e})}) = \mathtt{e}_0$, letting $\mathtt{S} = \mathtt{S}_0$ finishes the case.
Case: $CT(\mathtt{T}) = \mathtt{class\ C\ extends\ T'\ \{\overline{U}\ \overline{g};\ ...\}}$ $\quad \mathtt{T}' <: \mathtt{U.D}$
We have four subcases depending on whether $\mathtt{T}$ (or $\mathtt{T}'$) is a simple name or not. We only show a subcase where $\mathtt{T} = \mathtt{C}$ and $\mathtt{T}' = \mathtt{S}_0\mathtt{.D}_0$ since the other cases are similar. It suffices to show the first part where we have $\Gamma \vdash \mathtt{new\ C(\overline{e})} \in \mathtt{C}$. (We never have $\Gamma \vdash \mathtt{e}_0\mathtt{.new<T_0>\ C(\overline{e})} \in \mathtt{C}$.) By the rule IT-NewTop, we have

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

Also, we have $\overline{\mathtt{T}}\ \overline{\mathtt{f}} = fields(\mathtt{S}_0\mathtt{.D}_0)$, $\mathtt{S}_0\ \mathtt{this\$S_0\$D_0}$, $\overline{\mathtt{U}}\ \overline{\mathtt{g}}$. Thus, $\Gamma \vdash \mathtt{d}_0\mathtt{.new<S_0>\ D_0(\overline{d})} \in \mathtt{T}_0\mathtt{.D}_0$ where $\overline{\mathtt{e}} = \overline{\mathtt{d}}, \mathtt{d}_0, \overline{\mathtt{c}}$ and $\#(\overline{\mathtt{c}}) = \#(\overline{\mathtt{g}})$. On the other hand, by definition,

$$encl_{\mathtt{U.D}}(\mathtt{new\ C(\overline{d},\ d_0,\ \overline{c})}) = encl_{\mathtt{U.D}}(\mathtt{d}_0\mathtt{.new<T_0>\ D_0(\overline{d})}).$$

Finally, the induction hypothesis finishes the case. ∎

3.7.1.7 Lemma [Term Substitution]: If $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{U}} \vdash \mathtt{e} \in \mathtt{T}$ and $\Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{T}}$ where $\overline{\mathtt{T}} <: \overline{\mathtt{U}}$, then $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{S}$ and $\mathtt{S} <: \mathtt{T}$.

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

Case IT-Var: $\mathtt{e} = \mathtt{x}$ $\quad \mathtt{T} = \Gamma(\mathtt{x})$
If $\mathtt{x} \notin \overline{\mathtt{x}}$, then it's trivial since $[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x} = \mathtt{x}$. On the other hand, if $\mathtt{x} = \mathtt{x}_i$ and $\mathtt{T} = \mathtt{U}_i$, then, since $[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x} = \mathtt{d}_i$, letting $\mathtt{S} = \mathtt{T}_i$ finishes the case.
Case IT-Field: $\mathtt{e} = \mathtt{e}_0\mathtt{.f}_i \quad \mathtt{T} = \mathtt{S}_i \quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{U}} \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \mathit{fields}(\mathtt{T}_0) = \overline{\mathtt{S}}\ \overline{\mathtt{f}}$
By the induction hypothesis, we have some $\mathtt{S}_0$ such that $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 \in \mathtt{S}_0$ and $\mathtt{S}_0 \mathrel{<:} \mathtt{T}_0$. It is easy to show that

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

for some $\overline{\mathtt{D}}\ \overline{\mathtt{g}}$. Therefore, by the rule IT-Field, $\Gamma \vdash ([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0)\mathtt{.f}_i \in \mathtt{S}_i$.
Case IT-Invk: $\mathtt{e} = \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{U}} \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{V}} \to \mathtt{T}$
$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{U}} \vdash \overline{\mathtt{e}} \in \overline{\mathtt{T}} \quad \overline{\mathtt{T}} \mathrel{<:} \overline{\mathtt{V}}$
By the induction hypothesis, we have some $\mathtt{S}_0$ and $\overline{\mathtt{S}}$ such that

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

By Lemma 3.7.1.5, $\mathit{mtype}(\mathtt{m},\ \mathtt{S}_0) = \overline{\mathtt{V}} \to \mathtt{T}$. Moreover, $\overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{V}}$ by transitivity of $<:$. Therefore, by the rule IT-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 IT-NewTop: $\mathtt{e} = \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \quad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \quad \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{U}} \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \quad \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}}$
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 $\overline{\mathtt{V}} \mathrel{<:} \overline{\mathtt{S}}$. Moreover $\overline{\mathtt{V}} \mathrel{<:} \overline{\mathtt{T}}$, by transitivity of $<:$. Therefore, by the rule IT-NewTop, $\Gamma \vdash \mathtt{new\ C(}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}$.
The case for IT-NewInner is similar. ■

3.7.1.8 Lemma: If $\mathit{mtype}(\mathtt{m},\ \mathtt{T}) = \overline{\mathtt{U}} \to \mathtt{U}_0$ and $\mathit{mbody}(\mathtt{m},\ \mathtt{T}) = (\overline{\mathtt{x}},\ \mathtt{e}_0,\ \mathtt{C}_1.\cdots.\mathtt{C}_n)$, then $\mathtt{T} \mathrel{<:} \mathtt{C}_1.\cdots.\mathtt{C}_n$ 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 $\mathtt{T}_0$ where $\mathtt{T}_0 \mathrel{<:} \mathtt{U}_0$.

Proof: By induction on the derivation of $\mathit{mbody}(\mathtt{m}, \mathtt{T})$. The base case (where $\mathtt{m}$ is defined in $\mathtt{T}$ 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 $\mathtt{T}_0 \mathrel{<:} \mathtt{U}_0$ by IT-Method. The induction step also straightforward. ■

Proof of Theorem 3.7.1.1: By induction on a derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$, with a case analysis on the reduction rule used.
Case IR-FieldT: $\mathtt{e} = \mathtt{(new\ C}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)).f}_i \quad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \quad \mathtt{e}' = \mathtt{e}_i$
By the rule IT-Field, we have

$$\begin{array}{l} \Gamma \vdash \mathtt{new\ C}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T}_0 \\ \mathtt{C} = \mathtt{T}_i \end{array}$$

for some $\mathtt{T}_0$. Again, by the rule IT-NewTop,

$$\begin{array}{l} \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \\ \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}} \\ \mathtt{T}_0 = \mathtt{C}_0 \end{array}$$

In particular, $\Gamma \vdash \mathtt{e}_i \in \mathtt{S}_i$, finishing the case since $\mathtt{S}_i \mathrel{<:} \mathtt{T}_i$.
Case IR-FieldI:
Similar to the case for IR-FieldT.

Case IR-InvkT: $\mathtt{e} = \mathtt{(new\ C_0(\overline{e})).m(\overline{d})}$
$mbody(\mathtt{m}, \mathtt{C_0}) = (\overline{\mathtt{x}}, \mathtt{e_0}, \mathtt{C_1.\cdots.C_n})$
$$\mathtt{c}_i = \begin{cases} \mathtt{new\ C_0(\overline{e})} & (i = n) \\ encl_{\mathtt{C_1.\cdots.C_{i+1}}}(\mathtt{c}_{i+1}) & (i \in 1 \ldots n-1) \end{cases}$$
$\mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{c}_n/\mathtt{this},\ \mathtt{c}_i/\mathtt{C}_i\mathtt{.this}\ ^{i\in 1\ldots n}]\mathtt{e_0}$

By the rules IT-Invk and IT-NewTop, we have

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

By Lemma 3.7.1.8,

$$\overline{\mathtt{x}} : \overline{\mathtt{D}},\ \mathtt{this} : \mathtt{C_1.\cdots.C_n},\ \mathtt{C}_i\mathtt{.this} : \mathtt{C_1.\cdots.C}_i\ ^{i\in 1\ldots n} \vdash \mathtt{e_0} \in \mathtt{S_0}$$

where $\mathtt{S_0} \mathrel{<:} \mathtt{T}$ and $\mathtt{C} \mathrel{<:} \mathtt{C_1.\cdots.C_n}$. By Lemma 3.7.1.4,

$$\Gamma,\ \overline{\mathtt{x}} : \overline{\mathtt{D}},\ \mathtt{this} : \mathtt{C_1.\cdots.C_n},\ \mathtt{C}_i\mathtt{.this} : \mathtt{C_1.\cdots.C}_i\ ^{i\in 1\ldots n} \vdash \mathtt{e_0} \in \mathtt{S_0}.$$

By using the fact that $\mathtt{C} \mathrel{<:} \mathtt{C_1.\cdots.C_n}$ and Lemma 3.7.1.6 repeatedly, we have $\overline{\mathtt{U}}$ such that $\Gamma \vdash \mathtt{c}_i \in \mathtt{U}_i$ and $\mathtt{U}_i \mathrel{<:} \mathtt{C_1.\cdots.C}_i$ for $i \in 1 \ldots n$. Then, by Lemma 3.7.1.7,

$$\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{c}_n/\mathtt{this},\ \mathtt{c}_i/\mathtt{C}_i\mathtt{.this}\ ^{i\in 1\ldots n}]\mathtt{e_0} \in \mathtt{U_0}$$

for some $\mathtt{U_0} \mathrel{<:} \mathtt{S_0}$. Finally, letting $\mathtt{T}' = \mathtt{U_0}$ finishes this case.

Case IR-InvkI:

Similar to the case IR-InvkT. Cases for congruence rules (IRC-$\cdots$) are straightforward. ∎

Proof of Theorem 3.7.1.2: (1) If $\mathtt{e}$ has $\mathtt{new\ C_0(\overline{e}).f}$ (or $\mathtt{e_0.new\text{<}T_0\text{>}\ C(\overline{e}).f}$) as a subexpression, then, by well-typedness of the subexpression, it's easy to check that $fields(\mathtt{C_0})$ (or $fields(\mathtt{T_0.C})$) is well-defined and $\mathtt{f}$ appears in it.

(2) If $\mathtt{e}$ has $\mathtt{new\ C_0(\overline{e}).m(\overline{d})}$ as a subexpression, then, it's also easy to show $mbody(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e_0}, \mathtt{C_0.\cdots.C_n})$ and $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$ from the fact that $mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{D}} \rightarrow \mathtt{D}$ where $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{D}})$. Finally, by Lemma 3.7.1.6,

$$encl_{\mathtt{C_1.\cdots.C_n}}(\mathtt{e_0.new\text{<}T_0\text{>}\ C(\overline{e}).m(\overline{d})}),$$
$$\vdots$$
$$encl_{\mathtt{C_1}}(\cdots encl_{\mathtt{C_1.\cdots.C_n}}(\mathtt{e_0.new\text{<}T_0\text{>}\ C(\overline{e}).m(\overline{d})}))$$

are well defined. Similarly for a subexpression of the form $\mathtt{e_0.new\text{<}T_0\text{>}\ C(\overline{e}).m(\overline{d})}$. ∎

3.7.2 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 3.7.2.1). 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 3.7.2.2) and vice versa (Theorem 3.7.2.3).

3.7.2.1 Theorem [Compilation preserves typing]: When $\Gamma = \bar{\mathtt{x}} : \bar{\mathtt{T}}$, we write $|\Gamma|$ for $\bar{\mathtt{x}} : \left|\bar{\mathtt{T}}\right|$. If an FJI class table CT is ok and $\bar{\mathtt{x}} : \bar{\mathtt{T}}, \mathtt{this} : \mathtt{C}_1.\cdots.\mathtt{C}_n, \mathtt{C}_i.\mathtt{this} : \mathtt{C}_1.\cdots.\mathtt{C}_i \;{}^{i\in 1\ldots n} \vdash_{\text{FJI}} \mathtt{e} \in \mathtt{T}$ with respect to CT, then $|CT|$ is ok and $\bar{\mathtt{x}} : \left|\bar{\mathtt{T}}\right|, \mathtt{this} : |\mathtt{C}_1.\cdots.\mathtt{C}_n| \vdash_{\text{FJ}} |\mathtt{e}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n} \in |\mathtt{T}|$ with respect to $|CT|$.

3.7.2.2 Theorem [Compilation commutes with reduction]: If $\Gamma \vdash_{\text{FJI}} \mathtt{e} \in \mathtt{T}$ where $dom(\Gamma)$ does not include `this` or `C.this` for any `C`, and $\mathtt{e} \longrightarrow_{\text{FJI}} \mathtt{e}'$, then $|\mathtt{e}|_\star \longrightarrow_{\text{FJ}}{}^* |\mathtt{e}'|_\star$. In other words, the following diagram commutes.

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

3.7.2.3 Theorem [Compilation preserves termination]: If $\Gamma \vdash_{\text{FJI}} \mathtt{e} \in \mathtt{T}$ where $dom(\Gamma)$ does not include `this` or `C.this`, and $|\mathtt{e}|_\star \longrightarrow_{\text{FJ}} \mathtt{e}'$, then $\mathtt{e} \longrightarrow_{\text{FJI}} \mathtt{e}''$ for some $\mathtt{e}''$, and $\mathtt{e}' \longrightarrow_{\text{FJ}}{}^* |\mathtt{e}''|_\star$. In other words, the following diagram commutes.

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

Unfortunately, Theorems 3.7.2.2 and 3.7.2.3 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, but, in the compiled FJ program, `C.this` is translated to `this` followed by a sequence of field access, which is not necessarily an expression to be evaluated next. Thus, for call-by-value FJI, another technique for proving equivalence, such as contextual equivalence [Plo77], is required to show correctness of compilation.

Before giving their proofs, we develop a number of required lemmas.

3.7.2.4 Lemma: Suppose $|CT|$ is well defined. (1) If $\mathit{fields}_{\text{FJI}}(\mathtt{C}) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}$, then $\mathit{fields}_{\text{FJ}}(|\mathtt{C}|) = \left|\bar{\mathtt{T}}\right|\ \bar{\mathtt{f}}$. (2) If $\mathit{fields}_{\text{FJI}}(\mathtt{T.C}) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}$ then $\mathit{fields}_{\text{FJ}}(|\mathtt{T.C}|) = \left|\bar{\mathtt{T}}\right|\ \bar{\mathtt{f}}, |\mathtt{T}|\ \mathtt{this\$}|\mathtt{T.C}|$.

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

3.7.2.5 Lemma: If $|CT|$ is well defined and $mtype_{\text{FJI}}(\texttt{m}, \texttt{T}) = \overline{\texttt{T}}\!\to\!\texttt{U}$, then $mtype_{\text{FJ}}(\texttt{m}, |\texttt{T}|) = |\overline{\texttt{T}}| \to |\texttt{U}|$.

Proof: By induction on the derivation of $mtype_{\text{FJI}}(\texttt{m}, \texttt{T})$ with an analysis of the compilation rules for classes and methods. ■

3.7.2.6 Lemma: If $|CT|$ is well defined and $mbody_{\text{FJI}}(\texttt{m}, \texttt{T}) = (\overline{\texttt{x}}, \texttt{e}_0, \texttt{S})$, then $mbody_{\text{FJ}}(\texttt{m}, |\texttt{T}|) = (\overline{\texttt{x}}, |\texttt{e}_0|_{\texttt{S}})$.

Proof: By induction on the derivation of $mbody_{\text{FJI}}(\texttt{m}, \texttt{T})$ with an analysis of compilation rules for classes and methods. ■

3.7.2.7 Lemma: $\texttt{S} <:_{\text{FJI}} \texttt{T}$ if and only if $|\texttt{S}| <:_{\text{FJ}} |\texttt{T}|$.

Proof: Straightforward induction on the derivation of $\texttt{S} <:_{\text{FJI}} \texttt{T}$. ■

Proof of Theorem 3.7.2.1: We prove the theorem in three steps: first, we show $|CT|$ is well defined; second, it is shown that, if $\Gamma, \texttt{this} : \texttt{C}_1.\cdots.\texttt{C}_n, \texttt{C}_i\texttt{.this} : \texttt{C}_1.\cdots.\texttt{C}_i\ ^{i\in 1\ldots n} \vdash_{\text{FJI}} \texttt{e} \in \texttt{T}$ with respect to CT, then $|\Gamma|, \texttt{this} : |\texttt{C}_1.\cdots.\texttt{C}_n| \vdash_{\text{FJ}} |\texttt{e}|_{\texttt{C}_1.\cdots.\texttt{C}_n} \in |\texttt{T}|$ 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 $\texttt{C.this}$ that $\texttt{C} \not\in \texttt{S}$ where $\texttt{S}$ is the class name to which the method belongs. Note that well-definedness of $|CT|$ implies well-definedness of auxiliary functions.

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

Case IT-Field:
$\texttt{e} = \texttt{e}_0\texttt{.f}_i$
$\Gamma, \texttt{this} : \texttt{C}_1.\cdots.\texttt{C}_n, \texttt{C}_i\texttt{.this} : \texttt{C}_1.\cdots.\texttt{C}_i\ ^{i\in 1\ldots n} \vdash_{\text{FJI}} \texttt{e}_0 \in \texttt{T}_0$
$fields_{\text{FJI}}(\texttt{T}_0) = \overline{\texttt{T}}\ \overline{\texttt{f}}$
$\texttt{T} = \texttt{T}_i$

By the induction hypothesis, $|\Gamma|, \texttt{this} : |\texttt{C}_1.\cdots.\texttt{C}_n| \vdash_{\text{FJ}} |\texttt{e}_0|_{\texttt{C}_1.\cdots.\texttt{C}_n} \in |\texttt{T}_0|$. By Lemma 3.7.2.4, $fields_{\text{FJ}}(|\texttt{T}_0|) = \ldots, |\texttt{T}_i|\ \ \texttt{f}_i, \ldots$. Then, the rule IT-Field finishes the case.

Case IT-NewInner:
$\texttt{e} = \texttt{e}_0\texttt{.new<T}_0\texttt{> C(}\overline{\texttt{e}}\texttt{)}$
$\Gamma, \texttt{this} : \texttt{C}_1.\cdots.\texttt{C}_n, \texttt{C}_i\texttt{.this} : \texttt{C}_1.\cdots.\texttt{C}_i\ ^{i\in 1\ldots n} \vdash_{\text{FJI}} \texttt{e}_0 \in \texttt{S}_0$
$\texttt{S}_0 <:_{\text{FJI}} \texttt{T}_0$
$fields_{\text{FJI}}(\texttt{T}_0\texttt{.C}) = \overline{\texttt{T}}\ \overline{\texttt{f}}$
$\Gamma, \texttt{this} : \texttt{C}_1.\cdots.\texttt{C}_n, \texttt{C}_i\texttt{.this} : \texttt{C}_1.\cdots.\texttt{C}_i\ ^{i\in 1\ldots n} \vdash_{\text{FJI}} \overline{\texttt{e}} \in \overline{\texttt{S}}$
$\overline{\texttt{S}} <:_{\text{FJI}} \overline{\texttt{T}}$
$\texttt{T} = \texttt{T}_0\texttt{.C}$

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

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

and

$$|\Gamma|, \texttt{this} : |\texttt{C}_1.\cdots.\texttt{C}_n| \vdash_{\text{FJ}} |\overline{\texttt{e}}|_{\texttt{C}_1.\cdots.\texttt{C}_n} \in |\overline{\texttt{S}}|.$$

By Lemma 3.7.2.4, $fields_{\text{FJ}}(|\texttt{T}_0\texttt{.C}|) = |\overline{\texttt{T}}|\ \ \overline{\texttt{f}}', |\texttt{T}_0|\ \ \texttt{this\$}|\texttt{T}_0\texttt{.C}|$. Since $|\texttt{S}_0| <:_{\text{FJ}} |\texttt{T}_0|$ and $|\overline{\texttt{S}}| <:_{\text{FJ}} |\overline{\texttt{T}}|$ by Lemma 3.7.2.7, the rule T-New finishes the case.

Finally, the third step is proved by analyzing the derivation of $\vdash_{\text{FJI}} \texttt{L OK IN P}$ with the result of the second step. ■

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

$$|\mathtt{new\ C}(\overline{\mathtt{e}})|_\star.\mathtt{this\$C_1\$\cdots\$C}_n \longrightarrow_{\text{FJ}} |encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{new\ C}(\overline{\mathtt{e}}))|_\star.$$

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

$$|\mathtt{e}_0\mathtt{.new<S>\ C}(\overline{\mathtt{e}})|_\star.\mathtt{this\$C_1\$\cdots\$C}_n \longrightarrow_{\text{FJ}} |encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e}_0\mathtt{.new<S>\ C}(\overline{\mathtt{e}}))|_\star.$$

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

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

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 $fields_{\text{FJ}}(|\mathtt{S.C}|) = |\overline{\mathtt{T}}|\ \ \overline{\mathtt{f}},\ |\mathtt{S}|\ \ \mathtt{this\$C_1\$\cdots\$C}_n$ for some $\overline{\mathtt{T}}$ and $\overline{\mathtt{f}}$,

$$|\mathtt{e}_0\mathtt{.new<S>\ C}(\overline{\mathtt{e}})|_\star.\mathtt{this\$C_1\$\cdots\$C}_n \longrightarrow_{\text{FJ}} |\mathtt{e}_0|_\star.$$

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: $CT(\mathtt{T}) = \mathtt{class\ C\ extends\ U} \qquad \mathtt{U} <:_{\text{FJI}} \mathtt{C}_1.\cdots.\mathtt{C}_n$

We have four cases depending on whether `T` (respectively `U`) is a top-level class or not. We show the case where $\mathtt{T} = \mathtt{C}$ and $\mathtt{U} = \mathtt{U}_0\mathtt{.D}$ for some `C`, $\mathtt{U}_0$ and `D` since the other cases are similar. Since $\Gamma \vdash_{\text{FJI}} \mathtt{new\ C}(\overline{\mathtt{e}}) \in \mathtt{C}$, we have $\Gamma \vdash_{\text{FJI}} \mathtt{e}_0\mathtt{.new<U_0>\ D}(\overline{\mathtt{d}}) \in \mathtt{U}_0\mathtt{.D}$ for some $\mathtt{e}_0$ and $\overline{\mathtt{d}}$ such that $\overline{\mathtt{e}} = \overline{\mathtt{d}}, \mathtt{e}_0, \overline{\mathtt{c}}$ and $\#(\overline{\mathtt{c}})$ is equal to the number of fields declared in `C`. By the induction hypothesis,

$$\left|\mathtt{e}_0\mathtt{.new<U_0>\ D}(\overline{\mathtt{d}})\right|_\star.\mathtt{this\$C_1\$\cdots\$C}_n \longrightarrow_{\text{FJ}} \left|encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e}_0\mathtt{.new<U_0>\ D}(\overline{\mathtt{d}}))\right|_\star.$$

It is easy to show that

$$|\mathtt{new\ C}(\overline{\mathtt{e}})|_\star.\mathtt{this\$C_1\$\cdots\$C}_n \longrightarrow_{\text{FJ}} \left|encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e}_0\mathtt{.new<U_0>\ D}(\overline{\mathtt{d}}))\right|_\star.$$

By definition,

$$encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e}_0\mathtt{.new<U_0>\ D}(\overline{\mathtt{d}})) = encl_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{new\ C}(\overline{\mathtt{e}}))$$

finishing the case. ∎

Proof of Theorem 3.7.2.2: By induction on the derivation of $\mathtt{e} \longrightarrow_{\text{FJI}} \mathtt{e}'$ with a case analysis on the last rule used. We show only the cases for IR-FieldT and IR-InvkT since the other base cases are similar to either of them. The cases for congruence rules are straightforward.

Case IR-FieldT: $\quad \mathtt{e} = \mathtt{new\ C}(\overline{\mathtt{e}})\mathtt{.f}_i \qquad \mathtt{e}' = \mathtt{e}_i \qquad fields_{\text{FJI}}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$

By Lemma 3.7.2.4, $fields_{\text{FJ}}(|\mathtt{C}|) = \ldots, |\mathtt{T}_i|\ \ \mathtt{f}_i, \ldots$, and thus,

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

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

By Lemma 3.7.2.6, $\mathit{mbody}_{\mathrm{FJ}}(\mathtt{m}, |\mathtt{C}|) = (\overline{\mathtt{x}}, |\mathtt{e}_0|_{\mathtt{C}_1.\cdots.\mathtt{C}_n})$. Thus,

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

Since $|\mathtt{C}_i\mathtt{.this}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n} = \mathtt{this.this\$C_1\$}\cdots\mathtt{\$C}_n.\cdots.\mathtt{this\$C_1\$}\cdots\mathtt{\$C}_i$, by Lemma 3.7.2.8,

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

■

Proof of Theorem 3.7.2.3: 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: $|\mathtt{e}|_\star = \mathtt{new\ C(\overline{e}).f}_i \qquad \mathit{fields}_{\mathrm{FJ}}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}} \qquad \mathtt{e}' = \mathtt{e}_i$

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 3.7.1.2 and Lemma 3.7.2.4, we have $\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{e}_i{}'$, and $|\mathtt{e}_i{}'|_\star = \mathtt{e}_i$, finishing the case.

Case R-Invk:
$$\begin{array}{l}
|\mathtt{e}|_\star = \mathtt{new\ C(\overline{e}).m(\overline{d})} \\
\mathit{mbody}_{\mathrm{FJ}}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0) \\
\mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C(\overline{e})/this}]\mathtt{e}_0
\end{array}$$

By inspecting compilation rules, e must be method invocation on a top-level object or an inner class object. Moreover, e is well typed; by Theorem 3.7.1.2, we have $\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{e}''$ by using IR-InvokeI or IR-InvokeT. In either case, by Theorem 3.7.2.2, we have $|\mathtt{e}|_\star \longrightarrow_{\mathrm{FJ}}{}^* |\mathtt{e}''|_\star$. Then, it is easy to check that

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

by Lemma 3.7.2.6. (Refer to the case for IR-InvokeT in the proof of Theorem 3.7.2.2.)

Case RC-New-Arg: $|\mathtt{e}|_\star = \mathtt{new\ C(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \qquad \mathtt{e}_i \longrightarrow_{\mathrm{FJ}} \mathtt{e}_i{}' \qquad \mathtt{e}' = \mathtt{new\ C(}\ldots\mathtt{,e}_i{}'\mathtt{,}\ldots\mathtt{)}$

By inspecting compilation rules, e must be an object constructor. We have three subcases according to the form of e.

Subcase:
$$\begin{array}{ll}
\mathtt{e} = \mathtt{d}_0\mathtt{.new{<}T_0{>}\ C_0(}\ldots\mathtt{,d}_i\mathtt{,}\ldots\mathtt{)} & |\mathtt{d}_j|_\star = \mathtt{e}_j\ (j \in \{1,\ldots n-1\}) \\
|\mathtt{d}_0|_\star = \mathtt{e}_n &
\end{array}$$

By the induction hypothesis, $\mathtt{d}_i \longrightarrow_{\mathrm{FJI}} \mathtt{d}_i{}'$ for some $\mathtt{d}_i{}'$. By IRC-Inner-Arg,

$$\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{d}_0\mathtt{.new{<}T_0{>}\ C_0(}\ldots\mathtt{,d}_i{}'\mathtt{,}\ldots\mathtt{)}.$$

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. ■

3.8 Elaboration of Source Programs

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 superclasses, 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(`$\overline{\mathtt{e}}$`)` 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 $\mathtt{T}'$. 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 T(`$\overline{\mathtt{e}}$`)` 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`).

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

3.8.1 Pre-elaborated Syntax

In pre-elaborated programs, pre-elaborated names `X` are used where types are required; it is allowed to write a field access `f` and a method invocation $\texttt{m}(\overline{\texttt{e}})$ without a receiver, and constructor invocation $\texttt{new X}(\overline{\texttt{e}})$ of pre-elaborated type name without an enclosing instance. We assume the sanity conditions, only (1)–(3); (4) will be automatically satisfied when elaboration succeeds (see Theorem 3.8.3.1 (1)) and (5)–(7) can be checked after the elaboration of types.

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

We do not deal with omitted qualifications of `super` constructor invocations. An explicit qualification using a constructor argument is needed.

3.8.2 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. The step (2) must be performed after (1) since elaboration of expressions require type names (in particular `X` after `extends`) to be elaborated.

Elaboration of Types

First, we need an auxiliary function to look up the definition of an inner class in enclosing classes. The function *type-encl*(`C`, `P`) defined below returns the innermost enclosing class name (or $\star$) where the inner class of name `C` is defined:

$$\frac{\texttt{P.C} \in dom(CT)}{\textit{type-encl}(\texttt{C}, \texttt{P}) = \texttt{P}} \qquad \frac{\texttt{P.D.C} \notin dom(CT)}{\textit{type-encl}(\texttt{C}, \texttt{P.D}) = \textit{type-encl}(\texttt{C}, \texttt{P})}$$

The elaboration relation for types is written $\texttt{P} \vdash \texttt{X} \Rightarrow \texttt{T}$, read as "`X` is elaborated to `T` in `P`." We write *type-encl*(`C`, `P`) $\uparrow$, which means there is no `T` such that *type-encl*(`C`, `P`) = `T`. Similarly for the other functions. We also write $\texttt{P} \vdash \texttt{X} \Uparrow$, which means there is no `T` such that $\texttt{P} \vdash \texttt{X} \Rightarrow \texttt{T}$. The key rules are ET-SimpEncl and ET-SimpSuper. The rule ET-SimpEncl is used when the ambiguous name `D` is defined in neither the current class `P.C` nor superclasses ($\texttt{P} \vdash \texttt{X.D} \Uparrow$); it is resolved in the context of the direct enclosing class `P`. On the other hand, ET-SimpSuper is used when the ambiguous name `D` is defined in a superclass ($\texttt{P} \vdash \texttt{X.D} \Rightarrow \texttt{T}$). Then, the nearest declaration in enclosing classes, if any, should be the same ($\textit{type-encl}(\texttt{D}, \texttt{P}) = \texttt{U}$ and $\texttt{U.D} = \texttt{T}$).

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

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

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

$$\frac{\begin{array}{c}\texttt{P.C.D} \notin \mathit{dom}(CT) \\ CT(\texttt{P.C}) = \texttt{class C extends X \{...\}} \qquad \texttt{P} \vdash \texttt{X.D} \Rightarrow \texttt{T} \\ \textit{type-encl}(\texttt{D}, \texttt{P}) \uparrow \text{ or } (\textit{type-encl}(\texttt{D}, \texttt{P}) = \texttt{U} \text{ and } \texttt{U.D} = \texttt{T})\end{array}}{\texttt{P.C} \vdash \texttt{D} \Rightarrow \texttt{T}} \qquad \text{(ET-SimpSuper)}$$

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

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

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-LongSuper, it tries to find `T` such that $\star \vdash \texttt{A.B.B} \Rightarrow \texttt{T}$ since `A` does not have `B` and `A.B` is specified as a superclass of `A`; it then tries to find $\texttt{T}'$ such that $\star \vdash \texttt{A.B.B.B} \Rightarrow \texttt{T}'$ and so on. To prevent divergence, an elaboration algorithm should detect circularity by keeping previous inputs for recursive calls: in this example, the algorithm will try to find `T` such that $\star \vdash \texttt{A.B} \Rightarrow \texttt{T}$ twice.

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.

Similarly to elaboration of types, we need auxiliary functions to lookup a field/method definition in enclosing classes. The functions $\textit{field-encl}(\texttt{f}, \texttt{T})$ and $\textit{meth-encl}(\texttt{m}, \texttt{T})$ defined below returns which enclosing class has the declaration of the simple name `f` or `m`, when it is mentioned in `T`. (Note that, unlike $\textit{type-encl}(\texttt{C}, \texttt{P})$, the second argument of $\textit{field-encl}(\texttt{f}, \texttt{T})$ and $\textit{meth-encl}(\texttt{m}, \texttt{T})$ cannot be $\star$ since fields and methods are always defined in some class.) The function $\textit{subty-encl}(\texttt{U}, \texttt{C}_1.\cdots.\texttt{C}_n)$, used in elaboration of constructor invocations, returns a simple name $\mathtt{C}_i$ of the innermost enclosing class such that $\mathtt{C}_1.\cdots.\mathtt{C}_i$ <: U.

$$\frac{\mathtt{f}\text{ is defined in }CT(\mathtt{T})}{\textit{field-encl}(\mathtt{f},\,\mathtt{T}) = \mathtt{T}} \qquad \frac{\mathtt{f}\text{ is not defined in }CT(\mathtt{T.D})}{\textit{field-encl}(\mathtt{f},\,\mathtt{T.D}) = \textit{field-encl}(\mathtt{f},\,\mathtt{T})}$$

$$\frac{\mathtt{m}\text{ is defined in }CT(\mathtt{T})}{\textit{meth-encl}(\mathtt{m},\,\mathtt{T}) = \mathtt{T}} \qquad \frac{\mathtt{m}\text{ is not defined in }CT(\mathtt{T.D})}{\textit{meth-encl}(\mathtt{m},\,\mathtt{T.D}) = \textit{meth-encl}(\mathtt{m},\,\mathtt{T})}$$

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

We also need auxiliary functions to lookup a field/method definition in superclasses. The functions *field-super*(f, T) and *meth-super*(m, T) defined below return the path of the nearest superclass where a field/method name f or m is defined.

$$\frac{\mathtt{f}\text{ is defined in }CT(\mathtt{T})}{\textit{field-super}(\mathtt{f},\,\mathtt{T}) = \mathtt{T}} \qquad \frac{\begin{array}{c}\mathtt{f}\text{ is not defined in }CT(\mathtt{T.D})\\ CT(\mathtt{T.D}) = \mathtt{class\ D\ extends\ S\ \{...\}}\end{array}}{\textit{field-super}(\mathtt{f},\,\mathtt{T.D}) = \textit{field-super}(\mathtt{f},\,\mathtt{S})}$$

$$\frac{\mathtt{m}\text{ is defined in }CT(\mathtt{T})}{\textit{meth-super}(\mathtt{m},\,\mathtt{T}) = \mathtt{T}} \qquad \frac{\begin{array}{c}\mathtt{m}\text{ is not defined in }CT(\mathtt{T.D})\\ CT(\mathtt{T.D}) = \mathtt{class\ D\ extends\ S\ \{...\}}\end{array}}{\textit{meth-super}(\mathtt{m},\,\mathtt{T.D}) = \textit{meth-super}(\mathtt{m},\,\mathtt{S})}$$

Then, we can define the functions *whereis*(f, T) and *whereis*(m, T) to resolve the field/method name binding. Given a field/method name and the path where the name is referred to, they return a simple name C of the innermost enclosing class that has the definition of the field/method in itself or one of its superclasses. The rules below closely follow the algorithm described in the introduction to this section, including check of ambiguity.

$$\frac{\mathtt{S} = \textit{field-super}(\mathtt{f},\,\mathtt{P.C}) \qquad \textit{field-encl}(\mathtt{f},\,\mathtt{P.C})\uparrow \text{ or } \textit{field-encl}(\mathtt{f},\,\mathtt{P.C}) = \mathtt{S}}{\textit{whereis}(\mathtt{f},\,\mathtt{P.C}) = \mathtt{C}} \tag{EF-This}$$

$$\frac{\textit{field-super}(\mathtt{f},\,\mathtt{T.C})\uparrow \qquad \textit{whereis}(\mathtt{f},\,\mathtt{T}) = \mathtt{D}}{\textit{whereis}(\mathtt{f},\,\mathtt{T.C}) = \mathtt{D}} \tag{EF-Encl}$$

$$\frac{\mathtt{S} = \textit{meth-super}(\mathtt{m},\,\mathtt{P.C}) \qquad \textit{meth-encl}(\mathtt{m},\,\mathtt{P.C})\uparrow \text{ or } \textit{meth-encl}(\mathtt{m},\,\mathtt{P.C}) = \mathtt{S}}{\textit{whereis}(\mathtt{m},\,\mathtt{P.C}) = \mathtt{C}} \tag{EM-This}$$

$$\frac{\textit{meth-super}(\mathtt{m},\,\mathtt{T.C})\uparrow \qquad \textit{whereis}(\mathtt{m},\,\mathtt{T}) = \mathtt{D}}{\textit{whereis}(\mathtt{m},\,\mathtt{T.C}) = \mathtt{D}} \tag{EM-Encl}$$

Elaboration relation of 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." Thanks to auxiliary functions, most rules are straightforward. Note that the elaborated expression is not an FJI expression yet. The static types of the enclosing instances in inner class constructors may be still omitted; they are recovered during typechecking.

$$\frac{\mathtt{x} \in \overline{\mathtt{x}}}{\mathtt{T};\overline{\mathtt{x}} \vdash \mathtt{x} \Rightarrow \mathtt{x}} \tag{E-Var}$$

$$\frac{\mathtt{f} \notin \overline{\mathtt{x}} \qquad \mathit{whereis}(\mathtt{f}, \mathtt{T}) = \mathtt{C}}{\mathtt{T}; \overline{\mathtt{x}} \vdash \mathtt{f} \Rightarrow \mathtt{C.this.f}} \tag{E-FieldSimp}$$

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

$$\frac{\mathit{whereis}(\mathtt{m}, \mathtt{T}) = \mathtt{C} \qquad \mathtt{T}; \overline{\mathtt{x}} \vdash \overline{\mathtt{e}} \Rightarrow \overline{\mathtt{e}}'}{\mathtt{T}; \overline{\mathtt{x}} \vdash \mathtt{m}(\overline{\mathtt{e}}) \Rightarrow \mathtt{C.this.m}(\overline{\mathtt{e}}')} \tag{E-InvkSimp}$$

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

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

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

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

$$\mathtt{T}; \overline{\mathtt{x}} \vdash \mathtt{this} \Rightarrow \mathtt{this} \tag{E-This}$$

$$\frac{\mathtt{C} \in \mathtt{T}}{\mathtt{T}; \overline{\mathtt{x}} \vdash \mathtt{C.this} \Rightarrow \mathtt{C.this}} \tag{E-QlThis}$$

Elaboration of Methods and Classes

Elaboration of methods, written $\mathtt{T} \vdash \mathtt{M} \Rightarrow \mathtt{M}'$ read "method M in class T is elaborated to M', just replaces the method body since return type and argument types are already elaborated.

$$\frac{\mathtt{T}; \overline{\mathtt{x}} \vdash \mathtt{e} \Rightarrow \mathtt{e}'}{\mathtt{T} \vdash \mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return\ e;}\ \} \Rightarrow \mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return\ e';}\ \}} \tag{E-Method}$$

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.

$$\frac{\mathtt{P.C} \vdash \overline{\mathtt{L}} \Rightarrow \overline{\mathtt{L}}' \qquad \mathtt{P.C} \vdash \overline{\mathtt{M}} \Rightarrow \overline{\mathtt{M}}'}{\mathtt{P} \vdash \mathtt{class\ C\ extends\ T}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{L}}\ \overline{\mathtt{M}}\ \} \Rightarrow \mathtt{class\ C\ extends\ T}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{L}}'\ \overline{\mathtt{M}}'\}} \tag{E-Class}$$

3.8.3 Properties of Elaboration

A minimal requirement for elaboration is that guessed information is sensible 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.

3.8.3.1 Theorem: The following propositions hold.

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 $\mathtt{T} = \mathtt{P.C.S}$ and $\mathit{fields}(\mathtt{P.C}) = \ldots, \mathtt{U\ f}, \ldots$ for some `P`, `S` and `U`.
3. If $\mathtt{T}; \overline{\mathtt{x}} \vdash \mathtt{m}(\overline{\mathtt{e}}) \Rightarrow \mathtt{C.this.m}(\overline{\mathtt{e}}')$, then $\mathtt{T} = \mathtt{P.C.S}$ and $\mathit{mtype}(\mathtt{P.C})$ is well defined for some `P` and `S`.
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 $\mathtt{T} = \mathtt{P.C.S}$ and $\mathtt{U.D} \in \mathit{dom}(CT)$ and $\mathtt{P.C} <: \mathtt{U}$.

Proof: Each of them is proved by induction of the derivation of the condition. Note that *field-super*(`f`, `T`) = `S` implies *fields*(`T`) = `...,U f,...` for some `U`, and *meth-super*(`m`, `T`) = `U` implies well-definedness of *mtype*(`m`, `T`). ■

3.9 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 [Jav97], 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.

3.10 Summary

We have studied FJI, an extension of FJ with inner classes, whose semantics is complicated because of interaction with inheritance. Two styles of semantics for inner classes have been considered: 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 direct style is type sound and the translation preserves typing; finally, it has been proved that the two styles correspond, in the sense that the translation commutes with the high-level reduction relation in the direct semantics. The correspondence justifies the current compilation scheme of inner classes with respect to the direct semantics.

In addition, we have shown the elaboration rules to recover fully-qualified types from abbreviated type names, receivers of field accesses (or method invocations), and enclosing instances of inner class constructor invocations.

We have also pointed out an underspecification in the official specification, which allowed significant change of an interpretation of `C.this` expression in different versions of the JDK compilers.

Chapter 4

Featherweight GJ

It is well known that parametric polymorphism, found in functional languages such as ML and Haskell, is useful for *generic* data structures, such as lists or trees. For example, a procedure to extract the n-th element of a list is uniform, regardless of the type of elements. Such a procedure can be written in polymorphically typed langauges in such a way that it takes the element type as a parameter; every time a concrete element type is given, it (conceptually) produces a definition specialized to the given type. Hence, parametric polymorphism encourages code reuse and makes it easier to maintain programs.

The basic idea of parametric polymorphism itself can naturally fit into the idea of classes—for example, list class would take the element type as a parameter, and some of its method may take other type parameters—for example, a "map" method may take a type parameter which represents the element type of the resulted list. However, there are many design issues on interaction with features of the base language; in fact, even for a single base language, Java, several proposals [AFM97, MBL97, OW97, BOSW98b, CS98] have been proposed and they are significantly different from each other in details.

Among them, the design of GJ [BOSW98b] is heavily constrained by compatibility with the legacy Java system, including Java Virtual Machine (JVM) [LY99] and Java libraries. Distinctive features of GJ include the following:

- Compilation from GJ programs to JVM code. GJ compiler erases type parameter information from GJ programs and generates JVM code, which cannot maintain such information. As we will see later, this compilation scheme has significantly affected some of the language constructs.

- Introduction of raw types to maintain compatibility with legacy code. In GJ, parameterized classes can be referred to as if they were non-parameterized, by using the mechanism of raw types. They are useful to refine library code with type parameters: for example, the raw type `List` can be used for the parametric class `List<X>` where `X` is the type parameter that represents the element type. Thus, the client side does not need to change the program even when the old library `List` is replaced with the new version `List<X>`.

- Method type parameter inference. GJ compiler supports partial type inference of type parameters for method invocations.

Although their mechanisms and correctness are not trivial at all, discussion has been given only informally in prose. Moreover, its semantics is given in an indirect manner [BOSW98a], in terms of its compilation to the Java Virtual Machine Language (JVML) [LY99]. This chapter discusses semantics and the compilation scheme of GJ. Raw types are discussed in the next chapter; type parameter inference is left for future work.

We extend FJ with type parameters and obtain Featherweight GJ (or FGJ). The main results in this chapter are:

1. Formalization of direct semantics of FGJ with a reduction relation, as in FJ, and its type soundness proofs. This semantics would correspond to an implementation that augments the run-time system to carry information about type parameters.

2. Formalization of translation from FGJ to FJ and development of correctness proofs: preservation of types and execution will be proved. This semantics models the current implementation style, compilation from GJ to JVML.

Since compiled FJ programs involve typecasts, the base language for the extension is actually FJ with typecasts, introduced in Section 2.6. Aside from that, since use of typecasts in GJ are restricted due to the compilation scheme, FGJ also includes typecasts. In this chapter, we refer to FJ with typecasts by FJ.

The rest of this chapter is organized as follows. Section 4.1 gives an informal overview of FGJ. The following Sections 4.2, 4.3, and 4.4 give the formal definition of FGJ with its syntax, type system, and reduction semantics, respectively. Then, we develop a type soundness result in Section 4.5. After Section 4.6 defines compilation rules from FGJ to FJ, theorems for their correctness are developed in Section 4.7. Finally, Section 4.8 summarizes this chapter.

4.1 Overview of Featherweight GJ

We begin with the class definition for pairs from Chapter 2, rewritten with type parameters in FGJ.

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

Both classes and methods may take type parameters. Here `X` and `Y` are parameters of the class, and `Z` is a parameter of the `setfst` method. Each type parameter has a *bound*; the actual parameter for a type variable must be subtype of its bound. Here `X`, `Y`, and `Z` must be subtype of `Object`—that is, any type. To instantiate an object from a polymorphic class, we have to specify actual type parameters as found in the body of `setfst` method; similarly, polymorphic method invocation is written $\texttt{e.m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$ where $\overline{\texttt{T}}$ denotes actual type parameters. To keep backward compatibility with FJ, we allow to omit the empty angle brackets `<>` from class/method declarations; thus, classes `A` and `B` may remain the same:

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

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

Moreover, we can omit type parameters for instantiation and method invocation when they are empty—for example, `new A<>()` may be abbreviated to `new A()`. In the context of the above definitions, the expression

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

evaluates to the expression

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

as follows (in direct semantics):

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

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

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

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

The bound of a type variable may not be a type variable, but may be a type expression involving type variables, and may be recursive (or even, if there are several variables and bounds, mutually recursive)—that is, FGJ supports F-bounded polymorphism [CCH+89]. For example, if `C<X>` and `D<Y>` are classes with one parameter each, one may have bounds such as `<X extends C<X>>` or even `<X extends C<Y>, Y extends D<X>>`. Here is an example of the use of recursive bounds in FGJ.

```
class Max<X extends Max<X>> extends Object {
  Max() { super(); }
  X max(X that) { return this.max(that); }
}

class Integer extends Max<Integer> {
  ...
  Integer max(Integer that) { ... }
}

class MaxPair<X extends Max<X>, Y extends Max<Y>> extends Max<MaxPair<X,Y>> {
  X fst;
  Y snd;
  MaxPair(X fst, Y snd) { super(); this.fst=fst; this.snd=snd; }
  MaxPair<X,Y> max(MaxPair<X,Y> that) {
    return new MaxPair<X,Y>(this.fst.max(that.fst), this.snd.max(that.snd));
  }
}
```

Class `Max` has a type parameter `X` bounded by `Max<X>`; thus, an actual parameter `T` for `X` must extend `Max<T>`. Such type can be explicitly declared: for example, class `Integer` is declared to extend `Max<Integer>` and, as a result, the method `max` will take a parameter of type `Integer` and return an object of type `Integer`. As such, we can enforce every class that (directly) extends `Max<X>` to have a method `max` that takes another object of the same type and returns an object of the same type. More complicated use appears in parametric class `MaxPair`. The method invocation $\mathtt{e.max(e')}$ where `e` and $\mathtt{e'}$ are of type `MaxPair<S,T>` returns a pair consisting of the greater elements obtained by comparing each element from `e` and $\mathtt{e'}$. Since the types `S` and `T` of elements themselves extend `Max<S>` and `Max<T>`, the method `max` can be invoked on the elements of pairs.

4.2 Syntax of FGJ and Auxiliary Definitions

We use the notational conventions as in FJ, plus the following ones: the metavariables `X`, `Y`, and `Z` range over type variables; `S`, `T`, `U`, and `V` range over types; and `N`, `P` and `Q` range over nonvariable types (types other than type variables). In what follows, for the sake of conciseness we abbreviate the keyword `extends` to the symbol $\triangleleft$ and the keyword `return` to the symbol $\uparrow$. We write $\overline{\mathtt{X}}$ as shorthand for $\mathtt{X}_1,\ldots,\mathtt{X}_n$ (and similarly for $\overline{\mathtt{T}}$, $\overline{\mathtt{N}}$, etc.), write $\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}$ for $\mathtt{X}_1 \triangleleft \mathtt{N}_1,\ldots,\mathtt{X}_n \triangleleft \mathtt{N}_n$, and assume sequences of type variables contain no duplicate names.

The abstract syntax of FGJ is given as follows:

$$
\begin{array}{lcl}
\mathtt{N} & ::= & \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \\[1ex]
\mathtt{T} & ::= & \mathtt{X} \\
 & | & \mathtt{N} \\[1ex]
\mathtt{L} & ::= & \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \\[1ex]
\mathtt{K} & ::= & \mathtt{C(}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\ \{\ super(}\overline{\mathtt{f}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{\ =\ }\overline{\mathtt{f}}\mathtt{;\ \}} \\[1ex]
\mathtt{M} & ::= & \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>\ T\ m\ (}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}} \\[1ex]
\mathtt{e} & ::= & \mathtt{x} \\
 & | & \mathtt{e.f} \\
 & | & \mathtt{e.m<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \\
 & | & \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)} \\
 & | & \mathtt{(N)e}
\end{array}
$$

A non-variable type is a class name `C` followed by type parameters $\overline{\mathtt{T}}$ enclosed by `<>`. A type is either a non-variable type or a type variable. A class declaration `L` has its name `C`, formal type arguments $\overline{\mathtt{X}}$, their bounds $\overline{\mathtt{N}}$, supertype `N`, field declarations $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$, a constructor declaration `K`, and method declarations $\overline{\mathtt{M}}$. A method declaration `M` can also have formal type arguments $\overline{\mathtt{X}}$ with their bounds $\overline{\mathtt{N}}$. An expression is a variable, field access, method invocation, object constructor, or typecast. Unlike GJ, method invocation in FGJ needs explicit type parameters for the polymorphic method. Note that type variables cannot be bounds, the instantiated type of an object constructor, and the target of a typecast expression: since, in erasure semantics, type parameter information is not available at run-time, we could neither check bounds and typecast nor instantiate an object, if we allowed type variable for them.

The type variables $\overline{\mathtt{X}}$ are bound in $\overline{\mathtt{N}}$, `N`, and the class body $\mathtt{\{}\overline{\mathtt{T}}\ \mathtt{f;\ K}\ \overline{\mathtt{M}}\mathtt{\}}$ of a class declaration $\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}}$. Similarly, the type variables $\overline{\mathtt{X}}$ are bound in $\overline{\mathtt{N}}$, `T`, $\overline{\mathtt{T}}$, and `e` of a method declaration $\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>\ T\ m\ (}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}}$. Note that $\overline{\mathtt{X}}$ binds their occurrences in their bounds $\overline{\texttt{N}}$. We define α-conversions of bound (type) variables in a customary manner and identify α-convertible classes. We use the notation $[\overline{\texttt{T}}/\overline{\texttt{X}}]$ for a capture-avoiding substitution of $\overline{\texttt{T}}$ for $\overline{\texttt{X}}$, defined in the standard manner. The head of a non-variable type, written *head*(N), is defined by *head*(C<$\overline{\texttt{T}}$>) = C.

We allow a pair of angle brackets to be omitted when the sequence between them is empty: for example, a non-variable type `C<>` may be written `C`, and a method invocation expression `e.m<>(`$\overline{\texttt{e}}$`)` may be written `e.m(`$\overline{\texttt{e}}$`)`, and a method declaration `<> T m (`$\overline{\texttt{T}}$` x) {`$\uparrow$`e;}` may be written `T m (`$\overline{\texttt{T}}$` x) {`$\uparrow$`e;}`.

As in FJ (and FJI), we assume a fixed class table CT, which is a mapping from class names C to class declarations L, obeying the same sanity conditions given for FJ except for (5) (The definition of subtyping in FGJ is not syntactic, as we will see below). We use, instead, the partial order $\texttt{C} \trianglelefteq \texttt{D}$, which is the reflexive and transitive closure of the relation between two class names in the `extends` relation; then, given CT, $\texttt{C} \trianglelefteq \texttt{D}$ must be antisymmetric.

We define auxiliary functions *fields*, *mtype*, *mbody*, and *dcast* for reduction and typing rules. The fields of non-variable type C<$\overline{\texttt{T}}$>, written *fields*(C<$\overline{\texttt{T}}$>), is a sequence $\overline{\texttt{S}}\ \overline{\texttt{f}}$ pairing the type of a field with its name, for all the fields declared in class C and all of its superclasses.

$$\textit{fields}(\texttt{Object}) = \bullet \qquad \text{(F-OBJECT)}$$

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \quad \textit{fields}([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{\textit{fields}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{g}}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}}} \qquad \text{(F-CLASS)}$$

It first looks up the fields $\overline{\texttt{U}}\ \overline{\texttt{g}}$ of the superclass and append the fields defined in the current class $\overline{\texttt{S}}\ \overline{\texttt{f}}$ after it. Actual type arguments $\overline{\texttt{T}}$ are substituted for formal arguments $\overline{\texttt{X}}$ in $\overline{\texttt{S}}$ and supertype N.

4.2.1 Example: Under a class table including class `Pair`,

$$\textit{fields}(\texttt{Pair<A,B>}) = [\texttt{A/X}, \texttt{B/Y}](\texttt{X fst, Y snd}) = \texttt{A fst, B snd}$$

holds.

The body of the method m with type arguments $\overline{\texttt{T}}$ in non-variable type N, written *mbody*(m<$\overline{\texttt{T}}$>, N), is a pair, written ($\overline{\texttt{x}}$, e), of a sequence of formal parameters $\overline{\texttt{x}}$ and an expression e.

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \quad \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{> U m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e}_0\texttt{;\}} \in \overline{\texttt{M}}}{\textit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}, [\overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{e}_0)} \qquad \text{(MB-CLASS)}$$

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \quad \texttt{m} \text{ is not defined in } \overline{\texttt{M}}}{\textit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \textit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})} \qquad \text{(MB-SUPER)}$$

When a method definition is found, the formal type parameters $\overline{\texttt{X}}$ and $\overline{\texttt{Y}}$ are replaced with actual type parameters $\overline{\texttt{T}}$ and $\overline{\texttt{V}}$, respectively.

4.2.2 Example: Under a class table including class `Pair`,

$$\begin{array}{rl} & \mathit{mbody}(\texttt{setfst<B>}, \texttt{Pair<A,B>}) \\ = & (\texttt{newfst}, [\texttt{A}/\texttt{X}, \texttt{B}/\texttt{Y}, \texttt{B}/\texttt{Z}](\texttt{new Pair<Z,Y>(newfst, this.snd)})) \\ = & (\texttt{newfst}, \texttt{new Pair<B,B>(newfst, this.snd)}) \end{array}$$

holds.

The type of the method `m` in non-variable type $\texttt{C<}\overline{\texttt{T}}\texttt{>}$, written $\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>})$, is of the form $\texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}}\rightarrow\texttt{T}_0$ consisting of formal type arguments $\overline{\texttt{Y}}$ with their bound $\overline{\texttt{P}}$, argument types $\overline{\texttt{T}}$ and a result type $\texttt{T}_0$.

$$\frac{\begin{array}{c} \mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> U m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e;\}} \in \overline{\texttt{M}} \end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{U})} \qquad (\text{MT-Class})$$

$$\frac{\begin{array}{c} \mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{m} \text{ is not defined in } \overline{\texttt{M}} \end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \mathit{mtype}(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})} \qquad (\text{MT-Super})$$

The type variables $\overline{\texttt{Y}}$ are bound in $\overline{\texttt{P}}$, $\overline{\texttt{T}}$ and $\texttt{T}_0$ in $\texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}}\rightarrow\texttt{T}_0$.

Unlike *mbody*, *mtype* does not take actual type argument for the method as a part of arguments: it is checked whether actual type arguments satisfy constraints of the bounds *after* looking up the class table, as we will see in typing rules. (We could integrate that check in the rule MT-Class, but, for simplicity, it is separated.)

4.2.3 Example: Under a class table including class `Pair`,

$$\begin{array}{rl} & \mathit{mtype}(\texttt{setfst}, \texttt{Pair<A,B>}) \\ = & [\texttt{A}/\texttt{X}, \texttt{B}/\texttt{Y}](\texttt{<Z} \triangleleft \texttt{Object>Z}\rightarrow\texttt{Pair<Z,Y>}) \\ = & \texttt{<Z} \triangleleft \texttt{Object>Z}\rightarrow\texttt{Pair<Z,B>} \end{array}$$

dcast(`C`, `D`) is the least partial order closed under the following rule:

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{...\}} \qquad \overline{\texttt{X}} = \mathit{FV}(\texttt{N})}{\mathit{dcast}(\texttt{C}, \texttt{D})}$$

where $FV(\texttt{N})$ denotes the set of type variables in `N`. For example, *dcast*(`A`, `Object`) holds but *dcast*(`Pair`, `Object`) does not.

4.3 Typing

An *environment* Γ is a finite mapping from variables to types, written $\overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}$. A *type environment* Δ is a finite mapping from type variables to nonvariable types, written $\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}$, that takes each type variable to its bound. We write $\mathit{bound}_\Delta(\texttt{T})$ for the least non-variable supertype of `T` under Δ, as defined below.

$$\begin{array}{rcl} \mathit{bound}_\Delta(\texttt{X}) & = & \Delta(\texttt{X}) \\ \mathit{bound}_\Delta(\texttt{N}) & = & \texttt{N}. \end{array}$$

Unlike calculi such as $F_\leq$ [CMMS94], this promotion relation does not need to be defined recursively: the bound of a type variable in a type environment is always a nonvariable type.

4.3.1 Subtyping

We write $\Delta \vdash \texttt{S} <: \texttt{T}$ if `S` is a subtype of `T` under the assumption given by Δ. As before, subtyping is derived from the `extends` relation.

$$\Delta \vdash \texttt{T} <: \texttt{T} \tag{S-Refl}$$

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

$$\Delta \vdash \texttt{X} <: \Delta(\texttt{X}) \tag{S-Var}$$

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

Type parameters are *invariant* with regard to subtyping, so $\Delta \vdash \overline{\texttt{T}} <: \overline{\texttt{U}}$ does *not* (and must not) imply $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{C<}\overline{\texttt{U}}\texttt{>}$. For example, consider the following classes:

```
class Id extends Object {
  Id() { super(); }
  Id id() { return this; }
}

class Cell<X extends Object> extends Object{
  X elm;
  Cell (X elm) { super(); this.elm=elm; }
  Cell<X> set(X newelm) { return new Cell<X>(newelm);}
}

class IdCell extends Cell<Id> {
  IdCell(Id elm) { super(elm); }
  Cell<Id> set(Id newelm) { return new Cell<Id>(newelm.id()); }
}
```

We have polymorphic `Cell` class and a subclass `IdCell` where the element type is specialized to `Id`; moreover, `set` method is overridden so that it works only for an argument of the type `Id`. Now, if we allowed `Cell<Id>` (and `IdCell`) to be subtype of `Cell<Object>`, the expression

```
((Cell<Object>)new IdCell(new Id())).set(new Object())
```

would be well typed (by the typing rule given later) because the argument type of `set` from `Cell`, the static type of the receiver, is `Object`. But, it would crash when it tries to invoke the `id` method on `new Object()`.

4.3.2 Well-Formed Types

A type expression $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ makes sense only when each $\texttt{T}_i$ satisfies the constraint enforced by the bound of $\texttt{X}_i$. We write $\Delta \vdash \texttt{T}$ ok if the type T is well formed in the type environment Δ. The rules for well-formed types are given as follows:

$$\Delta \vdash \texttt{Object}\ \text{ok} \qquad \text{(WF-OBJECT)}$$

$$\frac{\texttt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \texttt{X}\ \text{ok}} \qquad \text{(WF-VAR)}$$

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

The type Object is always well formed and a type variable is well formed if it is in the domain of the type environment. If the declaration of a class C begins with $\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>}$, then a type like $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ is well formed only if substituting $\overline{\texttt{T}}$ for $\overline{\texttt{X}}$ respects the bounds $\overline{\texttt{N}}$, that is if $\overline{\texttt{T}} \texttt{ <: } [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}}$. Note that we perform a simultaneous substitution, and so any variable in $\overline{\texttt{X}}$ may appear in $\overline{\texttt{N}}$, permitting recursion and mutual recursion between variables and bounds.

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

4.3.3 Typing Rules

The typing judgment for expressions is of the form $\Delta; \Gamma \vdash \texttt{e} \in \texttt{T}$, read as "in the type environment Δ and the environment Γ, e has type T."

As before, the type of a variable is determined by the environment.

$$\Delta; \Gamma \vdash \texttt{x} \in \Gamma(\texttt{x}) \qquad \text{(GT-VAR)}$$

The rule GT-FIELD below for field access is essentially the same as in FJ. In case the type of $\texttt{e}_0$ is a type variable, the least non-variable supertype $\mathit{bound}_\Delta(\texttt{T}_0)$ is used to obtain the fields of the type.

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

The rule GT-INVK for method invocation looks complicated but it is actually straightforward, too.

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

It first obtain the type $\texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \to \texttt{U}$ of the method m; then, it is checked that the type arguments $\overline{\texttt{V}}$ are well-formed and respect the bounds $\overline{\texttt{P}}$ of the method. As in FJ, the actual type arguments must be subtypes of those of the formal arguments (where the formal type arguments $\overline{\texttt{Y}}$ are replaced with the actual type arguments $\overline{\texttt{V}}$); the type of the whole expression is the result type of the method.

The rule GT-NEW for object constructors is simple:

$$\frac{\Delta \vdash \mathtt{N}\ \text{ok} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}}}{\Delta;\Gamma \vdash \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{N}} \tag{GT-New}$$

As in FJ (with typecasts), we have three rules for typecasts:

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \mathrel{<:} \mathtt{N}}{\Delta;\Gamma \vdash \mathtt{(N)e}_0 \in \mathtt{N}} \tag{GT-UCast}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{N}\ \text{ok} \\ \Delta \vdash \mathtt{N} \mathrel{<:} \mathit{bound}_\Delta(\mathtt{T}_0) \qquad \mathtt{N} \neq \mathit{bound}_\Delta(\mathtt{T}_0) \\ \mathit{dcast}(\mathit{head}(\mathtt{N}), \mathit{head}(\mathit{bound}_\Delta(\mathtt{T}_0)))\end{array}}{\Delta;\Gamma \vdash \mathtt{(N)e}_0 \in \mathtt{N}} \tag{GT-DCast}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{N}\ \text{ok} \\ \mathtt{C} = \mathit{head}(\mathtt{N}) \qquad \mathtt{D} = \mathit{head}(\mathit{bound}_\Delta(\mathtt{T}_0)) \qquad \mathtt{C} \not\trianglelefteq \mathtt{D} \qquad \mathtt{D} \not\trianglelefteq \mathtt{C} \\ \mathit{stupid\ warning}\end{array}}{\Delta;\Gamma \vdash \mathtt{(N)e}_0 \in \mathtt{N}} \tag{GT-SCast}$$

The rules GT-DCAST and GT-SCAST are equipped with conditions to ensure that the result of the cast will be the same at run time, no matter whether we use the high-level (type-passing) reduction rules defined in the following section or the erasure semantics considered in Section 4.6. For example, suppose we have defined:

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

Now, if `o` has type `Object`, then the cast `(List<C>)o` is not permitted since `List<D>` is another subtype of `Object`. (At run time, `o` may be bound to `new List<D>()`; then the cast would fail in the type-passing semantics but succeed in the erasure semantics, since `(List<C>)o` erases to `(List)o` by removing type parameter while both `new List<C>()` and `new List<D>()` erase to `new List()`.) On the other hand, if `cl` is given type `List<C>`, then the cast `(LinkedList<C>)cl` is permitted, since the type-passing and erased versions of the cast are guaranteed to either both succeed or both fail. Similarly, GT-SCAST prevents a stupid cast from succeeding in erasure semantics.

The typing judgment for method declarations has the form $\mathtt{M\ OK\ IN\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}$, read "method declaration `M` is ok if it occurs in class $\mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}$," and is derived by the following rule:

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X}}\mathrel{<:}\overline{\mathtt{N}}, \overline{\mathtt{Y}}\mathrel{<:}\overline{\mathtt{P}} \qquad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}, \overline{\mathtt{P}}\ \text{ok} \\ \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 \in \mathtt{S} \qquad \Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T} \\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N\ \{...\}} \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U}, \text{ then } \overline{\mathtt{P}}, \overline{\mathtt{T}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}](\overline{\mathtt{Q}}, \overline{\mathtt{U}}) \text{ and } \Delta \vdash \mathtt{T} \mathrel{<:} [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}]\mathtt{U}\end{array}}{\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>\ T\ m\ (}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e}_0\mathtt{;\}\ OK\ IN\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}} \tag{GT-Method}$$

First of all, the declared bounds $\overline{\mathtt{P}}$ and types $\overline{\mathtt{T}}$ and $\mathtt{T}_0$ must be well-formed. The method body $\mathtt{e}_0$ must be given a subtype of the declared result type. The last conditions contains one additional subtlety that FJ (and Java) do not have. In FGJ (and GJ), unlike in FJ (and Java), covariant subtyping of method results is allowed. That is, the result type of a method may be subtype of the result type of the corresponding method in the superclass, while the bounds of type variables and the argument types must be identical (modulo renaming of type variables).

The typing judgment for class declarations has the form `L OK`, read "class declaration `L` is ok," and is derived by the following rule:

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

It checks well-formedness of the bounds $\overline{\mathtt{N}}$ of type parameters, supertype `N`, and the type of the fields $\overline{\mathtt{T}}$; constructor arguments must agree with the types of all the fields from this class and its superclasses; and methods must be ok. Finally, a class table is said to be ok if all its class definitions are ok.

4.4 Reduction Semantics

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

The reduction rules for basic computation are given as follows.

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{(new\ N(}\overline{\mathtt{e}}\mathtt{)).f}_i \longrightarrow \mathtt{e}_i} \qquad \text{(GR-FIELD)}$$

$$\frac{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\mathtt{(new\ N(}\overline{\mathtt{e}}\mathtt{)).m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{d}}\mathtt{)} \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)/this}]\mathtt{e}_0} \qquad \text{(GR-INVK)}$$

$$\frac{\emptyset \vdash \mathtt{N} \mathrel{\mathtt{<:}} \mathtt{P}}{\mathtt{(P)(new\ N(}\overline{\mathtt{e}}\mathtt{))} \longrightarrow \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)}} \qquad \text{(GR-CAST)}$$

There are three reduction rules, one for field access, one for method invocation, and one for typecasts. Actually, thanks to auxiliary functions, they are a little more complicated than we had in FJ.

The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules as before.

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0\mathtt{.f} \longrightarrow \mathtt{e}_0{}'\mathtt{.f}} \qquad \text{(GRC-FIELD)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \longrightarrow \mathtt{e}_0{}'\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}} \qquad \text{(GRC-INV-RECV)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{e}_0\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>(}\ldots\mathtt{e}_i{}'\mathtt{,}\ldots\mathtt{)}} \qquad \text{(GRC-INV-ARG)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{new\ N(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{new\ N(}\ldots\mathtt{e}_i{}'\mathtt{,}\ldots\mathtt{)}} \qquad \text{(GRC-NEW-ARG)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{(N)e}_0 \longrightarrow \mathtt{(N)e}_0{}'} \qquad \text{(GRC-CAST)}$$

4.5 Properties of Reduction Semantics

4.5.1 Type Soundness

FGJ programs enjoy subject reduction and progress properties exactly like programs in FJ (Theorems 2.5.1 and 2.5.2) The basic structures of the proofs are similar to those of Theorem 2.5.1 and 2.5.2. For subject reduction, however, since we now have parametric polymorphism combined with subtyping, we need a few more lemmas. The main lemmas required are a term substitution lemma as before, plus similar lemmas about the preservation of subtyping and typing under *type* substitution. (Readers familiar with proofs of subject reduction for typed lambda-calculi like $F_{\leq}$ [CMMS94] will notice many similarities).

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

Proof: See below. ∎

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

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

(2) If `e` includes $\mathtt{new\ N_0(\bar{e}).m<\bar{V}>(\bar{d})}$ as a subexpression, then $\mathit{mbody}(\mathtt{m<\bar{V}>}, \mathtt{N_0}) = (\bar{\mathtt{x}}, \mathtt{e_0})$ and $\#(\bar{\mathtt{x}}) = \#(\bar{\mathtt{d}})$.

Proof: Similar to the proof of Theorem 2.5.2. ∎

We will develop the proof of Theorem 4.5.1.1; we begin with required lemmas.

4.5.1.3 Lemma [Weakening]: Suppose $\Delta, \bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{N}} \vdash \bar{\mathtt{N}}$ ok and $\Delta \vdash \mathtt{U}$ ok.

1. If $\Delta \vdash \mathtt{S} \mathtt{<:} \mathtt{T}$, then $\Delta, \bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{N}} \vdash \mathtt{S} \mathtt{<:} \mathtt{T}$.
2. If $\Delta \vdash \mathtt{S}$ ok, then $\Delta, \bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{N}} \vdash \mathtt{S}$ ok.
3. If $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$, then $\Delta; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e} \in \mathtt{T}$, and $\Delta, \bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{N}}; \Gamma \vdash \mathtt{e} \in \mathtt{T}$.

Proof: Each of them is proved by straightforward induction on the derivation of $\Delta \vdash \mathtt{S} \mathtt{<:} \mathtt{T}$ and $\Delta \vdash \mathtt{S}$ ok and $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$. ∎

4.5.1.4 Lemma: If $\Delta \vdash \mathtt{E<\bar{V}>} \mathtt{<:} \mathtt{D<\bar{U}>}$ and $\mathtt{D} \not\trianglelefteq \mathtt{C}$ and $\mathtt{C} \not\trianglelefteq \mathtt{D}$, then $\mathtt{E} \not\trianglelefteq \mathtt{C}$ and $\mathtt{C} \not\trianglelefteq \mathtt{E}$.

Proof: It is easy to see that $\Delta \vdash \mathtt{E<\bar{V}>} \mathtt{<:} \mathtt{D<\bar{U}>}$ implies $\mathtt{E} \trianglelefteq \mathtt{D}$. The conclusions are easily proved by contradiction. (A similar argument is found in the proof of Lemma 2.6.1.) ∎

4.5.1.5 Lemma: Suppose $\mathit{dcast}(\mathtt{C}, \mathtt{D})$ and $\Delta \vdash \mathtt{C<\bar{T}>} \mathtt{<:} \mathtt{D<\bar{U}>}$. If $\Delta \vdash \mathtt{C<\bar{T}'>} \mathtt{<:} \mathtt{D<\bar{U}>}$, then $\bar{\mathtt{T}}' = \bar{\mathtt{T}}$.

Proof: The case where $\mathtt{C} = \mathtt{D}$ is easy: since *dcast* is antisymmetric, if $\Delta \vdash \mathtt{C<\bar{T}>} \mathtt{<:} \mathtt{D<\bar{U}>}$, then $\mathtt{C<\bar{T}>}$ and $\mathtt{D<\bar{U}>}$ must be equal. The case where $\mathit{dcast}(\mathtt{C}, \mathtt{D})$ because $\mathit{dcast}(\mathtt{C}, \mathtt{E})$ and $\mathit{dcast}(\mathtt{E}, \mathtt{D})$ is also easy: note that, from every judgment $\Delta \vdash \mathtt{C<\bar{T}>} \mathtt{<:} \mathtt{D<\bar{U}>}$, we can have $\Delta \vdash \mathtt{C<\bar{T}>} \mathtt{<:} \mathtt{E<\bar{V}>}$ and $\Delta \vdash \mathtt{E<\bar{V}>} \mathtt{<:} \mathtt{D<\bar{U}>}$. Finally, if `D` is the direct superclass of `C`, $\mathtt{C<\bar{T}>}$ is uniquely determined by $\mathtt{D<\bar{U}>}$ because $FV(\mathtt{N}) = \bar{\mathtt{X}}$ where $CT(\mathtt{C}) = \mathtt{class\ C<\bar{X} \triangleleft \bar{N}> \triangleleft N\{\ldots\}}$, finishing the proof. ∎

4.5.1.6 Lemma: If $\mathit{dcast}(\mathtt{C}, \mathtt{E})$ and $\mathtt{C} \trianglelefteq \mathtt{D} \trianglelefteq \mathtt{E}$, then $\mathit{dcast}(\mathtt{C}, \mathtt{D})$ and $\mathit{dcast}(\mathtt{D}, \mathtt{E})$.

Proof: Easy. ∎

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

Proof: By induction on the derivation of $\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$.
Case S-Refl:
Trivial.
Case S-Trans, S-Class:
Easy.
Case S-Var: $\mathtt{S} = \mathtt{X} \qquad \mathtt{T} = (\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2)(\mathtt{X})$
If $\mathtt{X} \in \mathit{dom}(\Delta_1) \cup \mathit{dom}(\Delta_2)$, then it's trivial. On the other hand, if $\mathtt{X} = \mathtt{X}_i$, then, by assumption, we have $\Delta_1 \vdash \mathtt{U}_i \mathrel{\mathtt{<:}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{N}_i$. Finally, Lemma 4.5.1.3 finishes the case. ∎

4.5.1.8 Lemma [Type substitution preserves type well-formedness]: If $\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{T}$ ok and $\Delta_1 \vdash \overline{\mathtt{U}} \mathrel{\mathtt{<:}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ with $\Delta_1 \vdash \overline{\mathtt{U}}$ ok and none of $\overline{\mathtt{X}}$ appearing in Δ_1, then $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$ ok.

Proof: By induction on the derivation of $\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{T}$ ok, with a case analysis on the last rule used.
Case WF-Object:
Trivial.
Case WF-Var:
It follows from $\Delta_1 \vdash \overline{\mathtt{U}}$ ok and Lemma 4.5.1.3.
Case WF-Class: $\mathtt{T} = \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>} \qquad \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \overline{\mathtt{T}}$ ok $\qquad \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \overline{\mathtt{T}} \mathrel{\mathtt{<:}} [\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$
$\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>} \triangleleft \mathtt{N}\ \{\ldots\}$
By the induction hypothesis,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}} \text{ ok.}$$

On the other hand, by Lemma 4.5.1.7, $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}} \mathrel{\mathtt{<:}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$. Since $\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \vdash \overline{\mathtt{P}}$ by the rule GT-Class, $\overline{\mathtt{P}}$ does not include any of $\overline{\mathtt{X}}$ as a free variable. Thus, $[\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} = [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$, and finally, we have $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{C}\mathtt{<}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}\mathtt{>}$ ok by WF-Class. ∎

4.5.1.9 Lemma: Suppose $\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{T}$ ok and $\Delta_1 \vdash \overline{\mathtt{U}} \mathrel{\mathtt{<:}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ with $\Delta_1 \vdash \overline{\mathtt{U}}$ ok and none of $\overline{\mathtt{X}}$ appearing in Δ_1. Then, $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}) \mathrel{\mathtt{<:}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2}(\mathtt{T}))$.

Proof: The case where $\mathtt{T}$ is a nonvariable type is trivial. The case where $\mathtt{T}$ is a type variable $\mathtt{X}$ and $\mathtt{X} \in \mathit{dom}(\Delta_1) \cup \mathit{dom}(\Delta_2)$ is also easy. Finally, if $\mathtt{T}$ is a type variable $\mathtt{X}_i$, then $\mathit{bound}_{\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}) = \mathtt{U}_i$ and $[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2}(\mathtt{T})) = [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{N}_i$; the assumption $\Delta_1 \vdash \overline{\mathtt{U}} \mathrel{\mathtt{<:}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ and Lemma 4.5.1.3 finish the proof. ∎

4.5.1.10 Lemma: If $\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$ and $\mathit{fields}(\mathit{bound}_\Delta(\mathtt{T})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, then $\mathit{fields}(\mathit{bound}_\Delta(\mathtt{S})) = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$ and $\mathtt{S}_i = \mathtt{T}_i$ and $\mathtt{g}_i = \mathtt{f}_i$ for all $i \leq \#(\overline{\mathtt{f}})$.

Proof: By straightforward induction on the derivation of $\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$.

Case S-TRANS:
Trivial.
Case S-VAR:
Trivial because $\mathit{bound}_\Delta(\mathtt{S}) = \mathit{bound}_\Delta(\mathtt{T})$.
Case S-TRANS:
Easy.
Case S-CLASS: $\mathtt{S} = \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ $\quad \mathtt{T} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}$ $\quad \mathit{CT}(\mathtt{C}) = \mathtt{class\ C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{g}};\ \ldots\}$
By the rule F-CLASS, $\mathit{fields}(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{g}}$ where $\overline{\mathtt{U}}\ \overline{\mathtt{f}} = \mathit{fields}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})$. ■

4.5.1.11 Lemma: If $\Delta \vdash \mathtt{T}$ ok and $\mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T})) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U}_0$, then for any $\mathtt{S}$ such that $\Delta \vdash \mathtt{S} \texttt{<:} \mathtt{T}$ and $\Delta \vdash \mathtt{S}$ ok, we have $\mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{S})) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U}_0{}'$ and $\Delta, \overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{P}} \vdash \mathtt{U}_0{}' \texttt{<:} \mathtt{U}_0$.

Proof: By straightforward induction on the derivation of $\Delta \vdash \mathtt{S} \texttt{<:} \mathtt{T}$ with a case analysis by the last rule used.
Case S-VAR:
Trivial because $\mathit{bound}_\Delta(\mathtt{S}) = \mathit{bound}_\Delta(\mathtt{T})$.
Case S-TRANS:
Easy.
Case S-CLASS: $\mathtt{S} = \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ $\quad \mathtt{T} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}$ $\quad \mathit{CT}(\mathtt{C}) = \mathtt{class\ C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>} \triangleleft \mathtt{N}\ \{\ \ldots\ \ \overline{\mathtt{M}}\}$
If $\overline{\mathtt{M}}$ do not include a declaration of $\mathtt{m}$, it is easy to show the conclusion, since

$$\mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{S})) = \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}))$$

by the rule MT-SUPER.

On the other hand, suppose $\overline{\mathtt{M}}$ includes a declaration of $\mathtt{m}$. By straightforward induction on the derivation of $\mathit{mtype}(\mathtt{m}, \mathtt{T})$, we can show

$$\mathit{mtype}(\mathtt{m}, \mathtt{T}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}'\texttt{>}\overline{\mathtt{U}}' \rightarrow \mathtt{U}_0{}'$$

where $\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}'\texttt{>}\overline{\mathtt{U}}' \rightarrow \mathtt{U}_0{}'' = \mathit{mtype}(\mathtt{m}, \mathtt{N})$. Without loss of generality, we can assume that $\overline{\mathtt{X}}$ and $\overline{\mathtt{Y}}$ are distinct and, in particular, that $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0{}'' = \mathtt{U}_0$. By GT-METHOD, it must be the case that

$$\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}'\texttt{>}\ \mathtt{W}_0{}'\ \mathtt{m}\ (\overline{\mathtt{U}}'\ \overline{\mathtt{x}})\ \{\ldots\} \in \overline{\mathtt{M}}$$

and

$$\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}, \overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{P}}' \vdash \mathtt{W}_0{}'\texttt{<:}\mathtt{U}_0{}''.$$

By Lemmas 4.5.1.7 and 4.5.1.3, we have

$$\Delta, \overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{P}} \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{W}_0{}'\texttt{<:}\mathtt{U}_0.$$

Since $\mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{S})) = \mathit{mtype}(\mathtt{m}, \mathtt{S}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}'\texttt{>}\overline{\mathtt{U}}' \rightarrow \mathtt{W}_0{}'$ by MT-CLASS, letting $\mathtt{U}_0{}' = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{W}_0{}'$ finishes the case. ■

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

Proof: By induction on the derivation of $\Delta_1, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}, \Delta_2; \Gamma \vdash \mathtt{e} \in \mathtt{T}$ with a case analysis on the last rule used.

Case GT-Var:
Trivial.

Case GT-Field: $\mathtt{e} = \mathtt{e}_0\mathtt{.f}_i \quad \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \mathit{fields}(\mathit{bound}_{\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2}(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$
$\mathtt{T} = \mathtt{T}_i$

By the induction hypothesis, $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0 \in \mathtt{S}_0$ and $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S}_0 \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0$. By Lemma 4.5.1.9,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{T}_0) \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathit{bound}_{\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2}(\mathtt{T}_0).$$

Then, it is easy to show

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0) \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathit{bound}_{\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2}(\mathtt{T}_0).$$

By Lemma 4.5.1.10, $\mathit{fields}(\mathit{bound}_{\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0)) = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$ and we have $\mathtt{f}_j = \mathtt{g}_j$ and $\mathtt{S}_j = [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_j$ for $j \le \#(\overline{\mathtt{f}})$. By the rule GT-Field, $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0\mathtt{.f}_i \in \mathtt{S}_i$. Letting $\mathtt{S} = \mathtt{S}_i$ $(= [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_i)$ finishes the case.

Case GT-Invk: $\mathtt{e} = \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \quad \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0$
$\mathit{mtype}(\mathtt{m}, \mathit{bound}_{\Delta_1,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}},\Delta_2}(\mathtt{T}_0)) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{W}}\mathtt{\to}\mathtt{W}_0$
$\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \overline{\mathtt{V}}\ \mathrm{ok} \quad \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \overline{\mathtt{V}} \mathtt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$
$\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \quad \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \overline{\mathtt{S}} \mathtt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{W}}$
$\mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{W}_0$

By the induction hypothesis,

$$\begin{array}{l}\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0 \in \mathtt{S}_0 \\ \Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S}_0 \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0\end{array}$$

and

$$\begin{array}{l}\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{e}} \in \overline{\mathtt{S}}' \\ \Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \overline{\mathtt{S}}' \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}.\end{array}$$

By using Lemma 4.5.1.9, it is easy to show

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0) \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathit{bound}_{\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2}(\mathtt{T}_0).$$

Then, by Lemma 4.5.1.11,

$$\begin{array}{l}\mathit{mtype}(\mathtt{m}, \mathit{bound}_{\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0)) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}\mathtt{>}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}\mathtt{\to}\mathtt{W}_0{}' \\ \Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2, \overline{\mathtt{Y}}\mathtt{<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{P}} \vdash \mathtt{W}_0{}' \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W}_0.\end{array}$$

By Lemma 4.5.1.8,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}\ \mathrm{ok}$$

Without loss of generality, we can assume that $\overline{\mathtt{X}}$ and $\overline{\mathtt{Y}}$ are distinct and that none of $\overline{\mathtt{Y}}$ appear in $\overline{\mathtt{U}}$; then $[\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}] = [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]$. By Lemma 4.5.1.7,

$$\begin{array}{ll}\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}} \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} & (= [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}) \\ \Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}} \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{W}} & (= [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}).\end{array}$$

By the rule S-Trans,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \overline{\mathtt{S}}' \mathtt{<:} [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}.$$

By Lemma 4.5.1.7, we have

$$\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{W}_0{}' \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{W}_0 \quad (= [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W}_0).$$

Finally, by the rule GT-Invk,

$$\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash ([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0)\mathtt{.m<}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}\mathtt{>(}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{d}}\mathtt{)} \in \mathtt{S}$$

where $\mathtt{S} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{W}_0{}'$, finishing the case.

Case GT-New, GT-UCast:

Easy.

Case GT-DCast: $\mathtt{e} = (\mathtt{N})\mathtt{e}_0$ $\quad \Delta = \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2$ $\quad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0$
$\mathtt{C} = \mathit{head}(\mathtt{N})$ $\quad \mathtt{E} = \mathit{head}(\mathit{bound}_\Delta(\mathtt{T}_0))$ $\quad \Delta \vdash \mathtt{N} \mathrel{<:} \mathit{bound}_\Delta(\mathtt{T}_0)$
$\mathtt{N} \neq \mathit{bound}_\Delta(\mathtt{T}_0)$ $\quad \mathit{dcast}(\mathtt{C}, \mathtt{E})$

By the induction hypothesis, $\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0 \in \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S}_0 \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0$. Let $\Delta' = \Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2$. We have three subcases according to a relation between $\mathtt{S}_0$ and $\mathtt{N}$.

Subcase: $\quad \Delta' \vdash \mathit{bound}_\Delta(\mathtt{S}_0) \mathrel{<:} \mathtt{N}$

By the rule GT-UCast, $\Delta';\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]((\mathtt{N})\mathtt{e}_0) \in \mathtt{N}$.

Subcase: $\quad \Delta' \vdash \mathtt{N} \mathrel{<:} \mathit{bound}_{\Delta'}(\mathtt{S}_0)$

By using Lemma 4.5.1.9 and the fact that $\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ implies $\Delta \vdash \mathit{bound}_\Delta(\mathtt{S}) \mathrel{<:} \mathit{bound}_\Delta(\mathtt{T})$, we have $\Delta' \vdash \mathit{bound}_{\Delta'}(\mathtt{S}_0) \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathit{bound}_\Delta(\mathtt{T}_0)$. Then, $\mathtt{C} \trianglelefteq \mathtt{D} \trianglelefteq \mathtt{E}$ where $\mathtt{D} = \mathit{head}(\mathit{bound}_{\Delta'}(\mathtt{S}_0))$. By Lemma 4.5.1.6, we have $\mathit{dcast}(\mathtt{C}, \mathtt{D})$; finally, the rule GT-DCast finishes the subcase.

Subcase: $\quad \Delta' \vdash \mathtt{N} \not\mathrel{<:} \mathit{bound}_\Delta(\mathtt{S}_0) \qquad \Delta' \vdash \mathit{bound}_\Delta(\mathtt{S}_0) \not\mathrel{<:} \mathtt{N}$

By using Lemma 4.5.1.9 and the fact that $\Delta' \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ implies $\Delta' \vdash \mathit{bound}_\Delta(\mathtt{S}) \mathrel{<:} \mathit{bound}_\Delta(\mathtt{T})$, we have $\Delta' \vdash \mathit{bound}_{\Delta'}(\mathtt{S}_0) \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathit{bound}_\Delta(\mathtt{T}_0)$.

Let $\mathtt{D} = \mathit{head}(\mathit{bound}_\Delta(\mathtt{S}_0))$. We show below that, by contradiction, that neither $\mathtt{C} \not\trianglelefteq \mathtt{D}$ nor $\mathtt{D} \not\trianglelefteq \mathtt{C}$ holds. Suppose $\mathtt{C} \trianglelefteq \mathtt{D}$. Then, there exist some $\overline{\mathtt{V}}'$ such that $\Delta' \vdash \mathtt{C<}\overline{\mathtt{V}}'\mathtt{>} \mathrel{<:} \mathit{bound}_\Delta(\mathtt{S}_0)$. By Lemma 4.5.1.6, we have $\mathit{dcast}(\mathtt{C}, \mathtt{D})$; it follows from Lemma 4.5.1.5 that $\mathtt{C<V'>} = \mathtt{N}$, contradicting the assumption; thus, $\mathtt{C} \not\trianglelefteq \mathtt{D}$. On the other hand, suppose $\mathtt{D} \trianglelefteq \mathtt{C}$. Since we have $\Delta' \vdash \mathit{bound}_{\Delta'}(\mathtt{S}_0) \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_\Delta(\mathtt{T}_0))$, we can have $\mathtt{C<}\overline{\mathtt{V}}'\mathtt{>}$ such that $\Delta' \vdash \mathit{bound}_{\Delta'}(\mathtt{S}_0) \mathrel{<:} \mathtt{C<}\overline{\mathtt{V}}'\mathtt{>}$ and $\Delta' \vdash \mathtt{C<}\overline{\mathtt{V}}'\mathtt{>} \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_\Delta(\mathtt{T}_0))$. Then, $\mathtt{N} = \mathtt{C<}\overline{\mathtt{V}}'\mathtt{>}$ by Lemma 4.5.1.5, contradicting the assumption $\Delta' \vdash \mathit{bound}_{\Delta'}(\mathtt{S}_0) \mathrel{<:} \mathtt{N}$; thus, $\mathtt{D} \not\trianglelefteq \mathtt{C}$.

Finally, by the rule GT-SCast, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]((\mathtt{N})\mathtt{e}_0) \in \mathtt{N}$ with *stupid warning.* ■

Case GT-SCast: $\mathtt{e} = (\mathtt{N})\mathtt{e}_0$ $\quad \Delta = \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2$ $\quad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0$
$\mathtt{C} = \mathit{head}(\mathtt{N})$ $\quad \mathtt{E} = \mathit{head}(\mathit{bound}_\Delta(\mathtt{T}_0))$ $\quad \mathtt{C} \not\trianglelefteq \mathtt{E}$ $\quad \mathtt{E} \not\trianglelefteq \mathtt{C}$

By the induction hypothesis, $\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0 \in \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S}_0 \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0$. Using Lemma 4.5.1.9, we have $\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0) \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathit{bound}_\Delta(\mathtt{T}_0)$. Since $\mathit{head}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathit{bound}_\Delta(\mathtt{T}_0)) = \mathit{head}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \mathtt{E}$, by Lemma 4.5.1.4, $\mathit{head}(\mathit{bound}_\Delta(\mathtt{S}_0)) \not\trianglelefteq \mathtt{C}$ and $\mathtt{C} \not\trianglelefteq \mathit{head}(\mathit{bound}_\Delta(\mathtt{S}_0))$. By the rule GT-SCast, $\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathtt{N})\mathtt{e}_0 \in \mathtt{N}$ with *stupid warning*, finishing the case.

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

Proof: By induction on the derivation of $\Delta;\Gamma,\overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} \in \mathtt{T}$ with a case analysis on the last rule used.

Case GT-VAR: $\quad$ $\mathtt{e} = \mathtt{x}$

If $\mathtt{x} \in dom(\Gamma)$, then it's trivial since $[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x} = \mathtt{x}$. On the other hand, if $\mathtt{x} = \mathtt{x}_i$ and $\mathtt{T} = \mathtt{T}_i$, then letting $\mathtt{S} = \mathtt{S}_i$ finishing the case.

Case GT-FIELD: $\quad$ $\mathtt{e} = \mathtt{e}_0\mathtt{.f}_i \qquad \Delta;\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathtt{T} = \mathtt{T}_i$

By the induction hypothesis, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 \in \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta \vdash \mathtt{S}_0$ <: $\mathtt{T}_0$. By Lemma 4.5.1.10, $\mathit{fields}(\mathit{bound}_\Delta(\mathtt{S}_0)) = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$ such that $\mathtt{S}_j = \mathtt{T}_j$ and $\mathtt{f}_j = \mathtt{g}_j$ for all $j \leq \#(\overline{\mathtt{T}})$. Therefore, by the rule GT-FIELD, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\mathtt{.f}_i \in \mathtt{T}$

Case GT-INVK:

$$\begin{array}{lll} \mathtt{e} = \mathtt{e}_0\mathtt{.m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{\mathtt{e}}) & \Delta;\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e}_0 \in \mathtt{T}_0 & \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0)) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U} \\ \Delta \vdash \overline{\mathtt{V}}\ \text{ok} & \Delta \vdash \overline{\mathtt{V}} \texttt{ <: } [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} & \Delta;\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \\ \Delta \vdash \overline{\mathtt{S}} \texttt{ <: } [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} & \mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U} & \end{array}$$

By the induction hypothesis, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 \in \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta \vdash \mathtt{S}_0$ <: $\mathtt{T}_0$ and $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} \in \overline{\mathtt{W}}$ for some $\overline{\mathtt{W}}$ such that $\Delta \vdash \overline{\mathtt{W}}$ <: $\overline{\mathtt{S}}$. By Lemma 4.5.1.11, $\mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{S}_0)) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U}'$ and Δ, $\overline{\mathtt{Y}}$<:$\overline{\mathtt{P}} \vdash \mathtt{U}'$ <: $\mathtt{U}$. By Lemma 4.5.1.7, $\Delta \vdash [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}'$ <: $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$. By the rule GT-METHOD, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}](\mathtt{e}_0\mathtt{.m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{\mathtt{e}})) \in [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}'$. Letting $\mathtt{S} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}'$ finishes the case.

Case GT-NEW, GT-UCAST:

Easy.

Case GT-DCAST:

$$\begin{array}{lll} \Delta;\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e}_0 \in \mathtt{T}_0 & \mathtt{C} = \mathit{head}(\mathtt{N}) & \mathtt{E} = \mathit{head}(\mathit{bound}_\Delta(\mathtt{T}_0)) \\ \Delta \vdash \mathtt{N} \texttt{ <: } \mathit{bound}_\Delta(\mathtt{T}_0) & \mathtt{N} \neq \mathit{bound}_\Delta(\mathtt{T}_0) & \mathit{dcast}(\mathtt{C}, \mathtt{E}) \end{array}$$

By the induction hypothesis, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 \in \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta \vdash \mathtt{S}_0$ <: $\mathtt{T}_0$. We have three subcases according to a relation between $\mathtt{S}_0$ and $\mathtt{N}$.

Subcase: $\quad$ $\Delta \vdash \mathit{bound}_\Delta(\mathtt{S}_0)$ <: $\mathtt{N}$

By the rule GT-UCAST, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]((\mathtt{N})\,\mathtt{e}_0) \in \mathtt{N}$.

Subcase: $\quad$ $\Delta \vdash \mathtt{N}$ <: $\mathit{bound}_\Delta(\mathtt{S}_0)$

Since $\Delta \vdash \mathtt{S}_0$ <: $\mathtt{T}_0$ implies $\Delta \vdash \mathit{bound}_\Delta(\mathtt{S}_0)$ <: $\mathit{bound}_\Delta(\mathtt{T}_0)$, for any $\overline{\mathtt{S}}$ such that $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}$ <: $\mathit{bound}_\Delta(\mathtt{S}_0)$ and $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}$ ok, we have $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}$ <: $\mathit{bound}_\Delta(\mathtt{T}_0)$, which implies $\overline{\mathtt{S}} = \overline{\mathtt{U}}$ for all $\overline{\mathtt{S}}$. Finally, the rule GT-DCAST finishes the subcase.

Subcase: $\quad$ $\Delta \vdash \mathtt{N} \not<: \mathit{bound}_\Delta(\mathtt{S}_0) \qquad \Delta \vdash \mathit{bound}_\Delta(\mathtt{S}_0) \not<: \mathtt{N}$

Let $\mathit{bound}_\Delta(\mathtt{S}_0) = \mathtt{D}\texttt{<}\overline{\mathtt{V}}\texttt{>}$ and $\mathit{bound}_\Delta(\mathtt{T}_0) = \mathtt{E}\texttt{<}\overline{\mathtt{W}}\texttt{>}$. We show that, by contradiction, that $\mathtt{C} \ntrianglelefteq \mathtt{D}$ or $\mathtt{D} \ntrianglelefteq \mathtt{C}$.

Suppose $\mathtt{C} \trianglelefteq \mathtt{D}$. Then, we can have $\mathtt{C}\texttt{<}\overline{\mathtt{U}}'\texttt{>}$ such that $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{U}}'\texttt{>}$ <: $\mathtt{D}\texttt{<}\overline{\mathtt{V}}\texttt{>}$. By transitivity of <: and the fact that $\Delta \vdash \mathtt{S}_0$ <: $\mathtt{T}_0$ implies $\Delta \vdash \mathit{bound}_\Delta(\mathtt{S}_0)$ <: $\mathit{bound}_\Delta(\mathtt{T}_0)$, we have $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{U}}'\texttt{>}$ <: $\mathit{bound}_\Delta(\mathtt{T}_0)$. Thus, $\overline{\mathtt{U}}' = \overline{\mathtt{U}}$, contradicting the assumption $\Delta \vdash \mathtt{N} \not<: \mathit{bound}_\Delta(\mathtt{S}_0)$ $\;(= \mathtt{D}\texttt{<}\overline{\mathtt{V}}\texttt{>})$. On the other hand, suppose $\mathtt{D}$ `ext` $\mathtt{C}$. Then, we can have a sequence of types $\overline{\mathtt{T}}'$ where $\mathtt{T_1}' = \mathit{bound}_\Delta(\mathtt{S}_0)$ and $\mathtt{T}_n{}' = \mathit{bound}_\Delta(\mathtt{T}_0)$ and $\Delta \vdash \mathtt{T}_i{}'$ <: $\mathtt{T}_{i+1}{}'$ derived by S-CLASS; moreover it must include $\mathtt{C}\texttt{<}\overline{\mathtt{U}}'\texttt{>}$; then, $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{U}}'\texttt{>}$ <: $\mathit{bound}_\Delta(\mathtt{T}_0)$ and $\overline{\mathtt{U}}' = \overline{\mathtt{U}}$, contradicting the assumption $\Delta \vdash \mathit{bound}_\Delta(\mathtt{S}_0) \not<: \mathtt{N}$. Therefore, $\mathtt{C} \ntrianglelefteq \mathtt{D}$ and $\mathtt{D} \ntrianglelefteq \mathtt{C}$.

By the rule GT-SCAST, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]((\mathtt{N})\,\mathtt{e}_0) \in \mathtt{N}$ with *stupid warning.* ■

Case GT-SCAST:

$$\begin{array}{lll} \Delta;\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e}_0 \in \mathtt{T}_0 & \mathtt{C} = \mathit{head}(\mathtt{N}) & \mathtt{D} = \mathit{head}(\mathit{bound}_\Delta(\mathtt{T}_0)) \\ \mathtt{C} \ntrianglelefteq \mathtt{D} & \mathtt{D} \ntrianglelefteq \mathtt{C} & \end{array}$$

By the induction hypothesis, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 \in \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta \vdash \mathtt{S}_0$ <: $\mathtt{T}_0$, which implies $\Delta \vdash \mathit{bound}_\Delta(\mathtt{S}_0)$ <: $\mathit{bound}_\Delta(\mathtt{T}_0)$. Let $\mathtt{E} = \mathit{head}(\mathit{bound}_\Delta(\mathtt{S}_0))$. By Lemma 4.5.1.4, we have $\mathtt{E} \ntrianglelefteq \mathtt{C}$ and $\mathtt{C} \ntrianglelefteq \mathtt{E}$. Then, by the rule GT-SCAST, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]((\mathtt{N})\,\mathtt{e}_0) \in \mathtt{N}$ with *stupid warning.*

4.5.1.14 Lemma: If $\mathit{mtype}(\mathtt{m}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}}\to\mathtt{U}$ and $\mathit{mbody}(\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$ where $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ ok and $\Delta \vdash \overline{\mathtt{V}}$ ok and $\Delta \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$, then there exist some $\mathtt{N}$ and $\mathtt{S}$ such that $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} \texttt{<:} \mathtt{N}$ and $\Delta \vdash \mathtt{N}$ ok and $\Delta \vdash \mathtt{S} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$ and $\Delta \vdash \mathtt{S}$ ok and $\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{N} \vdash \mathtt{e}_0 \in \mathtt{S}$.

Proof: By induction on the derivation of $\mathit{mbody}(\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = (\overline{\mathtt{x}}, \mathtt{e})$ using Lemmas 4.5.1.7 and 4.5.1.12.

Case MB-Class: $\mathit{CT}(\mathtt{C}) = \mathtt{class\ C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>} \triangleleft \mathtt{P}\ \{\ldots\ \overline{\mathtt{M}}\}$
$\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{Q}}\texttt{>}\ \mathtt{T}_0\ \mathtt{m}\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\} \in \overline{\mathtt{M}}$

Let $\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{this} : \mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}$ and $\Delta' = \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}, \overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{Q}}$. By the rules GT-Class and GT-Method, we have $\Delta'; \Gamma \vdash \mathtt{e} \in \mathtt{S}_0$ and $\Delta'; \Gamma \vdash \mathtt{S}_0 \texttt{<:} \mathtt{T}_0$ for some $\mathtt{S}_0$. Since $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ ok, we have $\Delta \vdash \overline{\mathtt{T}} \texttt{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ by the rule WF-Class. By Lemmas 4.5.1.3, 4.5.1.7, and 4.5.1.12,

$$\Delta, \overline{\mathtt{Y}}\texttt{<:}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}} \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0 \texttt{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_0$$

and

$$\Delta, \overline{\mathtt{Y}}\texttt{<:}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}; \overline{\mathtt{x}} : [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}, \mathtt{this} : \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e} \in \mathtt{S}_0{}'$$

where

$$\Delta, \overline{\mathtt{Y}}\texttt{<:}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}} \vdash \mathtt{S}_0{}' \texttt{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0.$$

By the rule MT-Class, we have

$$[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}} = \overline{\mathtt{P}} \quad [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}} = \overline{\mathtt{U}} \quad [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_0 = \mathtt{U}.$$

Again, by Lemmas 4.5.1.7 and 4.5.1.12,

$$\Delta \vdash [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{S}_0{}' \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$$

and

$$\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} \vdash [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e} \in \mathtt{S}_0{}''.$$

where

$$\Delta \vdash \mathtt{S}_0{}'' \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{S}_0{}'.$$

Since any of $\overline{\mathtt{Y}}$ does not occur in $\overline{\mathtt{T}}$,

$$\mathtt{e}_0 = [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{e} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}.$$

Letting $\mathtt{N} = \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ and $\mathtt{S} = \mathtt{S}_0{}''$ finishes the case.

Case MB-Super: $\mathit{CT}(\mathtt{C}) = \mathtt{class\ C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>} \triangleleft \mathtt{N}\ \{\ldots\ \overline{\mathtt{M}}\}$
$\mathtt{m}$ is not defined in $\overline{\mathtt{M}}$.

Easy from the induction hypothesis and the fact that $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} \texttt{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}$. ■

Now we can prove Theorem 4.5.1.1.

Proof of Theorem 4.5.1.1: By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$ with a case analysis on the reduction rule used. We will show main cases.

Case GR-Field: $\quad \mathtt{e} = \mathtt{new\ N(\overline{e}).f}_i \quad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \quad \mathtt{e}' = \mathtt{e}_i$

By the rules GT-Field and GT-New, we have

$$\begin{array}{l} \Delta;\Gamma \vdash \mathtt{new\ N(\overline{e})} \in \mathtt{N} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \\ \Delta \vdash \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}}. \end{array}$$

In particular, $\Delta;\Gamma \vdash \mathtt{e}_i \in \mathtt{S}_i$ finishes the case.

Case GR-Invk: $\quad \mathtt{e} = \mathtt{new\ N(\overline{e}).m<\overline{V}>(\overline{d})} \quad \mathit{mbody}(\mathtt{m<\overline{V}>}, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$
$\mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(\overline{e})/this}]\mathtt{e}_0$

By the rules GT-Invk and GT-New, we have

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

By Lemma 4.5.1.14, $\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{P} \vdash \mathtt{e}_0 \in \mathtt{S}$ for some $\mathtt{P}$ and $\mathtt{S}$ such that $\Delta \vdash \mathtt{N} \mathrel{<:} \mathtt{P}$ where $\Delta \vdash \mathtt{P}$ ok, and $\Delta \vdash \mathtt{S} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$ where $\Delta \vdash \mathtt{S}$ ok. Then, by Lemmas 4.5.1.3 and 4.5.1.13, $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(\overline{e})/this}]\mathtt{e}_0 \in \mathtt{T}_0$ for some $\mathtt{T}_0$ such that $\Delta \vdash \mathtt{T}_0 \mathrel{<:} \mathtt{S}$. By the rule S-Trans, we have $\Delta \vdash \mathtt{T}_0 \mathrel{<:} \mathtt{T}$. Finally, letting $\mathtt{T}' = \mathtt{T}_0$ finishes the case.

Case GR-Cast:

Easy.

Case GRC-Field: $\quad \mathtt{e} = \mathtt{e}_0\mathtt{.f} \quad \mathtt{e}' = \mathtt{e}_0'\mathtt{.f} \quad \mathtt{e}_0 \longrightarrow \mathtt{e}_0'$

By the rule GT-Field, we have

$$\begin{array}{l} \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \mathtt{T} = \mathtt{T}_i \end{array}$$

By the induction hypothesis, $\Delta;\Gamma \vdash \mathtt{e}_0' \in \mathtt{T}_0'$ for some $\mathtt{T}_0'$ such that $\Delta \vdash \mathtt{T}_0' \mathrel{<:} \mathtt{T}_0$. By Lemma 4.5.1.10, $\mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0')) = \overline{\mathtt{T}'}\ \overline{\mathtt{g}}$, and for $j \leq \#(\overline{\mathtt{f}})$, we have $\mathtt{g}_i = \mathtt{f}_i$ and $\mathtt{T}_i' = \mathtt{T}_i$. Therefore, by the rule GT-Field, $\Delta;\Gamma \vdash \mathtt{e}_0'\mathtt{.f} \in \mathtt{T}_i'$. Letting $\mathtt{T}' = \mathtt{T}_i'$ finishes the case.

Case GRC-Inv-Recv: $\quad \mathtt{e} = \mathtt{e}_0\mathtt{.m<\overline{V}>(\overline{e})} \quad \mathtt{e}' = \mathtt{e}_0'\mathtt{.m<\overline{V}>(\overline{e})} \quad \mathtt{e}_0 \longrightarrow \mathtt{e}_0'$

By the rule GT-Invk, we have

$$\begin{array}{ll} \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 & \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0)) = \mathtt{<\overline{Y} \triangleleft \overline{P}>\overline{T}\to U} \\ \Delta \vdash \overline{\mathtt{V}} \text{ ok} & \Delta \vdash \overline{\mathtt{V}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \\ \Delta \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} & \Delta \vdash \overline{\mathtt{S}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}} \\ \mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U} & \end{array}$$

By the induction hypothesis, $\Delta;\Gamma \vdash \mathtt{e}_0' \in \mathtt{T}_0'$ for some $\mathtt{T}_0'$ such that $\Delta \vdash \mathtt{T}_0' \mathrel{<:} \mathtt{T}_0$. By Lemma 4.5.1.11, $\mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0')) = \mathtt{<\overline{Y} \triangleleft \overline{P}>\overline{T}\to V}$ and $\Delta, \overline{\mathtt{Y}} \mathrel{<:} \overline{\mathtt{P}} \vdash \mathtt{V} \mathrel{<:} \mathtt{U}$. By Lemma 4.5.1.7, $\Delta \vdash [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{V} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$. Then, by the rule GT-Invk, $\Delta;\Gamma \vdash \mathtt{e}_0'\mathtt{.m<\overline{V}>(\overline{e})} \in [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{V}$. Letting $\mathtt{T}_0' = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{V}$ finishes the case.

Case GRC-Cast: $\quad \mathtt{e} = \mathtt{(N)e}_0 \quad \mathtt{e}' = \mathtt{(N)e}_0' \quad \mathtt{e}_0 \longrightarrow \mathtt{e}_0'$

There are three subcases according to the last typing rule GT-UCast, GT-DCast or GT-SCast. These subcases are similar to the subcases in the case for GT-DCast in the proof of Lemma 4.5.1.13.

Case GRC-Inv-Arg, GRC-New-Arg:

Easy. ∎

4.5.2 Backward Compatibility

FGJ is backward compatible with FJ. Intuitively, this means that an implementation of FGJ can be used to typecheck and execute FJ programs without changing their meaning. In the following statements, we use subscripts FJ or FGJ to show which set of rules is used.

4.5.2.1 Lemma: If CT is an FJ class table, then $\mathit{fields}_{\mathrm{FJ}}(\mathtt{C}) = \mathit{fields}_{\mathrm{FGJ}}(\mathtt{C})$ for all $\mathtt{C} \in \mathit{dom}(CT)$.

4.5.2.2 Lemma: Suppose CT is an FJ class table. Then, $\mathit{mtype}_{\mathrm{FJ}}(\mathtt{m},\ \mathtt{C}) = \overline{\mathtt{C}} \rightarrow \mathtt{C}$ if and only if $\mathit{mtype}_{\mathrm{FGJ}}(\mathtt{m},\ \mathtt{C}) = \overline{\mathtt{C}} \rightarrow \mathtt{C}$.

4.5.2.3 Lemma: Suppose CT is an FJ class table. Then, $\mathit{mbody}_{\mathrm{FJ}}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e})$ if and only if $\mathit{mbody}_{\mathrm{FGJ}}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e})$.

Proof: All these lemmas are easy. Note that all substitutions in the derivations are always empty and there are no methods with type arguments. ∎

We can show that a well-typed FJ program is always a well-typed FGJ program and that FJ and FGJ reduction correspond. (Note that it isn't quite the case that the well-typedness of an FJ program under the FGJ rules implies its well-typedness in FJ, because FGJ allows covariant overriding of methods and FJ does not.)

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

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

4.6 Compiling FGJ to FJ

We now explore the second implementation style for GJ and FGJ. The current GJ compiler works by translation into the standard JVM, which maintains no information about type parameters at run-time. In the literature [OW97], there have been proposed two strategies for compiling parametric classes to non-parametric classes. One strategy, called *heterogeneous* translation adopted by implementation of C++ templates [Str97], is as follows. Suppose we have a parametric class `C<X>`; for each type parameter `T` used in a program, the compiler generates a specialized class obtained by replacing `X` with `T`. The other strategy, we discuss here, is called *homogeneous* style: the compiler translates one parametric class to one class by removing type parameter information. The class `C<X>` is compiled to a single class definition of `C`. We model this compilation in our framework by an *erasure* translation from FGJ into FJ. We show that this translation maps well-typed FGJ programs into well-typed FJ programs, and that the behavior of a program in FGJ matches (in a suitable sense) the behavior of its erasure under the FJ reduction rules.

We begin with a brief overview of the erasure translation with a concrete example. A program is erased by removing type parameters and inserting downcasts where required. The erasure of a non-variable type is obtained just by removing type parameters, and the erasure of type variables is the erasure of their bounds. For example, `Pair<A,B>` is erased to `Pair` and `X` in the definition of `Pair<X,Y>` class is erased to `Object`; the class `Pair<X,Y>` of Section 4.1 erases to the following:

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

For expressions, the erasure inserts downcasts to recover type information of the original program. For example, the field selection

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

erases to

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

Since the type of field `snd` is `Object` in the compiled definition, the downcast `(B)` is needed to use the result of the field access as a `B` object. We call such downcasts inserted by erasure *synthetic.* A key property of the erasure transformation is that it satisfies a so-called *cast-iron guarantee*: if the FGJ program is well-typed, then no synthetic downcast will fail at run-time. In the following discussion, we often distinguish synthetic casts from typecasts derived from original FGJ programs by superscripting typecast expression, writing $(\texttt{C})^s$. Otherwise, they behave exactly the same as ordinary typecasts.

In FGJ (and GJ), a subclass may extend an instantiated superclass. This means that, unlike in FJ (and Java), the types of the fields and the methods in the subclass may not be identical to the types in the superclass. For example, we may declare a specialized subclass `PairOfA` as a subclass of the instantiation `Pair<A,A>`, which instantiates both `X` and `Y` to a given class `A`.

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

Note that, in the `setfst` method, the argument type `A` matches the argument type of `setfst` in `Pair<A,A>`, while the result type `PairOfA` is a subtype of the result type in `Pair<A,A>`; this is permitted by FGJ's covariant overriding, found in the rule GT-METHOD. Erasing the class `PairOfA` yields the following:

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

Here arguments to the constructor and the method are given type `Object`, even though the erasure of `A` is itself; and the result of the method is given type `Pair`, even though the erasure of `PairOfA` is itself. In both cases, the types are chosen to correspond to types in `Pair`, the highest superclass in which the fields and method are defined.[1] Two synthetic downcasts $\texttt{(A)}^s$ are inserted to remember the intended types of `newfst` and the field `snd` are `A` (although, in this example, these synthetic casts do not have significance).

Now, we proceed to the formal definition of erasure.

4.6.1 Erasure of Types

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

$$|\texttt{T}|_\Delta = \mathit{head}(\mathit{bound}_\Delta(\texttt{T}))$$

4.6.2 Auxiliary Definitions

As we have seen above, in order to specify a type-preserving erasure from FGJ to FJ, it is necessary to know the type of a field or method in the *highest* superclass in which it is defined. The maximum field types of a class `C`, written *fieldsmax*(`C`), is the sequence of pairs of a type and a field name defined as follows:

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

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

The maximum method type of `m` in `C`, written *mtypemax*(`m`, `C`), is defined as follows:

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{...\}} \qquad \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}}\texttt{→T} = \mathit{mtype}(\texttt{m}, \texttt{N})}{\mathit{mtypemax}(\texttt{m}, \texttt{C}) = \mathit{mtypemax}(\texttt{m}, \mathit{head}(\texttt{D}))}$$

$$\frac{\begin{array}{c}\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{...\}} \\ \mathit{mtype}(\texttt{m}, \texttt{N}) \text{ undefined} \\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}}\texttt{→T} = \mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{X}}\texttt{>}) \qquad \Delta = \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}, \overline{\texttt{Y}}\texttt{<:}\overline{\texttt{P}}\end{array}}{\mathit{mtypemax}(\texttt{m}, \texttt{C}) = |\overline{\texttt{T}}|_\Delta \to |\texttt{T}|_\Delta}$$

Notice that *mtypemax*(`m`, `C`) returns only when `m` is defined in `C` but it is not defined in any superclasses of `C`: it ignores all overriding definitions of `m`.

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

[1]In Java, it is allowed that the argument types of the constructor to be `A`.

4.6.3 Erasure of Expressions

We write $|\mathtt{e}|_{\Delta,\Gamma}$ for the erasure of a well-typed expression $\mathtt{e}$ with respect to environment Γ and type environment Δ. The erasure of an expression depends on the typing of that expression, since the types are used to determine which downcasts to insert. The erasure rules are defined below.

The erasure of variables is the identity function.

$$|\mathtt{x}|_{\Delta,\Gamma} = \mathtt{x} \qquad \text{(E-VAR)}$$

We have two rules each for a field access and a method invocation. The erasure rules are optimized to omit synthetic casts when it is trivially safe to do so; this happens when the maximum type is equal to the erased type.

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.f} \in \mathtt{T} \qquad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{fieldsmax}(|\mathtt{T}_0|_\Delta)(\mathtt{f}) = |\mathtt{T}|_\Delta}{|\mathtt{e}_0\mathtt{.f}|_{\Delta,\Gamma} = |\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.f}} \qquad \text{(E-FIELD)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.f} \in \mathtt{T} \qquad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{fieldsmax}(|\mathtt{T}_0|_\Delta)(\mathtt{f}) \neq |\mathtt{T}|_\Delta}{|\mathtt{e}_0\mathtt{.f}|_{\Delta,\Gamma} = \mathtt{(}|\mathtt{T}|_\Delta\mathtt{)}^s|\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.f}}$$
(E-FIELD-CAST)

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T} \qquad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{mtypemax}(\mathtt{m}, |\mathtt{T}_0|_\Delta) = \overline{\mathtt{C}}\to\mathtt{D} \qquad \mathtt{D} = |\mathtt{T}|_\Delta}{|\mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma} = |\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.m(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}}$$
(E-INVK)

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T} \qquad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{mtypemax}(\mathtt{m}, |\mathtt{T}_0|_\Delta) = \overline{\mathtt{C}}\to\mathtt{D} \qquad \mathtt{D} \neq |\mathtt{T}|_\Delta}{|\mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma} = \mathtt{(}|\mathtt{T}|_\Delta\mathtt{)}^s|\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.m(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}}$$
(E-INVK-CAST)

The rules for object constructors and typecasts are straightforward:

$$|\mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma} = \mathtt{new}\ |\mathtt{N}|_\Delta\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)} \qquad \text{(E-NEW)}$$

$$|\mathtt{(N)e}_0|_{\Delta,\Gamma} = \mathtt{(}|\mathtt{N}|_\Delta\mathtt{)}\ |\mathtt{e}_0|_{\Delta,\Gamma} \qquad \text{(E-CAST)}$$

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

4.6.4 Erasure of Methods and Classes

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

$$\frac{\begin{array}{c}\Gamma = \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}},\mathtt{this:C<}\overline{\mathtt{X}}\mathtt{>} \qquad \Delta' = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}},\ \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \\ \mathit{mtypemax}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{D}}\to\mathtt{D} \qquad \mathtt{e}_i = \begin{cases} \mathtt{x}_i{}' & \text{if } \mathtt{D}_i = |\mathtt{T}_i|_{\Delta'} \\ \mathtt{(}|\mathtt{T}_i|_{\Delta'}\mathtt{)}^s\mathtt{x}_i{}' & \text{otherwise}\end{cases}\end{array}}{|\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>\ T\ m\ (}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}}|_{\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}},\mathtt{C}} = \mathtt{D\ m\ (}\overline{\mathtt{D}}\ \overline{\mathtt{x}}'\mathtt{)\ \{}\uparrow[\overline{\mathtt{e}}/\overline{\mathtt{x}}]|\mathtt{e}|_{\Delta',\Gamma}\mathtt{;\}}} \qquad \text{(E-METHOD)}$$

It erases the method body under the relevant environments; in case an erasure $|\mathtt{T}_i|_{\Delta'}$ of an argument type differs from the corresponding argument type $\mathtt{D}_i$ obtained from *mtypemax*(m, C), a synthetic cast is inserted for method arguments.

Remark: In GJ, the actual erasure is somewhat more complex, involving the introduction of *bridge* methods, so that one ends up with two overloaded methods: one with the maximum type, and one with the instantiated type. For example, the erasure of `PairOfA` would be:

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

where the second definition of `setfst` is the bridge method, which overrides the definition of `setfst` in `Pair`. We don't model that extra complexity here, because it depends on overloading of method names, which is not modeled in FJ.

The erasures of constructors and classes are straightforward:

$$|\mathtt{C(\overline{U}\ \overline{g},\ \overline{T}\ \overline{f})\ \{super(\overline{g});\ this.\overline{f}\ =\ \overline{f};\}}|_{\mathtt{C}} = \mathtt{C(}\mathit{fieldsmax}(\mathtt{C})\mathtt{)\ \{super(\overline{g});\ this.\overline{f}\ =\ \overline{f};\}} \qquad \text{(E-CONSTR)}$$

The erasure of a constructor just replaces argument declarations with *fieldsmax*(`C`).

$$\frac{\Delta = \mathtt{\overline{X}{<:}\overline{N}}}{|\mathtt{class\ C{<}\overline{X}\triangleleft\overline{N}{>}\triangleleft N\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}}| = \mathtt{class\ C}\triangleleft|\mathtt{N}|_{\Delta}\mathtt{\{}|\mathtt{\overline{T}}|_{\Delta}\ \mathtt{\overline{f};}\ |\mathtt{K}|_{\mathtt{C}}\ |\mathtt{\overline{M}}|_{\Delta,\mathtt{C}}\mathtt{\}}} \qquad \text{(E-CLASS)}$$

We write $|CT|$ for the erasure of a class table CT, defined in an obvious way.

4.7 Properties of Compilation

Having defined erasure, we may investigate some of its properties. As in the discussion of backward compatibility in Section 4.5.2, we often use subscripts FJ or FGJ to avoid confusion.

4.7.1 Preservation of Typing

First, a well-typed FGJ program erases to a well-typed FJ program, as expected; moreover, synthetic casts are not stupid.

4.7.1.1 Theorem [Erasure preserves typing]: If an FGJ class table CT is ok and $\Delta;\Gamma \vdash_{\text{FGJ}} \mathtt{e} \in \mathtt{T}$, then $|CT|$ is ok using the FJ typing rules and $|\Gamma|_{\Delta} \vdash_{\text{FJ}} |\mathtt{e}|_{\Delta,\Gamma} \in |\mathtt{T}|_{\Delta}$. Moreover, every synthetic cast in $|CT|$ and $|\mathtt{e}|_{\Delta,\Gamma}$ does not involve *stupid warning.*

First, we show that, if an expression is well-typed, then its type is well formed (Lemma 4.7.1.5).

4.7.1.2 Lemma: If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta \vdash \mathtt{S}$ ok for some well-formed type environment Δ, then $\Delta \vdash \mathtt{T}$ ok.

Proof: By induction on the derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$ with a case analysis on the last rule used. The cases for S-REFL and S-TRANS are easy.

Case S-Var: $\mathtt{S} = \mathtt{X}$ $\mathtt{T} = \Delta(\mathtt{X})$

$\mathtt{T}$ must be well formed since Δ is well formed.

Case S-Class: $\mathtt{S} = \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ $\mathtt{T} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}$ $CT(\mathtt{C}) = \mathtt{class\ C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>} \triangleleft \mathtt{N}\ \{\ldots\}$
$\Delta \vdash \overline{\mathtt{T}}$ ok $\Delta \vdash \overline{\mathtt{T}}$ <: $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$

Since $CT(\mathtt{C})$ is ok, we also have $\overline{\mathtt{X}}$<:$\overline{\mathtt{N}} \vdash \mathtt{N}$ ok by the rule GT-Class. Then, by Lemmas 4.5.1.3 and 4.5.1.8, $\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}$ ok. ■

4.7.1.3 Lemma: If $\Delta \vdash \mathtt{N}$ ok for some well-formed type environment Δ and $\mathit{fields}_{\mathrm{FGJ}}(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$, then $\Delta \vdash \overline{\mathtt{U}}$ ok.

Proof: By induction on the derivation of $\mathit{fields}_{\mathrm{FGJ}}(\mathtt{N})$ with a case analysis on the last rule used. The case for F-Object is trivial.

Case F-Class: $\mathtt{N} = \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ $CT(\mathtt{C}) = \mathtt{class\ C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>} \triangleleft \mathtt{P}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}$
$\mathit{fields}_{\mathrm{FGJ}}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}$

Since $CT(\mathtt{C})$ is ok, by the rule GT-Class, $\overline{\mathtt{X}}$<:$\overline{\mathtt{N}} \vdash \mathtt{P}$ ok. By Lemmas 4.5.1.3 and 4.5.1.8, $\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P}$ ok. Then, by the induction hypothesis, $\Delta \vdash \overline{\mathtt{U}}$ ok. Since $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ ok, we have $\Delta \vdash \overline{\mathtt{T}}$ ok and $\Delta \vdash \overline{\mathtt{T}}$ <: $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ by the rule WF-Class. On the other hand, by the rule GT-Class, we have $\overline{\mathtt{X}}$<:$\overline{\mathtt{N}} \vdash \overline{\mathtt{S}}$ ok. Finally, by Lemmas 4.5.1.3 and 4.5.1.8, $\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}$ ok finishing the case. ■

4.7.1.4 Lemma: If $\Delta \vdash \mathtt{N}$ ok for some well-formed type environment Δ and $\mathit{mtype}_{\mathrm{FGJ}}(\mathtt{m},\ \mathtt{N}) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}}{\rightarrow}\mathtt{U}_0$, then $\Delta, \overline{\mathtt{Y}}$<:$\overline{\mathtt{P}} \vdash \mathtt{U}_0$ ok.

Proof: By induction on the derivation of $\mathit{mtype}_{\mathrm{FGJ}}(\mathtt{m},\ \mathtt{N})$ with a case analysis on the last rule used.

Case MT-Class: $\mathtt{N} = \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$
$CT(\mathtt{C}) = \mathtt{class\ C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>} \triangleleft \mathtt{P}\ \{\ldots\ \ \overline{\mathtt{M}}\}$
$\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{Q}}\texttt{>}\ \mathtt{S}_0\ \mathtt{m}\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e}_0;\} \in \overline{\mathtt{M}}$
$[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{Q}}\texttt{>}\overline{\mathtt{S}}{\rightarrow}\mathtt{S}_0) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}}{\rightarrow}\mathtt{U}_0$

Without loss of generality, we can assume that $\overline{\mathtt{X}}$ and $\overline{\mathtt{Y}}$ are distinct and that $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}} = \overline{\mathtt{P}}$ and $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0 = \mathtt{U}_0$. By the rule GT-Method, we have

$$\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}},\ \overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{Q}} \vdash \mathtt{S}_0 \text{ ok.}$$

By the rule WF-Class, we have $\Delta \vdash \overline{\mathtt{T}}$ ok and $\Delta \vdash \overline{\mathtt{T}}$ <: $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$. Then, by Lemma 4.5.1.3 and 4.5.1.8,

$$\Delta,\ \overline{\mathtt{Y}}\texttt{<:}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}} \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0 \text{ ok.}$$

finishing the case.

Case MT-Super:

Since $CT(\mathtt{C})$ is ok, by the rule GT-Class, $\overline{\mathtt{X}}$<:$\overline{\mathtt{N}} \vdash \mathtt{P}$ ok. By Lemmas 4.5.1.3 and 4.5.1.8, $\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P}$ ok. The induction hypothesis finishes the case. ■

4.7.1.5 Lemma: If $\Delta \vdash \Gamma$ ok and $\Delta; \Gamma \vdash_{\mathrm{FGJ}} \mathtt{e} \in \mathtt{T}$ for some well-formed type environment Δ, then $\Delta \vdash \mathtt{T}$ ok.

Proof: By induction on the derivation of $\Delta; \Gamma \vdash_{\mathrm{FGJ}} \mathtt{e} \in \mathtt{T}$ with a case analysis on the last rule used.

Case GT-VAR:

Trivial by the definition of well-formedness of Γ.

Case GT-FIELD: $\Delta;\Gamma \vdash_{\text{FGJ}} \texttt{e}_0 \in \texttt{T}_0 \quad \textit{fields}_{\text{FGJ}}(\textit{bound}_\Delta(\texttt{T}_0)) = \overline{\texttt{T}}\ \overline{\texttt{f}}$

By the induction hypothesis, $\Delta \vdash \texttt{T}_0$ ok. Since Δ is well formed, $\Delta \vdash \textit{bound}_\Delta(\texttt{T}_0)$ ok. Then, by Lemma 4.7.1.3, we have $\Delta \vdash \overline{\texttt{T}}$ ok, finishing the case.

Case GT-INVK:
$$\begin{array}{ll} \Delta;\Gamma \vdash_{\text{FGJ}} \texttt{e}_0 \in \texttt{T}_0 & \textit{mtype}_{\text{FGJ}}(\texttt{m},\ \textit{bound}_\Delta(\texttt{T}_0)) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{U} \\ \Delta \vdash \overline{\texttt{V}} \text{ ok} & \Delta \vdash \overline{\texttt{V}} \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{P}} \\ \Delta;\Gamma \vdash_{\text{FGJ}} \overline{\texttt{e}} \in \overline{\texttt{S}} & \Delta \vdash \overline{\texttt{S}} \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}} \\ \texttt{T} = [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0 & \end{array}$$

By the induction hypothesis, $\Delta \vdash \texttt{T}_0$ ok. Since Δ is well formed, $\Delta \vdash \textit{bound}_\Delta(\texttt{T}_0)$ ok. Then, by Lemma 4.7.1.4, $\Delta, \overline{\texttt{Y}}\texttt{<:}\overline{\texttt{P}} \vdash \texttt{U}_0$ ok. Finally, by Lemma 4.5.1.8, we have $\Delta \vdash [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0$ ok finishing the case.

Case GT-UCAST: $\Delta;\Gamma \vdash_{\text{FGJ}} \texttt{e}_0 \in \texttt{T}_0 \quad \Delta \vdash \texttt{T}_0 \texttt{ <: N}$

By the induction hypothesis, $\Delta \vdash \texttt{T}_0$ ok. By Lemma 4.7.1.2, $\Delta \vdash \texttt{N}$ ok finishes the case.

Case GT-NEW, GT-DCAST, GT-SCAST:

Trivial since T is well formed by assumption. ■

After developing several lemmas about erasure, we prove Theorem 4.7.1.1.

4.7.1.6 Lemma: If $\Delta \vdash \texttt{S} \texttt{ <:}_{\text{FGJ}} \texttt{T}$, then $|\texttt{S}|_\Delta \texttt{ <:}_{\text{FJ}} |\texttt{T}|_\Delta$.

Proof: Straightforward induction on the derivation of $\Delta \vdash \texttt{S} \texttt{ <:}_{\text{FGJ}} \texttt{T}$. ■

4.7.1.7 Lemma: If $\Delta_1, \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}, \Delta_2 \vdash \texttt{U}$ ok where none of $\overline{\texttt{X}}$ appear in Δ_1, and $\Delta_1 \vdash \overline{\texttt{T}} \texttt{ <:}_{\text{FGJ}} [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}}$, then $|[\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{U}|_{\Delta_1,\,[\overline{\texttt{T}}/\overline{\texttt{X}}]\Delta_2} \texttt{ <:}_{\text{FJ}} |\texttt{U}|_\Delta$.

Proof: If U is nonvariable or a type variable $\texttt{Y} \notin \overline{\texttt{X}}$, then the result is trivial. If U is a type variable $\texttt{X}_i$, it's also easy since $[\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{U} = \texttt{T}_i$ and, by Lemma 4.7.1.6, $|\texttt{T}_i|_{\Delta_1,\,[\overline{\texttt{T}}/\overline{\texttt{X}}]\Delta_2} = |\texttt{T}_i|_{\Delta_1} \texttt{ <:}_{\text{FJ}} |[\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}_i|_{\Delta_1} = |\texttt{N}_i|_\Delta = |\texttt{X}|_\Delta$. ■

4.7.1.8 Lemma: If $\Delta \vdash \texttt{C<}\overline{\texttt{U}}\texttt{>}$ ok and $\textit{fields}_{\text{FGJ}}(\texttt{C<}\overline{\texttt{U}}\texttt{>}) = \overline{\texttt{V}}\ \overline{\texttt{f}}$, then $\textit{fieldsmax}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{f}}$ and $|\overline{\texttt{V}}|_\Delta \texttt{ <:}_{\text{FJ}} \overline{\texttt{D}}$.

Proof: By induction on the derivation of $\textit{fields}_{\text{FGJ}}(\texttt{C<}\overline{\texttt{U}}\texttt{>})$ using Lemma 4.7.1.7 and the fact that $\Delta \vdash \overline{\texttt{U}} \texttt{ <: } [\overline{\texttt{U}}/\overline{\texttt{X}}]\overline{\texttt{N}}$, where $\textit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \ldots$, derived from the rule WF-CLASS. ■

4.7.1.9 Lemma: If $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}$ ok and $\textit{mtype}_{\text{FGJ}}(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{U}_0$ where $\Delta \vdash \overline{\texttt{V}} \texttt{ <:}_{\text{FGJ}} [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{P}}$, then $\textit{mtypemax}(\texttt{m},\ \texttt{C}) = \overline{\texttt{C}}\rightarrow\texttt{C}_0$ and $|[\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}}|_\Delta \texttt{ <:}_{\text{FJ}} \overline{\texttt{C}}$ and $|[\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0|_\Delta \texttt{ <:}_{\text{FJ}} \texttt{C}_0$.

Proof: Since $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}$ ok, we can have a sequence of type $\overline{\texttt{S}}$ such that $\texttt{S}_1 = \texttt{C<}\overline{\texttt{T}}\texttt{>}$ and $\texttt{S}_n = \texttt{Object}$ and $\Delta \vdash \texttt{S}_i \texttt{ <:}_{\text{FGJ}} \texttt{S}_{i+1}$ derived by the rule S-CLASS for any i. We prove by induction on the length n of the sequence.

Case: $n = 2$
It must be the case that

$$\begin{array}{l} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{Object \{ ...} \\ \qquad\quad \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{Q}}\texttt{>W}_0\texttt{ m (}\overline{\texttt{W}}\ \overline{\texttt{x}}\texttt{) \{...\} ...\}}. \end{array}$$

By the definition of *mtypemax*, $\overline{\texttt{C}} = |\overline{\texttt{W}}|_{\overline{\texttt{X}}<:\overline{\texttt{N}},\,\overline{\texttt{Y}}<:\overline{\texttt{Q}}}$ and $\texttt{C}_0 = |\texttt{W}_0|_{\overline{\texttt{X}}<:\overline{\texttt{N}},\,\overline{\texttt{Y}}<:\overline{\texttt{Q}}}$. Without loss of generality, we can assume $\overline{\texttt{X}}$ and $\overline{\texttt{Y}}$ are distinct. By the definition of $\mathit{mtype}_{\text{FGJ}}$,

$$\begin{array}{l} [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{Q}} = \overline{\texttt{P}} \\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{W}} = \overline{\texttt{U}} \\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{W}_0 = \texttt{U}_0, \end{array}$$

and therefore

$$\Delta \vdash \overline{\texttt{V}} <:_{\text{FGJ}} [\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{Q}}.$$

Moreover, by the rule WF-CLASS, we have

$$\Delta \vdash \overline{\texttt{T}} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}} \quad (= [\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}} \text{ since } \overline{\texttt{Y}} \text{ do not appear in } [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}}).$$

By Lemma 4.7.1.7, $|[\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{W}}|_\Delta <:_{\text{FJ}} \overline{\texttt{C}}$ and $|[\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{W}_0|_\Delta <:_{\text{FJ}} \texttt{C}_0$, finishing the case.

Case: $n = k + 1$
Suppose

$$CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{ ...\}}.$$

Note that $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <:_{\text{FGJ}} [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}$ by the rule S-CLASS. Now, we have three subcases:

Subcase: $\mathit{mtype}_{\text{FGJ}}(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})$ is not well defined.
The method `m` must be declared in `C`. Similarly for the base case above.

Subcase: $\mathit{mtype}_{\text{FGJ}}(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})$ is well defined and `m` is defined in `C`.
By the rule GT-METHOD, it must be the case that

$$\mathit{mtype}_{\text{FGJ}}(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\texttt{→}\texttt{U}_0{}'$$

where $\Delta,\ \overline{\texttt{Y}}\texttt{<:}\overline{\texttt{P}} \vdash \texttt{U}_0 <:_{\text{FGJ}} \texttt{U}_0{}'$. By Lemmas 4.5.1.7 and 4.7.1.6, $|[\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0|_\Delta <:_{\text{FJ}} |[\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0{}'|_\Delta$. The induction hypothesis and transitivity of `<:` finish the subcase.

Subcase: $\mathit{mtype}_{\text{FGJ}}(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})$ is well defined and `m` is not defined in `C`.
It is easy because $\mathit{mtype}_{\text{FGJ}}(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}) = \mathit{mtype}_{\text{FGJ}}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>})$, by the rule MT-SUPER. The induction hypothesis finishes the subcase. ■

Proof of Theorem 4.7.1.1: We prove the theorem in three steps: first, we show $|CT|$ is well defined; second, it is shown that, if $\Delta;\Gamma \vdash_{\text{FGJ}} \texttt{e} \in \texttt{T}$, then $|\Gamma|_\Delta \vdash_{\text{FJ}} |\texttt{e}|_{\Delta,\Gamma} \in |\texttt{T}|_\Delta$; and third, we show $|CT|$ is ok.

The first part is easy because every method body is well typed and every type is well formed under appropriate (type) environments. Now, by the definition of erasure, it is obvious that $\mathit{fields}_{\text{FJ}}(\texttt{C}) = \mathit{fieldsmax}(\texttt{C})$ and $\mathit{mtype}_{\text{FJ}}(\texttt{m}, \texttt{C}) = \mathit{mtypemax}(\texttt{m}, \texttt{C})$ for all `m` and `C`.

The second part is proved by induction on the derivation of $\Delta;\Gamma \vdash_{\text{FGJ}} \texttt{e} \in \texttt{T}$ with a case analysis on the last rule used.

Case GT-Field: $\quad e = e_0.f_i \quad \Delta;\Gamma \vdash_{FGJ} e_0 \in T_0 \quad \mathit{fields}_{FGJ}(\mathit{bound}_\Delta(T_0)) = \overline{T}\ \overline{f} \quad T = T_i$

By the induction hypothesis, we have $|\Gamma|_\Delta \vdash_{FJ} |e_0|_\Delta \in |T_0|_\Delta$. By Lemma 4.7.1.5, $\Delta \vdash T_0$ ok. Then, whether T_0 is a type variable or not, we have, by Lemma 4.7.1.8, $\mathit{fieldsmax}(|T_0|_\Delta) = \overline{C}\ \overline{f}$ and $|\overline{T}|_\Delta$ <: $\overline{C}$. By the rule T-Field, we have $|\Gamma|_\Delta \vdash_{FJ} |e_0|_{\Delta,\Gamma}.f_i \in C_i$.

If $|T_i|_\Delta = C_i$, then the equation $|e_0.f_i|_{\Delta,\Gamma} = |e_0|_{\Delta,\Gamma}.f_i$ derived from the rule E-Field finishes the case. On the other hand, if $(|T_i|_\Delta \neq C_i)$, then

$$|e_0.f_i|_{\Delta,\Gamma} = (|T_i|_\Delta)^s|e_0|_{\Delta,\Gamma}.f_i$$

by the rule E-Field-Cast and $|\Gamma|_\Delta \vdash_{FJ} (|T|_\Delta)^s|e_0|_{\Delta,\Gamma}.f_i \in |T|_\Delta$ by the rule T-DCast, finishing the case. Note that the synthetic cast is not stupid.

Case GT-Invk:
Similar to the case above.

Case GT-New, GT-UCast, GT-DCast, GT-SCast:
Easy. Notice that the nature of the cast (up, down, or stupid) is also preserved.

The third part ($|CT|$ is ok) follows from the first part with examination of the rules GT-Method and GT-Class. We show that, if M OK IN C<$\overline{X} \triangleleft \overline{N}$> and $|M|_{\overline{X}<:\overline{N},\,C} = M'$, then M$'$ OK IN C. Suppose

$$\begin{aligned}
&M = \texttt{<}\overline{Y} \triangleleft \overline{P}\texttt{> T m (}\overline{T}\texttt{ x) \{}\uparrow\texttt{e;\}}\\
&M' = \texttt{D m (}\overline{D}\texttt{ x}'\texttt{) \{}\uparrow\texttt{e}'\texttt{;\}}\\
&\mathit{mtypemax}(m, C) = \overline{D}\to D\\
&\Gamma = \overline{x} : \overline{T}\\
&\Delta = \overline{X}<:\overline{N}, \overline{Y}<:\overline{P}\\
&e_i = \begin{cases} x_i' & \text{if } D_i = |T_i|_\Delta\\ (|T_i|_\Delta)^s x_i' & \text{otherwise}\end{cases}\\
&e' = [\overline{e}/\overline{x}]|e|_{\Delta,\,(\Gamma,\texttt{this}:C<\overline{X}>)}.
\end{aligned}$$

By the rule GT-Method, we have

$$\begin{aligned}
&\Delta \vdash \overline{T}, T, \overline{P} \text{ ok}\\
&\Delta;\Gamma, \texttt{this} : C<\overline{X}> \vdash_{FGJ} e \in S\\
&\Delta \vdash S <:_{FGJ} T\\
&\text{if } \mathit{mtype}_{FGJ}(m, N) = \texttt{<}\overline{Z} \triangleleft \overline{Q}\texttt{>}\overline{U}\to U\text{, then } \overline{P}, \overline{T} = [\overline{Y}/\overline{Z}](\overline{Q}, \overline{U}) \text{ and } \Delta \vdash T <:_{FGJ} [\overline{Y}/\overline{Z}]U
\end{aligned}$$

where CT(C) = class C<$\overline{X} \triangleleft \overline{N}$>$\triangleleft$N {...}. We must show that

$$\begin{aligned}
&\overline{x}' : \overline{D}, \texttt{this} : C \vdash_{FJ} e' \in E\\
&E <:_{FJ} D\\
&\text{if } \mathit{mtype}_{FJ}(m, |N|_\Delta) = \overline{E}\to D'\text{, then } \overline{E} = \overline{D} \text{ and } D' = D.
\end{aligned}$$

for some E. By the result of the second part, $|\Gamma|_\Delta$, this : C $\vdash_{FJ} |e|_{\Delta,\Gamma} \in |S|_\Delta$. Since, by Lemma 4.7.1.9, $|T_i|_\Delta$ <: D_i, we have $x_i' : D_i \vdash e_i \in |T_i|_\Delta$. By Lemma 4.5.1.13,

$$\overline{x}' : \overline{D}, \texttt{this} : C \vdash e' \in C_0$$

for some C_0 where $C_0 <:_{FJ} |S|_\Delta$. On the other hand, by Lemma 4.7.1.9, $|T|_\Delta$ <: D. Since we have $|S|_\Delta$ <: $|T|_\Delta$ by Lemma 4.7.1.6, C_0 <: D by transitivity of <:. Let E be C_0. Finally, suppose $\mathit{mtypemax}(m, |N|_\Delta)$ is well defined. Then, $\mathit{mtype}_{FGJ}(m, N)$ is also well defined. By definition, $\mathit{mtypemax}(m, |N|_\Delta) = \overline{D}\to D = \mathit{mtype}_{FJ}(m, |N|_\Delta)$.

It is easy to show that L OK in FGJ implies |L| OK in FJ. ■

4.7.2 Preservation of Execution

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

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

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

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

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

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

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

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

$\mathsf{e} = \texttt{new Pair<A,B>(a, b).setfst<B>(b}'\texttt{)}$

and its erasure

$|\mathsf{e}|_{\Delta,\Gamma} = \texttt{new Pair(}|\mathsf{a}|_{\Delta,\Gamma}\texttt{,}|\mathsf{b}|_{\Delta,\Gamma}\texttt{).setfst(}|\mathsf{b}'|_{\Delta,\Gamma}\texttt{).}$

where `a` is an expression of type `A` and `b` and `b′` are of type `B`. In FGJ,

$\mathsf{e} \longrightarrow_{\mathrm{FGJ}} \texttt{new Pair<B,B>(b}'\texttt{,new Pair<A,B>(a,b).snd).}$

In FJ, on the other hand,

$|\mathsf{e}|_{\Delta,\Gamma} \longrightarrow_{\mathrm{FJ}} \texttt{new Pair(}|\mathsf{b}'|_{\Delta,\Gamma}\texttt{,new Pair(}|\mathsf{a}|_{\Delta,\Gamma}\texttt{,}|\mathsf{b}|_{\Delta,\Gamma}\texttt{).snd)}$

which has fewer synthetic casts than

$\texttt{new Pair(}|\mathsf{b}'|_{\Delta,\Gamma}\texttt{,(B)}^{s}\texttt{new Pair(}|\mathsf{a}|_{\Delta,\Gamma}\texttt{,}|\mathsf{b}|_{\Delta,\Gamma}\texttt{).snd),}$

which is the erasure of the reduced expression in FGJ. The subtlety we observe here is that, when the erased term is reduced, synthetic casts may become "coarser" than the casts inserted when the reduced term is erased, or may be removed entirely as in this example. (Removal of downcasts can be considered as a combination of two operations: replacement of $\texttt{(A)}^s$ with the coarser cast $\texttt{(Object)}^s$ and removal of the upcast $\texttt{(Object)}^s$, which does not affect the result of computation.)

To formalize both of these observations, we define an auxiliary relation that relates FJ expressions differing only by the addition and replacement of some synthetic casts. Suppose $\Gamma \vdash_{\mathrm{FJ}} \texttt{e} \in \texttt{C}$. Let us call an expression `d` an *expansion* of `e` under Γ, written $\Gamma \vdash \texttt{e} \overset{\text{exp}}{\Longrightarrow} \texttt{d}$, if `d` is obtained from `e` by some combination of (1) addition of zero or more synthetic upcasts, (2) replacement of some synthetic casts `(D)` with `(C)`, where `C` is a supertype of `D`, or (3) removal of some synthetic casts, and $\Gamma \vdash_{\mathrm{FJ}} \texttt{d} \in \texttt{D}$ for some `D`.

4.7.2.1 Example: Suppose Γ = `x:A`, `y:B`, `z:B` for given classes `A` and `B`. Then,

$$\Gamma \vdash \texttt{x} \overset{\text{exp}}{\Longrightarrow} \texttt{(A)}^s\texttt{x}$$

and

$$\Gamma \vdash \texttt{new Pair(z,(B)}^s\texttt{new Pair(x,y).snd)} \overset{\text{exp}}{\Longrightarrow} \texttt{new Pair(z,new Pair(x,y).snd)}.$$

Then, reduction commutes with erasure modulo expansion:

4.7.2.2 Theorem [Erasure preserves reduction modulo expansion]: If $\Delta; \Gamma \vdash \texttt{e} \in \texttt{T}$ and $\texttt{e} \longrightarrow_{\mathrm{FGJ}}{}^* \texttt{e}'$, then there exists some FJ expression $\texttt{d}'$ such that $|\Gamma|_\Delta \vdash |\texttt{e}'|_{\Delta,\Gamma} \overset{\text{exp}}{\Longrightarrow} \texttt{d}'$ and $|\texttt{e}|_{\Delta,\Gamma} \longrightarrow_{\mathrm{FJ}} \texttt{d}'$. In other words, the following diagram commutes.

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

Conversely, for the execution of an erased expression, there is a corresponded execution in FGJ semantics:

4.7.2.3 Theorem [Erased program reflects FGJ execution]: Suppose $\Delta; \Gamma \vdash \texttt{e} \in \texttt{T}$ and $|\Gamma|_\Delta \vdash |\texttt{e}|_{\Delta,\Gamma} \overset{\text{exp}}{\Longrightarrow} \texttt{d}$. If `d` reduces to $\texttt{d}'$ with zero or more steps by reducing at synthetic casts, followed by one step by other kinds of reduction, then $\texttt{e} \longrightarrow_{\mathrm{FGJ}} \texttt{e}'$ for some $\texttt{e}'$ and $|\Gamma|_\Delta \vdash |\texttt{e}'|_{\Delta,\Gamma} \overset{\text{exp}}{\Longrightarrow} \texttt{d}'$. In other words, the following diagram commutes.

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

As easy corollaries of these theorems, it can be shown that, if an FGJ expression $\mathtt{e}$ reduces to a "fully-evaluated expression," then the erasure of $\mathtt{e}$ reduces to exactly its erasure and vice versa. Similarly, if FGJ reduction gets stuck at a stupid cast, then FJ reduction also gets stuck because of the same typecast and vice versa. We use the metavariable $\mathtt{v}$ (or $\mathtt{w}$) for fully evaluated FGJ (or FJ, respectively) expressions, defined as follows:

$$\mathtt{v} ::= \mathtt{new\ N(\overline{v})}$$
$$\mathtt{w} ::= \mathtt{new\ C(\overline{w})}$$

4.7.2.4 Corollary [Erasure preserves execution results]: If $\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\mathtt{e} \longrightarrow_{\mathrm{FGJ}}{}^{*}\ \mathtt{v}$, then $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\mathrm{FJ}}{}^{*}\ |\mathtt{v}|_{\Delta,\Gamma}$. Similarly, if $\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\mathrm{FJ}}{}^{*}\ \mathtt{w}$, then there exists an FGJ expression $\mathtt{v}$ such that $\mathtt{e} \longrightarrow_{\mathrm{FGJ}}{}^{*}\ \mathtt{v}$ and $|\mathtt{v}|_{\Delta,\Gamma} = \mathtt{w}$.

4.7.2.5 Corollary [Erasure preserves typecast errors]: If $\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\mathtt{e} \longrightarrow_{\mathrm{FGJ}}{}^{*}\ \mathtt{e}'$, where $\mathtt{e}'$ has a stuck subexpression $\mathtt{(C<\overline{S}>)new\ D<\overline{T}>(\overline{e})}$, then $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\mathrm{FJ}}{}^{*}\ \mathtt{d}'$ such that $\mathtt{d}'$ has a stuck subexpression $\mathtt{(C)new\ D(\overline{d})}$, where $\overline{\mathtt{d}}$ are expansions of the erasures of $\overline{\mathtt{e}}$, at the same position (modulo synthetic casts) as the erasure of $\mathtt{e}'$. Similarly, if $\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\mathrm{FJ}}{}^{*}\ \mathtt{e}'$, where $\mathtt{e}'$ has a stuck subexpression $\mathtt{(C)new\ D<\overline{T}>(\overline{e})}$, then there exists an FGJ expression $\mathtt{d}$ such that $\mathtt{e} \longrightarrow_{\mathrm{FGJ}}{}^{*}\ \mathtt{d}$ and $|\Gamma|_{\Delta} \vdash |\mathtt{d}|_{\Delta,\Gamma} \overset{\mathrm{exp}}{\Longrightarrow}{}^{*}\ \mathtt{e}'$ and $\mathtt{d}$ has a stuck subexpression $\mathtt{(C<\overline{S}>)new\ D<\overline{T}>(\overline{d})}$, where $\overline{\mathtt{e}}$ are expansions of the erasures of $\overline{\mathtt{d}}$, at the same position (modulo synthetic casts) as $\mathtt{e}'$.

In the rest of this section, we prove these theorems and corollaries; we first prove the required lemmas.

4.7.2.6 Lemma: Suppose $\mathit{dom}(\Gamma) = \mathit{dom}(\Gamma')$ and $\Delta = \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2$ where none of $\overline{\mathtt{X}}$ appears in Δ_1. If $\Delta;\Gamma \vdash_{\mathrm{FGJ}} \mathtt{e} \in \mathtt{T}$ and $\Delta_1 \vdash \overline{\mathtt{U}} \mathrel{\mathtt{<:}_{\mathrm{FGJ}}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ where $\Delta_1 \vdash \overline{\mathtt{U}}$ ok, and $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \Gamma'(\mathtt{x}) \mathrel{\mathtt{<:}_{\mathrm{FGJ}}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma(\mathtt{x})$ for all $\mathtt{x} \in \mathit{dom}(\Gamma)$, then $|\mathtt{e}|_{\Delta,\Gamma}$ is obtained from $|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\Gamma'}$ by some combination of replacements of some synthetic casts $\mathtt{(D)}^{s}$ with $\mathtt{(C)}^{s}$ where $\mathtt{D} \mathrel{\mathtt{<:}} \mathtt{C}$, or removals of some synthetic casts.

Proof: By induction on the derivation of $\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T}$ with a case analysis on the last rule used.
Case GT-VAR:
Trivial.
Case GT-FIELD: $\quad \mathtt{e} = \mathtt{e}_0\mathtt{.f} \qquad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{fields}_{\mathrm{FGJ}}(\mathit{bound}_{\Delta}(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathtt{T} = \mathtt{T}_i$
By the induction hypothesis, $|\mathtt{e}_0|_{\Delta,\Gamma}$ is obtained from $|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'}$ by some combination of replacements of some synthetic casts $\mathtt{(D)}^{s}$ with $\mathtt{(C)}^{s}$ where $\mathtt{D} \mathrel{\mathtt{<:}} \mathtt{C}$, or removals of some synthetic casts. By Theorem 4.7.1.1, $|\Gamma|_{\Delta} \vdash_{\mathrm{FJ}} |\mathtt{e}_0|_{\Delta,\Gamma} \in |\mathtt{T}_0|_{\Delta}$. By Lemma 4.7.1.8, $\mathit{fieldsmax}(|\mathtt{T}_0|_{\Delta}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ and $|\overline{\mathtt{T}}|_{\Delta} \mathrel{\mathtt{<:}_{\mathrm{FJ}}} \overline{\mathtt{C}}$.
Now we have two subcases.
Subcase: $\quad |\mathtt{T}_i|_{\Delta} \neq \mathtt{C}_i$
By the rule E-FIELD-CAST,

$$|\mathtt{e}|_{\Delta,\Gamma} = \mathtt{(}|\mathtt{T}_i|_{\Delta}\mathtt{)}^{s}|\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.f}_i.$$

Now we must show that $|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'} = \mathtt{(D)}^{s}|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'}\mathtt{.f}_i$ for some $\mathtt{D} \mathrel{\mathtt{<:}_{\mathrm{FJ}}} |\mathtt{T}|_{\Delta}$. By Lemmas 4.5.1.12 and 4.5.1.13,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;\Gamma' \vdash_{\mathrm{FGJ}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0 \in \mathtt{S}_0$$
$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S}_0 \mathrel{\mathtt{<:}_{\mathrm{FGJ}}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0.$$

By Lemmas 4.5.1.9 and 4.5.1.10 and

$$\textit{fields}_{\text{FGJ}}(\textit{bound}_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0)) = [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}}, \overline{\mathtt{T}'}\ \overline{\mathtt{g}}$$

Then, by Lemma 4.7.1.7,

$$|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_i|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2} \mathrel{<:_{\text{FJ}}} |\mathtt{T}_i|_{\Delta}.$$

On the other hand,

$$\textit{fieldsmax}(|\mathtt{S}_0|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}, \overline{\mathtt{D}}\ \overline{\mathtt{g}}.$$

Therefore, by the rule E-FIELD-CAST,

$$|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'} = \mathtt{(}|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_i|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}\mathtt{)}^s|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'}\mathtt{.f}_i.$$

finishing the subcase.

Subcase: $|\mathtt{T}_i|_\Delta = \mathtt{C}_i$

Similar to the above subcase.

Case GT-METHOD: $\mathtt{e} = \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{d}}\mathtt{)}$ $\quad \Delta;\Gamma \vdash_{\text{FGJ}} \mathtt{e}_0 \in \mathtt{T}_0$
$\textit{mtype}_{\text{FGJ}}(\mathtt{m}, \textit{bound}_\Delta(\mathtt{T}_0)) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U}_0$
$\Delta \vdash \overline{\mathtt{V}}$ ok $\quad \Delta \vdash \overline{\mathtt{V}} \mathrel{<:_{\text{FGJ}}} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$
$\Delta;\Gamma \vdash_{\text{FGJ}} \overline{\mathtt{d}} \in \overline{\mathtt{S}} \quad \Delta \vdash \overline{\mathtt{S}} \mathrel{<:_{\text{FGJ}}} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$
$\mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0$

By the induction hypothesis, $|\overline{\mathtt{d}}|_{\Delta,\Gamma}$ are obtained from $|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{d}}|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'}$ by some combination of replacements of some synthetic casts $\mathtt{(D)}^s$ with $\mathtt{(C)}^s$ where $\mathtt{D} <: \mathtt{C}$, or removals of some synthetic casts. By Theorem 4.7.1.1, $|\Gamma|_\Delta \vdash_{\text{FJ}} |\mathtt{e}_0|_{\Delta,\Gamma} \in |\mathtt{T}_0|_\Delta$. By Lemma 4.7.1.9, $\textit{mtypemax}(\mathtt{m}, |\mathtt{T}_0|_\Delta) = \overline{\mathtt{E}}\to\mathtt{E}_0$ and $|\mathtt{T}|_\Delta \mathrel{<:_{\text{FJ}}} \mathtt{E}_0$.

Now we have two subcases:

Subcase: $|\mathtt{T}|_\Delta \neq \mathtt{E}_0$

By the rule E-INVK-CAST,

$$|\mathtt{e}|_{\Delta,\Gamma} = \mathtt{(}|\mathtt{T}|_\Delta\mathtt{)}^s|\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.m(}|\overline{\mathtt{d}}|_{\Delta,\Gamma}\mathtt{)}.$$

Now, we must show that

$$|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'} = \mathtt{(D)}^s|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'}\mathtt{.m(}|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{d}}|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'}\mathtt{)}$$

for some $\mathtt{D} \mathrel{<:_{\text{FJ}}} |\mathtt{T}|_\Delta$. By Lemmas 4.5.1.12 and 4.5.1.13,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; \Gamma' \vdash_{\text{FGJ}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0 \in \mathtt{S}_0$$
$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S}_0 \mathrel{<:_{\text{FGJ}}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0.$$

Without loss of generality, we can assume $\overline{\mathtt{X}}$ and $\overline{\mathtt{Y}}$ are distinct. By Lemmas 4.5.1.9 and 4.5.1.11, we have

$$\textit{mtype}_{\text{FGJ}}(\mathtt{m}, \textit{bound}_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0)) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}\mathtt{>}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}\to\mathtt{U}_0{}'$$
$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2, \overline{\mathtt{Y}} \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{P}} \vdash \mathtt{U}_0{}' \mathrel{<:_{\text{FGJ}}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{U}_0.$$

By Lemma 4.5.1.7,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}} \mathrel{<:_{\text{FGJ}}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \quad (= [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}]([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}))$$

and by the same lemma,

$$\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0{}' <:_{\text{FGJ}} [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{U}_0 \quad (= [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 = [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}).$$

Then, by Lemmas 4.7.1.6 and 4.7.1.7,

$$|[[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0{}'|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2} <:_{\text{FJ}} |[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2} <:_{\text{FJ}} |\mathtt{T}|_{\Delta}.$$

On the other hand, it is easy to show that

$$\mathit{mtypemax}(\mathtt{m},\ |\mathtt{S}_0|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}) = \mathit{mtypemax}(\mathtt{m},\ |[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}) = \overline{\mathtt{E}}\rightarrow\mathtt{E}_0.$$

Then, by the rule E-Invk-Cast,

$$|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'} = (|[[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0{}'|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2})^s|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'}.\mathtt{m}(|[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{d}}|_{\Delta_1,\,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2,\,\Gamma'})$$

finishing the subcase.

Subcase: $|\mathtt{T}|_{\Delta,\Gamma} = \mathtt{E}_0$

Similar to the subcase above.

Case GT-New, GT-UCast, GT-DCast, GT-SCast:

Easy. ■

4.7.2.7 Lemma: Suppose

1. $\mathit{mbody}_{\text{FGJ}}(\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>},\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = (\overline{\mathtt{x}},\ \mathtt{e})$,
2. $\mathit{mtype}_{\text{FGJ}}(\mathtt{m},\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$,
3. $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ ok,
4. $\Delta \vdash \overline{\mathtt{V}} <:_{\text{FGJ}} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$, and
5. $\mathit{mbody}_{\text{FJ}}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e}')$.

Then, $|\overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}|_{\Delta} \vdash |\mathtt{e}|_{\Delta,\ \overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\mathtt{this}:\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}} \overset{\text{exp}}{\Longrightarrow} \mathtt{e}'$.

Proof: By induction on the derivation of $\mathit{mbody}_{\text{FGJ}}(\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>},\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})$ with a case analysis on the last rule used.

Case MB-Class: $\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>} \triangleleft \mathtt{N}\ \{\ \ldots$
$\qquad\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{Q}}\texttt{>}\ \mathtt{S}_0\ \mathtt{m}\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e}_0;\}$
$[\overline{\mathtt{T}}/\overline{\mathtt{X}},\ \overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{e}_0 = \mathtt{e}$
$[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}} = \overline{\mathtt{P}}$
$[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}} = \overline{\mathtt{U}}$
$[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0 = \mathtt{U}_0$

Let $\Delta' = \overline{\mathtt{X}}<:\overline{\mathtt{N}},\ \overline{\mathtt{Y}}<:\overline{\mathtt{P}}$ and $\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{S}},\ \mathtt{this} : \mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}$. By Lemma 4.7.1.9, $\mathit{mtypemax}(\mathtt{m},\ \mathtt{C}) = \overline{\mathtt{D}}\rightarrow\mathtt{D}$ and $|[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}|_{\Delta} <:_{\text{FJ}} \overline{\mathtt{D}}$. By WF-Class, $\Delta \vdash \overline{\mathtt{T}} <:_{\text{FGJ}} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}\ \ (= [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}})$. By Lemma 4.7.2.6, $|\mathtt{e}_0|_{\Delta',\ \overline{\mathtt{x}}:\overline{\mathtt{S}},\mathtt{this}:\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}}$ is obtained from $|\mathtt{e}|_{\Delta,\ \overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\mathtt{this}:\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}}$ by some combination of replacements of some synthetic casts $(\mathtt{D})^s$ with $(\mathtt{C})^s$ where $\mathtt{D} <: \mathtt{C}$, or removals of some synthetic casts. By Theorem 4.7.1.1, $|\overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{this} : \mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}|_{\Delta'} \vdash_{\text{FJ}} |\mathtt{e}_0|_{\Delta',\ \overline{\mathtt{x}}:\overline{\mathtt{S}},\mathtt{this}:\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}} \in |\mathtt{S}_0|_{\Delta'}$ Now, let

$$\mathtt{e}_i = \begin{cases} \mathtt{x}_i & \text{if } \mathtt{D}_i = |\mathtt{S}_i|_{\Delta'} \\ (|\mathtt{S}_i|_{\Delta'})^s\mathtt{x}_i & \text{otherwise} \end{cases}$$

for $i = 1, \ldots, \#(\overline{\mathtt{x}})$. Since $\mathtt{e}' = [\overline{\mathtt{e}}/\overline{\mathtt{x}}]|\mathtt{e}_0|_{\Delta',\Gamma}$ and $|[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}|_\Delta \mathrel{\mathtt{<:}_{\mathrm{FJ}}} |\overline{\mathtt{S}}|_{\Delta'}$ by Lemma 4.7.1.7, each $\mathtt{e}_i$ is either a variable or a variable with an upcast under the environment $|\overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta$.

$$|\overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta \vdash_{\mathrm{FJ}} \mathtt{e}' \in \mathtt{D}$$

for some $\mathtt{D}$ such that $\mathtt{D} \mathrel{\mathtt{<:}_{\mathrm{FJ}}} |\mathtt{S}_0|_{\Delta'}$ by Lemma 2.5.4. Therefore, we have

$$|\overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta \vdash |\mathtt{e}_0|_{\Delta,\,\Delta',\,\overline{\mathtt{x}}:\overline{\mathtt{S}},\mathtt{this}:\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}} \overset{\mathrm{exp}}{\Longrightarrow} \mathtt{e}'$$

finishing the case.

Case MB-Super: $\quad CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{\ \ldots\}}$
$\mathtt{m}$ is not defined in $CT(\mathtt{C})$.

By the induction hypothesis,

$$|\overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}|_\Delta \vdash |\mathtt{e}|_{\Delta,\,\overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\mathtt{this}:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}} \overset{\mathrm{exp}}{\Longrightarrow} \mathtt{e}'.$$

Then, by Lemma 4.7.2.6,

$$|\overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta \vdash |\mathtt{e}|_{\Delta,\,\overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\mathtt{this}:\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}} \overset{\mathrm{exp}}{\Longrightarrow} |\mathtt{e}|_{\Delta,\,\overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\mathtt{this}:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}}$$

and, by Lemma 2.5.4,

$$|\overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta \vdash_{\mathrm{FJ}} \mathtt{e}' \in E$$

for some $\mathtt{E}$. Therefore,

$$|\overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta \vdash |\mathtt{e}|_{\Delta,\,\overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\mathtt{this}:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}} \overset{\mathrm{exp}}{\Longrightarrow} \mathtt{e}'$$

finishing the case. ∎

4.7.2.8 Lemma: If $\Delta; \Gamma \vdash_{\mathrm{FGJ}} \mathtt{e} \in \mathtt{T}$ and $\mathtt{e} \longrightarrow_{\mathrm{FGJ}} \mathtt{e}'$, then there exists some FJ expression $\mathtt{d}'$ such that $|\Gamma|_\Delta \vdash_{\mathrm{FJ}} |\mathtt{e}'|_{\Delta,\Gamma} \overset{\mathrm{exp}}{\Longrightarrow} \mathtt{d}'$ and $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\mathrm{FJ}} \mathtt{d}'$. In other words, the following diagram commutes.

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

Proof: By induction on the derivation of $\mathtt{e} \longrightarrow_{\mathrm{FGJ}} \mathtt{e}'$ with a case analysis on the last reduction rule used. We show the main base cases.

Case GR-Field: $\quad \mathtt{e} = \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).f}_i \quad \mathtt{e}' = \mathtt{e}_i \quad \mathit{fields}_{\mathrm{FGJ}}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$

We have two subcases depending on the last erasure rule used.

Subcase E-Field-Cast: $\quad |\mathtt{e}|_{\Delta,\Gamma} = \mathtt{(D)}^s\mathtt{(new\ C(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{).f}_i\mathtt{)}$

We have $|\mathtt{N}|_\Delta = \mathtt{C}$ by definition of erasure. Since $\mathit{fields}_{\mathrm{FJ}}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ for some $\overline{\mathtt{C}}$, we have $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\mathrm{FJ}} \mathtt{(D)}^s|\mathtt{e}_i|_{\Delta,\Gamma}$. On the other hand, by Theorem 4.5.1.1, $\Delta; \Gamma \vdash_{\mathrm{FGJ}} \mathtt{e}_i \in \mathtt{T}_i$ such that $\Delta \vdash \mathtt{T}_i \mathrel{\mathtt{<:}_{\mathrm{FGJ}}} \mathtt{T}$. By Theorem 4.7.1.1, $|\mathtt{T}|_\Delta = \mathtt{D}$ and $|\Gamma|_\Delta \vdash_{\mathrm{FJ}} |\mathtt{e}_i|_{\Delta,\Gamma} \in |\mathtt{T}_i|_\Delta$. Since $|\mathtt{T}_i|_\Delta \mathrel{\mathtt{<:}_{\mathrm{FJ}}} \mathtt{D}$ by Lemma 4.7.1.6, $\mathtt{(D)}^s|\mathtt{e}_i|_{\Delta,\Gamma}$ is obtained by adding an upcast to $|\mathtt{e}_i|_{\Delta,\Gamma}$.

Subcase E-Field: $|\texttt{e}|_{\Delta,\Gamma} = \texttt{new C(}|\overline{\texttt{e}}|_{\Delta,\Gamma}\texttt{).f}_i$

Follows from the induction hypothesis.

Case GR-Invk: $\texttt{e} = \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)}$ $\quad \texttt{e}' = [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new N(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}_0$
$\mathit{mbody}_{\mathrm{FGJ}}(\texttt{m<}\overline{\texttt{V}}\texttt{>, N}) = (\overline{\texttt{x}}, \texttt{e}_0)$

We have two subcases depending on the last erasure rule used.

Subcase E-Invk-Cast: $|\texttt{e}|_{\Delta,\Gamma} = \texttt{(D)}^s\texttt{(new C(}|\overline{\texttt{e}}|_{\Delta,\Gamma}\texttt{).m(}|\overline{\texttt{d}}|_{\Delta,\Gamma}\texttt{))}$

By Theorem 4.7.1.1, we have $|\texttt{T}|_\Delta = \texttt{D}$. Since

$$|\texttt{e}'|_{\Delta,\Gamma} = [|\overline{\texttt{d}}|_{\Delta,\Gamma}/\overline{\texttt{x}}, |\texttt{new N(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma}/\texttt{this}]|\texttt{e}_0|_{\Delta,\Gamma'}$$

where $\Gamma' = \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{N}$ and $\overline{\texttt{T}}$ are types of $\overline{\texttt{d}}$, we have, by Theorems 4.5.1.1 and 4.7.1.1,

$$|\Gamma|_\Delta \vdash_{\mathrm{FJ}} [|\overline{\texttt{d}}|_{\Delta,\Gamma}/\overline{\texttt{x}}, |\texttt{new N(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma}/\texttt{this}]|\texttt{e}_0|_{\Delta,\Gamma'} \in |\texttt{T}'|_\Delta$$

for some $\texttt{T}'$ such that $\Delta \vdash \texttt{T}' \texttt{<:}_{\mathrm{FGJ}} \texttt{T}$. By Lemma 4.7.1.6, $|\texttt{T}'|_\Delta \texttt{<:}_{\mathrm{FJ}} \texttt{D}$. Thus,

$$|\Gamma|_\Delta \vdash [|\overline{\texttt{d}}|_{\Delta,\Gamma}/\overline{\texttt{x}}, |\texttt{new N(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma}/\texttt{this}]|\texttt{e}_0|_{\Delta,\Gamma'} \overset{\text{exp}}{\Longrightarrow} \texttt{(D)}^s[|\overline{\texttt{d}}|_{\Delta,\Gamma}/\overline{\texttt{x}}, |\texttt{new N(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma}/\texttt{this}]|\texttt{e}_0|_{\Delta,\Gamma'}.$$

Now, because $\mathit{mbody}_{\mathrm{FGJ}}(\texttt{m, C})$ is well defined, $\mathit{mbody}_{\mathrm{FJ}}(\texttt{m, C})$ is well defined. Suppose it is equal to $(\overline{\texttt{x}}, \texttt{e}')$. By Lemma 4.7.2.7,

$$|\Gamma'|_\Delta \vdash |\texttt{e}_0|_{\Delta,\Gamma'} \overset{\text{exp}}{\Longrightarrow} \texttt{e}'.$$

Therefore,

$$|\Gamma|_\Delta \vdash [|\overline{\texttt{d}}|_{\Delta,\Gamma}/\overline{\texttt{x}}, |\texttt{new N(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma}/\texttt{this}]|\texttt{e}_0|_{\Delta,\Gamma'} \overset{\text{exp}}{\Longrightarrow} \texttt{(D)}^s[|\overline{\texttt{d}}|_{\Delta,\Gamma}/\overline{\texttt{x}}, |\texttt{new N(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma}/\texttt{this}]\texttt{e}',$$

finishing the subcase. Note that $\texttt{new C(}|\overline{\texttt{e}}|_{\Delta,\Gamma}\texttt{).m(}|\overline{\texttt{d}}|_{\Delta,\Gamma}\texttt{)} \longrightarrow_{\mathrm{FJ}} [|\overline{\texttt{d}}|_{\Delta,\Gamma}/\overline{\texttt{x}}, |\texttt{new N(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma}/\texttt{this}]\texttt{e}'$.

Subcase E-Invk:

Similarly to the subcase above.

Case GR-Cast:

Easy. ∎

4.7.2.9 Lemma: If $\Gamma \vdash_{\mathrm{FJ}} \texttt{e} \in \texttt{C}$ and $\texttt{e} \longrightarrow_{\mathrm{FJ}} \texttt{e}'$ and $\Gamma \vdash \texttt{e} \overset{\text{exp}}{\Longrightarrow} \texttt{d}$, then there exists some FJ expression $\texttt{d}'$ such that $\Gamma \vdash \texttt{e}' \overset{\text{exp}}{\Longrightarrow} \texttt{d}'$ and $\texttt{d} \longrightarrow_{\mathrm{FJ}}{}^* \texttt{d}'$. In other words, the following diagram commutes.

$$\begin{tikzcd}
\texttt{e} \arrow[r, "\text{reduce (FJ)}"] \arrow[d, Rightarrow] & \texttt{e}' \arrow[d, Rightarrow, dotted] \\
\texttt{d} \arrow[r, dotted, "*", "\text{reduce (FJ)}"'] & \texttt{d}'
\end{tikzcd}$$

Proof: By induction on the derivation of $\texttt{e} \longrightarrow_{\mathrm{FJ}} \texttt{e}'$ with a case analysis on the last reduction rule used.

Case R-Field: $\texttt{e} = \texttt{new C(}\overline{\texttt{e}}\texttt{).f}_i$ $\quad \mathit{fields}_{\mathrm{FJ}}(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{f}}$ $\quad \texttt{e}' = \texttt{e}_i$

The expansion $\texttt{d}$ must have a form of $\texttt{(D}_1\texttt{)}^s \cdots \texttt{(D}_n\texttt{)}^s\texttt{(new C(}\overline{\texttt{d}}\texttt{).f}_i\texttt{)}$ where $\Gamma \vdash \overline{\texttt{e}} \overset{\text{exp}}{\Longrightarrow} \overline{\texttt{d}}$ and $\texttt{C} \texttt{<:}_{\mathrm{FJ}} \texttt{D}_i$ for $1 \leq i \leq n$ because each $\texttt{D}_i$ is introduced as an upcast. Thus, $\texttt{d} \longrightarrow_{\mathrm{FJ}}{}^* \texttt{new C(}\overline{\texttt{d}}\texttt{).f}_i \longrightarrow_{\mathrm{FJ}} \texttt{d}_i$.

The other base cases are similar and the cases for induction steps are straightforward. ∎

Proof of Theorem 4.7.2.2: By induction on the length n of reduction sequence $\texttt{e} \longrightarrow_{\mathrm{FGJ}}{}^* \texttt{e}'$.

Case: $n = 0$

Trivial since $\mathtt{e} = \mathtt{e}'$.

Case: $\mathtt{e} \longrightarrow_{\text{FGJ}} \mathtt{e}' \longrightarrow_{\text{FGJ}}{}^{*}\ \mathtt{e}''$

We have the following commuting diagram.

$$\begin{tikzcd}
\mathtt{e} \arrow[rr, "\text{reduce (FGJ)}"] \arrow[dddd, "\text{erase}"'] & & \mathtt{e}' \arrow[rr, "\text{reduce (FGJ)}^{*}"] \arrow[dd, dotted, "\text{erase}"'] & & \mathtt{e}'' \arrow[d, dotted, "\text{erase}"] \\
 & & & (2) & {|\mathtt{e}''|} \arrow[d, dotted] \\
 & (1) & {|\mathtt{e}'|} \arrow[rr, dotted, "\text{reduce (FJ)}^{*}"] \arrow[dd, dotted] & & \mathtt{d}' \arrow[dd, dotted] \\
 & & & (3) & \\
{|\mathtt{e}|} \arrow[rr, dotted, "\text{reduce (FJ)}"'] & & \mathtt{d} \arrow[rr, dotted, "\text{reduce (FJ)}^{*}"'] & & \mathtt{d}''
\end{tikzcd}$$

Commutation (1) is proved by Lemma 4.7.2.8, (2) by the induction hypothesis and (3) by Lemma 4.7.2.9.
■

4.7.2.10 Lemma: Suppose $\Delta;\Gamma \vdash_{\text{FGJ}} \mathtt{e} \in \mathtt{T}$. If $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\text{FJ}} \mathtt{d}$, then $\mathtt{e} \longrightarrow_{\text{FGJ}} \mathtt{e}'$ for some $\mathtt{e}'$ and $|\Gamma|_{\Delta} \vdash |\mathtt{e}'|_{\Delta,\Gamma} \overset{\text{exp}}{\Longrightarrow} \mathtt{d}$. In other words, the following diagram commutes:

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

Proof: By induction on the derivation of $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\text{FJ}} \mathtt{d}$ with a case analysis by the last rule used.

Case RC-CAST:

We have two subcases according to whether the cast is synthetic ($|\mathtt{e}|_{\Delta,\Gamma} = \mathtt{(C)}^{s}\mathtt{e}_0$) or not ($|\mathtt{e}|_{\Delta,\Gamma} = \mathtt{(C)}\,\mathtt{e}_0$). The latter case follows from the induction hypothesis. We show the former case where

$$\begin{array}{l}
|\mathtt{e}|_{\Delta,\Gamma} = \mathtt{(C)}^{s}\mathtt{e}_0 \\
\mathtt{e}_0 \longrightarrow_{\text{FJ}} \mathtt{d}_0 \\
\mathtt{d} = \mathtt{(C)}^{s}\mathtt{d}_0
\end{array}$$

Then $\mathtt{e}_0$ must be either a field access or a method invocation. We have another case analysis with the last reduction rule for the derivation of $\mathtt{e}_0 \longrightarrow_{\text{FJ}} \mathtt{d}_0$. The cases for RC-FIELD, RC-INVK-RECV and RC-INVK-ARG are omitted since they follow from the induction hypothesis.

Subcase R-FIELD: $\mathtt{e}_0 = \mathtt{new\ D(\overline{e}).f}_i$
$\mathtt{d}_0 = \mathtt{e}_i$
$\mathit{fields}_{\mathrm{FJ}}(\mathtt{D}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$

By inspecting the derivation of $|\mathtt{e}|_{\Delta,\Gamma}$, it must be the case that

$$\begin{aligned}
&\mathtt{e} = \mathtt{new\ D<\overline{T}>(\overline{e}').f}_i\\
&|\overline{\mathtt{e}}'|_{\Delta,\Gamma} = \overline{\mathtt{e}}\\
&\mathit{fieldsmax}(\mathtt{D}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}\\
&|\mathtt{T}|_\Delta = \mathtt{C} \neq \mathtt{C}_i.
\end{aligned}$$

By Theorems 4.5.1.2 and 4.5.1.1, we have $\mathtt{e} \longrightarrow_{\mathrm{FGJ}} \mathtt{e}_i{}'$ and $\Delta;\Gamma \vdash_{\mathrm{FGJ}} \mathtt{e}_i{}' \in \mathtt{S}$ and $\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}}_{\mathrm{FGJ}} \mathtt{T}$. By Theorem 4.7.1.1, $|\Gamma|_\Delta \vdash_{\mathrm{FJ}} |\mathtt{e}_i{}'|_{\Delta,\Gamma} \in |\mathtt{S}|_\Delta$. By Lemma 4.7.1.6, $|\mathtt{S}|_\Delta \mathrel{\mathtt{<:}}_{\mathrm{FJ}} |\mathtt{T}|_\Delta$. Then, $|\Gamma|_\Delta \vdash \mathtt{e}_i \overset{\mathrm{exp}}{\Longrightarrow} (|\mathtt{T}|_\Delta)\mathtt{e}_i$, finishing the case.

Subcase R-INVK: $\mathtt{e}_0 = \mathtt{new\ D(\overline{d}).m(\overline{e})}$
$\mathtt{d}_0 = [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D(\overline{d})/this}]\mathtt{e}_m$
$\mathit{mbody}_{\mathrm{FJ}}(\mathtt{m}, \mathtt{D}) = (\overline{\mathtt{x}}, \mathtt{e}_m)$

By inspecting the derivation of $|\mathtt{e}|_{\Delta,\Gamma}$, it must be the case that

$$\begin{aligned}
&\mathtt{e} = \mathtt{new\ D<\overline{T}>(\overline{d}').m<\overline{V}>(\overline{e}')}\\
&|\overline{\mathtt{d}}'|_{\Delta,\Gamma} = \overline{\mathtt{d}}\\
&|\overline{\mathtt{e}}'|_{\Delta,\Gamma} = \overline{\mathtt{e}}\\
&\mathit{mtype}_{\mathrm{FGJ}}(\mathtt{m}, \mathtt{D<\overline{T}>}) = \mathtt{<\overline{Y} \triangleleft \overline{P}>\overline{U}{\rightarrow}U_0}\\
&[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 = \mathtt{T}\\
&\mathit{mtypemax}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{C}}{\rightarrow}\mathtt{C}_0\\
&|\mathtt{T}|_\Delta = \mathtt{C} \neq \mathtt{C}_0.
\end{aligned}$$

By Theorems 4.5.1.2 and 4.5.1.1, $\mathtt{e} \longrightarrow_{\mathrm{FJ}} [\overline{\mathtt{e}}'/\overline{\mathtt{x}}, \mathtt{new\ D<\overline{T}>(\overline{d}')/this}]\mathtt{e}_m{}'$ where $\mathit{mbody}_{\mathrm{FGJ}}(\mathtt{m<\overline{V}>}, \mathtt{D<\overline{T}>}) = (\overline{\mathtt{x}}, \mathtt{e}_m{}')$ and $\Delta;\Gamma \vdash_{\mathrm{FGJ}} [\overline{\mathtt{e}}'/\overline{\mathtt{x}}, \mathtt{new\ D<\overline{T}>(\overline{d}')/this}]\mathtt{e}_m{}' \in \mathtt{S}$ for some $\mathtt{S}$ such that $\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$. Then,

$$|[\overline{\mathtt{e}}'/\overline{\mathtt{x}}, \mathtt{new\ D<\overline{T}>(\overline{d}')/this}]\mathtt{e}_m{}'|_{\Delta,\Gamma} = [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D(\overline{d})/this}]|\mathtt{e}_m{}'|_{\Delta,\ \overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\ \mathtt{this:D<\overline{T}>}}$$

By Theorem 4.7.1.1,

$$|\Gamma|_\Delta \vdash_{\mathrm{FJ}} [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D(\overline{d})/this}]|\mathtt{e}_m{}'|_{\Delta,\ \overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\ \mathtt{this:D<\overline{T}>}} \in |\mathtt{S}|_\Delta.$$

Since we have $|\mathtt{S}|_\Delta \mathrel{\mathtt{<:}}_{\mathrm{FJ}} |\mathtt{T}|_\Delta$ by Lemma 4.7.1.6,

$$\begin{aligned}
|\Gamma|_\Delta \vdash {} & [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D(\overline{d})/this}]|\mathtt{e}_m{}'|_{\Delta,\ \overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\ \mathtt{this:D<\overline{T}>}}\\
& \overset{\mathrm{exp}}{\Longrightarrow} (|\mathtt{T}|_\Delta)[\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D(\overline{d})/this}]|\mathtt{e}_m{}'|_{\Delta,\ \overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\ \mathtt{this:D<\overline{T}>}}
\end{aligned}$$

Then, by Lemma 4.7.2.7,

$$|\Gamma|_\Delta \vdash |\mathtt{e}_m{}'|_{\Delta,\ \overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\ \mathtt{this:D<\overline{T}>}} \overset{\mathrm{exp}}{\Longrightarrow} \mathtt{e}_m.$$

and, finally,

$$|\Gamma|_\Delta \vdash |[\overline{\mathtt{e}}'/\overline{\mathtt{x}}, \mathtt{new\ D<\overline{T}>(\overline{d}')/this}]\mathtt{e}_m{}'|_{\Delta,\Gamma} \overset{\mathrm{exp}}{\Longrightarrow} (|\mathtt{T}|_\Delta)[\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D(\overline{d})/this}]\mathtt{e}_m$$

Case R-FIELD:
Similar to the subcase for R-FIELD above.

Case R-INVK:
Similar to the subcase for R-INVK above. Other cases are straightforward. ■

4.7.2.11 Lemma: Suppose $\Delta;\Gamma \vdash_{\text{FGJ}} \mathtt{e} \in \mathtt{T}$ and $|\Gamma|_\Delta \vdash |\mathtt{e}|_{\Delta,\Gamma} \overset{\text{exp}}{\Longrightarrow} \mathtt{d}$. If $\mathtt{d}$ reduces to $\mathtt{d}'$ with zero or more steps by reducing synthetic casts, followed by one step by other kinds of reduction, then $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\text{FJ}} \mathtt{e}'$ and $|\Gamma|_\Delta \vdash \mathtt{e}' \overset{\text{exp}}{\Longrightarrow} \mathtt{d}'$. In other words, the following diagram commutes:

$$\begin{tikzcd}
|\mathtt{e}| \arrow[rr, dotted, "\text{reduce (FJ)}"] \arrow[d, Rightarrow] & & \mathtt{e}' \arrow[d, dotted] \\
\mathtt{d} \arrow[r, "*", "\text{R-Cast}"'] & {} \arrow[r, "\text{reduce (FJ)}"'] & \mathtt{d}'
\end{tikzcd}$$

Proof: By induction on the derivation of the last reduction step with a case analysis by the last rule used.

Case R-FIELD: $\quad \mathtt{d} \longrightarrow_{\text{FJ}}{}^* \mathtt{new\ C(\overline{e}).f}_i \qquad \mathit{fields}_{\text{FJ}}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}} \qquad \mathtt{d}' = \mathtt{e}_i$

The expression $\mathtt{d}$ must be of the form $\mathtt{((D_1)}^s \ldots \mathtt{(D}_n\mathtt{)}^s\mathtt{new\ C(\overline{e}')).f}_i$ where $\mathtt{C} \mathrel{<:} \mathtt{D}_i$ for any i and each $\mathtt{e}_i{}'$ reduces to $\mathtt{e}_i$ by reducing upcasts (in several steps). In other words, $|\Gamma|_\Delta \vdash \overline{\mathtt{e}}' \overset{\text{exp}}{\Longrightarrow} \overline{\mathtt{e}}$. Moreover, since $|\Gamma|_\Delta \vdash |\mathtt{e}|_{\Delta,\Gamma} \overset{\text{exp}}{\Longrightarrow} \mathtt{d}$, $|\mathtt{e}|_{\Delta,\Gamma}$ must be of the form, either $\mathtt{new\ C(\overline{e}'').f}_i$ or $\mathtt{(D)}^s\mathtt{new\ C(\overline{e}'').f}_i$, where $|\Gamma|_\Delta \vdash \overline{\mathtt{e}}'' \overset{\text{exp}}{\Longrightarrow} \overline{\mathtt{e}}'$. Therefore, $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\text{FJ}} \mathtt{e}_i{}''$ or $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\text{FJ}} \mathtt{(D)}^s\mathtt{e}_i{}''$. It is easy to see

$$|\Gamma|_\Delta \vdash \mathtt{(D)}^s\mathtt{e}_i{}'' \overset{\text{exp}}{\Longrightarrow} \mathtt{e}_i$$

and

$$|\Gamma|_\Delta \vdash \mathtt{e}_i{}'' \overset{\text{exp}}{\Longrightarrow} \mathtt{e}_i.$$

Other base cases are similar; induction steps are straightforward. ■

Proof of Theorem 4.7.2.3: Follows from Lemmas 4.7.2.10 and 4.7.2.11. ■

Proof of Corollary 4.7.2.4: By Theorem 4.7.2.2, we have an FJ expression $\mathtt{d}$ such that $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{\text{FJ}}{}^*\mathtt{d}$ and $|\Gamma|_\Delta \vdash |\mathtt{v}|_{\Delta,\Gamma} \overset{\text{exp}}{\Longrightarrow} \mathtt{d}$. Since $|\mathtt{v}|_{\Delta,\Gamma}$ does not include any typecasts, $\mathtt{d}$ is obtained only by adding some (synthetic) upcasts. Therefore, $\mathtt{d}$ reduces to $|\mathtt{v}|_{\Delta,\Gamma}$.

The second part follows from a similar argument using Theorem 4.7.2.3. ■

Proof of Corollary 4.7.2.5: Similar to the proof of Corollary 4.7.2.4. ■

4.8 Summary

We have studied the core of GJ-style parametric classes with Featherweight GJ, an extension of FJ with parametric classes. We have defined the two styles of semantics of FGJ: the direct semantics, based on reduction of FGJ expressions, and the translation semantics, based on the erasure translation from FGJ to FJ. The type system is proved to be sound with respect to the reduction semantics. As for the translation semantics, the main results are preservation of typing and behavior of a program. The proof of preservation of program behavior is not trivial at all because a reduction step of execution of an FGJ program does not correspond to a sequence of reduction steps of execution of its erasure. We have proved slightly weaker theorems that guarantee the execution results correspond to each other. The result also guarantees that synthetic casts, inserted by the compiler, can never fail during the execution of the compiled program.

Chapter 5

Raw Types

Compatibility is one of the important issues in the design of language extensions; especially, when the base language is already wide spread, its importance will increase. If old programs cannot run on the new environment, users of the base language won't use the extension regardless of its usefulness.

The design of GJ is significantly affected by issues concerning compatibility with the current Java language environment. For example, the language is backward compatible with Java in the sense that every Java program is also a GJ program, and GJ programs are compiled to Java Virtual Machine Language so that they can run on the old Java environment.

Moreover, GJ's novel feature called *raw types* makes a shift from the Java environment to the GJ environment easier. Every parameterized class `C<`$\overline{\texttt{X}}$`>` provides the raw type `C`, which can be used as usual type. So, even when a monomorphic class `C` (in Java library or your program) is upgraded into a parameterized version `C<`$\overline{\texttt{X}}$`>`, the code that uses it by using `C` may remain the same; it accesses the parameterized version through the raw type `C` instead of `C<`$\overline{\texttt{T}}$`>`. Figure 5.1 summarizes the process, which we call *program evolution*: beginning with a Java program, which is also a GJ program, we replace each class by a polymorphic version one by one to make the whole program polymorphic; during this process, old classes refer to new classes through raw types but it remains accepted by the compiler and still runs. Thus, programmers will not be bothered too much by incompatibility with new polymorphic classes and legacy monomorphic classes; they may add type parameters to the Java code at any time.

A raw type roughly corresponds to the set of all objects instantiated from one parameterized class (possibly with different type parameters). In other words, a type with type parameters, which we call *cooked type*, is subtype of the corresponding raw type—that is, `Pair<X,Y>` <: `Pair` for any `X` and `Y`. As for field and method types, raw types provide erased types: for example, the type of the `fst` and `snd` fields of the raw type `Pair` is `Object`, which is the erasure of `X` and `Y`.

Not only are raw types useful to support smooth program evolution but also they lighten the restriction on downcasts. As we discussed in the previous chapter, downcasts are prohibited when the run-time check will require type parameter information of objects. For example, you cannot write an `equal` method like this:

```
class Pair<X extends Object, Y extends Object> extends Object {
  ...
  boolean equal(Object p) {
    return this.fst.equal.(((Pair<X,Y>)p).fst) &&
           this.snd.equal.(((Pair<X,Y>)p).snd);
  }
```

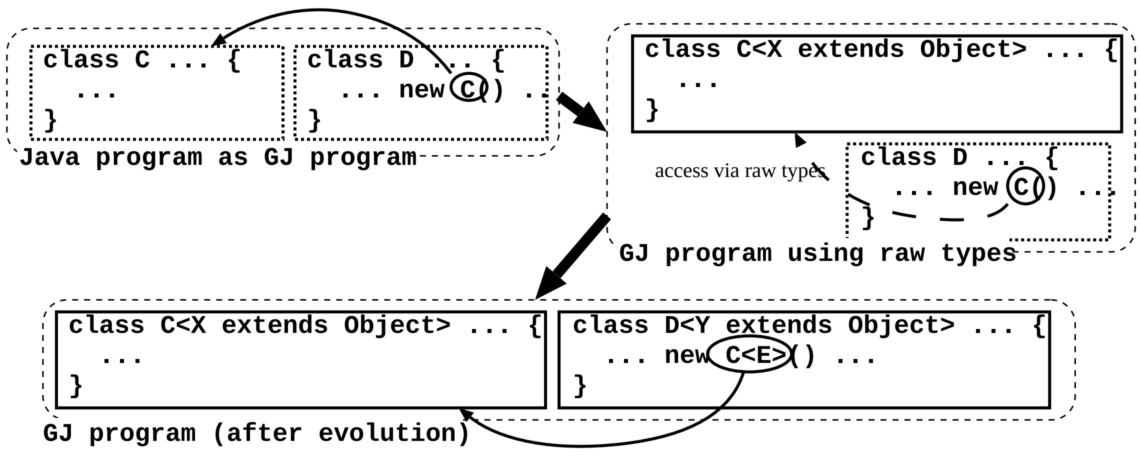
class C ... {
...
}
class D ... {
... new C() ...
}
Java program as GJ program
class C<X extends Object> ... {
...
}
access via raw types
class D ... {
... new C() ...
}
GJ program using raw types
class C<X extends Object> ... {
...
}
class D<Y extends Object> ... {
... new C<E>() ...
}
GJ program (after evolution)

Figure 5.1: Program evolution

```
}
```

(where `&&` is a binary operator for conjunction of two booleans). Here, the downcast `(Pair<X,Y>)p` is prohibited because the check requires the actual types bound to `X` and `Y`. Raw types solve this problem: instead of `(Pair<X,Y>)p`, you are allowed to write `(Pair)p`. Since it just checks whether `p`'s run-time type is `Pair<X,Y>` for some `X` and `Y`[1], it does not require run-time type parameter information and can be executed by using erasure semantics.

Although raw types provide a nice solution for the problems about compatibility and restriction on downcasts, abuse of them leads to an unsound type system, as we see in detail. In fact, GJ programs are characterized into three classes: ill-typed programs, which will be rejected by the compiler; *checked* programs, which correspond to well-typed programs in the standard sense and are guaranteed to run safely; and *unchecked* programs, which are accepted by the compiler but may cause run-time errors. Even though they are unsafe, unchecked programs help program evolution since many programs during evolution are exected to be unchecked. We hope that, since rules for checked programs are conservative, many unchecked programs can run without run-time errors.

In this chapter, we augment FGJ with raw types and obtain Raw FGJ. The main theoretical results are formalization of Raw FGJ, and proof of type soundness for checked programs with respect to reduction semantics. Typing rules for checked programs are very subtle, and, in fact, the current GJ compiler (and specification) overlooks one source of unsoundness, making some compiled GJ programs fail on a synthetic cast. One of the main contributions in this chapter is to propose a fixed type system and to prove its type soundness. We also discuss desired properties of unchecked programs with respect to program evolution. Unfortunately, we find that the current GJ design does not satisfy some of the important properties, either; a possible fix for this problem is proposed.

The rest of this chapter is organized as follows. Section 5.1 gives more details about raw types and discusses the border between checked programs and unchecked programs, in detail. The following Sections 5.2, 5.3, and 5.4 define Raw FGJ, with its syntax, type system, and reduction

[1]In this sense, raw types resembles to existential types [MP88].

semantics, respectively. We prove type soundness in Section 5.5. Section 5.6 discusses desired properties of unchecked programs. Finally, Section 5.7 summarizes this chapter.

5.1 Overview of Raw FGJ

5.1.1 Basic of Raw Types

As mentioned above, each class `C` provides the raw type `C`, which is supertype of any parametric type `C<`$\overline{\texttt{T}}$`>`. Basically, the fields and methods of `C` are given the same types as those of the erasure of `C<`$\overline{\texttt{X}}$`>`. Consider, for example, the following Raw FGJ classes `Pair` and `PairClient`.

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

class PairClient extends Object {
  PairClient() { super(); }
  A m(Pair p, A a) {
    return (A)p.setfst(a).fst;
  }
}
```

The method `m` takes an argument `p` of the raw type `Pair` and invokes the method `setfst` on `p`; the argument type will be an `Object`, the erasure of `X`, and the return type will be `Pair`, the erasure of `Pair<Z,Y>`. Then, the `fst` field, whose type is `Object`, is accessed; finally the downcast `(A)` is required for the return type of `m` to be `A`, although we know the content of the `fst` field is `a` of the type `A`.

Now, since a class name without type parameters can be used as a type, it poses a question about the previous assumption that `C` used without angle brackets is an abbreviation of `C<>`: is `C` still an abbreviation of `C<>`, or the raw type for the class `C`? In GJ, `C` is an abbreviation of `C<>` when the class `C` does not take type parameters (methods in `C` may take type parameters). In other words, there is no raw type for the class without type parameters since every occurrence of `C` is interpreted as the cooked type `C<>`. However, it will turn out that this rule create a problem about the evolution property, as we discuss later in Section 5.6. Therefore, in Raw FGJ, the use of `C` and `C<>` as types are strictly distinguished. (The *declaration* of a class without type parameters `class C<> extends ...` is abbreviated to `class C extends ...`. Also, we still omit `<>` before method declarations and use an abbreviation `e.m(`$\overline{\texttt{e}}$`)` for a method invocation `e.m<>(`$\overline{\texttt{e}}$`)` without type parameters.)

5.1.2 Checked and Unchecked Programs

In Raw FGJ, as well as GJ, well-typed programs are further classified into two categories, checked and unchecked. The typechecker issues *unchecked warnings* for unchecked programs. Checked programs are guaranteed to be safe in the sense that the subject reduction property holds, while unchecked programs are not. Even so, unchecked programs are still important to make program evolution easier. In fact, many unchecked programs can be executed without run-time errors since unchecked warnings are conservative. In this subsection, we show a few case studies of unchecked programs and explain why they can be unsafe.

First, in Raw FGJ (and GJ), a raw type `C` can be subtype of `C<`$\overline{\texttt{T}}$`>` with unchecked warnings. However, this subtyping apparently leads to unsoundness because it essentially allows two cooked types with different type parameters to be compatible. Consider the following classes:

```
class A extends Object {
  A() { super(); }
  A<> m() { return this; }
  A<> n(Pair<A<>,A<>> p) { return p.fst.m(); }
}

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

and the expression

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

The expression is well typed since `Pair<B,B>` is (safe) subtype of `Pair` and `Pair` is (unsafe) subtype of `Pair<A,A>`.[2] However, it eventually invokes the method `m` on a `B` object, which does not have such a method.

Second, method invocation on an expression of raw type is unchecked when the argument types involve any type parameters—that is, when they are different from their erasures. Suppose `a` is given raw type `A`. Then, the expression

```
a.n(new Pair<B<>,B<>>(new B<>(), new B<>()))
```

is well typed but unchecked since the argument type is changed by erasure from `Pair<A<>,A<>>` to `Pair`. Actually, following the same story as before, it will cause a run-time error. Consider another example:

```
class Cell<X extends Object> extends Object {
  Cell() { super(); }
  X id(X x) { return x; }
}

class CellOfA extends Cell<A<>> {
  CellOfA() { super(); }
  A<> id (A<> x) { return x.m(); }
}
```

[2]The unchecked subtyping relation may not be combined with checked subtyping: in Raw FGJ and GJ, it is not the case that `Pair<B,B>` is subtype of `Pair<A,A>` whether it is checked or not, even though `Pair<B,B>` is subtype of `Pair` and `Pair` is unchecked subtype of `Pair<A,A>`. Thus, the upcast `(Pair)` is required here.

and an expression `c.id(o)` where `c` and `o` are given type `Cell` and `Object`, respectively. Since, erasing the argument type `X` generates another type `Object`, the method invocation `c.id(o)` will be unchecked; in fact, if `c` is bound to a `CellOfA` object and `o` is an object of type other than `A`, then it will cause a run-time error when the overriding method is invoked.

Finally, an instantiation of an object using a raw type should be unchecked. Consider the following classes:

```
class Cell<X extends Object> {
  X f;
  Cell(X f) { this.f = f; }
}

class E<X extends Cell<A>> {
  X g;
  E(X g) { this.g = g; }
  A loophole() { return this.g.f.m(); }
}
```

In the method `loophole`, `this` is given type `E<X>` under the constraint `X <: Cell<A>`, and so `this.g.f` will be given type `A` and so `this.g.f.m()` is well typed. On the other hand, the type of the field `g` accessed through the raw type `E` is given type `Cell`, the erasure of `X`. Therefore, the constructor invocation `new E(c)` seems safe if `c` has type `Cell`. However, consider the expression `new E(new Cell<B<>>(new B<>())).loophole()`. Although it is a well-typed expression, it eventually tries to invoke the method `m` on `new B()`, causing a run-time error. Similarly, if a class extends the raw type `E` (not `E<T>`), invocation of `loophole` on the object of the subclass causes a run-time error.

An problem here is similar to the case for unchecked method invocations above: when `loophole` is invoked, `this` of type `E<X>` is replaced with a receiver object of raw type `E`, which is not a safe subtype of `E<X>`. There are at least two choices where unchecked warning should be signaled. One is where such a method is invoked and the other is where an object of raw type is instantiated. However, the former seems too restrictive because it implies that every method invocation on an expression of raw type should signal an unchecked warning. Therefore, we adopted the latter rule here. This rule is also preferable from the point of view of program evolution since this rule is helpful to detect the problem where one forget to attach type parameters, creating a raw type by mistake.

Actually, this source of unsoundness is overlooked by GJ designers and the current GJ compiler does not signal an unchecked warning for instantiation of raw types. As a result, a program compiled without unchecked warnings, such as the above one, fails on synthetic casts at run-time.

5.1.3 Raw Method Invocation and the Bottom Type

Introduction of raw types makes semantics of method invocations much more complicated. It is not straightforward at all to define type-preserving reduction since the number of actual type arguments may not be equal to the one expected by the method. Such a situation occurs when the expression on which a method is invoked is given a raw type. Statically, no type arguments are required for a method invocation on an expression of raw types since the argument types and return type are erased. However, at run-time, the actual method receiver may be an object of cooked type and the method may require type parameters! For example, suppose a variable `p` is of raw type `Pair`; then, the method `setfst` takes an argument of `Object`, which is erasure of `Z`, and the method invocation `p.setfst(x)` without type parameters is well typed. But, at run-time, a cooked object such as `new Pair<A,B>(a,b)` may be bound to `p`; now the invoked method requires a type parameter bound to `Z`. In general, when type arguments are missing from the expression, we have to synthesize type arguments that satisfy constraints on the type variables.

A key innovation to solve this problem is introduction of the bottom type `*`, which is subtype of every type. Intuitively, the bottom type represents the empty set; there is no *value* of the bottom type. In our reduction semantics, defined later, the bottom type `*` is bound to the formal type arguments when the actual type parameters are missing from the method invocation expression. In the example above, `*` is substituted for `Z` and reduces as follows:

$$\texttt{new Pair<A,B>(a,b).setfst(a}'\texttt{)}\longrightarrow\texttt{new Pair<*,B>(a}'\texttt{, new Pair<A,B>(a,b).snd)}.$$

Actually, the reduced expression may become ill typed when the method invocation is unchecked; indeed, expression `new Pair<*,B>(a′, new Pair<A,B>(a,b).snd)` is not well typed since the actual argument `a′` cannot given type `*`. (Notice that it is unchecked because erasure changes the original argument type `Z` to `Object`). On the other hand, if the method invocation is checked—that is, erasure does not change method parameter types—then the reduced expression remains well typed. We give a discussion why reduction preserves well-typedness, for a simple case. (A proof of subject reduction is given in Section 5.5.) Consider the following reduction steps:

$$\texttt{e}_0\texttt{.m(d)} \longrightarrow^* \texttt{new C<T>(}\overline{\texttt{e}}\texttt{).m(d)} \longrightarrow [\texttt{d/x, new C<T>(}\overline{\texttt{e}}\texttt{)/this}]\texttt{d}_0$$

where the definition of the class `C` is

```
class C<X extends Object> extends ... {
  ...
  <Y extends T'> U m (S x) { return d_0; }
}
```

and suppose $\texttt{e}_0$ is given raw type `C` and $\texttt{e}_0\texttt{.m(d)}$ is well typed without unchecked warnings. Since the original method invocation is checked, erasure of `S` is equal to `S`—that is, `S` does not involve any type parameters and should be another raw type `D`—and the result type is erasure of `U`. (Note that `U` may include type parameters.) Moreover, the actual argument `d` is given type `S′`, which is subtype of `S`. Now, the method body of `m` should be typed as

$$\texttt{X<:Object,Y<:T}';\texttt{S : x,this : C<X>} \vdash \texttt{d}_0 \in \texttt{U}'$$

where

$$\texttt{X<:Object,Y<:T}' \vdash \texttt{U}' \texttt{ <: U}.$$

Since `T` is subtype of `Object` and `*` is subtype of any type, it is safe to substitute `T` and `*` for `X` and `Y`, respectively; we obtain

$$\emptyset;\texttt{S : x,this : C<T>} \vdash [\texttt{T/X, */Y}]\texttt{d}_0 \in [\texttt{T/X, */Y}]\texttt{U}'$$

and

$$\emptyset \vdash [\texttt{T/X, */Y}]\texttt{U}' \texttt{ <: } [\texttt{T/X, */Y}]\texttt{U}.$$

The important point here is the argument type `S` remains the same after the substitution; thus, it is safe to substitute the actual argument for the formal argument. (On the other hand, if erasure changes `S`, `S′` may not be subtype of [`T/X, */Y`]`S` and the substitution may not be safe.) Finally, we have

$$\emptyset;\emptyset \vdash [\texttt{d/x, new C<T>(}\overline{\texttt{e}}\texttt{)/this}][\texttt{T/X, */Y}]\texttt{d}_0 \in [\texttt{T/X, */Y}]\texttt{U}'$$

and it is easy to show [`T/X, */Y`]`U′` is subtype of the erasure of `U`, the type of the original expression $\texttt{e}_0\texttt{.m(d)}$.

5.2 Syntax of Raw FGJ

We use the same notational conventions as in FGJ. The abstract syntax of Raw FGJ is as follows:

$$
\begin{array}{lcl}
\texttt{N} & ::= & \texttt{C<}\overline{\texttt{T}}\texttt{>} \\
 & | & \texttt{C} \\[1ex]
\texttt{T} & ::= & \texttt{X} \\
 & | & * \\
 & | & \texttt{N} \\[1ex]
\texttt{L} & ::= & \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\[1ex]
\texttt{K} & ::= & \texttt{C(}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{) \{super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{;\}} \\[1ex]
\texttt{M} & ::= & \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> T m (}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e;\}} \\[1ex]
\texttt{e} & ::= & \texttt{x} \\
 & | & \texttt{e.f} \\
 & | & \texttt{e.m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \\
 & | & \texttt{new N(}\overline{\texttt{e}}\texttt{)} \\
 & | & \texttt{(N)e}
\end{array}
$$

Now that raw types are introduced, a non-variable type N is either *cooked*, which has type parameters, or *raw*. We use $head(\texttt{N})$ to know the class name of the non-variable type N: $head(\texttt{C}) = head(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{C}$. We often use $\overline{*}$ for a sequence of the bottom types $*,\ldots,*$ (without subscripts). We enforce to the class table a sanity condition that the bottom type appear only in e, as well as the ones for FGJ; the additional condition means, in other words, that types of fields or method argument/return types cannot include $*$.

We require erasure of types to define the type system (including auxiliary functions defined below). The definitions of type environments Δ, $bound_\Delta(\texttt{T})$ and $|\texttt{T}|_\Delta$ remain the same:

$$bound_\Delta(\texttt{X}) = \Delta(\texttt{X})$$
$$bound_\Delta(\texttt{N}) = \texttt{N}$$
$$|\texttt{T}|_\Delta = head(bound_\Delta(\texttt{T}))$$

Note that $bound_\Delta(*)$ and $|*|_\Delta$ are undefined.

We define auxiliary functions *fields*, *mtype*, *mbody*, and *dcast* below. Their definitions get more complicated because of raw types. First, the definition of $fields(\texttt{N})$ is as follows:

$$fields(\texttt{Object}) = \bullet \qquad \text{(F-OBJECT)}$$

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

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad fields(head(\texttt{N})) = \overline{\texttt{C}}\ \overline{\texttt{g}}}{fields(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{g}}, |\overline{\texttt{S}}|_{\overline{\texttt{X}}<:\overline{\texttt{N}}}\ \overline{\texttt{f}}} \qquad \text{(F-RAW)}$$

The cases for Object and cooked types are the same as before; the case for raw types returns the erased types of the fields.

5.2.1 Example: Under a class table including class `Pair`,

$$\begin{aligned} \mathit{fields}(\texttt{Pair}) &= |\texttt{X}|_{\texttt{X<:Object, Y<:Object}}\ \texttt{fst}, |\texttt{Y}|_{\texttt{X<:Object, Y<:Object}}\ \texttt{snd} \\ &= \texttt{Object fst, Object snd} \end{aligned}$$

The function *mtype* now takes actual type arguments, as well as the method name and the type of a receiver, written $\mathit{mtype}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{N})$. Actual type arguments $\overline{\texttt{V}}$ are used to determine whether the invocation is raw or not. Remember that, even if the type of the receiver is cooked, type arguments may be missing. Therefore, the length of $\overline{\texttt{V}}$ is zero and the method takes more than zero type parameters, it returns the erased types, whether `N` is cooked or not; if the erasure changes the argument types, unchecked warning is signaled. (the rules MT-ClassRaw and MT-Raw).

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> U m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e;\}} \in \overline{\texttt{M}} \qquad \#(\overline{\texttt{V}}) = \#(\overline{\texttt{Y}}) \end{array}}{\mathit{mtype}(\texttt{m<}\overline{\texttt{V}}\texttt{>, C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \rightarrow \texttt{U})} \quad \text{(MT-Class)}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> U m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e;\}} \in \overline{\texttt{M}} \qquad \#(\overline{\texttt{Y}}) > 0 \\ \Delta' = \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}, \overline{\texttt{Y}}\texttt{<:}\overline{\texttt{P}} \qquad \textit{unchecked warning} \text{ if } |\overline{\texttt{U}}|_{\Delta'} \neq \overline{\texttt{U}} \end{array}}{\mathit{mtype}(\texttt{m<>, C<}\overline{\texttt{T}}\texttt{>}) = |\overline{\texttt{U}}|_{\Delta'} \rightarrow |\texttt{U}|_{\Delta'}} \quad \text{(MT-ClassRaw)}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> U m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e;\}} \in \overline{\texttt{M}} \\ \Delta' = \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}, \overline{\texttt{Y}}\texttt{<:}\overline{\texttt{P}} \qquad \textit{unchecked warning} \text{ if } |\overline{\texttt{U}}|_{\Delta'} \neq \overline{\texttt{U}} \end{array}}{\mathit{mtype}(\texttt{m<>, C}) = |\overline{\texttt{U}}|_{\Delta'} \rightarrow |\texttt{U}|_{\Delta'}} \quad \text{(MT-Raw)}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{m} \text{ is not defined in } \overline{\texttt{M}} \end{array}}{\mathit{mtype}(\texttt{m<}\overline{\texttt{V}}\texttt{>, C<}\overline{\texttt{T}}\texttt{>}) = \mathit{mtype}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})} \quad \text{(MT-Super)}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{m} \text{ is not defined in } \overline{\texttt{M}} \end{array}}{\mathit{mtype}(\texttt{m<>, C}) = \mathit{mtype}(\texttt{m<>}, \mathit{head}(\texttt{N}))} \quad \text{(MT-Super-Raw)}$$

As in FGJ, $\mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{N})$ returns the method body where type variables are replaced by actual ones. When actual type parameters are missing ($\overline{\texttt{V}}$ is empty or `N` is raw), the bottom type is substituted for type variables (the rules MB-ClassRaw and MB-Raw).

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> U m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e}_0\texttt{;\}} \in \overline{\texttt{M}} \qquad \#(\overline{\texttt{V}}) = \#(\overline{\texttt{Y}}) \end{array}}{\mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>, C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}, [\overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{e}_0)} \quad \text{(MB-Class)}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> U m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e}_0\texttt{;\}} \in \overline{\texttt{M}} \qquad \#(\overline{\texttt{Y}}) > 0 \end{array}}{\mathit{mbody}(\texttt{m<>, C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}, [\overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{*}}/\overline{\texttt{Y}}]\texttt{e}_0)} \quad \text{(MB-ClassRaw)}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> U m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e}_0\texttt{;\}} \in \overline{\texttt{M}} \end{array}}{\mathit{mbody}(\texttt{m<>, C}) = (\overline{\texttt{x}}, [\overline{\texttt{*}}/\overline{\texttt{X}}, \overline{\texttt{*}}/\overline{\texttt{Y}}]\texttt{e}_0)} \quad \text{(MB-Raw)}$$

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

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \texttt{m} \text{ is not defined in } \overline{\texttt{M}}}{mbody(\texttt{m<>, C}) = mbody(\texttt{m<>}, head(\texttt{N}))} \quad \text{(MB-SUPER-RAW)}$$

The definition of $dcast(\texttt{C}, \texttt{D})$ is exactly the same as before. $dcast(\texttt{C}, \texttt{D})$ is the least partial order closed under the following rule:

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{...\}} \qquad \overline{\texttt{X}} = FV(\texttt{N})}{dcast(\texttt{C}, \texttt{D})}$$

where $FV(\texttt{N})$ denotes the set of type variables in N.

5.3 Typing

5.3.1 Subtyping

We write $\Delta \vdash \texttt{S <: T}$ if S is a safe subtype of T under the assumption given by Δ. The first four rules below are from the FGJ subtyping rules:

$$\Delta \vdash \texttt{T <: T} \quad \text{(S-REFL)}$$

$$\frac{\Delta \vdash \texttt{S <: T} \qquad \Delta \vdash \texttt{T <: U}}{\Delta \vdash \texttt{S <: U}} \quad \text{(S-TRANS)}$$

$$\Delta \vdash \texttt{X <: } \Delta(\texttt{X}) \quad \text{(S-VAR)}$$

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

Besides these rules, we have rules for the bottom types and raw types. First, the bottom type is a subtype of any type.

$$\Delta \vdash * \texttt{ <: T} \quad \text{(S-BOT)}$$

For raw types, we have two rules; a raw type C is subtype of D if D is a name of a superclass of C (strictly speaking, the rule S-TRANS is required), and a cooked type C<$\overline{\texttt{T}}$> is a subtype of the corresponding raw type C.

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{...\}}}{\Delta \vdash \texttt{C <: } head(\texttt{N})} \quad \text{(S-RAW)}$$

$$\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{> <: C} \quad \text{(S-COOKED/RAW)}$$

The judgment of unsafe subtyping is written $\Delta \vdash \texttt{S <:}_{\text{unsafe}} \texttt{T}$. As we will see in the typing rules, unsafe subtyping is used in comparing the types of formal arguments (of a constructor or method) with those of actual arguments. First of all, if S is a safe subtype of T, then S is also a unsafe subtype of T.

$$\frac{\Delta \vdash \texttt{S} <: \texttt{T}}{\Delta \vdash \texttt{S} <:_{\text{unsafe}} \texttt{T}} \qquad \text{(SU-SAFE)}$$

If a raw type C is a safe subtype of another raw type D, then we allow C to be an unsafe subtype of the cooked type D<$\overline{\texttt{T}}$> with unchecked warning.

$$\frac{\Delta \vdash \texttt{C} <: \texttt{D} \qquad \textit{unchecked warning}}{\Delta \vdash \texttt{C} <:_{\text{unsafe}} \texttt{D<}\overline{\texttt{T}}\texttt{>}} \qquad \text{(SU-UNCHECKED)}$$

5.3.2 Well-Formed Types

We write $\Delta \vdash \texttt{T}$ ok if type T is well formed in the context Δ. The first four rules below are the same as ones for FGJ:

$$\Delta \vdash \texttt{Object ok} \qquad \text{(WF-OBJECT)}$$

$$\frac{\texttt{X} \in \textit{dom}(\Delta)}{\Delta \vdash \texttt{X} \text{ ok}} \qquad \text{(WF-VAR)}$$

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

Besides, we have rules for the bottom type and raw types: the bottom type is always well formed and a raw type is well formed if it is in the domain of *CT*.

$$\Delta \vdash * \text{ ok} \qquad \text{(WF-BOT)}$$

$$\frac{\textit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{...\}}}{\Delta \vdash \texttt{C} \text{ ok}} \qquad \text{(WF-RAW)}$$

5.3.3 Typing Rules

As before, an *environment* Γ is a finite mapping from variables to types, written $\overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}$ and the typing judgment for expressions is of the form $\Delta; \Gamma \vdash \texttt{e} \in \texttt{T}$, read as "in the type environment Δ and the environment Γ, the expression e has type T." Most rules are the same as in FGJ:

$$\Delta; \Gamma \vdash \texttt{x} \in \Gamma(\texttt{x}) \qquad \text{(GT-VAR)}$$

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

In the rules GT-INVK and GT-NEW below, the types $\overline{\texttt{S}}$ of the actual arguments $\overline{\texttt{e}}$ can be unsafe subtypes of the expected argument types $[\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}}$.

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

$$\frac{\begin{array}{c}\Delta \vdash \texttt{N}\ \text{ok} \qquad \textit{fields}(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \\ \Delta;\Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \qquad \Delta \vdash \overline{\texttt{S}}\ \texttt{<:}_{\text{unsafe}}\ \overline{\texttt{T}} \\ \textit{unchecked warning}\ \text{if}\ \Delta \vdash \texttt{N <: C}\ \text{and}\ \texttt{C} \neq \texttt{Object}\ \text{but}\ \Delta \vdash \texttt{N} \not\texttt{<:}\ \texttt{C<}\overline{\texttt{T}}\texttt{>}\ \text{for all}\ \overline{\texttt{T}}\end{array}}{\Delta;\Gamma \vdash \texttt{new N(}\overline{\texttt{e}}\texttt{)} \in \texttt{N}} \qquad \text{(GT-New)}$$

The last condition of the rule GT-New make sure that the class of N does not extend (whether directly or not) a raw type.

The rules for typecasts are essentially the same as before except some notational differences: <: is substituted for $\trianglelefteq$ since $\texttt{C} \trianglelefteq \texttt{D}$ if and only if C <: D.

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

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \qquad \Delta \vdash \texttt{N}\ \text{ok} \\ \Delta \vdash \texttt{N}\ \texttt{<:}\ \textit{bound}_\Delta(\texttt{T}_0) \qquad \texttt{N} \neq \textit{bound}_\Delta(\texttt{T}_0) \\ \textit{dcast}(|\texttt{N}|_\Delta,\ |\texttt{T}_0|_\Delta)\end{array}}{\Delta;\Gamma \vdash \texttt{(N)}\texttt{e}_0 \in \texttt{N}} \qquad \text{(GT-DCast)}$$

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

In addition to the rules above, we require the following two rules relevant to the bottom type:

$$\frac{\Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{*}}{\Delta;\Gamma \vdash \texttt{e}_0\texttt{.f}_i \in \texttt{*}} \qquad \text{(GT-Bot-Field)}$$

$$\frac{\Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{*} \qquad \Delta \vdash \overline{\texttt{V}}\ \text{ok} \qquad \Delta;\Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}}}{\Delta;\Gamma \vdash \texttt{e}_0\texttt{.m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in \texttt{*}} \qquad \text{(GT-Bot-Invk)}$$

We allow a field access and a method invocation on an expression of the bottom type since the bottom type is subtype of any type; the resulted expression is also given the bottom type. Roughly speaking, the bottom type corresponds the empty set. Thus, any operation on the element of the empty set is vacuously allowed and the result also belongs to the empty set. Similar rules can be found in an extension of System $F_\leq$ [CMMS94] with the bottom type [Pie97].

These rules are required mainly for technical reasons. The subject reduction theorem (Theorem 5.5.1) is proved by induction on the derivation of $\texttt{e} \longrightarrow \texttt{e}'$. The rules above can easily handle with the case where the reduction step is derived from the context rule GRC-Field or GRC-Invk-Recv defined later. For example, suppose we have $\texttt{e}_0\texttt{.f} \longrightarrow \texttt{e}_0'\texttt{.f}$ from $\texttt{e}_0 \longrightarrow \texttt{e}_0'$.

By the induction hypothesis $\mathtt{e}_0{}'$ is given some subtype of the type of $\mathtt{e}_0$, which might be the bottom type. In that case, the rule GT-Bot-Field is used to give $\mathtt{e}_0{}'\mathtt{.f}$ the bottom type. Although it makes the proof of subject reduction easier and clearer, it complicates other portions of the formalization, such as properties of erasure (see Section 5.6). Thus, it is desirable to have a proof without depending on these rules; it should be possible because we expect that an expression is always given type other than `*` as long as the program satisfies the syntactic condition that the bottom type appear only in expressions. But, it is left for future work for now.

The typing rules for method declarations and classes remain the same.

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \qquad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}, \overline{\mathtt{P}}\ \text{ok} \\ \Delta; \overline{\mathtt{x}}\colon\overline{\mathtt{T}}, \mathtt{this}\colon\mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 \in \mathtt{S} \qquad \Delta \vdash \mathtt{S <: T} \\ CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \{\ldots\} \\ \text{if } \mathit{mtype}(\mathtt{m<>}, \mathit{head}(\mathtt{N})) \text{ is defined, then } \mathit{mtype}(\mathtt{m<}\overline{\mathtt{Y}}\mathtt{>}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}}\to\mathtt{U} \text{ and } \Delta \vdash \mathtt{T <: U}\end{array}}{\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>\ T\ m\ (}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e}_0\mathtt{;\}\ OK\ IN\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}} \quad \text{(GT-Method)}$$

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

5.4 Reduction Semantics

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 $\longrightarrow$.

Thanks to the auxiliary definitions, the reduction rules are exactly the same as before. We show them just for a reminder:

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

$$\frac{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\mathtt{(new\ N(}\overline{\mathtt{e}}\mathtt{)).m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{d}}\mathtt{)} \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)/this}]\mathtt{e}_0} \quad \text{(GR-Invk)}$$

$$\frac{\emptyset \vdash \mathtt{N <: P}}{\mathtt{(P)(new\ N(}\overline{\mathtt{e}}\mathtt{))} \longrightarrow \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)}} \quad \text{(GR-Cast)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0\mathtt{.f} \longrightarrow \mathtt{e}_0{}'\mathtt{.f}} \quad \text{(GRC-Field)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \longrightarrow \mathtt{e}_0{}'\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}} \quad \text{(GRC-Inv-Recv)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{e}_0\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>(}\ldots\mathtt{e}_i{}'\mathtt{,}\ldots\mathtt{)}} \quad \text{(GRC-Inv-Arg)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{new\ N(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{new\ N(}\ldots\mathtt{e}_i{}'\mathtt{,}\ldots\mathtt{)}} \quad \text{(GRC-New-Arg)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0{}'}{\texttt{(N)e}_0 \longrightarrow \texttt{(N)e}_0{}'} \qquad \text{(GRC-Cast)}$$

5.5 Properties of Checked Programs

In this section, we show type soundness for checked programs. Although the proof steps are similar to those of FGJ, each proof of the lemmas becomes complicated due to raw types and the bottom type.

5.5.1 Theorem [Subject reduction]: If $\Delta;\Gamma \vdash \texttt{e} \in \texttt{T}$ without *unchecked warning* and $\texttt{e} \longrightarrow \texttt{e}'$, then $\Delta;\Gamma \vdash \texttt{e}' \in \texttt{T}'$ without *unchecked warning*, for some $\texttt{T}'$ such that $\Delta \vdash \texttt{T}' \texttt{<:} \texttt{T}$.

Proof: See below. ■

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

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

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

Proof: Similar to the proof of Theorem 4.5.1.2. ■

We develop a proof of Theorem 5.5.1 below.

5.5.3 Lemma [Weakening]: Suppose $\Delta, \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}} \vdash \overline{\texttt{N}}$ ok and $\Delta \vdash \texttt{U}$ ok.

1. If $\Delta \vdash \texttt{S} \texttt{<:} \texttt{T}$, then $\Delta, \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}} \vdash \texttt{S} \texttt{<:} \texttt{T}$.

2. If $\Delta \vdash \texttt{S}$ ok, then $\Delta, \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}} \vdash \texttt{S}$ ok.

3. If $\Delta;\Gamma \vdash \texttt{e} \in \texttt{T}$, then $\Delta;\Gamma, \texttt{x} : \texttt{U} \vdash \texttt{e} \in \texttt{T}$, and $\Delta, \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}};\Gamma \vdash \texttt{e} \in \texttt{T}$.

Proof: Each of them is proved by straightforward induction on the derivation of $\Delta \vdash \texttt{S} \texttt{<:} \texttt{T}$ and $\Delta \vdash \texttt{S}$ ok and $\Delta;\Gamma \vdash \texttt{e} \in \texttt{T}$. ■

5.5.4 Lemma: If $\Delta \vdash \texttt{N} \texttt{<:} \texttt{P}$ and $\Delta \vdash \textit{head}(\texttt{P}) \not\texttt{<:} \texttt{C}$ and $\Delta \vdash \texttt{C} \not\texttt{<:} \textit{head}(\texttt{P})$, then $\Delta \vdash \textit{head}(\texttt{N}) \not\texttt{<:} \texttt{C}$ and $\Delta \vdash \texttt{C} \not\texttt{<:} \textit{head}(\texttt{N})$.

Proof: It is easy to see that $\Delta \vdash \texttt{N} \texttt{<:} \texttt{P}$ implies $\Delta \vdash \textit{head}(\texttt{E}) \texttt{<:} \textit{head}(\texttt{D})$. The conclusions are easily proved by contradiction. ■

5.5.5 Lemma: Suppose $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \texttt{<:} \texttt{N}$ and $\textit{dcast}(\texttt{C}, \textit{head}(\texttt{N}))$. If $\Delta \vdash \texttt{C<}\overline{\texttt{T}}'\texttt{>} \texttt{<:} \texttt{N}$, then $\overline{\texttt{T}}' = \overline{\texttt{T}}$.

Proof: The case where $\texttt{C} = \textit{head}(\texttt{N})$ is easy; since *dcast* is antisymmetric, if $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \texttt{<:} \texttt{N}$, then $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ and N must be equal. The case where $\textit{dcast}(\texttt{C}, \texttt{D})$ because $\textit{dcast}(\texttt{C}, \texttt{E})$ and $\textit{dcast}(\texttt{E}, \textit{head}(\texttt{N}))$ is also easy; note that, from every judgment $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \texttt{<:} \texttt{N}$, we can have $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \texttt{<:} \texttt{P}$ and $\Delta \vdash \texttt{P} \texttt{<:} \texttt{N}$ where $\textit{head}(\texttt{P}) = \texttt{E}$. Finally, if $\textit{head}(\texttt{N})$ is a direct superclass of C, $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ is uniquely determined by N because $FV(\texttt{N}') = \overline{\texttt{X}}$ where $CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N}'\ \texttt{\{...\}}$, finishing the proof. ■

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

Proof: By induction on the derivation of $\Delta_1, \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}, \Delta_2 \vdash \texttt{S} \texttt{ <: } \texttt{T}$. Similar to the proof of Lemma 5.5.6. Note that the cases (S-Bot, S-Raw, S-Cooked/Raw) are trivial. ∎

5.5.7 Lemma [Type substitution preserves type well-formedness]: If $\Delta_1, \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}, \Delta_2 \vdash \texttt{T}$ ok and $\Delta_1 \vdash \overline{\texttt{U}} \texttt{ <: } [\overline{\texttt{U}}/\overline{\texttt{X}}]\overline{\texttt{N}}$ with $\Delta_1 \vdash \overline{\texttt{U}}$ ok and none of $\overline{\texttt{X}}$ appearing in Δ_1, then $\Delta_1, [\overline{\texttt{U}}/\overline{\texttt{X}}]\Delta_2 \vdash [\overline{\texttt{U}}/\overline{\texttt{X}}]\texttt{T}$ ok.

Proof: By induction on the derivation of $\Delta_1, \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}, \Delta_2 \vdash \texttt{T}$ ok, with a case analysis on the last rule used. Similar to the proof of Lemma 5.5.7. Note that the cases (WF-Bot and WF-Raw) are trivial. ∎

5.5.8 Lemma: If $\Delta \vdash \texttt{S} \texttt{ <: } \texttt{T}$ and $\Delta \vdash \texttt{S}$ ok and $\Delta \vdash \texttt{T}$ ok and neither S nor T is $*$, then $\emptyset \vdash |\texttt{S}|_\Delta \texttt{ <: } |\texttt{T}|_\Delta$.

Proof: By straightforward induction on the derivation of $\Delta \vdash \texttt{S} \texttt{ <: } \texttt{T}$. ∎

5.5.9 Lemma: Suppose $\Delta_1, \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}, \Delta_2 \vdash \texttt{T}$ ok where $\texttt{T} \neq *$ and $\Delta_1 \vdash \overline{\texttt{U}} \texttt{ <: } [\overline{\texttt{U}}/\overline{\texttt{X}}]\overline{\texttt{N}}$ with $\Delta_1 \vdash \overline{\texttt{U}}$ ok and none of $\overline{\texttt{X}}$ appearing in Δ_1. Then, $\Delta_1, [\overline{\texttt{U}}/\overline{\texttt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1, [\overline{\texttt{U}}/\overline{\texttt{X}}]\Delta_2}([\overline{\texttt{U}}/\overline{\texttt{X}}]\texttt{T}) \texttt{ <: } [\overline{\texttt{U}}/\overline{\texttt{X}}](\mathit{bound}_{\Delta_1,\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}},\Delta_2}(\texttt{T}))$.

Proof: The case where T is a nonvariable type is trivial. The case where T is a type variable X and $\texttt{X} \in \mathit{dom}(\Delta_1) \cup \mathit{dom}(\Delta_2)$ is also easy. Finally, if T is a type variable $\texttt{X}_i$, then $\mathit{bound}_{\Delta_1, [\overline{\texttt{U}}/\overline{\texttt{X}}]\Delta_2}([\overline{\texttt{U}}/\overline{\texttt{X}}]\texttt{T}) = \texttt{U}_i$ and $[\overline{\texttt{U}}/\overline{\texttt{X}}](\mathit{bound}_{\Delta_1,\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}},\Delta_2}(\texttt{T})) = [\overline{\texttt{U}}/\overline{\texttt{X}}]\texttt{N}_i$; the assumption $\Delta_1 \vdash \overline{\texttt{U}} \texttt{ <: } [\overline{\texttt{U}}/\overline{\texttt{X}}]\overline{\texttt{N}}$ and Lemma 5.5.3 finish the proof. ∎

5.5.10 Lemma: If $\Delta \vdash \texttt{S} \texttt{ <: } \texttt{T}$ and $\texttt{S} \neq *$ and $\Delta \vdash \texttt{S}$ ok and $\mathit{fields}(\mathit{bound}_\Delta(\texttt{T})) = \overline{\texttt{T}}\ \overline{\texttt{f}}$, then $\mathit{fields}(\mathit{bound}_\Delta(\texttt{S})) = \overline{\texttt{S}}\ \overline{\texttt{g}}$ and $\Delta \vdash \texttt{S}_i \texttt{ <: } \texttt{T}_i$ and $\texttt{g}_i = \texttt{f}_i$ for all $i \leq \#(\overline{\texttt{f}})$.

Proof: By induction on the derivation of $\Delta \vdash \texttt{S} \texttt{ <: } \texttt{T}$ with a case analysis on the last rule used. The cases similar to those in the proof of Lemma 4.5.1.10 are omitted.

Case S-Bot:

Can't happen.

Case S-Raw: $\texttt{S} = \texttt{C}$ $\quad \texttt{T} = \texttt{D}$ $\quad \mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\triangleleft\texttt{D<}\overline{\texttt{T}}\texttt{> \{}\overline{\texttt{S}}\ \overline{\texttt{g}}\texttt{; ...\}}$

Easy. By the rule F-Raw, $\mathit{fields}(\texttt{C}) = \overline{\texttt{U}}\ \overline{\texttt{f}}, |\overline{\texttt{S}}|_{\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}}\overline{\texttt{g}}$ where $\overline{\texttt{U}}\ \overline{\texttt{f}} = \mathit{fields}([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})$.

Case S-Cooked/Raw: $\texttt{S} = \texttt{C<}\overline{\texttt{T}}\texttt{>}$ $\quad \texttt{T} = \texttt{C}$

By induction on the derivation of $\mathit{fields}(\texttt{C})$. Suppose $\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\triangleleft\texttt{D<}\overline{\texttt{T}}\texttt{> \{}\overline{\texttt{S}}\ \overline{\texttt{g}}\texttt{; ...\}}$. Since $\Delta \vdash \texttt{S}$ ok, we have $\Delta \vdash \overline{\texttt{T}} \texttt{ <: } [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}}$. Then, it is easy to show that $\Delta \vdash [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}} \texttt{ <: } |\overline{\texttt{S}}|_{\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}}$. ∎

5.5.11 Lemma: If $\Delta \vdash \texttt{T}$ ok and $\mathit{mtype}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \mathit{bound}_\Delta(\texttt{T})) = \texttt{<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\to\texttt{U}_0$ without *unchecked warning*, then for any S such that $\Delta \vdash \texttt{S} \texttt{ <: } \texttt{T}$ and $\Delta \vdash \texttt{S}$ ok and $\texttt{S} \neq *$, we have $\mathit{mtype}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \mathit{bound}_\Delta(\texttt{S})) = \texttt{<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\to\texttt{U}_0{}'$ without *unchecked warning* and $\Delta, \overline{\texttt{Y}}\texttt{<:}\overline{\texttt{P}} \vdash \texttt{U}_0{}' \texttt{ <: } \texttt{U}_0$.

Proof: By induction on the derivation of $\Delta \vdash \texttt{S} \texttt{ <: } \texttt{T}$ with a case analysis on the last rule used. The cases similar to those in the proof of Lemma 4.5.1.11 are omitted.

Case S-Bot:

Can't happen.

Case S-Raw: $\mathtt{S} = \mathtt{C}$ $\mathtt{T} = \mathtt{D}$ $CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft \overline{N}> \triangleleft N\ \{\ ...\ \ \overline{M}\}}$
$head(\mathtt{N}) = \mathtt{D}$

The length of $\overline{\mathtt{V}}$ must be zero. If $\overline{\mathtt{M}}$ do not include $\mathtt{m}$, it is easy to show the conclusion, since $mtype(\mathtt{m<>},\ bound_\Delta(\mathtt{S})) = mtype(\mathtt{m<>},\ bound_\Delta(\mathtt{T}))$ by the rule MT-Super-Raw.

On the other hand, suppose $\overline{\mathtt{M}}$ includes the declaration $\mathtt{<\overline{Z} \triangleleft \overline{Q}>T_0{}'\ \ m\ \ (\overline{T}\ \ \overline{x})\ \ \{\uparrow e;\}}$. We have two subcases according to whether $\mathtt{N}$ is raw or not.

Subcase: $\mathtt{N} = \mathtt{D<\overline{S}>}$

Since $mtype(\mathtt{m<>}, \mathtt{D})$ is well defined, by the rule GT-Method, $mtype(\mathtt{m<\overline{Z}>}, \mathtt{N}) = \mathtt{<\overline{Z} \triangleleft \overline{Q}>\overline{T} \rightarrow T_0}$ and $\mathtt{\overline{X}{<:}\overline{N}},\ \mathtt{\overline{Z}{<:}\overline{Q}} \vdash \mathtt{T_0{}'} <: \mathtt{T_0}$. By Lemma 5.5.8, $\emptyset \vdash |\mathtt{T_0{}'}|_{\mathtt{\overline{X}<:\overline{N}},\ \mathtt{\overline{Z}<:\overline{Q}}} <: |\mathtt{T_0}|_{\mathtt{\overline{X}<:\overline{N}},\ \mathtt{\overline{Z}<:\overline{Q}}}$. It is easy to see that, by induction on the derivation of $mtype(\mathtt{m<\overline{Z}>}, \mathtt{N})$, $\mathtt{<\overline{Y} \triangleleft \overline{P}>\overline{U} \rightarrow U_0} = \mathtt{\overline{T} \rightarrow E}$ and $\emptyset \vdash |\mathtt{T_0}|_{\mathtt{\overline{X}<:\overline{N}},\ \mathtt{\overline{Z}<:\overline{Q}}} <: \mathtt{E}$ and $|\mathtt{\overline{T}}|_{\mathtt{\overline{X}<:\overline{N}},\ \mathtt{\overline{Z}<:\overline{Q}}} = \mathtt{\overline{T}}$.

Subcase: $\mathtt{N} = \mathtt{D}$

Similar.

Case S-Cooked/Raw: $\mathtt{S} = \mathtt{C<\overline{T}>}$ $\mathtt{T} = \mathtt{C}$

By induction on the derivation of $mtype(\mathtt{m<>}, \mathtt{C})$. If $\mathtt{m}$ is defined in $CT(\mathtt{C})$ then, it must be of the form $\mathtt{<\overline{Y} \triangleleft \overline{P}>T_0\ \ m\ \ (\overline{D}\ \ \overline{x})\ \ \{\uparrow e;\}}$ and it is easy to show $\Delta, \mathtt{\overline{Y}{<:}\overline{P}} \vdash \mathtt{T_0} <: |\mathtt{T_0}|_{\Delta,\ \mathtt{\overline{Y}<:\overline{P}}}$. ■

5.5.12 Lemma [Type substitution preserves typing]: If $\Delta_1,\ \mathtt{\overline{X}{<:}\overline{N}},\ \Delta_2; \Gamma \vdash \mathtt{e} \in \mathtt{T}$ without *unchecked warning* and $\Delta_1 \vdash \mathtt{\overline{U}} <: [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\mathtt{\overline{N}}$ where $\Delta_1 \vdash \mathtt{\overline{U}}$ ok and none of $\mathtt{\overline{X}}$ appears in Δ_1, then $\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2; [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Gamma \vdash [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\mathtt{e} \in \mathtt{S}$ without *unchecked warning* for some $\mathtt{S}$ such that $\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2 \vdash \mathtt{S} <: [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\mathtt{T}$.

Proof: By induction on the derivation of $\Delta_1, \mathtt{\overline{X}{<:}\overline{N}}, \Delta_2; \Gamma \vdash \mathtt{e} \in \mathtt{T}$ with a case analysis on the last rule used. The cases similar to those in the proof of Lemma 4.5.1.12 are omitted.

Case GT-Field: $\mathtt{e} = \mathtt{e_0.f}_i$ $\Delta_1, \mathtt{\overline{X}{<:}\overline{N}},\ \Delta_2; \Gamma \vdash \mathtt{e_0} \in \mathtt{T_0}$ $fields(bound_{\Delta_1,\ \mathtt{\overline{X}<:\overline{N}},\ \Delta_2}(\mathtt{T_0})) = \mathtt{\overline{T}\ \ \overline{f}}$
$\mathtt{T} = \mathtt{T}_i$

By the induction hypothesis, $\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2; [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Gamma \vdash [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\mathtt{e_0} \in \mathtt{S_0}$ and $\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2 \vdash \mathtt{S_0} <: [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\mathtt{T_0}$. Now we have two subcases:

Subcase: $\mathtt{S_0} = *$

By the rule GT-Bot-Field, $\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2 \vdash [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\mathtt{e} \in *$.

Subcase: $\mathtt{S_0} \neq *$

By Lemma 5.5.9,

$$\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2 \vdash bound_{\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2}(\mathtt{T_0}) <: [\mathtt{\overline{U}}/\mathtt{\overline{X}}]bound_{\Delta_1, \mathtt{\overline{X}<:\overline{N}},\ \Delta_2}(\mathtt{T_0}).$$

Then, it is easy to show

$$\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2 \vdash bound_{\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2}(\mathtt{S_0}) <: [\mathtt{\overline{U}}/\mathtt{\overline{X}}]bound_{\Delta_1, \mathtt{\overline{X}<:\overline{N}},\ \Delta_2}(\mathtt{T_0}).$$

By Lemma 5.5.10, $fields(bound_{\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2}(\mathtt{S_0})) = \mathtt{\overline{S}\ \ \overline{g}}$ and we have $\mathtt{f}_j = \mathtt{g}_j$ and $\Delta \vdash \mathtt{S}_j <: [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\mathtt{T}_j$ for $j \leq \#(\mathtt{\overline{f}})$. By the rule GT-Field, $\Delta_1,\ [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Delta_2; [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\Gamma \vdash [\mathtt{\overline{U}}/\mathtt{\overline{X}}]\mathtt{e_0.f}_i \in \mathtt{S}_i$. Letting $\mathtt{S} = \mathtt{S}_i$ finishes the case.

Case GT-Invk: $\mathtt{e} = \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}$ $\quad \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0$
$\mathit{mtype}(\mathtt{m}, \mathit{bound}_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}(\mathtt{T}_0)) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{W}}\to\mathtt{W}_0$
$\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \overline{\mathtt{V}}$ ok $\quad \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \overline{\mathtt{V}} \mathtt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$
$\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}}$ $\quad \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \overline{\mathtt{S}} \mathtt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{W}}$
$\mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{W}_0$

By the induction hypothesis,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0 \in \mathtt{S}_0$$
$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S}_0 \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0$$

and

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{e}} \in \overline{\mathtt{S}}'$$
$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \overline{\mathtt{S}}' \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}.$$

We have two subcases:

Subcase: $\mathtt{S}_0 = *$

By the rule GT-Bot-Invk, $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{e} \in *$.

Subcase: $\mathtt{S}_0 \neq *$

This subcase is essentially similar to the case for GT-Invk in the proof of Lemma 4.5.1.12.
By using Lemma 4.5.1.9, it is easy to show

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0) \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathit{bound}_{\Delta_1,\, \overline{\mathtt{X}}<:\overline{\mathtt{N}},\, \Delta_2}(\mathtt{T}_0).$$

Then, by Lemma 5.5.11,

$$\mathit{mtype}(\mathtt{m}, \mathit{bound}_{\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0)) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}\mathtt{>}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}\to\mathtt{W}_0{}'$$
$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2, \overline{\mathtt{Y}}\mathtt{<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{P}} \vdash \mathtt{W}_0{}' \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W}_0.$$

By Lemma 5.5.7,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}} \text{ ok}$$

Without loss of generality, we can assume that $\overline{\mathtt{X}}$ and $\overline{\mathtt{Y}}$ are distinct and that none of $\overline{\mathtt{Y}}$ appear in $\overline{\mathtt{U}}$, and thus $[\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}] = [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]$. By Lemma 5.5.6,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}} \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \quad (= [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{P}})$$
$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}} \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{W}} \quad (= [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}).$$

By the rule S-Trans,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \overline{\mathtt{S}}' \mathtt{<:} [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}.$$

By Lemma 5.5.6, we have

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{W}_0{}' \mathtt{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{W}_0 \quad (= [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W}_0).$$

Finally, by the rule GT-Invk,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash ([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0)\mathtt{.m<}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}\mathtt{>(}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{d}}\mathtt{)} \in \mathtt{S}$$

where $\mathtt{S} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{W}_0{}'$, finishing the case.

Case GT-Bot-Field:
Straightforward.
Case GT-Bot-Method:
Straightforward. ■

5.5.13 Lemma [Term substitution preserves typing]: If $\Delta;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{T}}\vdash\mathtt{e}\in\mathtt{T}$ and, $\Delta;\Gamma\vdash\overline{\mathtt{d}}\in\overline{\mathtt{S}}$ both without *unchecked warning* where $\Delta\vdash\overline{\mathtt{S}}$ <: $\overline{\mathtt{T}}$, then $\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}\in\mathtt{S}$ without *unchecked warning* for some S such that $\Delta\vdash\mathtt{S}$ <: T.

Proof: By induction on the derivation of $\Delta;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{T}}\vdash\mathtt{e}\in\mathtt{T}$. The cases similar to those in the proof of Lemma 4.5.1.13 are omitted.

Case GT-Field: $\mathtt{e}=\mathtt{e}_0\mathtt{.f}_i$ $\quad\Delta;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{T}}\vdash\mathtt{e}_0\in\mathtt{T}_0$ $\quad fields(bound_\Delta(\mathtt{T}_0))=\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ $\quad\mathtt{T}=\mathtt{T}_i$

By the induction hypothesis, $\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\in\mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta\vdash\mathtt{S}_0$ <: $\mathtt{T}_0$.
We have two subcases:

Subcase: $\mathtt{S}_0=*$
By the rule GT-Bot-Field, $\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}\in *$.

Subcase: $\mathtt{S}_0=*$
Essentially similar to the case for GT-Field in the proof of Lemma 4.5.1.13. By Lemma 5.5.10, $fields(bound_\Delta(\mathtt{S}_0))=\overline{\mathtt{S}}\ \overline{\mathtt{g}}$ such that $\Delta\vdash\mathtt{S}_j$ <: $\mathtt{T}_j$ and $\mathtt{f}_j=\mathtt{g}_j$ for all $j\leq\#(\overline{\mathtt{T}})$. Therefore, by the rule GT-Field, $\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\mathtt{.f}_i\in\mathtt{S}_i$.

Case GT-Invk: $\mathtt{e}=\mathtt{e}_0\mathtt{.m}$<$\overline{\mathtt{V}}$>$(\overline{\mathtt{e}})$ $\quad\Delta;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{T}}\vdash\mathtt{e}_0\in\mathtt{T}_0$ $\quad mtype(\mathtt{m},\ bound_\Delta(\mathtt{T}_0))=$ <$\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}$>$\overline{\mathtt{U}}\to\mathtt{U}$
$\Delta\vdash\overline{\mathtt{V}}$ ok $\quad\Delta\vdash\overline{\mathtt{V}}$ <: $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$ $\quad\Delta;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{T}}\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$
$\Delta\vdash\overline{\mathtt{S}}$ <: $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$ $\quad\mathtt{T}=[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$

By the induction hypothesis, $\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\in\mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta\vdash\mathtt{S}_0$ <: $\mathtt{T}_0$ and $\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\in\overline{\mathtt{W}}$ for some $\overline{\mathtt{W}}$ such that $\Delta\vdash\overline{\mathtt{W}}$ <: $\overline{\mathtt{S}}$.
We have two subcases now:

Subcase: $\mathtt{S}_0=*$
By the rule GT-Bot-Invk, $\Delta;\Gamma\vdash\mathtt{e}\in *$.

Subcase: $\mathtt{S}_0\neq *$
This subcase is essentially similar to the case for GT-Invk in the proof of Lemma 4.5.1.13. By Lemma 5.5.11, $mtype(\mathtt{m},\ bound_\Delta(\mathtt{S}_0))=$ <$\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{Q}}$>$\overline{\mathtt{U}}'\to\mathtt{U}'$ where $\overline{\mathtt{Q}}=[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$ and $\overline{\mathtt{U}}'=[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$ and $\Delta,\overline{\mathtt{Z}}$<:$\overline{\mathtt{Q}}\vdash[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\mathtt{U}'$ <: U. By Lemma 5.5.6, $\Delta\vdash[\overline{\mathtt{V}}/\overline{\mathtt{Z}}]\mathtt{U}'$ <: $[\overline{\mathtt{V}}/\overline{\mathtt{Z}}]\mathtt{U}$. (Note that $\Delta\vdash\overline{\mathtt{V}}$ <: $[\overline{\mathtt{V}}/\overline{\mathtt{Z}}]\overline{\mathtt{Q}}$ since $\overline{\mathtt{Q}}=[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$.) By the rule GT-Method, $\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}](\mathtt{e}_0\mathtt{.m}$<$\overline{\mathtt{V}}$>$(\overline{\mathtt{e}}))\in[\overline{\mathtt{V}}/\overline{\mathtt{Z}}]\mathtt{U}'$. Letting $\mathtt{S}=[\overline{\mathtt{V}}/\overline{\mathtt{Z}}]\mathtt{U}'$ finishes the case.

■

5.5.14 Lemma: Suppose $\Delta\vdash\mathtt{N}$ ok and for any C such that $\Delta\vdash\mathtt{N}$ <: C, there exists $\overline{\mathtt{T}}$ such that $\Delta\vdash\mathtt{N}$ <: C<$\overline{\mathtt{T}}$>. If $mtype(\mathtt{m}$<$\overline{\mathtt{V}}$>$,\ \mathtt{N})=$ <$\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}$>$\overline{\mathtt{U}}\to\mathtt{U}$ without *unchecked warning* and $mbody(\mathtt{m}$<$\overline{\mathtt{V}}$>$,\ \mathtt{N})=(\overline{\mathtt{x}},\ \mathtt{e}_0)$ with $\Delta\vdash\overline{\mathtt{V}}$ ok and $\Delta\vdash\overline{\mathtt{V}}$ <: $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$, then there exist some P and S such that $\Delta\vdash\mathtt{N}$ <: P where $\Delta\vdash\mathtt{P}$ ok, and $\Delta\vdash\mathtt{S}$ <: $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$ where $\Delta\vdash\mathtt{S}$ ok and $\Delta;\overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\ \mathtt{this}:\mathtt{P}\vdash\mathtt{e}_0\in\mathtt{S}$ without *unchecked warning.*

Proof: By induction on the derivation of $mbody(\mathtt{m}$<$\overline{\mathtt{V}}$>$,\ \mathtt{C}$<$\overline{\mathtt{T}}$>$)=(\overline{\mathtt{x}},\ \mathtt{e})$.

Case MB-Raw, MB-SuperRaw: $\quad \mathtt{N} = \mathtt{C}$

Can't happen since there is no $\mathtt{C<\overline{T}>}$ such that $\Delta \vdash \mathtt{N} \mathrel{<:} \mathtt{C<\overline{T}>}$.

Case MB-Class:

Similar to the case for MB-Class in the proof of Lemma 4.5.1.14.

Case MB-Class/Raw: $\quad \mathtt{N} = \mathtt{C<\overline{T}>}$

$CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft \overline{N}> \triangleleft P\ \{\ \ldots\ \ \overline{M}\ \}}$

$\mathtt{<\overline{Y} \triangleleft \overline{Q}>T_0\ m\ (\overline{S}\ \overline{x})\ \{\uparrow e_0;\}} \in \overline{\mathtt{M}}$

$\#(\overline{\mathtt{V}}) = 0$

Let $\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{this} : \mathtt{C<\overline{X}>}$ and $\Delta' = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{Q}}$. By the rules GT-Class and GT-Method, we have $\Delta'; \Gamma \vdash \mathtt{e} \in \mathtt{S_0}$ and $\Delta'; \Gamma \vdash \mathtt{S_0} \mathrel{<:} \mathtt{T_0}$ for some $\mathtt{S_0}$. Since $\Delta \vdash \mathtt{C<\overline{T}>}$ ok, we have $\Delta \vdash \overline{\mathtt{T}} \mathrel{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ by the rule WF-Class. By Lemmas 5.5.3, 5.5.12 and 5.5.6, we have

$$\Delta; [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\Gamma \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\mathtt{e_0} \in [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\mathtt{S_0}$$

and

$$\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\mathtt{S_0} \mathrel{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\mathtt{T_0}.$$

Since $\#(\overline{\mathtt{V}}) = 0$, $mtype(\mathtt{m<>}, \mathtt{N})$ is derived by the rule MT-ClassRaw; moreover, since there is no *unchecked warning*, we have $|\overline{\mathtt{S}}|_{\Delta'} = \overline{\mathtt{S}}$ and $mtype(\mathtt{m<>}, \mathtt{N}) = \overline{\mathtt{S}} \rightarrow |\mathtt{T_0}|_{\Delta'}$. Then, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\Gamma = \Gamma$. Now, we have

$$\Delta; \overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{this} : \mathtt{C<\overline{T}>} \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\mathtt{e_0} \in [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\mathtt{S_0}$$

and

$$\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\mathtt{S_0} \mathrel{<:} |\mathtt{T_0}|_{\Delta'}$$

finishing the case.

Case MB-Super: $\quad \mathtt{N} = \mathtt{C<\overline{T}>}$

$CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft \overline{N}> \triangleleft P\ \{\ \ldots\ \ \overline{M}\ \}}$

$\mathtt{m}$ is not defined in $\overline{\mathtt{M}}$

Follows from the induction hypothesis. Note that $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P}$ also satisfies the property that, for any $\mathtt{D}$ such that $\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P} \mathrel{<:} \mathtt{D}$, $\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P} \mathrel{<:} \mathtt{D<\overline{S}>}$ for some $\overline{\mathtt{S}}$ because we can have $\Delta \vdash \mathtt{N} \mathrel{<:} \mathtt{D<\overline{S}>}$, from which we have $\Delta \vdash \mathtt{N} \mathrel{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P}$ and $\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P} \mathrel{<:} \mathtt{D<\overline{S}>}$. ■

Proof of Theorem 5.5.1: By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e'}$ with a case analysis on the reduction rule used. The cases similar to those in the proof of Theorem 4.5.1.1 are omitted.

Case GR-Invk: $\quad \mathtt{e} = \mathtt{new\ N(\overline{e}).m<\overline{V}>(\overline{d})} \qquad mbody(\mathtt{m<\overline{V}>}, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e_0})$

$\mathtt{e'} = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(\overline{e})/this}]\mathtt{e_0}$

By the rules GT-Invk and GT-New, we have

$\Delta; \Gamma \vdash \mathtt{new\ N(\overline{e})} \in \mathtt{N} \qquad mtype(\mathtt{m}, bound_\Delta(\mathtt{N})) = \mathtt{<\overline{Y} \triangleleft \overline{P}>\overline{U} \rightarrow U}$

$\Delta \vdash \overline{\mathtt{V}}$ ok $\qquad \Delta \vdash \overline{\mathtt{V}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$

$\Delta; \Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$

$\mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U} \qquad \Delta \vdash \mathtt{N}$ ok

for $\mathtt{D}$ such that $\Delta \vdash \mathtt{N} \mathrel{<:} \mathtt{D}$, there exist $\overline{\mathtt{S}'}$ such that $\Delta \vdash \mathtt{N} \mathrel{<:} \mathtt{D<\overline{S'}>}$.

By Lemma 5.5.14, $\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this} : \mathtt{P} \vdash \mathtt{e_0} \in \mathtt{S}$ for some $\mathtt{P}$ and $\mathtt{S}$ such that $\Delta \vdash \mathtt{N} \mathrel{<:} \mathtt{P}$ where $\Delta \vdash \mathtt{P}$ ok, and $\Delta \vdash \mathtt{S} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$ where $\Delta \vdash \mathtt{S}$ ok. Then, by Lemmas 5.5.3 and 5.5.13, $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(\overline{e})/this}]\mathtt{e_0} \in \mathtt{T_0}$ for some $\mathtt{T_0}$ such that $\Delta \vdash \mathtt{T_0} \mathrel{<:} \mathtt{S}$. By the rule S-Trans, we have $\Delta \vdash \mathtt{T_0} \mathrel{<:} \mathtt{T}$. Finally, letting $\mathtt{T'} = \mathtt{T_0}$ finishes the case.

Case GRC-FIELD: $\texttt{e} = \texttt{e}_0\texttt{.f} \qquad \texttt{e}' = \texttt{e}_0'\texttt{.f} \qquad \texttt{e}_0 \longrightarrow \texttt{e}_0'$

By the rule GT-FIELD, we have

$$\begin{array}{l} \Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \\ \mathit{fields}(\mathit{bound}_\Delta(\texttt{T}_0)) = \overline{\texttt{T}}\ \overline{\texttt{f}} \\ \texttt{T} = \texttt{T}_i \end{array}$$

By the induction hypothesis, $\Delta;\Gamma \vdash \texttt{e}_0' \in \texttt{T}_0'$ for some $\texttt{T}_0'$ such that $\Delta \vdash \texttt{T}_0' \mathrel{<:} \texttt{T}_0$. Now we have two subcases:

Subcase: $\texttt{T}_0' = *$

By the rule GT-BOT-FIELD, $\Delta;\Gamma \vdash \texttt{e}_0'\texttt{.f} \in *$.

Subcase: $\texttt{T}_0' \neq *$

By Lemma 5.5.10, $\mathit{fields}(\mathit{bound}_\Delta(\texttt{T}_0')) = \overline{\texttt{T}'}\ \overline{\texttt{g}}$, and for $j \leq \#(\overline{\texttt{f}})$, we have $\texttt{g}_i = \texttt{f}_i$ and $\Delta \vdash \texttt{T}_i' \mathrel{<:} \texttt{T}_i$. Therefore, by the rule GT-FIELD, $\Delta;\Gamma \vdash \texttt{e}_0'\texttt{.f} \in \texttt{T}_i'$. Letting $\texttt{T}' = \texttt{T}_i'$ finishes the case.

Case GRC-INV-RECV: $\texttt{e} = \texttt{e}_0\texttt{.m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \qquad \texttt{e}' = \texttt{e}_0'\texttt{.m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \qquad \texttt{e}_0 \longrightarrow \texttt{e}_0'$

By the rule GT-INVK, we have

$$\begin{array}{ll} \Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 & \mathit{mtype}(\texttt{m}, \mathit{bound}_\Delta(\texttt{T}_0)) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{U} \\ \Delta \vdash \overline{\texttt{V}}\text{ ok} & \Delta \vdash \overline{\texttt{V}} \mathrel{<:} [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{P}} \\ \Delta \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} & \Delta \vdash \overline{\texttt{S}} \mathrel{<:} [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{T}} \\ \texttt{T} = [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U} & \end{array}$$

By the induction hypothesis, $\Delta;\Gamma \vdash \texttt{e}_0' \in \texttt{T}_0'$ for some $\texttt{T}_0'$ such that $\Delta \vdash \texttt{T}_0' \mathrel{<:} \texttt{T}_0$.

Subcase: $\texttt{T}_0' = *$

By the rule GT-BOT-INVK, $\Delta;\Gamma \vdash \texttt{e}_0'\texttt{.m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in *$.

Subcase: $\texttt{T}_0' \neq *$

By Lemma 5.5.11, $\mathit{mtype}(\texttt{m}, \mathit{bound}_\Delta(\texttt{T}_0')) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{V}$ and $\Delta, \overline{\texttt{Y}} \texttt{<:} \overline{\texttt{P}} \vdash \texttt{V} \mathrel{<:} \texttt{U}$. By Lemma 5.5.6, $\Delta \vdash [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{V} \mathrel{<:} [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}$. Then, by the rule GT-INVK, $\Delta;\Gamma \vdash \texttt{e}_0'\texttt{.m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{V}$. Letting $\texttt{T}_0' = [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{V}$ finishes the case. ∎

5.6 Program Evolution and Unchecked Programs

As discussed at the beginning of this chapter, unchecked programs are important when program evolution is taken into account. The goal of program evolution is to augment monomorphic Java classes with type parameters and obtain checked GJ classes. Let us review the process of program evolution:

1. To begin with, a Java program has to be considered a GJ program. After this step, we can discuss program evolution within the framework of GJ. Here, it is desirable for every Java program to be a *checked* GJ program, which behaves exactly the same as before; otherwise, programmers are forced to worry about a possible crash of the program, from the beginning.

2. Programmers pick up a monomorphic class, written in the Java syntax, and upgrade it to a polymorphic version one by one, repeatedly. During this step, the whole program may become unchecked, but it continues to run.

3. Finally, a checked GJ program is obtained.

In this section, we discuss desired properties of Raw FGJ about program evolution and some observed difficulties for the proof; their rigorous treatment are left for future work.

5.6.1 Backward Compatibility

For the first step, it should be guaranteed that an FJ program is also a checked Raw FGJ program and behaves exactly the same. However, it is not quite true if we interpret an FJ program as it is, since, in Raw FGJ, every object instantiation `new C(`$\overline{\texttt{e}}$`)` is unchecked (remember that `C` is raw even if the class `C` does not take type parameters). Thus, to ensure this property, it seems to important to use a trick similar to the one used in FGJ (and GJ)—that is, the notation `C` is the abbreviation of the cooked type `C<>` when the class does not take type parameters. Then, the property could be proved just as in FGJ (Theorem 4.5.2.4). In fact, the current GJ compiler adopts this rule and seems to achieve backward compatibility with Java. The minimal requirement for the rule to achieve backward compatibility would be that `C` is interpreted as `C<>` if the class definition of `C` is written in the Java syntax. Thus, the rule adopted by the GJ compiler regards more occurrences of `C` as `C<>`. However, the GJ rule causes a problem to prove the evolution property discussed below.

5.6.2 Compilation of Raw FGJ

During the iteration of the second step, a program may become unchecked; even so, we expect that it continues to run. We have modeled the half of the story by defining direct semantics of unchecked programs. The other half, connected with erasure semantics, would be formalized as the following two properties: (1) an unchecked Raw FGJ program erases to a well-typed FJ program (like Theorem 4.7.1.1); and (2) semantic equivalence of an unchecked Raw FGJ program and its erasure (like Theorems 4.7.2.2 and 4.7.2.3).

The proof for the first property would be more complicated because of the bottom type, since, in FJ, there is no type corresponding to the bottom type. One solution might be to augment FJ with the bottom type. However, we do not expect that such an extension is essential to prove. It is expected that there is no expression whose type is the bottom type in the type derivation of the class table, if the bottom type appears only in the method body, as is ensured by the sanity conditions. If it is really the case, we will not be bothered by ill-definedness of the absence of the bottom type in FJ.

The semantic equivalence seems hard to prove since the proof technique in the previous chapter depends on well-typedness of expressions: erasure and expansion are not defined for ill-typed expressions.

5.6.3 Evolution Property

In the second step, each class will be replaced with another version. Then, we would like to guarantee the *evolution property* that each replacement does not affect the well-typedness and behavior of the whole program if the replacement is “sensible.”

First of all, it is crucial to have a reasonable definition of sensible replacement. The definition should reflect our intuition behind upgrade of classes without changing the behavior. One reasonable choice of sensible replacements seems to replace a class, written in the FJ syntax, with a well-typed class so that the erasure of the latter is the original one. For example, the class `Pair` in Chapter 2 can be replaced with the class `Pair<X,Y>` in Chapter 4. However, this definition seems too restricted; it is often the case that intuitively reasonable replacement makes the erasure different. Consider a method declaration

```
    Pair copy(Pair p) { return new Pair(p.fst, p.snd); }
```

that uses `Pair<X,Y>` via the raw type `Pair`. We might want to replace it with a new definition

```
 Pair<A,B> copy(Pair<A,B> p) { return new Pair<A,B>(p.fst, p.snd); }
```

but it would not be allowed since the erasure return

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

of the new method body involves synthetic casts. Therefore, we will need a more permissible definition. We may be able to ignore difference by the insertions of synthetic casts, as we did in the proof of correctness of FGJ compilation, since synthetic casts are supposed not to affect the behavior of programs.

Although we do not yet have a rigorous definition of such sensible replacements, we can observe Raw FGJ (and GJ) do not quite satisfy the evolution property, in particular, the preservation of well-typedness. The intuitive reason behind that property is that every occurrence of `C`, where the class `C` is written in the FJ (or Java) syntax, will become a raw type after the class is upgraded to a polymorphic version. However, it is now always the case in the presence of the GJ abbreviation rule mentioned above. Consider the following Raw FGJ classes:

```
class C<X> extends Object {
}
class A extends C<A> {
}
class B extends C<B> {
}

class Pair<X,Y> extends Object {
  X f; Y g;
  Pair(X f, Y g) { this.f = f; this.g = g; }
}

class Factory extends Object {
  Factory() { super(); }
  Pair doublet(C f, C g) { return new Pair(f,g); }
}

class Test extends Object {
  Test() { super(); }
  Pair foo() { return new Factory().doublet(new A(),new B()); }
}
```

The classes `A`, `B`, `C`, and `Pair` have already been evolved; now we are going to replace the class `Factory`. Consider the following new definition of `Factory`:

```
class Factory extends Object {
  Factory() { super(); }
  <X extends C<X>> Pair<X,X> doublet(X f, X g) {
      return new Pair<X,X>(f,g);
  }
}
```

Notice that the new class does not take type parameters, but the method `doublet` is augmented with a type parameter `X` bounded by `C<X>`. The new definition is well typed and erases to the old one. However, the class `Test` is not well typed any more; in particular, the method body of `foo` is ill-typed now since the occurrence of `Factory` in `foo` is considered the abbreviation of `Factory<>`

and the invocation of `doublet` lacks type parameters. Even if we had a GJ-style type parameter inference, it would not help very much since there is no appropriate type parameter for the method invocation. (Notice that there are no type `T` such that `A <: T` and `B <: T` and `T <: C<T>`.)

The abbreviation rule of GJ seems harmful for the evolution property. Since type inference may fail anyway, the new program should not trigger type inference. In the particular example above, if `new Factory()` instantiated the raw type `Factory`, the class `Test` would be well typed, even though it becomes a unchecked program, though. One possible fix for this problem would be to restrict how a cooked type `C<>` is abbreviated to `C` in program texts: our proposed fix is that `C` is the abbreviation of `C<>` only when the class `C` does not take type parameters and no method of `C` takes type parameters. This new rule will still keep the backward compatibility with FJ.

5.7 Summary

This chapter has further extended Featherweight GJ with raw types, a distinctive feature of GJ to make it smooth to upgrade a monomorphic program to a polymorphic version. Since raw types allow programmers to use parametric classes without type parameters, even when a monomorphic part of a program is replaced with its polymorphic variant, its clients still typecheck and the whole program will run.

Programs using raw types are classified into checked and unchecked programs. Although checked programs are supposed to run safely, we have found GJ's type system for raw types is flawed and some compiled program can fail at synthetic casts. We have proved type soundness of a fixed type system. In our formalization of semantics and proof, the key idea was introduction of the bottom type. On the other hand, as for unchecked programs, we have given an informal discussion on relationship with program evolution, and conjectured some desired properties of unchecked programs. The current GJ seems not to satisfy one of those properties.

Chapter 6

Related Work

6.1 Class-Based Object Calculi

6.1.1 Core Languages for Java

There are several known proofs in the literature of type soundness for subsets of Java. In the earliest, Drossopoulou, Eisenbach and Khurshid [DEK99] (using a technique later mechanically checked by Syme [Sym97]) prove soundness for a fairly large subset of sequential Java. Like us, they use a small-step operational semantics, but they avoid the subtleties of "stupid casts" by omitting casting entirely. Nipkow and Oheimb [NvO98] give a mechanically checked proof of soundness for a somewhat larger core language. Their language does include casts, but it is formulated using a "big-step" operational semantics, which sidesteps the stupid cast problem. Flatt, Krishnamurthi, and Felleisen [FKF98a, FKF98b] use a small-step semantics and formalize a language with both assignment and casting. Their system is somewhat larger than ours (the syntax, typing, and operational semantics rules take perhaps three times the space), and the soundness proof, though correspondingly longer, is of similar complexity. Their published proof of subject reduction in the earlier version is slightly flawed—the case that motivated our introduction of stupid casts is not handled properly—but the problem can be repaired by applying the same refinement we have used here.

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

6.1.2 Other Class-Based Object Calculi

The literature on foundations of object-oriented languages contains many papers formalizing class-based object-oriented languages, either taking classes as primitive (e.g., [Wan89, Bru94, BPSM99, BPS99]) or translating classes into lower-level mechanisms (e.g., [FM98, BF98, AC96, PT94]. Some of these systems (e.g. [PT94, Bru94]) include generic classes and methods, but only in fairly simple forms.

6.2 Generic Extensions of Java

A number of extensions of Java with parametric classes and methods have been proposed by various groups, including the language of Agesen, Freund, and Mitchell [AFM97]; PolyJ, by Myers, Bank, and Liskov [MBL97]; Pizza, by Odersky and Wadler [OW97]; GJ, by Bracha, Oderksy, Stoutamire, and Wadler [BOSW98b]; and NextGen, by Cartwright and Steele [CS98]. While all these languages are believed to be type-safe, our study of FGJ is the first to give rigorous proof of soundness for a generic extension of Java. We have used GJ as the basis for our generic extension, but similar techniques should apply to the forms of genericity found in the rest of these languages.

Recently, Duggan [Dug99] has proposed a technique to translate monomorphic classes to parametric classes by inference of type argument information. He has also defined a polymorphic extension of Java, slightly less expressive than GJ (for example, polymorphic methods are not allowed and a subclass must have the same number of type arguments as its superclass). Type soundness theorem of the language is mentioned but stupid cast problem is not taken into account.

As an alternative to parametric classes, virtual types [Tho97] have been proposed for an extension to Java. Since the original proposal was not statically type-safe, several type-safe variants have been proposed [Tor98, BOW98, BV99] later. Pierce and the author [IP99] have shown examples of an encoding of objects and classes involving virtual types in an extension of System $F^{\omega}_{\leq}$. However, since the target language is not class-based, comparison of the variants are done via encoding. It would be possible to formalize a variety of virtual types as an extension of FJ and obtain more direct accounts of the proposed variants.

6.3 Nested Declarations in Object-Oriented Languages

Beta [MMPN93] 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 Java-style inner classes. First, in Beta, inner classes are 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* [MMP89], in the sense that it depends on *run-time type* of the enclosing instance which constructor is invoked. A constructor invocation `e.new E(`$\overline{\texttt{e}}$`)` 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 [Mad99]. The algorithm is very close to the rules presented in Section 3.8, in the sense that the search order is the same as ours, although the presence of virtual classes complicates the algorithm.

Block expressions of Smalltalk [GR83] provides block structure of *objects* rather than classes. Since block expressions can offer only one method and cannot inherit from another class, their expressiveness is limited.

6.4 Microsoft's Delegates

Microsoft has proposed *delegates* [Mic99] 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.

6.5 Object Closure Conversion

Recently, Glew [Gle99a] 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. On the other hand, he has dealt with subtlety in the proof of correctness of conversion in the presence of a call-by-value reduction strategy.

Chapter 7

Conclusions

7.1 Summary of the Thesis

In this thesis, we have studied two advanced class mechanisms, Java-style inner classes and GJ-style parametric classes. We have designed a small class-based object-oriented language, FJ, and formalized the core features of those mechanisms on top of FJ. Several properties such as type soundness and correctness of the compilation schemes have been proved. The main contributions are summarized as follows:

- The design of FJ. FJ has been designed to be a sublanguage of Java. We have dropped as many features as possible and left the essential features required to understand the essence of the extensions. As a result, FJ is much smaller than any other existing core languages for Java; this extreme simplicity was important in order to make proofs for complex extensions tractable.

  We have also identified the stupid cast problem in the proof of subject reduction for FJ with typecasts. Although it is a rather small technicality, it is worth pointing out. In fact, it has been overlooked in other formalizations of the core languages for Java.

- Formalization of the core of inner classes. One of the main difficulties in understanding inner class lies in interaction with inheritance, which complicates semantics and scoping rules of the language. We have clarified the key notion of enclosing instance, which roughly corresponds to an environment under which a method is executed. We have extended FJ with inner classes to FJI and defined its semantics in the two styles: the direct style, defined by a reduction relation between FJI expressions, provides a model of direct execution of the program; and the translation style, defined by translation from FJI to FJ, captures the essence of the current compilation scheme, given by the official specification. Type soundness and equivalence of the two styles have been proved. The equivalence guarantees correctness of the current compilation scheme.

  As for scoping rules, we have defined formal rules of elaboration from user programs, where receivers ($\mathtt{e}_0$ of $\mathtt{e}_0$`.f` or $\mathtt{e}_0$`.m(`$\overline{\mathtt{e}}$`)`) may be omitted when they are `this` or `C.this`, to FJI programs, where they must be explicit.

- Formalization of the core of GJ. We have augmented FJ with type parameterization and obtained Featherweight GJ. As we did for FJI, we have defined the direct and translation styles of semantics and proved type soundness for each of them. The most challenging property was correctness of the translation style with respect to the direct style. Naive correspondence between FGJ reduction steps and FJ reduction steps did not hold. Nevertheless, we have proved a slightly weaker property that the execution results in both semantics correspond to each other.

- We have further extended FGJ with raw types to Raw FGJ. Raw FGJ programs are classified into checked and unchecked programs, where the former is guaranteed to be safe, whereas the latter is not. The main technical result is a proof of subject reduction property for checked Raw FGJ programs. Surprisingly, the current typing rules of GJ has turned out to be flawed; we have proved it for a fixed type system. In our definition of semantics and proof of subject reduction, the bottom type has played an important role.

  We have also discussed roles of raw types and unchecked programs in evolution from legacy monomorphic programs to its polymorphic variant. Some desired properties have been conjectured. Once again, we have found the current design of GJ does not satisfy one of those important properties.

To our knowledge, this work gives for the first time a direct style of formal semantics of both inner classes and parametric classes (as an extension for Java); in particular, the use of the bottom type in the operational semantics of Raw FGJ seems new. The direct style should be deemed important since semantics depending on translation makes reasoning about program behavior considerably hard.

Besides the technical contributions above, we have found several bugs in a few Java compilers and the GJ compiler (though most of them are not critical and have already been fixed in the currently available versions). Moreover, some significant underspecification in the official document of inner classes and bugs in the GJ design has been revealed.

In conclusion, although our approach, formalization of the core features with a small core language, is far from completeness, it is really useful not only for confirming expected properties but also for discovering flaws that even the designers did not know.

7.2 Future Work

Future work mainly consists of two directions: modeling of features left out from this work, and investigating connection between FJ (including the extensions) and more primitive calculi.

7.2.1 Formalization of Other Aspects of Inner Classes

Local Classes and Anonymous Classes

In Java, programmers are allowed to declare inner classes not only as members of a class but also in a method body as local classes, declared in a block just as local variables, or anonymous classes, considered a combination of a local class declaration and its instantiation. Formalizing these classes should face another subtle rule, found in the specification, that method formal parameters used in local/anonymous classes should be marked `final`, which prohibits assignment to the parameters. This rule is derived from the current compilation scheme; method parameters referred to in a local/anonymous class are translated to fields of the compiled class and they are initialized with the values of the method parameters. The annotation `final` is required to keep consistency between the values of the method parameters of the enclosing class and those of the fields of the local/anonymous class.

Moreover, anonymous classes raise an interesting question on the type system, namely, "what would be the type of instances of an anonymous class?" On the one hand, it seems reasonable to introduce structural types for anonymous classes. On the other hand, the introduction of structural types seems incompatible with translation semantics in the sense that different anonymous class types with the same structure may be mapped into the same types.

Inner Classes and Access Control

Combination of inner classes and access control is subtle especially in the presence of translation semantics. It has been already pointed out that the current compilation scheme may compromise access control annotation in the original program [Aba98, BP] since, when an inner class accesses `private` members of the enclosing class, those members are translated into members accessible from other classes of the same package. One solution has been proposed by Bhowmik and Pugh [BP]. In their translation, the inner class and its enclosing class will share a secret key to access private members; since the secret is generated at run-time, it is unforgeable. Formalizing their translation and showing its correctness should be interesting. To express such information sharing, assignments seem essential.

7.2.2 Formalization of Other Aspects of GJ

Raw Types

As discussed in Section 5.6, the properties about program evolution and unchecked programs should be proved. The main difficulties in the proof would lie in erasure semantics and the formal definition of evolution.

Since we do not have the bottom type in FJ, the erasure of the bottom type is not clear. It seems to require some workaround in a proof of the property that an unchecked program compiles to a well-typed FJ program. One possible solution is to extend FJ with the bottom type. However, we do not expect that such an extension is essential to prove it; it seems that, if a program is well typed, there is no type judgment that gives an expression the bottom type and, hence, we do not have to consider the erasure of the bottom type. This conjecture also affects our proof of subject reduction: we may not need the special typing rules for the bottom types.

The properties about program evolution seem much harder to proof because even the formal definition of program evolution is not clear yet.

Type Inference

The GJ compiler supports partial type inference for polymorphic method invocations. Basically, it is an instance of local type inference scheme [PT98] in the sense that missing type parameters are determined only by the types of the receiver and the actual arguments, not depending on the context of the invocation. Hence, a similar formalization seems possible.

However, the type system for type inference uses the bottom type, making argument of soundness of type inference very subtle. For example, the type `C<*>` is considered subtype of `C<A>` for any type `A`. In general, this kind of covariant subtyping should not be allowed for the type system to be sound. (Java's subtyping rule for array is such an example: in Java, `String[]` is subtype of `Object[]`, forcing run-time checks for insertions of a value into an array.) Another subtle rule is that, if `*` is inferred for an type parameter, the type parameter may not appear more than once in the result type. For example, if the result type of an method is `C<X,X>`, the bottom type cannot be a choice for `X`. We expect that these rules are significant in the presence of side-effects; therefore, the base language may have to be extended with assignments to capture subtleties of GJ type inference.

Combination of Inner Classes and Raw Types

It might be worth investigating a combination of inner classes and raw types. It is not immediately clear what a qualified type name consisting of cooked types and raw types since an inner class may refer to type parameters derived from enclosing classes. Does the type `C.D<A>` make sense when `C` is raw? Or, what does `C<A>.D` mean if `D` is raw? It seems that the current GJ compiler ignores type parameters when one of simple names in a qualified name is raw. For example, the following GJ class definitions

```
class A extends Object{
    void m() { return; }
}

class C<X extends Object> extends Object {
    class D<Y extends Object> extends Object {
        X f;
        D(X f) { this.f = f; }
    }
    void test (C<A>.D x) {
        x.f.m();
    }
}
```

are not accepted by the compiler, saying that method `m` not found in class `Object`, although one might expect that `x.f` is given type `A`, the type parameter for `X`.

7.2.3 Connection between FJ and Object-Based Calculi

As discussed in Chapter 1, the first step of our approach to this work was to create an appropriate setting to concentrate on study of the essence of inner classes and GJ. Hence, we did not choose existing object-based calculi such as Abadi-Cardelli's calculi [AC96] mainly because even basic class mechanisms seemed hard to express in such low-level calculi and, as a result, the essential points would be blurred.

It is interesting to fill the gap between FJ and more primitive object-based calculi: FJ may be encoded in a lower-level calculus. The biggest difference lies in their type systems; FJ is based on name-based subtyping and most of the calculi used for theoretical studies of object-oriented languages are based on structural subtyping. Above all, encoding of typecasts seems challenging. Although we can find a similar construct such as the so-called "typecase" construct [ACPR95, HM95], its detail is significantly different from typecasts because of name-based subtyping. Glew's recent work [Gle99b] on a language with type dispatch for generative and hierarchical types would be a good starting point.

7.2.4 Primitive Calculus for Compilation and Raw Types

When we design FJ, we dropped as many features as possible from Java. The simplicity was really useful to prove formal properties of the languages such as type soundness. However, some of the problems such as compilation from FGJ to FJ and raw types are not necessarily inherent in object-oriented programming and FJ seems still too big to study them. We could think of similar problems in, say, functional programming. For example, it might be possible to mimic programming with a polymorphic type system by a monomorphic language with subtyping and a dynamic typechecking construct similar to typecasts of Java.

Therefore, it will be worth investigating the essence of raw types and compilation from a polymorphic language to a monomorphic language with subtyping and typecasts in a simpler setting. Then, we will not be bothered by details of object-oriented features.

Bibliography

[Aba98] 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).

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Preliminary version in Proceedings of the ACM SIGPLAN Workshop on ML and its Applications, June 1992.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 49–65, Atlanta, GA, October 1997.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) and European Conference on Object-Oriented Programming (ECOOP)*, pages 303–311, Ottawa, ON Canada, October 1990. ACM Press, New York, NY, USA. Published as SIGPLAN Notices, volume 25, number 10.

[BF98] Viviana Bono and Kathleen Fisher. An imperative first-order calculus with object extension. In Rachid Guerraoui, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 462–497, Brussels, Belgium, 1998. Springer-Verlag.

[BOSW98a] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ specification. Manuscript, 1998. Available through `http://cm.bell-labs.com/cm/cs/who/wadler/gj/`.

[BOSW98b] 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, *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549, Brussels, Belgium, July 1998. Springer-Verlag.

[BP] 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`.

[BPS99] Viviana Bono, Amit J. Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66, Lisbon, Portugal, June 1999. Springer-Verlag.

[BPSM99] Viviana Bono, Amit J. Patel, Vitaly Shmatikov, and John C. Mitchell. A core calculus of classes and objects. In *Proceedings of the Fifteenth Conference on the Mathematical Foundations of Programming Semantics (MFPS XV)*, volume 20 of *Electronic Notes in Theoretical Computer Science*, New Orleans, LA, April 1999. Elsevier. Available through `http://www.elsevier.nl/locate/entcs/volume20.html`.

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. Preliminary version in POPL 1993, under the title "Safe type checking in a statically typed object-oriented programming language".

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 952, pages 27–51, Aarhus, Denmark, August 1995. Springer-Verlag.

[BV99] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Proceedings of the Fifteenth Conference on the Mathematical Foundations of Programming Semantics (MFPS XV)*, volume 20 of *Electronic Notes in Theoretical Computer Science*, New Orleans, LA, April 1999. Elsevier. Available through `http://www.elsevier.nl/locate/entcs/volume20.html`.

[Car90] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995. Short version in *Principles of Programming Languages (POPL)*, January 1995.

[CCH$^+$89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[Cha97] Craig Chambers. The Cecil language: Specification and rationale. Available through `http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html`, March 1997.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[CL91] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

[CL97] Patrick Chan and Rosanna Lee. *The Java Class Libraries.* Addison-Wesley, Reading, MA, second edition, October 1997. 2 volumes.

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. Preliminary version in TACS '91 (Sendai, Japan, pp. 750–770).

[Com94] Adriana B. Compagnoni. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F^\omega_\wedge$ is decidable".

[CS98] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, SIGPLAN Notices volume 33 number 10, pages 201–215, Vancouver, BC, October 1998. ACM.

[DEK99] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 7(1):3–24, 1999. Preliminary version in ECOOP '97.

[DG87] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. In Jean Bezivin et al., editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 276 of *Lecture Notes in Computer Science*, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[Dug99] Dominic Duggan. Modular type-based reverse engineering of parameterized types in Java code. In Linda M. Northrop, editor, *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, volume 34, number 10, pages 97–113, Denver, CO, October 1999. ACM Press.

[FF98] Matthias Felleisen and Daniel P. Friedman. *A little Java, A few Patterns.* The MIT Press, Cambridge, MA, 1998.

[FKF98a] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 171–183, San Diego, CA, January 1998. ACM Press.

[FKF98b] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR97-293, Computer Science Department, Rice University, February 1998. Corrected version appeared in June, 1999.

[FM98] Kathleen Fisher and John C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–25, 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[Gle99a] 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/volue26.html`.

[Gle99b] Neal Glew. Type dispatch for named hierarchical types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, volume 34, number 9, pages 172–182, Paris, France, September 1999. ACM Press.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[HJW$^+$92] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partian, and J. Peterson. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 130–141, San Francisco, CA, January 1995.

[IP99] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In Rachid Guerraoui, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 161–185, Lisbon, Portugal, June 1999. Springer-Verlag.

[Jav97] JavaSoft. Inner classes specification, February 1997. Available through `http://java.sun.com/products/JDK/1.1/`.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, second edition, 1999. 496 pages.

[Mad99] Ole Lehrmann Madsen. Semantic analysis of virtual classes and nested classes. In Linda M. Northrop, editor, *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, volume 34, number 10, pages 114–131, Denver, CO, October 1999. ACM Press.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 132–145, Paris, France, January 1997.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Mic99] Microsoft. Microsoft Java SDK 3.2 documentation. Available through `http://www.microsoft.com/Java/sdk/32/`, 1999.

[MMP89] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 397–406, New Orleans, LA, 1989.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proceedings of 12th ACM Symposium on Principles of Programming Languages,* 1985.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[NvO98] Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe — definitely. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 161–170, San Diego, CA, January 1998. ACM Press.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 146–159, Paris, France, January 1997.

[Pie97] Benjamin C. Pierce. Bounded quantification with bottom. Technical Report 492, Computer Science Department, Indiana University, 1997.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[PS94] Benjamin Pierce and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. Preliminary version in *Principles of Programming Languages (POPL)*, 1993.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 252–265, San Diego, CA, 1998. Full version available as Indiana University CSCI technical report #493.

[RV98] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998. Preliminary version appeared in *ACM Symposium on Principles of Programming Languages (POPL97)*, pages 40–53, Paris, France, January 1997.

[SL94] Alexander A. Stepanov and Minsuk Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, third edition, 1997.

[Sym97] Don Syme. Proving Java type soundness. Technical Report 427, Computer Laboratory, University of Cambridge, June 1997.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471, Jyväskylä, Finland, June 1997. Springer-Verlag.

[Tor98] Mads Torgersen. Virtual types are statically safe. In *Proceedings of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, San Diego, CA, January 1998. Available through `http://www.cs.williams.edu/~kim/FOOL/FOOL5.html`.

[US91] David Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, 1991. Preliminary version appeared in *Proceedings of OOPSLA*, 1987, pages 227–241.

[Wan89] Mitchell Wand. Type inference for objects with instance variables and inheritance. Technical Report NU-CCS-89-2, College of Computer Science, Northeastern University, February 1989. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.