JFP **18** (3): 285–331, 2008.
doi:10.1017/S0956796807006405 First published online 6 July 2007 Printed in the United Kingdom

*Lightweight family polymorphism**

CHIERI SAITO and ATSUSHI IGARASHI
Kyoto University, Japan
(*e-mail:* {saito,igarashi}@kuis.kyoto-u.ac.jp)

MIRKO VIROLI
Alma Mater Studiorum – Università di Bologna a Cesena, Italy
(*e-mail:* mirko.viroli@unibo.it)

Abstract

Family polymorphism has been proposed for object-oriented languages as a solution to supporting reusable yet type-safe mutually recursive classes. A key idea of family polymorphism is the notion of families, which are used to group mutually recursive classes. In the original proposal, due to the design decision that families are represented by objects, dependent types had to be introduced, resulting in a rather complex type system. In this article, we propose a simpler solution of *lightweight* family polymorphism, based on the idea that families are represented by classes rather than by objects. This change makes the type system significantly simpler without losing much expressive power of the language. Moreover, "family-polymorphic" methods now take a form of parametric methods; thus, it is easy to apply method type argument inference as in Java 5.0. To rigorously show that our approach is safe, we formalize the set of language features on top of Featherweight Java and prove that the type system is sound. An algorithm for type inference for family-polymorphic method invocations is also formalized and proved to be correct. Finally, a formal translation by erasure to Featherweight Java is presented; it is proved to preserve typing and execution results, showing that our new language features can be implemented in Java by simply extending the compiler.

1 Introduction

1.1 Mismatch Between Mutually Recursive Classes and Simple Inheritance

It is fairly well-known that, in object-oriented languages with simple name-based type systems such as C++ or Java, mutually recursive class definitions and extension by inheritance do not fit very well. Since classes are usually closed entities in a program, mutually recursive classes here really mean a set of classes whose method *signatures* refer to each other by their *names*. Thus, different sets of mutually recursive classes necessarily have different signatures, even though their structures are similar. On the other hand, in C++ or Java, it is not allowed to inherit a method from the superclass with a different signature (in fact, it is not safe, in general, to allow covariant change

* A preliminary summary appeared in the proceedings of the third Asian Symposium on Programming Languages and Systems (APLAS2005), volume 3780 of Lecture Notes in Computer Science, Tsukuba, Japan, November 2005, Springer-Verlag, pp. 161–177.

of method parameter types). As a result, deriving subclasses of mutually recursive classes yields another set of classes that do *not* refer to each other and, worse, this mismatch is often resolved by typecasting, which is a potentially unsafe operation (not to say unsafe, an exception may be raised). A lot of studies (Bruce *et al.* 1998; Bruce & Vanderwaart 1999; Thorup & Torgersen 1999; Ernst 2001; Bruce 2003; Odersky *et al.* 2003; Jolly *et al.* 2004; Nystrom *et al.* 2004; Ernst *et al.* 2006) have been recently done to develop a language mechanism with a static type system that allows "right" extension of mutually recursive classes without resorting to typecasting or other unsafe features.

1.2 Family polymorphism

Erik Ernst (2001) has recently coined the term "family polymorphism" for a particular programming style, using virtual classes (Madsen & Møller-Pedersen 1989) of `gbeta` (Ernst 1999), and applied it to solve the above-mentioned problem of mutually recursive classes.

In his proposal, mutually recursive classes are programmed as nested class members of another (top-level) class. These member classes are virtual in the same sense as virtual methods—a reference to a class member is resolved at run-time. Thus, the meaning of mutual references to class names will change when a subclass of the enclosing class is derived and these member classes are inherited. This late-binding of class names makes it possible to reuse implementation without the mismatch described above. The term "family" refers to such a set of mutually recursive classes grouped inside another class. He has also shown how a method that can uniformly work for different families can be written in a safe way: such "family-polymorphic" methods take as arguments not only instances of mutually recursive classes but also the identity of the family that they belong to, so that semantical analysis (or a static type checker) can check if these instances really belong to the same family.

Although family polymorphism seems very powerful, we feel that there may be a simpler solution to the present problem. In particular, in `gbeta`, nested classes really are members (or, more precisely, attributes) of an *object*, so types for mutually recursive classes include as part object references, which serve as identifiers of families. As a result, the semantical analysis of `gbeta` essentially involves a dependent type system (Odersky *et al.* 2003; Aspinall & Hofmann 2005; Ernst *et al.* 2006), which is rather complex (especially in the presence of side effects).

1.3 Contributions of this article

We identify a minimal, *lightweight* set of language features to solve the problem of typing mutually recursive classes, rather than introduce a new advanced mechanism. As done elsewhere (Jolly *et al.* 2004), we adopt what we call the "classes-as-families" principle, in which families are identified with classes, which are static entities, rather than with objects, which are dynamic. Although it loses some expressive power, programming extensible mutually recursive classes is still possible. Moreover,

for type safety reasons, we take the approach that inheritance is not subtyping and regard all types for nested classes as exact, in the sense of Bruce *et al.* (1998), while admitting subtyping for top-level types. These decisions simplify the type system a lot, making much easier a type soundness argument and application to practical languages such as Java. As a by-product, we can view family-polymorphic methods as a kind of parametric methods found, e.g. in Java generics and find that the technique of type argument synthesis as in GJ and Java 5.0 (Bracha *et al.* 1998; Odersky 2002) can be extended to our proposal as well.

Other technical contributions of the present article can be summarized as follows:

- simplification of the type system for family polymorphism with the support for family-polymorphic methods;
- a rigorous discussion of the safety issues by the development of a formal model called .FJ (read "dot FJ") of lightweight family polymorphism, on top of Featherweight Java (FJ) (Igarashi *et al.* 2001) with a correctness theorem of the type system;
- an algorithm of type argument synthesis for family-polymorphic methods and its correctness theorem; and
- a formal translation of .FJ to FJ, which is proved to preserve typing and semantics.

This article adds the formal translation and its correctness proof as a new contribution to the conference version (Igarashi *et al.* 2005).

We would like to emphasize that one of our main aims is to identify a minimal set of features to describe typical examples of family polymorphism, rather than to solve a wider range of problems, as tackled in the literature (Ernst 1999; Odersky *et al.* 2003; Jolly *et al.* 2004). As a result, our language has several restrictions not found in those proposals. Most restrictions, however, have been relaxed in succeeding work (Igarashi & Viroli 2007), which will be mentioned in Section 6.

1.4 The rest of this article

After Section 2 presents the overview of our language constructs through the standard example of graphs, Section 3 formalizes those mechanisms as the calculus .FJ and discusses its type safety. Then, Section 4 formalizes the translation of .FJ into FJ as a model of erasure compilation of lightweight family polymorphism to Java with correctness theorems. After Section 5, which discusses related work, Section 6 concludes.

2 Programming lightweight family polymorphism

We start by informally describing the main aspects of the language constructs we study in this article, used to support lightweight family polymorphism. To this end, we consider as a reference the example in Ernst (2001), properly adapted to fit our "classes-as-families" principle.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

This example features a *family* (or group) `Graph`, containing the classes `Node` and `Edge`, which are the *members* of the family, and are used as components to build graph instances. As typically happens, members of the same family can mutually refer to each other: in our example, for instance, each node holds a reference to (an array of) connected edges, while each edge holds references to its source and destination nodes. Now suppose that we are interested in defining a new family `ColorWeightGraph`, used to define graphs with colored nodes and weighted edges—nodes and edges with the new fields called `color` and `weight`, respectively—with the property that the weight of an edge depends on the color of its source and destination nodes. Note that in this way the members of the family `Graph` are not compatible with those of family `ColorWeightGraph` in the sense that an edge of a `ColorWeightGraph` cannot be used in a plain `Graph`. Nevertheless, to achieve code reuse, we would like to define the family `ColorWeightGraph` as an extension of the family `Graph`, and declare a member `Node` that automatically inherits all the attributes (fields and methods) of `Node` in `Graph`, and similarly for member `Edge`. Moreover, as advocated by the family polymorphism idea, we would like classes `Node` and `Edge` in `ColorWeightGraph` to mutually refer to each other automatically, as opposed to those solutions exploiting simple inheritance where class `Node` of `ColorWeightGraph` would refer to `Edge` of `Graph`—thus requiring extensive uses of typecasts.

2.1 Nested classes, relative path types, and extension of families

This graph example can be programmed using our lightweight family polymorphism solution as reported in Figure 1, whose code adheres to a Java-like syntax—which is also the basis for the syntax of the calculus .FJ we introduce in Section 3.

The first idea is to represent families as (top-level) classes, and their members as nested classes. Note that in particular we relied on the syntax of Java `static` member classes, which provide a grouping mechanism suitable to define a family. In spite of this similarity, however, we shall give a different semantics to those member classes, in order to support family polymorphism. The types of nodes and edges of class (family) `Graph` are denoted by notations `Graph.Node` and `Graph.Edge`, which we call *absolute path types*. Although such types are useful outside the family to declare variables and to create instances of such member classes, we do not use them to specify mutual references of family members. The notations `.Node` and `.Edge` are instead introduced for this purpose, meaning "member `Node` in the current family" and "member `Edge` in the current family," respectively. We call such types *relative path types*. A similar distinction between absolute and relative is found, for example, in UNIX file systems.

The importance of relative path types becomes clear when introducing the concept of family extension. To define a new family `ColorWeightGraph`, the new class `ColorWeightGraph` is declared to extend `Graph` and provide the member classes `Node` and `Edge`. Such new members, identified outside their family by the absolute path types `ColorWeightGraph.Node` and `ColorWeightGraph.Edge`, will inherit all the attributes of classes `Graph.Node` and `Graph.Edge`, respectively. In particular,

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

```
class Graph {
    static class Node {
        .Edge[] es=new .Edge[10]; int i=0;
        void add(.Edge e) { es[i++] = e; }}
    static class Edge {
        .Node src, dst;
        void connect(.Node s, .Node d) {
            src = s; dst = d; s.add(this); d.add(this);
        }
    }
}
class ColorWeightGraph extends Graph {
    static class Node { Color color; }
    static class Edge {
        int weight;
        void connect(.Node s, .Node d) {
            weight = colorToWeight(s.color, d.color);
            super.connect(s, d);
        }
    }
}
```

```
Graph.Edge e; Graph.Node n1, n2;
ColorWeightGraph.Edge we; ColorWeightGraph.Node cn1, cn2;
e.connect(n1, n2);    // 1: OK
we.connect(cn1, cn2); // 2: OK
we.connect(n1, cn2);  // 3: compile-time error
e.connect(n1, cn2);   // 4: compile-time error
```

```
<G extends Graph>
  void connectAll(G.Edge[] es, G.Node n1, G.Node n2){
    for (int i: es) es[i].connect(n1,n2);
}

Graph.Edge[] ges;            Graph.Node gn1,gn2;
ColorWeightGraph.Edge[] ces; ColorWeightGraph.Node cn1,cn2;
connectAll(ges, gn1, gn2);   // G as Graph
connectAll(ces, cn1, cn2);   // G as ColorWeightGraph
connectAll(ces, gn1, gn2);   // compile-time error
```

Fig. 1. `Graph` and `ColorWeightGraph` classes.

`ColorWeightGraph.Edge` will inherit method `connect()` from `Graph.Edge`, and can therefore override it as shown in the reference code, and even redirect calls by the invocation `super.connect()`. However, since `connect()` is declared to accept two arguments of relative path type `.Node`, it will accept a `Graph.Node` when invoked on a `Graph.Edge`, and a `ColorWeightGraph.Node` when invoked on a `ColorWeightGraph.Edge`. Notice that relative path types are essential to realize

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

family polymorphism, as they guarantee members of the extended family to mutually refer to each other, and not to refer to a different (i.e., super) family.

2.2 Member class inheritance is not subtyping

This covariance schema for relative path types—they change as we move from a family to a subfamily—resembles and extends the construct of *ThisType* (Bruce & Foster 2004), used to make method signatures of self-referencing classes change covariantly through inheritance hierarchies. As well known, however, such a covariance schema prevents inheritance and substitutability from correctly working together as happens in most of common object-oriented languages. In particular, when a relative path type is used as an argument type to a method in a family member, as in method `connect()` of class `Edge`, they prevent its instances from being substituted for those in the superfamily, even though the proper inheritance relation is supported. The following code fragment reveals this problem:

```
// If ColorWeightGraph.Edge were substitutable for Graph.Edge
Graph.Edge e = new ColorWeightGraph.Edge();
Graph.Node n1 = new Graph.Node();
Graph.Node n2 = new Graph.Node();
e.connect(n1,n2);  // Unsafe!!
```

If class `ColorWeightGraph.Edge` could be substituted for `Graph.Edge`, then `connect()` would be invoked on the object (`e`) of `ColorWeightGraph.Edge` with the objects (`n1` and `n2`) of `Graph.Node` as arguments. This invocation would lead to an attempt to access field `color` on `n1` and `n2`, which do *not* have such a field!

To prevent this form of unsoundness, our lightweight family polymorphism solution disallows such substitutability by adopting an "inheritance-without-subtyping" approach for family members. (Subtyping between top-level classes is retained as usual.) Applied to our graph example, it means that while `ColorWeightGraph.Node` inherits all the attributes of `Graph.Node` (for `ColorWeightGraph` extends `Graph`), `ColorWeightGraph.Node` is *not* a subtype of `Graph.Node`. In other words, `Graph.Node` and `Graph.Edge` (as well as `ColorWeightGraph.Node` and `ColorWeightGraph.Edge`) are exact types (Bruce *et al.* 1998). As a result, we can correctly typecheck the invocation of methods in member classes. In the client code in the middle of Figure 1, the first two invocations are correct as node arguments belong to the same family of the receiver edge, but the third and fourth are (statically) rejected, as we are passing as an argument a node belonging to a family different from the receiver edge: in other words, `Graph.Node` and `ColorWeightGraph.Node` are not in the subtype relation.

2.3 Family-polymorphic methods as parametric methods

To fully exploit the benefits of family polymorphism, it should be possible to write so-called *family-polymorphic methods*—methods that can work uniformly over different families. As an example, we consider the method `connectAll()` that takes as input an array of edges and two nodes of *any* family and connects each edge

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

to the two nodes. In our language this is realized through parametric methods as shown at the bottom of Figure 1. Method `connectAll()` is defined as parametric in a type `G`—which represents the family used for each invocation—with upper-bound `Graph` and correspondingly the arguments are of type `G.Edge[]`, `G.Node` and `G.Node`, respectively. As a result, in the first invocation of the example code, by passing edges and nodes of family `Graph`, the compiler would infer the type `Graph` for `G`, and similarly in the second invocation infers `ColorWeightGraph`. Finally, in the third invocation no type can be inferred for `G`, since types `G.Edge` and `G.Node` do not match `ColorWeightGraph.Edge` and `Graph.Node`, respectively, for any `G`.

It may be worth noting that we do not allow relative path types to appear directly in a top-level class: for instance, `.Node` cannot appear in `Graph` or `ColorWeightGraph`. This is because allowing both subtyping, which is used to realize family-polymorphic methods, and relative path types in a top-level class would also lead to unsoundness, as the following code fragment, which is similar to the previous one, shows:

```
// If Graph had a field f of type .Edge
Graph g = new ColorWeightGraph();
Graph.Edge e = g.f;
Graph.Node n1 = new Graph.Node();
Graph.Node n2 = new Graph.Node();
e.connect(n1,n2);  // Unsafe!!
```

3 .FJ: A formal model of lightweight family polymorphism

In this section, we formalize the ideas described in the previous section, namely, nested classes with simultaneous extension, relative path types, and family-polymorphic methods. This is realized through a small calculus named .FJ based on Featherweight Java (Igarashi *et al.* 2001), a functional core calculus of class-based object-oriented languages. After formally defining the syntax (Section 3.1), type system (Sections 3.2 and 3.3), and operational semantics (Section 3.4) of .FJ, we show a type soundness result (Section 3.5). Finally, we present an algorithm of type inference for polymorphic methods and show its soundness (Section 3.6).

For simplicity, we deal with only a single level of class nesting, as opposed to Java, which allows arbitrary levels of nesting. We believe that, for programming family polymorphism, little expressiveness is lost by this restriction. In succeeding work (Igarashi & Viroli 2007), this restriction is lifted and arbitrarily deep nesting is allowed. Although they are easy to add, typecasts, which appear in Featherweight Java and are essential to discuss erasure compilation of generics, are dropped since one of our aims here is to show that programming as in the previous section is possible without typecasts. In .FJ, every parametric method invocation has to provide its type arguments—type inference will be discussed in Section 3.6. Method invocation on `super` is also omitted since directly formalizing `super` would require several global changes to the calculus, because `super` invocation is not virtual (Flatt *et al.* 1998) and, more importantly, it does not really pose a new typing challenge. Invocations on `super` work for much the same reason for invocations of inherited

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

P,Q	::=	C | X	family names
A,B	::=	C | C.C	absolute class names
S,T,U	::=	P | P.C | .C	types
L	::=	class C ◁ C {$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; K $\overline{\texttt{M}}$ $\overline{\texttt{N}}$}	top-level class declarations
K	::=	C($\overline{\texttt{T}}$ $\overline{\texttt{f}}$){super($\overline{\texttt{f}}$); this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;}	constructor declarations
M	::=	<$\overline{\texttt{X}}$ ◁ $\overline{\texttt{C}}$>T m($\overline{\texttt{T}}$ $\overline{\texttt{x}}$){ return e; }	method declarations
N	::=	class C {$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; K $\overline{\texttt{M}}$}	nested class declarations
d,e	::=	x | e.f | e.<$\overline{\texttt{P}}$>m($\overline{\texttt{e}}$) | new A($\overline{\texttt{e}}$)	expressions
v	::=	new A($\overline{\texttt{v}}$)	values

Fig. 2. .FJ: Syntax.

methods on this to work. One may criticize that the formal model is functional unlike the models presented elsewhere (Odersky *et al.* 2003; Jolly *et al.* 2004; Nystrom *et al.* 2004; Ernst *et al.* 2006) and doubt the safety in the presence of imperative features such as assignments. We believe that a functional model is sufficient to catch most of the typing issues and that imperative features do not harm to type safety (we do not claim that type soundness of an imperative model *has been* proved, of course).

3.1 Syntax

The abstract syntax of top-level/nested class declarations, constructor declarations, method declarations, and expressions of .FJ is given in Figure 2. Here, the metavariables C, D, and E range over (simple) class names; X and Y range over type variable names; f and g range over field names; m ranges over method names; x and y range over variables.

We put an overline for a possibly empty sequence and denote the length of a sequence by $\#(\cdot)$. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\overline{\texttt{T}}$ $\overline{\texttt{f}}$" for "$\texttt{T}_1$ $\texttt{f}_1$, ... ,$\texttt{T}_n$ $\texttt{f}_n$," where n is the length of $\overline{\texttt{T}}$ and $\overline{\texttt{f}}$, and "this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;" as shorthand for "this.$\texttt{f}_1$=$\texttt{f}_1$; ... ;this.$\texttt{f}_n$=$\texttt{f}_n$;" and so on. Sequences of type variables, field declarations, parameter names, and method declarations are assumed to contain no duplicate names. We write the empty sequence as • and denote concatenation of sequences using a comma.

A family name P, used as a type argument to family-polymorphic methods, is either a top-level class name or a type variable. Absolute class names can be used to instantiate objects, so they play the role of run-time types of objects. A type can be an absolute path type P or P.C, or a relative path type .C. A top-level class declaration consists of its name, its superclass, field declarations, a constructor, methods, and nested classes. The symbol ◁ is read extends. On the other hand, a nested class does not have an extends clause since the class from which it inherits is implicitly determined. We have dropped the key word static, used in the previous section, for conciseness. As in Featherweight Java, a constructor is given in a stylized

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

syntax and just takes initial (and final) values for the fields and assigns them to corresponding fields. A method declaration can be parameterized by type variables, whose bounds are top-level class (i.e., family) names. Since the language is functional, the body of a method is a single `return` statement. An expression is a variable, field access, method invocation, or object creation. We assume that the set of variables includes the special variable `this`, which cannot be used as the name of a parameter to a method.

A class table CT is a mapping from top-level class names `C` to top-level class declarations. A program is a pair $(CT, \mathtt{e})$ of a class table and an expression. To lighten the notation in what follows, we always assume a *fixed* class table CT. As in Featherweight Java, we assume that `Object` has no members and its definition does *not* appear in the class table. We also assume some other sanity conditions on CT: (1) $CT(\mathtt{C}) = \mathtt{class\ C\ ..}$ for every $\mathtt{C} \in dom(CT)$; (2) for every class name `C` (except `Object`) appearing anywhere in CT, we have $\mathtt{C} \in dom(CT)$; and (3) there are no cycles in the transitive closure of `extends` relation. Given these conditions, we can identify a class table with a sequence of class declarations in an obvious way. Thus, in what follows, we write simply `class C ..` to mean $CT(\mathtt{C}) = \mathtt{class\ C\ ..}$.

3.2 Lookup functions

Before proceeding to the type system, we give functions to look up field or method definitions. The function *fields*(`A`) returns a sequence $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ of field names of the class `A` with their types. The function *mtype*(`m`, `A`) takes a method name and a class name as inputs and returns the corresponding method signature of the form $\mathtt{<}\overline{\mathtt{X}}\mathtt{<}\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\mathtt{\rightarrow}\mathtt{T}_0$, in which $\overline{\mathtt{X}}$ are bound in $\overline{\mathtt{T}}$ and $\mathtt{T}_0$. They are defined by the rules in Figure 3. Here, $\mathtt{m} \notin \overline{\mathtt{M}}$ (and $\mathtt{E} \notin \overline{\mathtt{N}}$) means the method of name `m` (and the nested class of name `E`, respectively) does not exist in $\overline{\mathtt{M}}$ (and $\overline{\mathtt{N}}$, respectively). In what follows, we identify method signatures modulo renaming of bound type variables.

As mentioned before, `Object` does not have any fields, methods, or nested classes, so *fields*(`Object`) = •, and *mtype*(`m`, `Object`) and *mtype*(`m`, `Object.C`) are undefined for any `C`. The definitions are straightforward extensions of the ones in Featherweight Java. Interesting rules are the last rules: when a nested class `C.E` does not exist, lookup proceeds in the nested class of the same name `E` in the superclass of the enclosing class `C`. When the method definition is found in a superclass, relative path types—whose meaning depends on the type of the receiver—in the method signature remain unchanged; they are resolved in typing rules. Note that we allow `Object.C` to be an argument of *fields*(·) for technical convenience, contrary to the fact that `Object` has no nested classes, to define *fields*(`C.D`) concisely. Moreover, *fields*(`C.D`) is defined for any `D` if $\mathtt{C} \in dom(CT)$. It does no harm since we never ask fields of such nonexisting classes, which will be rejected as ill-formed types (see the definition of type well-formedness below). We also think it would clutter the presentation to give a definition in which *fields*(`C.D`) is undefined when `C` or its superclasses do not have `D`.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Field Lookup

$$\mathit{fields}(\texttt{Object}) = \bullet \qquad \text{(F-TObject)}$$

$$\frac{\texttt{class C} \triangleleft \texttt{D}\{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}..\ \} \qquad \mathit{fields}(\texttt{D}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{U}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}}} \qquad \text{(F-TClass)}$$

$$\mathit{fields}(\texttt{Object.C}) = \bullet \qquad \text{(F-NObject)}$$

$$\frac{\texttt{class C} \triangleleft \texttt{D}\{..\ \texttt{class E}\{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}..\ \}..\ \} \qquad \mathit{fields}(\texttt{D.E}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C.E}) = \overline{\texttt{U}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}}} \qquad \text{(F-NClass)}$$

$$\frac{\texttt{class C} \triangleleft \texttt{D}\ \{..\ \overline{\texttt{N}}\} \qquad \texttt{E} \notin \overline{\texttt{N}} \qquad \mathit{fields}(\texttt{D.E}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C.E}) = \overline{\texttt{U}}\ \overline{\texttt{g}}} \qquad \text{(F-NSuper)}$$

Method Type Lookup

$$\frac{\texttt{class C} \triangleleft \texttt{D}\ \{..\ \overline{\texttt{M}}\} \quad \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{C}}\texttt{>T}_0\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\{\ \texttt{return e;}\ \} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0} \qquad \text{(MT-TClass)}$$

$$\frac{\texttt{class C} \triangleleft \texttt{D}\ \{..\ \overline{\texttt{M}}..\ \} \qquad \texttt{m} \notin \overline{\texttt{M}} \qquad \mathit{mtype}(\texttt{m}, \texttt{D}) = \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0} \qquad \text{(MT-TSuper)}$$

$$\frac{\texttt{class C} \triangleleft \texttt{D}\ \{..\ \texttt{class E}\ \{..\ \overline{\texttt{M}}\}..\ \} \quad \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{C}}\texttt{>T}_0\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\{\ \texttt{return e;}\ \} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C.E}) = \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0} \qquad \text{(MT-NClass)}$$

$$\frac{\texttt{class C} \triangleleft \texttt{D}\ \{..\ \texttt{class E}\ \{..\ \overline{\texttt{M}}\}..\ \} \quad \texttt{m} \notin \overline{\texttt{M}} \qquad \mathit{mtype}(\texttt{m}, \texttt{D.E}) = \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0}{\mathit{mtype}(\texttt{m}, \texttt{C.E}) = \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0} \qquad \text{(MT-NSuper1)}$$

$$\frac{\texttt{class C} \triangleleft \texttt{D}\ \{..\ \overline{\texttt{N}}\} \qquad \texttt{E} \notin \overline{\texttt{N}} \quad \mathit{mtype}(\texttt{m}, \texttt{D.E}) = \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0}{\mathit{mtype}(\texttt{m}, \texttt{C.E}) = \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0} \qquad \text{(MT-NSuper2)}$$

Fig. 3. .FJ: Lookup functions.

3.3 Type system

The main judgments of the type system consist of one for subtyping $\Delta \vdash \texttt{S} <: \texttt{T}$, one for type well-formedness $\Delta; \texttt{A} \vdash \texttt{T}$ ok, and one for typing $\Delta; \Gamma; \texttt{A} \vdash \texttt{e} : \texttt{T}$. Here, Δ is a *bound environment*, which is a finite mapping from type variables to their bounds, written $\overline{\texttt{X}}\texttt{<:}\overline{\texttt{C}}$, and Γ is a *type environment*, which is a finite mapping from variables to types, written $\overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}$. Since we are not concerned with more general forms

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Subtyping

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \qquad \text{(S-Refl)}$$

$$\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X}) \qquad \text{(S-Var)}$$

$$\Delta \vdash \mathtt{T} <: \mathtt{Object} \qquad \text{(S-Object)}$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D\ \{..\}}}{\Delta \vdash \mathtt{C} <: \mathtt{D}} \qquad \text{(S-Class)}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}} \qquad \text{(S-Trans)}$$

Type Well-formedness

$$\Delta; \mathtt{A} \vdash \mathtt{Object\ ok} \qquad \text{(WF-Object)}$$

$$\frac{\widehat{\Delta}(\mathtt{P}) \in \mathit{dom}(CT)}{\Delta; \mathtt{A} \vdash \mathtt{P\ ok}} \qquad \text{(WF-Abs)}$$

$$\frac{\mathtt{C} = \widehat{\Delta}(\mathtt{P}) \qquad \mathtt{class\ C} \triangleleft \mathtt{D\{..\ class\ E\ \{..\}\ ..\}}}{\Delta; \mathtt{A} \vdash \mathtt{P.E\ ok}} \qquad \text{(WF-NClass)}$$

$$\frac{\mathtt{C} = \widehat{\Delta}(\mathtt{P}) \qquad \mathtt{class\ C} \triangleleft \mathtt{D\{..}\ \overline{\mathtt{N}}\mathtt{\}} \qquad \mathtt{E} \notin \overline{\mathtt{N}} \qquad \Delta; \mathtt{A} \vdash \mathtt{D.E\ ok}}{\Delta; \mathtt{A} \vdash \mathtt{P.E\ ok}} \qquad \text{(WF-NClassSup)}$$

$$\frac{\Delta; \mathtt{C.D} \vdash \mathtt{C.E\ ok}}{\Delta; \mathtt{C.D} \vdash \mathtt{.E\ ok}} \qquad \text{(WF-Rel)}$$

Fig. 4. .FJ: Subtyping and type well-formedness.

of bounded polymorphism, upper bounds are always top-level class names. We write $\widehat{\Delta}(\mathtt{T})$ for the upper bound of $\mathtt{T}$ with respect to Δ, defined by $\widehat{\Delta}(\mathtt{A}) = \mathtt{A}$, $\widehat{\Delta}(\mathtt{X}) = \Delta(\mathtt{X})$ and $\widehat{\Delta}(\mathtt{X.C}) = \Delta(\mathtt{X}).\mathtt{C}$. We never ask the upper bound of a relative path type, so $\widehat{\Delta}(\mathtt{.C})$ is undefined. We abbreviate a sequence of judgments in the obvious way: $\Delta \vdash \mathtt{S}_1 <: \mathtt{T}_1, \ldots, \Delta \vdash \mathtt{S}_n <: \mathtt{T}_n$ to $\Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$; and $\Delta; \mathtt{A} \vdash \mathtt{T}_1\ \mathtt{ok}, \ldots, \Delta; \mathtt{A} \vdash \mathtt{T}_n\ \mathtt{ok}$ to $\Delta; \mathtt{A} \vdash \overline{\mathtt{T}}\ \mathtt{ok}$; and $\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Delta; \Gamma; \mathtt{A} \vdash \mathtt{e}_n : \mathtt{T}_n$ to $\Delta; \Gamma; \mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$.

Subtyping. The subtyping judgment $\Delta \vdash \mathtt{S} <: \mathtt{T}$, read as "$\mathtt{S}$ is subtype of $\mathtt{T}$ under Δ," is defined in Figure 4. This relation is the reflexive and transitive closure of the `extends` relation with `Object` being the top type. Note that a nested class, which does not have the `extends` clause, has only a trivial proper supertype, which is `Object`, even if some members are inherited from another (nested) class.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Type Well-formedness. The type well-formedness judgment $\Delta; \mathtt{A} \vdash \mathtt{T}$ ok, read as "T is a well-formed type in (the body of) class A under Δ." The rules for well-formed types appear also in Figure 4. `Object` and class names in the domain of the class table are well formed. Moreover, a nested class name `C.E` is well formed if `E` is inherited from `C`'s superclass `D`. Type `X` (possibly with a suffix) is well formed if its upper bound (with the suffix) is well formed. Finally, a relative path type `.E` is well formed in a nested class `C.D` if `C.E` is well formed.

Typing. Typing requires another auxiliary (but important) definition. The *resolution* `T@S` of `T` at `S` intuitively denotes the class name that `T` refers to in a given class `S`. One use of the resolution is to determine which class to look up for fields or methods, when a receiver's type is relative: *fields* and *mtype* require absolute path types as arguments. The definition is as follows:

$$\mathtt{.D@P.C} = \mathtt{P.D} \qquad \mathtt{.D@.C} = \mathtt{.D} \qquad \mathtt{P@T} = \mathtt{P} \qquad \mathtt{P.C@T} = \mathtt{P.C}.$$

The only interesting case is the first clause: it means that a relative path type `.D` in `P.C` refers to `P.D`. For example, `.Edge@Graph.Node` = `Graph.Edge`. Note that `.D@C` and `.D@X` are undefined since we never resolve a relative path type with respect to a family name.

The typing judgment for expressions is of the form $\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e:T}$, read as "under bound environment Δ and type environment Γ, expression `e` has type `T` in class `A`," in which we assume that for any $\mathtt{x} \in \mathit{dom}(\Gamma)$, $\Delta; \mathtt{A} \vdash \Gamma(\mathtt{x})$ ok. Typing rules are given in Figure 5. Interesting rules are T-Field and T-Invk, although the basic idea is as usual—for example, in T-Field, the field types are retrieved from the receiver's type $\mathtt{T}_0$, and the corresponding type of the accessed field is the type of the whole expression. We need some tricks to deal with relative path types (and type variables): if the receiver's type $\mathtt{T}_0$ is a relative path type, it has to be resolved in `A`, the class in which `e` appears; a type variable is taken to its upper bound by $\widehat{\Delta}(\cdot)$. Moreover, if the field type is a relative path type, it is resolved in the *receiver's type*. For example, if *fields*(`ColorWeightGraph.Node`) = `.Edge edg` and Γ = `x:ColorWeightGraph.Node, y:.Node`, then

$$\Delta; \Gamma; \mathtt{ColorWeightGraph.Node} \vdash \mathtt{x.edg : ColorWeightGraph.Edge} \text{ and}$$
$$\Delta; \Gamma; \mathtt{ColorWeightGraph.Node} \vdash \mathtt{y.edg : .Edge}.$$

In this way, accessing a field of relative path type gives a relative path type if and only if the receiver is also given a relative path type. Similarly, in T-Invk, the method type is retrieved from the receiver's type; then, it is checked whether the given type arguments are subtypes of bounds $\overline{\mathtt{C}}$ of formal type parameters and the types of actual value arguments are subtypes of those of formal parameters, where type arguments are substituted for variables. For example, if *mtype*(`connectAll`, `C`) = `<G◁Graph>(G.Node,G.Edge)→void`, then

$$\Delta; \mathtt{x:C, n:ColorWeightGraph.Node, e:ColorWeightGraph.Edge; A} \vdash$$
$$\mathtt{x.<ColorWeightGraph>connectAll(n,e) : void}.$$

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Expression Typing

$$\Delta;\Gamma;\mathtt{A} \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-Var)}$$

$$\frac{\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{fields}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{T}_i@\mathtt{T}_0} \qquad \text{(T-Field)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m}, \widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}_0 \\ \Delta;\mathtt{A} \vdash \overline{\mathtt{P}}\ \mathtt{ok} \qquad \Delta \vdash \overline{\mathtt{P}} \mathtt{<:}\ \overline{\mathtt{C}} \qquad \Delta;\Gamma;\mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \qquad \Delta \vdash \overline{\mathtt{T}} \mathtt{<:}\ ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@\mathtt{T}_0\end{array}}{\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} : ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U}_0)@\mathtt{T}_0} \qquad \text{(T-Invk)}$$

$$\frac{\Delta;\mathtt{A} \vdash \mathtt{A}_0\ \mathtt{ok} \qquad \mathit{fields}(\mathtt{A}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta;\Gamma;\mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta \vdash \overline{\mathtt{U}} \mathtt{<:}\ (\overline{\mathtt{T}}@\mathtt{A}_0)}{\Delta;\Gamma;\mathtt{A} \vdash \mathtt{new}\ \mathtt{A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{A}_0} \qquad \text{(T-New)}$$

Method Typing

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}} \qquad \Delta;\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathit{thistype}(\mathtt{A});\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{U}_0 \\ \Delta \vdash \mathtt{U}_0 \mathtt{<:}\ \mathtt{T}_0 \qquad \Delta;\mathtt{A} \vdash \mathtt{T}_0, \overline{\mathtt{T}}, \overline{\mathtt{C}}\ \mathtt{ok} \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathit{superclass}(\mathtt{A})) = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{D}}\mathtt{>}\overline{\mathtt{S}} \to \mathtt{S}_0, \text{ then } (\overline{\mathtt{X}}, \overline{\mathtt{C}}, \overline{\mathtt{T}}, \mathtt{T}_0) = (\overline{\mathtt{Y}}, \overline{\mathtt{D}}, \overline{\mathtt{S}}, \mathtt{S}_0)\end{array}}{\mathtt{A} \vdash \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{C}}\mathtt{>T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{return\ e}_0\mathtt{;\}}\ \mathtt{ok}} \qquad \text{(T-Method)}$$

Class Typing

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{E(}\overline{\mathtt{U}}\ \overline{\mathtt{g}}\mathtt{,}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{super(}\overline{\mathtt{g}}\mathtt{);this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;\}} \\ \mathit{fields}(\mathit{superclass}(\mathtt{C.E})) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \mathtt{C.E} \vdash \overline{\mathtt{M}}\ \mathtt{ok} \qquad \emptyset;\mathtt{C.E} \vdash \overline{\mathtt{T}}\ \mathtt{ok}\end{array}}{\mathtt{C} \vdash \mathtt{class\ E\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{M}}\mathtt{\}}\ \mathtt{ok}} \qquad \text{(T-NClass)}$$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C(}\overline{\mathtt{U}}\ \overline{\mathtt{g}}\mathtt{,}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{super(}\overline{\mathtt{g}}\mathtt{);this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;\}} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \mathtt{C} \vdash \overline{\mathtt{M}}\ \mathtt{ok} \qquad \mathtt{C} \vdash \overline{\mathtt{N}}\ \mathtt{ok} \qquad \emptyset;\mathtt{C} \vdash \overline{\mathtt{T}}, \mathtt{D}\ \mathtt{ok}\end{array}}{\vdash \mathtt{class\ C} \lhd \mathtt{D\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{M}}\ \overline{\mathtt{N}}\mathtt{\}}\ \mathtt{ok}} \qquad \text{(T-TClass)}$$

Fig. 5. .FJ: Typing.

A judgment for method typing is written $\mathtt{A} \vdash \mathtt{M}$ ok, and derived by T-Method. Here, *thistype*(A) and *superclass*(A) are defined by the following:

$$\begin{array}{ll}\mathit{thistype}(\mathtt{C}) = \mathtt{C} & \mathit{thistype}(\mathtt{C.E}) = \mathtt{.E} \\ \mathit{superclass}(\mathtt{C}) = \mathtt{D} & \mathit{superclass}(\mathtt{C.E}) = \mathtt{D.E}\end{array}$$

where `class C ◁ D{.. }`. It is checked that the body of the method is well typed under the bound and type environments obtained from the definition. Note that the type of `this` in a nested class is relative, as the meaning of `this` changes in subclasses. The last conditional premise checks that `m` correctly overrides (if it does) the method of the same name in the superclass with the same signature.

There are two class typing rules, one for top-level classes and one for nested classes. Both of them are essentially the same: they check that field types and

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

constructor argument types are the same, and that methods are ok in the class. The rule T-TCLASS for top-level classes also checks that nested classes are ok.

3.4 Operational semantics

The operational semantics is given by the reduction relation of the form $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression e reduces to e′ in one step." We require another lookup function *mbody*(m<$\overline{\mathtt{P}}$>, A), for the method body with formal parameters, written $\overline{\mathtt{x}}.\mathtt{e}$, of given method and class names. It is defined at the top of Figure 6. Similarly to *mtype*, *mbody*(m<$\overline{\mathtt{P}}$>, Object) and *mbody*(m<$\overline{\mathtt{P}}$>, Object.C) are undefined.

The reduction rules are given in the middle of Figure 6. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ for the expression obtained from $\mathtt{e}_0$ by replacing $\mathtt{x}_1$ with $\mathtt{d}_1, \ldots,$ $\mathtt{x}_n$ with $\mathtt{d}_n$, and y with e. There are two reduction rules, one for field access and one for method invocation, which are straightforward. The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules (if $\mathtt{e} \longrightarrow \mathtt{e}'$ then $\mathtt{e.f} \longrightarrow \mathtt{e}'\mathtt{.f}$, and the like). We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

3.5 Type soundness

The type system is sound with respect to the operational semantics, as expected. Type soundness is proved in the standard manner via subject reduction and progress (Wright & Felleisen 1994; Igarashi *et al.* 2001). (Recall that values are defined by $\mathtt{v} ::= \mathtt{new\ A}(\overline{\mathtt{v}})$, where $\overline{\mathtt{v}}$ can be empty.)

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

Proof
See Appendix A. □

Theorem 3.2 (*Progress*)
If $\emptyset;\emptyset;\mathtt{B} \vdash \mathtt{e}:\mathtt{A}$ and e is not a value, then $\mathtt{e} \longrightarrow \mathtt{e}'$, for some e′.

Proof
By induction on $\emptyset;\emptyset;\mathtt{B} \vdash \mathtt{e} : \mathtt{A}$ with case analysis on the last rule used.

Case T-VAR**:**

Cannot happen.

Case T-FIELD**:** $\emptyset;\emptyset;\mathtt{B} \vdash \mathtt{e}_0 : \mathtt{A}_0$ $\quad$ $\mathit{fields}(\widehat{\Delta}(\mathtt{A}_0@\mathtt{B})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$

If $\mathtt{e}_0 = \mathtt{new\ A}_0(\overline{\mathtt{v}})$, then, by R-FIELD, $\mathtt{e}_0 \longrightarrow \mathtt{v}_i$; otherwise, immediate from the induction hypothesis and RC-FIELD.

Case T-INVK**:** $\mathtt{e} = \mathtt{e}_0.<\overline{\mathtt{P}}>\mathtt{m}(\overline{\mathtt{v}})$

If there is a nonvalue subexpression, then the conclusion is immediate from the induction hypothesis and the rules RC-INV-RECV and RC-INV-ARG. Suppose that

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Method body lookup

$$\frac{\texttt{class C}\triangleleft\texttt{D\{..}\ \overline{\texttt{M}}\texttt{..\}} \quad \texttt{<}\overline{\texttt{X}}\triangleleft\overline{\texttt{C}}\texttt{>T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e}_0\texttt{; \}} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m<}\overline{\texttt{P}}\texttt{>, C}) = \overline{\texttt{x}}\texttt{.}\,[\overline{\texttt{P}}/\overline{\texttt{X}}]\overline{\texttt{e}}_0} \qquad \text{(MB-TClass)}$$

$$\frac{\texttt{class C}\triangleleft\texttt{D \{..}\ \overline{\texttt{M}}\texttt{..\}} \qquad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m<}\overline{\texttt{P}}\texttt{>, C}) = \mathit{mbody}(\texttt{m<}\overline{\texttt{P}}\texttt{>, D})} \qquad \text{(MB-TSuper)}$$

$$\frac{\texttt{class C}\triangleleft\texttt{D \{..}\ \overline{\texttt{N}}\texttt{\}} \quad \texttt{class E \{..}\ \overline{\texttt{M}}\texttt{\}} \in \overline{\texttt{N}} \quad \texttt{<}\overline{\texttt{X}}\triangleleft\overline{\texttt{C}}\texttt{>T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e}_0\texttt{; \}} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m<}\overline{\texttt{P}}\texttt{>, C.E}) = \overline{\texttt{x}}\texttt{.}\,[\overline{\texttt{P}}/\overline{\texttt{X}}]\texttt{e}_0} \qquad \text{(MB-NClass)}$$

$$\frac{\texttt{class C}\triangleleft\texttt{D \{..}\ \overline{\texttt{N}}\texttt{\}} \qquad \texttt{E} \notin \overline{\texttt{N}}}{\mathit{mbody}(\texttt{m<}\overline{\texttt{P}}\texttt{>, C.E}) = \mathit{mbody}(\texttt{m<}\overline{\texttt{P}}\texttt{>, D.E})} \qquad \text{(MB-NSuper1)}$$

$$\frac{\texttt{class C}\triangleleft\texttt{D \{..}\ \overline{\texttt{N}}\texttt{\}} \quad \texttt{class E \{..}\ \overline{\texttt{M}}\texttt{\}} \in \overline{\texttt{N}} \quad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m<}\overline{\texttt{P}}\texttt{>, C.E}) = \mathit{mbody}(\texttt{m<}\overline{\texttt{P}}\texttt{>, D.E})} \qquad \text{(MB-NSuper2)}$$

Computation

$$\frac{\mathit{fields}(\texttt{A}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\texttt{new A(}\overline{\texttt{e}}\texttt{).f}_i \longrightarrow \texttt{e}_i} \qquad \text{(R-Field)}$$

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

Congruence

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

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0'}{\texttt{e}_0\texttt{.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \longrightarrow \texttt{e}_0'\texttt{.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}} \qquad \text{(RC-Invk-Recv)}$$

$$\frac{\texttt{e}_i \longrightarrow \texttt{e}_i'}{\texttt{e}_0\texttt{.<}\overline{\texttt{P}}\texttt{>m(...,e}_i\texttt{,...)} \longrightarrow \texttt{e}_0\texttt{.<}\overline{\texttt{P}}\texttt{>m(...,e}_i'\texttt{,...)}} \qquad \text{(RC-Invk-Arg)}$$

$$\frac{\texttt{e}_i \longrightarrow \texttt{e}_i'}{\texttt{new A(...,e}_i\texttt{,...)} \longrightarrow \texttt{new A(...,e}_i'\texttt{,...)}} \qquad \text{(RC-New-Arg)}$$

Fig. 6. .FJ: Operational semantics.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

$\mathtt{e}_0$ and $\overline{\mathtt{e}}$ are values $\mathtt{v}_0$ and $\overline{\mathtt{v}}$, respectively. Then, by T-Invk,

$$\frac{\begin{array}{cc}\emptyset;\emptyset;\mathtt{B} \vdash \mathtt{v}_0 : \mathtt{A}_0 & \mathit{mtype}(\mathtt{m}, \widehat{\Delta}(\mathtt{A}_0@\mathtt{B})) = \mathtt{<}\overline{\mathtt{X}} \mathtt{\triangleleft} \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\mathtt{\to}\mathtt{T}_0 \\ \emptyset;\mathtt{B} \vdash \overline{\mathtt{P}}\ \mathrm{ok} \quad \emptyset \vdash \overline{\mathtt{P}} \mathtt{<:} \overline{\mathtt{C}} & \emptyset;\emptyset;\mathtt{B} \vdash \overline{\mathtt{v}} : \overline{\mathtt{A}} \quad \emptyset \vdash \overline{\mathtt{A}} \mathtt{<:} ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{T}})@\mathtt{A}_0\end{array}}{\emptyset;\emptyset;\mathtt{B} \vdash \mathtt{v}_0.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m(}\overline{\mathtt{v}}\mathtt{)} : ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0)@\mathtt{A}_0}$$

From premises of the rule, we have $\#(\overline{\mathtt{v}}) = \#(\overline{\mathtt{A}}) = \#(\overline{\mathtt{T}})$. Since $\mathit{mtype}(\mathtt{m}, \mathtt{A}_0) = \mathtt{<}\overline{\mathtt{X}} \mathtt{\triangleleft} \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\mathtt{\to}\mathtt{T}_0$, it is easy to show that $\mathit{mbody}(\mathtt{m}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}, \mathtt{A}_0) = \overline{\mathtt{x}}.\mathtt{e}_0$ for some $\overline{\mathtt{x}}$ and $\mathtt{e}_0$ with $\#(\overline{\mathtt{T}}) = \#(\overline{\mathtt{x}})$. Then, by R-Invk, $\mathtt{v}_0.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m(}\overline{\mathtt{v}}\mathtt{)} \longrightarrow [\overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{v}_0/\mathtt{this}]\mathtt{e}_0$.

Case T-New: $\mathtt{e} = \mathtt{new\ A(}\overline{\mathtt{e}}\mathtt{)}$

If one of $\overline{\mathtt{e}}$ is not a value, apply the induction hypothesis with the rule T-New-Arg; otherwise, e is a value. □

Theorem 3.3 (*Type Soundness*)
If $\emptyset;\emptyset;\mathtt{B} \vdash \mathtt{e} : \mathtt{A}$ and $\mathtt{e} \longrightarrow^* \mathtt{e}'$ with $\mathtt{e}'$ a normal form, then $\mathtt{e}'$ is a value v with $\emptyset;\emptyset;\mathtt{B} \vdash \mathtt{v} : \mathtt{A}'$ and $\emptyset \vdash \mathtt{A}' \mathtt{<:} \mathtt{A}$.

Proof
By easy induction on $\mathtt{e} \longrightarrow^* \mathtt{e}'$ using Theorems 3.1 and 3.2. □

3.6 Type inference for parametric method invocations

The language .FJ is considered an intermediate language in which every type argument to parametric methods is made explicit. In this section, we briefly discuss how type arguments can be recovered, give an algorithm for type argument inference, and prove its correctness theorem.

The basic idea of type inference is the same as Java 5.0: given a method invocation expression $\mathtt{e}_0.\mathtt{m(}\overline{\mathtt{e}}\mathtt{)}$ that appears in class A without specifying type arguments, we can at least compute the type $\mathtt{T}_0$ of $\mathtt{e}_0$, the signature $\mathtt{<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0$ of the method m, and the types $\overline{\mathtt{T}}$ of (value) arguments. Then, it is easy to see from the rule T-Invk that it suffices to find $\overline{\mathtt{P}}$ that satisfies $\overline{\mathtt{P}} \mathtt{<:} \overline{\mathtt{C}}$ and $\overline{\mathtt{T}} \mathtt{<:} ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@\mathtt{T}_0$. In other words, the goal of type inference is to solve the set $\{\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}}, \overline{\mathtt{T}}\mathtt{<:}(\overline{\mathtt{U}}@\mathtt{T}_0)\}$ of inequalities with respect to $\overline{\mathtt{X}}$.

We formalize this constraint-solving process as function $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S)$. It takes as input a set S of inequalities of the form either $\mathtt{X}\mathtt{<:}\mathtt{C}$ or $\mathtt{T}_1\mathtt{<:}\mathtt{T}_2$, where $\mathtt{T}_1$ does not contain $\mathtt{X}_i$, and returns a mapping from $\overline{\mathtt{X}}$ to types (more precisely, family names). The auxiliary parameter Δ records other variables' bounds, so $\overline{\mathtt{X}}$ and the domain of Δ are assumed to be disjoint. The definition of $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S)$ is shown in Figure 7. Here, $S_1 \uplus S_2$ is a union of S_1 and S_2, where $S_1 \cap S_2 = \emptyset$. $\mathtt{T}_1 \sqcup_{\Delta} \mathtt{T}_2$ is the least upper bound of $\mathtt{T}_1$ and $\mathtt{T}_2$ (the least upper bound of two given types always exists since we do not have interfaces, which can extend more than one interface). We assume that each clause is applied in the order shown—thus, for example, the fourth clause will not be applied until there is only one inequation of the form $\mathtt{T}\mathtt{<:}\mathtt{X}_i$.

The algorithm is explained as follows. The second clause is the case where a formal argument type is $\mathtt{X}_i.\mathtt{C}$ and the corresponding actual is P.C: since P.C has only a trivial supertype (namely, itself) except Object, $\mathtt{X}_i$ must be P. The third

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

$$
\begin{array}{lcl}
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(\emptyset) & = & [\,] \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{P}\,.\,\mathtt{C}{<:}\,\mathtt{X}_i\,.\,\mathtt{C}\}) & = & [\mathtt{P}/\mathtt{X}_i] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}\setminus\{\mathtt{X}_i\}}([\mathtt{P}/\mathtt{X}_i]S') \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T}_1{<:}\,\mathtt{X}_i\} \uplus \{\mathtt{T}_2{<:}\,\mathtt{X}_i\}) & = & \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{(\mathtt{T}_1 \sqcup_{\Delta} \mathtt{T}_2){<:}\,\mathtt{X}_i\}) \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T}{<:}\,\mathtt{X}_i, \mathtt{X}_i{<:}\,\mathtt{C}\}) & = & \begin{cases} [\mathtt{T}/\mathtt{X}_i] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}\setminus\{\mathtt{X}_i\}}(S') & \\ \qquad \text{(if } \Delta \vdash \mathtt{T}{<:}\,\mathtt{C} \text{ and } \mathtt{X}_i \notin S') & \\ \mathit{fail} \quad \text{(otherwise)} & \end{cases} \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{X}_i{<:}\,\mathtt{C}\}) & = & \begin{cases} [\mathtt{C}/\mathtt{X}_i] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}\setminus\{\mathtt{X}_i\}}(S') & (\text{if } \mathtt{X}_i \notin S') \\ \mathit{fail} & (\text{otherwise}) \end{cases} \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T}_1{<:}\,\mathtt{T}_2\}) & = & \begin{cases} \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S') & (\text{if } \Delta \vdash \mathtt{T}_1{<:}\,\mathtt{T}_2) \\ \mathit{fail} & (\text{otherwise}) \end{cases}
\end{array}
$$

Fig. 7. Type inference algorithm.

clause is the case where a type variable has more than one lower bound: we replace two inequalities by one using the least upper bound. The following two clauses are applied when no other constraints on $\mathtt{X}_i$ appear elsewhere; they check whether the constraint is satisfiable. Note that the fifth clause is applied only when the method signature contains a type variable that does not appear in parameters types, as in `<G◁Graph>()→void`. In this case, the given upper bound itself is chosen for the type variable and so the solution will not be least.

Now, we state the theorem of correctness of type inference. It means that, if type inference succeeds (and every type variable appears in argument types), it gives the least type arguments.

Theorem 3.4 (*Type inference correctness*)
If $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0$ and $\mathit{mtype}(\mathtt{m}, \widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\rangle\overline{\mathtt{U}}\to\mathtt{U}_0$ and $\Delta;\Gamma;\mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$ and $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(\{\overline{\mathtt{X}}{<:}\,\overline{\mathtt{C}}, \overline{\mathtt{T}}{<:}\,(\overline{\mathtt{U}}@\mathtt{T}_0)\})$ returns $\sigma = [\overline{\mathtt{P}}/\overline{\mathtt{X}}]$, then $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0\,.\langle\sigma\overline{\mathtt{X}}\rangle\mathtt{m}(\overline{\mathtt{e}}) : (\sigma\mathtt{U}_0)@\mathtt{T}_0$. Moreover, if every $\mathtt{X}_i$ occurs in $\overline{\mathtt{U}}$, then σ is least in the sense that for any σ' such that $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0\,.\langle\sigma'\overline{\mathtt{X}}\rangle\mathtt{m}(\overline{\mathtt{e}}) : (\sigma'\mathtt{U}_0)@\mathtt{T}_0$, it holds that $\Delta \vdash \sigma(\mathtt{X}_i) <: \sigma'(\mathtt{X}_i)$ for any $\mathtt{X}_i$.

Proof
We show that, if $\overline{\mathtt{X}} \cap \mathit{dom}(\Delta) = \emptyset$ and $\overline{\mathtt{X}}$ do not occur in $\overline{\mathtt{S}}$ and $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(\{\overline{\mathtt{X}}{<:}\,\overline{\mathtt{C}}, \overline{\mathtt{S}}{<:}\,\overline{\mathtt{T}}\}) = \sigma$, then $\Delta \vdash \sigma\overline{\mathtt{X}} <: \overline{\mathtt{C}}$ and $\Delta \vdash \mathtt{S} <: \sigma\mathtt{T}$ and σ is least, by induction on the number of steps to derive $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(\{\overline{\mathtt{X}}{<:}\,\overline{\mathtt{C}}, \overline{\mathtt{S}}{<:}\,\overline{\mathtt{T}}\}) = \sigma$. (It is easy to show $\Delta;\mathtt{A} \vdash \sigma(\overline{\mathtt{X}})$ ok from Lemma B.5.)

Case: $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(\{\}) = [\,]$

Trivial.

Case: $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{P}\,.\,\mathtt{C}{<:}\,\mathtt{X}_i\,.\,\mathtt{C}\}) = [\mathtt{P}/\mathtt{X}_i] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}\setminus\{\mathtt{X}_i\}}([\mathtt{P}/\mathtt{X}_i]S')$

Let $\sigma' = \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}\setminus\{\mathtt{X}_i\}}([\mathtt{P}/\mathtt{X}_i]S')$. By the induction hypothesis, for each $\mathtt{S}{<:}\,\mathtt{T} \in [\mathtt{P}/\mathtt{X}_i]S'$, it holds that $\Delta \vdash \sigma'(\mathtt{S} <: \mathtt{T})$. Thus, $\Delta \vdash (\sigma' \circ [\mathtt{P}/\mathtt{X}_i])(\mathtt{S}'{<:}\,\mathtt{T}')$ for each $\mathtt{S}'{<:}\,\mathtt{T}' \in S'$. Since $\mathtt{X}_i$ is not in the range of σ' and none of $\overline{\mathtt{X}}$ appears in $\mathtt{P}$, it holds that $\sigma' \circ [\mathtt{P}/\mathtt{X}_i] = [\mathtt{P}/\mathtt{X}_i] \circ \sigma'$. Thus, $\Delta \vdash ([\mathtt{P}/\mathtt{X}_i] \circ \sigma')(\mathtt{S}' <: \mathtt{T}')$ and $\Delta \vdash ([\mathtt{P}/\mathtt{X}_i] \circ \sigma')(\mathtt{P}\,.\,\mathtt{C} <: \mathtt{X}_i\,.\,\mathtt{C})$. Leastness is obvious since $\mathtt{X}_i$ must be instantiated to $\mathtt{P}$ for $\Delta \vdash ([\mathtt{P}/\mathtt{X}_i] \circ \sigma')(\mathtt{P}\,.\,\mathtt{C} <: \mathtt{X}_i\,.\,\mathtt{C})$ to hold.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Case: $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T}_1 <: \mathtt{X}_i\} \uplus \{\mathtt{T}_2 <: \mathtt{X}_i\}) = \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{(\mathtt{T}_1 \sqcup_{\Delta} \mathtt{T}_2) <: \mathtt{X}_i\})$

Similar. For leastness, recall that $\mathtt{T}_1 \sqcup_{\Delta} \mathtt{T}_2$ gives the least upper bound of $\mathtt{T}_1$ and $\mathtt{T}_2$ (with respect to Δ).

Case: $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T} <: \mathtt{X}_i, \mathtt{X}_i <: \mathtt{C}\}) = [\mathtt{T}/\mathtt{X}_i] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}} \setminus \{\mathtt{X}_i\}}(S')$
$\Delta \vdash \mathtt{T} <: \mathtt{C} \qquad \mathtt{X}_i \notin S'$

Similar. Obviously, instantiating $\mathtt{X}_i$ with $\mathtt{T}$ gives the least solution.

Case: $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{X}_i <: \mathtt{C}\}) = [\mathtt{C}/\mathtt{X}_i] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}} \setminus \{\mathtt{X}_i\}}(S') \qquad \mathtt{X}_i \notin S'$

Similar. Note that this case does not apply to the leastness proof.

Case: $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T}_1 <: \mathtt{T}_2\}) = \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S') \qquad \Delta \vdash \mathtt{T}_1 <: \mathtt{T}_2$

Similar. □

4 Translating .FJ to Featherweight Java

In this section, we discuss a possible implementation of the proposed language features by *erasure* translation. The basic idea of erasure translation, which is also used in the current implementation of Java generics (Bracha *et al.* 1998), is to erase refined type information such as type arguments to generic classes or, here, relative path types to conventional monomorphic types; since simply erasing some type information makes the program ill-typed, typecasts are inserted where required. Following Igarashi *et al.* (2001), we model erasure translation by presenting formal translation from .FJ to Featherweight Java (FJ). This is an abstraction of the actual translation that a real compiler would apply to abstract syntax trees to support the language extension—features like generics and nested classes are implemented in `javac` in this way. Then, we prove that the translation is correct with respect to typing and reduction.

Before proceeding to the formal definition of the translation, we describe how classes `Graph` and `ColorWeightGraph` and polymorphic method `connectAll()` would be translated by erasure in Figure 8. First of all, all nested classes are pulled out to the top level and fully qualified names are made simple as in `Graph$Edge`, in which `$` is another character that can be used to form a simple name. As mentioned above, all relative path types are changed to ordinary types; also, type parameters to polymorphic methods are erased by replacing the type variable `G` by its upper bound `Graph`. Since method `connect()` now takes two `Graph$Nodes`, typecasts to `ColorWeightGraph$Node` are inserted where field `color` is accessed. Another noticeable change includes `extends` clauses in nested classes to make all inheritance relations explicit.

The rest of this section proceeds as follows: we first briefly review Featherweight Java, the target language of the translation; then, present the formal definition of translation; and finally prove properties of the erasure translation.

4.1 Featherweight Java

We begin with briefly reviewing Featherweight Java. Since most definitions of .FJ can be reused, we show only main changes that would be required to make Featherweight

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

```
class Graph { }
class Graph$Node {
    Graph$Edge[] es=new Graph$Edge[10]; int i=0;
    void add(Graph$Edge e) { es[i++] = e; }
}
class Graph$Edge {
    Graph$Node src, dst;
    void connect(Graph$Node s, Graph$Node d) {
        src = s; dst = d; s.add(this); d.add(this);
    }
}

class ColorWeightGraph extends Graph { }
class ColorWeightGraph$Node extends Graph$Node { Color color; }
class ColorWeightGraph$Edge extends Graph$Edge {
    int weight;
    void connect(Graph$Node s, Graph$Node d) {
        weight = colorToWeight(((ColorWeightGraph$Node)s).color,
                               ((ColorWeightGraph$Node)d).color);
        super.connect(s, d);
    }
}
```

```
void connectAll(Graph$Edge[] es, Graph$Node n1, Graph$Node n2){
    for (int i: es) es[i].connect(n1,n2);
}
```

Fig. 8. Erased nested classes and family-polymorphic method.

Java, instead of reintroducing all definitions from scratch. The main changes consist of addition of type casts and restriction on the syntax. Figure 9 shows the syntax with typing and semantics for casting.

In the restricted syntax, top-level classes do not contain nested classes; all types are top-level class names; and methods are monomorphic. (Thus, neither polymorphic method invocation nor instantiation of a nested class appears in a well-typed program.) Note that, in the formal translation defined below, all nested classes are translated to top-level classes, as nesting does not play any significant role after erasure.

In the figure, we abuse the notation by omitting irrelevant information of bound environment Δ and enclosing class name $\mathtt{A}$ from subtyping and typing judgments. They are irrelevant in the sense that they cannot affect derivable judgments (modulo this information) within this restricted language. So, the existing rules could be simplified by dropping Δ and $\mathtt{A}$: for example, T-Field for FJ expressions would be

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

which is exactly the same rule as in the original formulation.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Syntax

$$
\begin{array}{lcll}
\texttt{L} & ::= & \texttt{class C}\lhd\texttt{C}\{\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{}\} & \text{classes} \\
\texttt{K} & ::= & \texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{f}})\texttt{;this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}\} & \text{constructors} \\
\texttt{M} & ::= & \texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\texttt{\{ return e; \}} & \text{methods} \\
\texttt{e} & ::= & \ldots \mid \texttt{(C)e} & \text{expressions} \\
\texttt{v} & ::= & \texttt{new C}(\overline{\texttt{v}}) & \text{values}
\end{array}
$$

Typing

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

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

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

Computation

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

Congruence

$$\frac{\texttt{e} \longrightarrow \texttt{e}'}{\texttt{(C)e} \longrightarrow \texttt{(C)e}'} \quad \text{(RC-Cast)}$$

Fig. 9. FJ syntax, and typing and reduction rules for casting.

The rules for typecasts are the same as those in original FJ. Although all casts introduced by translation are downcasts, they will (be proved to) turn into upcasts, which are removed by R-Cast, during execution. Thus, in fact, stupid casts, which necessarily fail if ever executed, could be omitted for correctness theorems below. Here, it is included mainly for consistency (and to make subject reduction hold).

In what follows, we write $\Gamma \vdash_{\text{FJ}} \texttt{e} : \texttt{C}$ for FJ typing judgments and $\texttt{C} <:_{\text{FJ}} \texttt{D}$ for FJ subtyping.

4.2 Erasure of types and expressions

Relative path types and type variables are translated to ordinary types by resolving with the enclosing class in which they appear, and promoting to their upper bounds, respectively. Since all classes are translated to top-level classes, absolute class names like `C.D` will become atomic names. The erasure $|\texttt{T}|_{\Delta,\texttt{A}}$ of .FJ type `T` with respect to

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

bound environment Δ and class A (in which T appears) is defined as follows:

$$\begin{array}{lcl} |\mathtt{P}|_{\Delta,\mathtt{A}} & = & \widehat{\Delta}(\mathtt{P}) \\ |\mathtt{P.C}|_{\Delta,\mathtt{A}} & = & \widehat{\Delta}(\mathtt{P})\mathtt{\$C} \\ |\mathtt{.E}|_{\Delta,\mathtt{C.D}} & = & \mathtt{C\$E} \end{array}$$

Note that, as mentioned above, C$E is a simple name. We sometimes omit Δ when Δ is empty and A when T is not a relative path type.

For the erasure of expressions, we define auxiliary functions *fieldsmax*(A) and *mtypemax*(m, A) to look up types of fields and methods after erasure. These functions will be used to determine where to insert downcasts, as we will see later.

fieldsmax(A) takes an absolute class name A as an input, and returns a sequence of pairs of an erased type and a field name for all the fields of A. Since inherited fields are declared in different classes, types are erased at the very class in which the corresponding fields are declared. *fieldsmax*(A) is defined as follows:

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathit{fieldsmax}(\mathtt{C}) = |\overline{\mathtt{T}}|\ \overline{\mathtt{f}}}$$

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

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\ \{\ ..\ class\ E\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ ..\ \}\ ..\ \}} \qquad \mathit{fieldsmax}(\mathtt{D.E}) = \overline{\mathtt{C}}\ \overline{\mathtt{g}}}{\mathit{fieldsmax}(\mathtt{C.E}) = \overline{\mathtt{C}}\ \overline{\mathtt{g}}, |\overline{\mathtt{T}}|_{\mathtt{C.E}}\ \overline{\mathtt{f}}}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\ \{\ ..\ }\overline{\mathtt{N}}\mathtt{\ \}} \qquad \mathtt{E} \notin \overline{\mathtt{N}} \qquad \mathit{fieldsmax}(\mathtt{D.E}) = \overline{\mathtt{C}}\ \overline{\mathtt{g}}}{\mathit{fieldsmax}(\mathtt{C.E}) = \overline{\mathtt{C}}\ \overline{\mathtt{g}}}$$

The basic structure of *fieldsmax* is the same as *fields*. Notice that types are erased in the class where they are found. Since field types can never be type variables or relative path types in top-level classes, we can use the result of *fields* and erase it at once. We write $\mathit{fieldsmax}(\mathtt{A})(\mathtt{f}_i) = \mathtt{D}_i$, where $\mathit{fieldsmax}(\mathtt{A}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$.

mtypemax(m, A) takes a method name m and an absolute class name A as inputs, and returns the erased signature of m of A. To deal with possible method overriding, method signatures will be erased with respect to *the superclass of* A *in which* m *is first defined* so that the signatures of all overriding m will be the same after erasure; if signatures were erased with respect to the class where the definition is found, the same relative path type would result in different types, depending on where they appear, and method overriding will not be preserved in the erased program.[1] The definition of *mtypemax* is as follows:

$$\frac{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \langle \overline{\mathtt{X}} \lhd \overline{\mathtt{C}} \rangle \overline{\mathtt{T}} \rightarrow \mathtt{T}_0 \qquad \Delta = \overline{\mathtt{X}} <: \overline{\mathtt{C}}}{\mathit{mtypemax}(\mathtt{m}, \mathtt{C}) = |\overline{\mathtt{T}}|_{\Delta} \rightarrow |\mathtt{T}_0|_{\Delta}}$$

[1] Another solution to this problem is to introduce *bridge methods* (Bracha *et al.* 1998), which exploit method overloading, not modeled here.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

$$\frac{\texttt{class C} \lhd \texttt{D \{ .. class E \{ .. } \overline{\texttt{M}}\texttt{\} .. \}} \qquad \mathit{mtype}(\texttt{m}, \texttt{D.E}) \text{ undefined} \qquad \texttt{<}\overline{\texttt{X}}\lhd\overline{\texttt{C}}\texttt{>T}_0\ \texttt{m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \in \overline{\texttt{M}} \qquad \Delta = \overline{\texttt{X}}\texttt{<:}\overline{\texttt{C}}}{\mathit{mtypemax}(\texttt{m}, \texttt{C.E}) = |\overline{\texttt{T}}|_{\Delta,\texttt{C.E}} \to |\texttt{T}_0|_{\Delta,\texttt{C.E}}}$$

$$\frac{\texttt{class C} \lhd \texttt{D \{ .. \}} \qquad \mathit{mtype}(\texttt{m}, \texttt{D.E}) = \texttt{<}\overline{\texttt{X}}\lhd\overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \to \texttt{T}_0}{\mathit{mtypemax}(\texttt{m}, \texttt{C.E}) = \mathit{mtypemax}(\texttt{m}, \texttt{D.E})}$$

Now, we define the erasure $|\texttt{e}|_{\Delta,\Gamma,\texttt{A}}$ of an expression e with respect to type environment Γ, bound environment Δ, and enclosing class A, by the following rules (here, e is assumed to be well typed under Δ, Γ, and A):

$$|\texttt{x}|_{\Delta,\Gamma,\texttt{A}} = \texttt{x} \tag{E-Var}$$

$$\frac{\Delta;\Gamma;\texttt{A} \vdash \texttt{e}_0\texttt{.f} : \texttt{T} \qquad \Delta;\Gamma;\texttt{A} \vdash \texttt{e}_0 : \texttt{T}_0 \qquad \mathit{fieldsmax}(\widehat{\Delta}(\texttt{T}_0@\texttt{A}))(\texttt{f}) = |\texttt{T}|_{\Delta,\texttt{A}}}{|\texttt{e}_0\texttt{.f}|_{\Delta,\Gamma,\texttt{A}} = |\texttt{e}_0|_{\Delta,\Gamma,\texttt{A}}\texttt{.f}} \tag{E-Field}$$

$$\frac{\Delta;\Gamma;\texttt{A} \vdash \texttt{e}_0\texttt{.f} : \texttt{T} \qquad \Delta;\Gamma;\texttt{A} \vdash \texttt{e}_0 : \texttt{T}_0 \qquad \mathit{fieldsmax}(\widehat{\Delta}(\texttt{T}_0@\texttt{A}))(\texttt{f}) \neq |\texttt{T}|_{\Delta,\texttt{A}}}{|\texttt{e}_0\texttt{.f}|_{\Delta,\Gamma,\texttt{A}} = (|\texttt{T}|_{\Delta,\texttt{A}})|\texttt{e}_0|_{\Delta,\Gamma,\texttt{A}}\texttt{.f}} \tag{E-Field-Cast}$$

$$\frac{\Delta;\Gamma;\texttt{A} \vdash \texttt{e}_0\texttt{.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} : \texttt{T} \qquad \Delta;\Gamma;\texttt{A} \vdash \texttt{e}_0 : \texttt{T}_0 \qquad \mathit{mtypemax}(\texttt{m}, \widehat{\Delta}(\texttt{T}_0@\texttt{A})) = \overline{\texttt{C}} \to \texttt{C} \qquad \texttt{C} = |\texttt{T}|_{\Delta,\texttt{A}}}{|\texttt{e}_0\texttt{.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma,\texttt{A}} = |\texttt{e}_0|_{\Delta,\Gamma,\texttt{A}}\texttt{.m(}|\overline{\texttt{e}}|_{\Delta,\Gamma,\texttt{A}}\texttt{)}} \tag{E-Invk}$$

$$\frac{\Delta;\Gamma;\texttt{A} \vdash \texttt{e}_0\texttt{.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} : \texttt{T} \qquad \Delta;\Gamma;\texttt{A} \vdash \texttt{e}_0 : \texttt{T}_0 \qquad \mathit{mtypemax}(\texttt{m}, \widehat{\Delta}(\texttt{T}_0@\texttt{A})) = \overline{\texttt{C}} \to \texttt{C} \qquad \texttt{C} \neq |\texttt{T}|_{\Delta,\texttt{A}}}{|\texttt{e}_0\texttt{.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma,\texttt{A}} = (|\texttt{T}|_{\Delta,\texttt{A}})|\texttt{e}_0|_{\Delta,\Gamma,\texttt{A}}\texttt{.m(}|\overline{\texttt{e}}|_{\Delta,\Gamma,\texttt{A}}\texttt{)}} \tag{E-Invk-Cast}$$

$$|\texttt{new A}_0\texttt{(}\overline{\texttt{e}}\texttt{)}|_{\Gamma,\Delta,\texttt{A}} = \texttt{new}\ |\texttt{A}_0|(|\overline{\texttt{e}}|_{\Delta,\Gamma,\texttt{A}}) \tag{E-New}$$

There are two rules for field access; the main difference is in whether or not typecasts are inserted. The rule E-Field-Cast is applied when a field type in the source program is a relative path type and the field is declared in a (proper) superclass of A. For example, consider `e.src` in a method defined in `ColorWeightGraph.Edge`, where e is of type `.Edge`. Then, the type of `e.src` (in the source program) is `.Node`, which becomes `ColorWeightGraph$Node` after erasure, whereas the field `src` declared in `Graph.Edge` is given type `Graph$Node` in the erased program. So, the typecast `(ColorWeightGraph$Node)` will be required to access new members, say field `color`, declared in the subclass. Similarly for method invocations.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

4.3 Erasure of methods, constructors, and classes

As mentioned above, family-polymorphic methods are translated to monomorphic methods by discarding type parameterization and erasing types and the method body. Although erased methods no longer maintain type variables, they can be applied to different families, thanks to subtyping between erased nested classes. The erasure of a method `M` with respect to class `A`, written $|\texttt{M}|_{\texttt{A}}$, is defined as follows:

$$\frac{\Gamma = \overline{\texttt{x}}{:}\overline{\texttt{T}}, \texttt{this}{:}\mathit{thistype}(\texttt{A}) \qquad \Delta = \overline{\texttt{X}}{<:}\overline{\texttt{C}} \qquad \mathit{mtypemax}(\texttt{m}, \texttt{A}) = \overline{\texttt{D}}{\rightarrow}\texttt{D} \qquad \texttt{e}_i = \begin{cases} \texttt{x}_i{}' & \text{if } \texttt{D}_i = |\texttt{T}_i|_{\Delta,\texttt{A}} \\ (|\texttt{T}_i|_{\Delta,\texttt{A}})\,\texttt{x}_i{}' & \text{otherwise} \end{cases}}{|\texttt{<}\overline{\texttt{X}}\texttt{<}\overline{\texttt{C}}\texttt{>T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e}_0\texttt{; \}}|_{\texttt{A}} = \texttt{D m(}\overline{\texttt{D}}\ \overline{\texttt{x}}'\texttt{)\{ return } [\overline{\texttt{e}}/\overline{\texttt{x}}]|\texttt{e}_0|_{\Delta,\Gamma,\texttt{A}}\texttt{; \}}} \quad \text{(E-Method)}$$

The method body is erased with respect to bound environment Δ taken from the parameterization, type environment Γ taken from the formal parameter declarations, and enclosing class `A`. Note that the signature of the erased method is obtained by *mtypemax*(`m`, `A`). Downcasts are inserted before each occurrence of the references to the parameters when the erasure $|\texttt{T}_i|_{\Delta,\texttt{A}}$ of the parameter type is different from the corresponding argument type from *mtypemax*. It is required for the same reason as erasure of field access or method invocation.

The erasure $|\texttt{K}|_{\texttt{A}}$ of a constructor `K` of class `A` is fairly straightforward:

$$\begin{aligned} &|\texttt{C(}\overline{\texttt{U}}\ \overline{\texttt{g}}\texttt{, }\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)\{ super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{; \}}|_{\texttt{A}} \\ &= |\texttt{A}|\,(\mathit{fieldsmax}(\texttt{A}))\texttt{\{ super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{; \}} \end{aligned} \quad \text{(E-Constructor)}$$

Argument types are replaced with erased field types and the name of the constructor becomes the erasure of `A` (rather than `C`, in case `K` is a constructor of a nested class).

The erasure $|\texttt{N}|_{\texttt{C}}$ of a nested class `N` in `C` is also straightforward:

$$\frac{\texttt{class C} \lhd \texttt{D \{ .. \}} \qquad \emptyset; \texttt{C} \vdash \texttt{D.E ok}}{|\texttt{class E \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}}|_{\texttt{C}} = \texttt{class C\$E} \lhd \texttt{D\$E \{ }|\overline{\texttt{T}}|_{\texttt{C.E}}\ \overline{\texttt{f}}\texttt{; }|\texttt{K}|_{\texttt{C.E}}\ |\overline{\texttt{M}}|_{\texttt{C.E}}\texttt{\}}} \quad \text{(E-NestedClass1)}$$

$$\frac{\texttt{class C} \lhd \texttt{D \{ .. \}} \qquad \emptyset; \texttt{C} \nvdash \texttt{D.E ok}}{|\texttt{class E \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}}|_{\texttt{C}} = \texttt{class C\$E} \lhd \texttt{Object \{ }|\overline{\texttt{T}}|_{\texttt{C.E}}\ \overline{\texttt{f}}\texttt{; }|\texttt{K}|_{\texttt{C.E}}\ |\overline{\texttt{M}}|_{\texttt{C.E}}\texttt{\}}} \quad \text{(E-NestedClass2)}$$

The erasure of a nested class declaration consists of the erasures of its fields, constructor and method declarations. Since superclasses are not explicit in the source program, translation has to recover it. The rule E-NestedClass1 is applied if its superclass exists (i.e., `D.E` is a well-formed type); the clause "$\lhd$ `D$E`" is inserted; otherwise, "$\lhd$ `Object`" is inserted (E-NestedClass2).

Erasing top-level classes is slightly more involved than one might have expected, because they may contain inherited nested classes, which do not explicitly appear

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

in the source program. To collect all nested classes—whether they are explicit or implicit—we define another auxiliary function *nestedclasses*(C), defined as follows:

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

$$\frac{\texttt{class C} \lhd \texttt{D \{ .. } \overline{\texttt{N}}\texttt{\}} \qquad nestedclasses(\texttt{D}) = \overline{\texttt{N}}' \qquad \overline{\texttt{N}}'' = \{\texttt{class E \{ E(}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)\{super(}\overline{\texttt{f}}\texttt{);\}\}} \mid \texttt{E} \in \overline{\texttt{N}}', \texttt{E} \notin \overline{\texttt{N}}, fields(\texttt{D.E}) = \overline{\texttt{T}}\ \overline{\texttt{f}}\}}{nestedclasses(\texttt{C}) = \overline{\texttt{N}}, \overline{\texttt{N}}''}$$

Here, E represents the name of a class that is inherited from D but not redefined in C. For such names, the empty class definition (consisting of only its constructor) is generated.

With the help of *nestedclasses*, the erasure |L| of a top-level class L is defined as follows:

$$\frac{\overline{\texttt{N}}' = nestedclasses(\texttt{C})}{|\texttt{class C} \lhd \texttt{D \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K}\ \overline{\texttt{M}}\ \overline{\texttt{N}}\texttt{\}}| = \texttt{class C} \lhd \texttt{D \{}\ |\overline{\texttt{T}}|_{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}\ |\texttt{K}|_{\texttt{C}}\ |\overline{\texttt{M}}|_{\texttt{C}}\texttt{\}}\ |\overline{\texttt{N}}'|_{\texttt{C}}} \quad \text{(E-TopClass)}$$

Note that it returns a set of FJ classes since nested classes are pulled out to the top level.

Another possible way to erase nested classes without generating empty classes for implicitly inherited classes is to erase the name of a nested class, not declared explicitly, to the name of the first superclass that explicitly appears in the source program. For example, `new ColorWeightGraph.Node(..)` would be erased to `new Graph$Node(..)` if `ColorWeightGraph` did not have `Node`. Although this somewhat optimized compilation scheme would work in this setting, too, we do not, however, take this approach since it would not always be sensible to change class names, for example, in the presence of reflection.

We write $|CT|$ for the (FJ) class table consisting of classes obtained by erasing all top-level classes in CT.

4.4 Properties of the translation

Now, we investigate correctness of the erasure translation in terms of typing and reduction. Proofs of the theorems are found in Appendix B. We will distinguish reduction steps involving R-Cast and other steps (i.e., field access and method invocation) by writing the former as $\texttt{e} \longrightarrow_c \texttt{e}'$, the latter as $\texttt{e} \longrightarrow_n \texttt{e}'$. Their union is written $\texttt{e} \longrightarrow_{\text{FJ}} \texttt{e}'$.

The following theorem says that a well-typed .FJ program is erased to a well-typed FJ program, as expected.

Theorem 4.1 (*Erasure Preserves Typing*)
If a .FJ class table CT is ok and $\Delta; \Gamma; \texttt{A} \vdash \texttt{e}:\texttt{T}$, then $|CT|$ is ok using the FJ typing rules and $|\Gamma|_{\Delta,\texttt{A}} \vdash_{\text{FJ}} |\texttt{e}|_{\Delta,\Gamma,\texttt{A}} : |\texttt{T}|_{\Delta,\texttt{A}}$.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

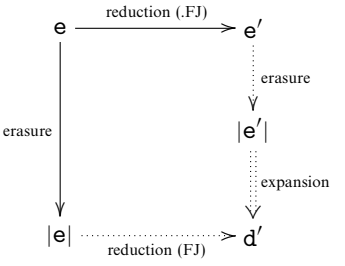

Fig. 10. Commuting diagram of Theorem 4.2.

Then, we will show that the evaluation of a .FJ program agrees with that of its erasure, in the sense that, if a .FJ expression `e` is reduced to a value, then the erasure of `e` will also be reduced to the erasure of that value and *vice versa*. Unfortunately, however, one step reduction does not really commute with erasure translation. One obvious reason is that erased execution takes more steps to remove inserted typecasts. Actually, there is a more subtle reason: one step reduction on .FJ does not preserve typecasts inserted by erasure. The same kind of problem is also observed in erasure translation from FGJ to FJ (Igarashi *et al.* 2001)—to which interested readers are referred for more detailed analysis of the problem. To solve the problem, we follow the same approach of using *expansions*, which relate two expressions with similar structures but different typecasts.

Suppose $\Gamma \vdash_{\mathrm{FJ}}$ `e:C`. We call an expression `d` an *expansion* of `e` under Γ, written $\Gamma \vdash$ `e` $\stackrel{\exp}{\Longrightarrow}$ `d`, if $\Gamma \vdash_{\mathrm{FJ}}$ `d:D` for some `D` and `d` is obtained from `e` by some combination of (1) addition of zero or more upcasts, (2) replacement of some casts `(D)` with `(C)`, where `C` is a supertype of `D`, or (3) removal of some casts.

Example 4.1
Suppose that $\Gamma =$ `x:ColorWeightGraph$Node, y:ColorWeightGraph$Edge`. Then,

$$\Gamma \vdash \texttt{x} \stackrel{\exp}{\Longrightarrow} \texttt{(Graph\$Node)x}$$

and

$$\Gamma \vdash \texttt{x} \stackrel{\exp}{\Longrightarrow} \texttt{(Graph\$Node)(Object)x}$$

and

$$\Gamma \vdash \texttt{y.connect((ColorWeightGraph\$Node)x,(ColorWeightGraph\$Node)x)} \stackrel{\exp}{\Longrightarrow} \texttt{y.connect(x, x)}\ .$$

Note that inserted upcasts may become downcasts due to other inserted casts, as in the second example, where `(Object)` is inserted after `(Graph$Node)` is inserted.

Then, first, one step reduction commutes with erasure modulo expansion, as illustrated in Figure 10 (we assume that *CT* ok in the following three theorems).

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

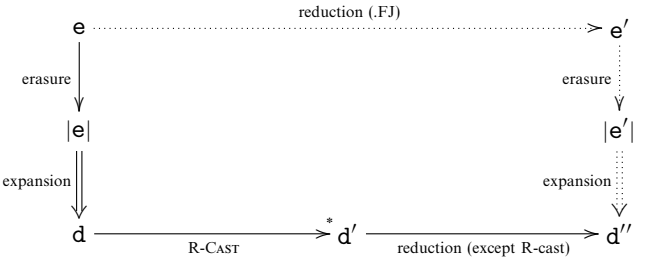

Fig. 11. Commuting diagram of Theorem 4.3.

Theorem 4.2 (*Erasure Preserves Reduction Modulo Expansion*)
If $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}:\mathtt{T}$ and $\mathtt{e} \longrightarrow \mathtt{e}'$, then there exists some FJ expression $\mathtt{d}'$ such that $|\Gamma|_{\Delta,\mathtt{A}} \vdash |\mathtt{e}'|_{\Delta,\Gamma,\mathtt{A}} \overset{\text{exp}}{\Longrightarrow} \mathtt{d}'$ and $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} \longrightarrow_{\text{FJ}} \mathtt{d}'$.

Conversely, for the execution of an erased expression, there is a corresponding execution in the .FJ semantics, as illustrated in Figure 11.

Theorem 4.3 (*Erased Program Reflects .FJ Execution*)
Suppose that $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}:\mathtt{T}$ and $|\Gamma|_{\Delta,\mathtt{A}} \vdash |\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} \overset{\text{exp}}{\Longrightarrow} \mathtt{d}$. If $\mathtt{d} \longrightarrow_c^* \mathtt{d}'$ and $\mathtt{d}' \longrightarrow_n \mathtt{d}''$, then $\mathtt{e} \longrightarrow \mathtt{e}'$ for some $\mathtt{e}'$ and $|\Gamma|_{\Delta,\mathtt{A}} \vdash |\mathtt{e}'|_{\Delta,\Gamma,\mathtt{A}} \overset{\text{exp}}{\Longrightarrow} \mathtt{d}''$.

Notice that a "real" execution step may be preceded by removal of casts to expose a redex.

Finally, the following theorem states that the evaluation result of a .FJ expression matches that of its erasure.

Theorem 4.4 (*Erasure Preserves Execution Results*)
If $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}:\mathtt{T}$ and $\mathtt{e} \longrightarrow^* \mathtt{v}$, then $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} \longrightarrow_{\text{FJ}}^* |\mathtt{v}|_{\Delta,\Gamma,\mathtt{A}}$. Similarly, if $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}:\mathtt{T}$ and $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} \longrightarrow_{\text{FJ}}^* \mathtt{v}$, then there exists a .FJ value $\mathtt{v}'$ such that $\mathtt{e} \longrightarrow^* \mathtt{v}'$ and $|\mathtt{v}'|_{\Delta,\Gamma,\mathtt{A}} = \mathtt{v}$.

5 Related work

As we have already mentioned, in the original formulation of family polymorphism (Ernst 2001) nested classes are members (or attributes) of an object of their enclosing class. Thus, to create node or edge objects, one first has to instantiate `Graph` and then to invoke `new` on a class attribute of that object. It would be written as

```
final Graph g = new Graph();
g.Node n = new g.Node(.. );  g.Edge e = new g.Edge(.. );
```

Notice that the types of nodes and edges would include a reference `g` to the `Graph` object. Relative path types `.Node` and `.Edge` would become `this.Node` and `this.Edge`, respectively, where the meaning of types changes as the meaning of `this` changes because of usual late-binding. Finally, `connectAll()` would take four *value* arguments instead of one type and three value arguments as follows:

```
void connectAll(final Graph g, g.Edge[] es, g.Node n1, g.Node n2){ .. }
```

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Notice that the first argument appears as part of types of the following arguments; it is required for a type system to guarantee that `es`, `n1`, and `n2` belong to the same graph. As a result, a type system is equipped with dependent types, such as `g.Edge`, which can be quite tricky (especially in the presence of side effects). For example, as the `final` modifier indicates, the family object `g` is required to be immutable. We deliberately avoid such dependent types by identifying families with classes. As a by-product, as shown in Section 3.6, we have discovered that it is easy to extend to this setting the GJ-style type inference, which lessens the burden of programmers to specify the argument that only represents a family (like `g` or `g.class` above). Although complex, the original approach has one apparent advantage: one can instantiate an arbitrary number of `Graph` objects and distinguish nodes and edges of different graphs by static typing. Formal accounts of such type systems can be found in the literature (Ernst *et al.* 2006; Clarke *et al.* 2007). Scala (Odersky *et al.* 2003) also supports family polymorphism, based on dependent types. JX (Nystrom *et al.* 2004) also uses `final` and dependent types, even though it is also based on the "classes-as-families" principle—in JX, `connectAll()` would be written

```
void connectAll(final Graph g,
                g.class.Edge[] es, g.class.Node n1, g.class.Node n2){ .. }
```

where `g.class` is used for `g` to denote the run-time class of `g`.

Historically, the mismatching problem of recursive class definitions has been studied in the context of binary methods (Bruce *et al.* 1996), which take an object of the same class as the receiver; hence, the interface is (self-)recursive. In particular, Bruce's series of work (1997, 2004) introduced the notion of *MyType* (or sometimes called *ThisType*), which is the type of `this` and changes its meaning along the inheritance chain, just as our relative path types. Later, he extended the idea to mutually recursive type/class definitions (Bruce *et al.* 1998; Bruce & Vanderwaart 1999; Bruce 2003) by introducing constructs to group mutually recursive definitions, and the notion of *MyGroup*, which is a straightforward extension of *MyType* to mutually recursion. While Bruce and his colleagues mainly focus on structural type systems (except for Bruce *et al.* 1998), Jolly *et al.* (2004) have designed the language called Concord by applying the notion of groups to a Java-like language with a name-based type system and have proved that the core type system is sound. Our approach is similar to them in the sense that dependent types are not used.[2]

One of our contributions in this work (over the work mentioned above) is (formal) modeling of family-polymorphic methods, which are absent from Concord, as a special form of parametric methods. Note that they can be considered a generalization of match-bound polymorphic methods in the language $\mathscr{LOOM}$ (Bruce *et al.* 1997) to mutually recursion—in fact, a similar idea has already been mentioned in Bruce *et al.* (1998) but never formalized.

Another contribution would be in how types are classified into exact types, which denote instances of one particular class and play a crucial role to achieve type safety

[2] By dependent types, we mean types that are dependent on *values*; in this sense, Concord does not use dependent types, contrary to the title of the paper.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

of extensible (mutually) recursive classes, and inexact types, which denote instances of one class and its subclasses as usual. In Bruce *et al.* (1998, 1997), one class name is used for both exact and inexact types, which are distinguished by a special marker. This approach brings a problem of which kind of types to use as a default—in fact, in Bruce *et al.* (1998), a class name without a marker is used as an inexact types (for compatibility with Java, on which their work is based) and a class name with `@` is used for the exact type for that class, but, in $\mathscr{LOOM}$, the default is exact and `#` is used for inexact. Instead of introducing special markers, we have designed our language such that top-level class names are always inexact and nested class names are always exact, resulting in a language much easier to understand.

In addition to simultaneous extension of nested classes, Concord, `gbeta`, and JX allow inheritance relations between nested classes. For example, `C.F` can be a subclass of `C.E` and, in a subclass `D` of `C`, the relationship is preserved while members can be added to both `E` and `F`. Although allowing virtual superclasses is useful to solve the "expression problem," we have carefully avoided this feature, too, which is not strongly required by family polymorphism—there is a semantic complication as in languages with multiple inheritance: `D.F` may inherit conflicting members of the same name from `C.F` and `D.E`.

Finally, we should note that programming described in Section 2 could be carried out in Java 5.0 proper, which is equipped with generics (Bracha *et al.* 1998) and F-bounded polymorphism (Canning *et al.* 1989), by using the technique (Torgersen 2004) used to solve the "expression problem." It requires, however, a lot of boilerplate code for type parameterization, which makes programs less easy to grasp.

6 Concluding remarks

We have identified a minimal set of language features to solve the problem of mismatching between mutually recursive classes and inheritance. Our proposal is lightweight in the sense that the type system, which avoids dependent types, is much simpler than the original formulation of family polymorphism and easy to apply to mainstream languages such as Java and C#. We have shown type safety of the language mechanism by proving a type soundness theorem for the formalized core language .FJ. We have also developed an algorithm for type argument inference for family polymorphic methods with its correctness theorem. Although .FJ is not equipped with generics, we believe they can be easily integrated.

As a possible implementation scheme for lightweight family polymorphism, we have described erasure translation, formalized it as translation from .FJ to Featherweight Java and proved its correctness. A prototype compiler for the language features presented is being implemented on top of `javac`.

Our main aim in this article was to identify a minimal set of language features for lightweight family polymorphism, based on the the principle of classes-as-families. We think, however, that it is also an interesting research question to what extent this principle can be extended to integrate advanced ideas, such as higher-order hierarchies (Ernst 2003) and generalized path types (Clarke *et al.* 2007), found in the work on object-based families. In fact, the second and third authors have developed

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

a new type system by lifting .FJ's restrictions on nesting levels and inheritance between members of the same family and have proposed for class-based families the new notions of exact and inexact qualifications to give programmers expressive subtyping relations over path types (Igarashi & Viroli 2007).

Acknowledgments

Part of the development of the prototype compiler was done by Stefano Olivieri and the third author during their visit to Kyoto University in the summer of 2005. Comments from anonymous referees helped us improve the final presentation. The second author would like to thank members of the Kumiki project for fruitful discussions on this subject. This work was supported in part by Grant-in-Aid for Scientific Research on Priority Areas Research No. 13224013 from MEXT of Japan (Igarashi), and from the Italian PRIN 2004 Project "Extensible Object Systems" (Viroli).

Appendix A

Proof of Subject Reduction Theorem

In this appendix, we detail our proof of Subject Reduction Theorem (Theorem 3.1). The structure of the proof is similar to those for typed λ-calculi with subtyping and parametric polymorphism and also Featherweight Java (Igarashi *et al.* 2001). So, we first prove various substitution lemmas (Lemmas A.6, A.7, A.8, and A.10). We also prove that resolution preserves typing as Lemma A.9, which amounts to say that relative path types are polymorphic over types in the inheritance relation.

We begin with developing a number of required lemmas. In what follows, the metavariables $\mathtt{V}$ and $\mathtt{W}$ range over types as well as $\mathtt{S}$, $\mathtt{T}$, and $\mathtt{U}$.

Lemma A.1 (*Weakening*)

1. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$, then $\Delta, \overline{\mathtt{X}}<:\overline{\mathtt{C}} \vdash \mathtt{S} <: \mathtt{T}$.
2. If $\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e} : \mathtt{T}$, then $\Delta; \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{T}}; \mathtt{A} \vdash \mathtt{e} : \mathtt{T}$.
3. If $\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e} : \mathtt{T}$, then $\Delta, \overline{\mathtt{X}}<:\overline{\mathtt{C}}; \Gamma; \mathtt{A} \vdash \mathtt{e} : \mathtt{T}$.

Proof
Straightforward induction on derivations. □

Lemma A.2 (*Properties of resolution*)

1. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\mathtt{U}@\mathtt{S} = \mathtt{V}$, then $\mathtt{U}@\mathtt{T} = \mathtt{V}$.
2. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$, then $\Delta \vdash \mathtt{S}@\mathtt{A} <: \mathtt{T}@\mathtt{A}$.
3. $(\mathtt{T}@\mathtt{S})@\mathtt{A} = \mathtt{T}@(\mathtt{S}@\mathtt{A})$.
4. $[\overline{\mathtt{P}}/\overline{\mathtt{X}}](\mathtt{T}@\mathtt{S}) = ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T})@([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S})$.

Proof
(1) is immediate since, if $\mathtt{S} \neq \mathtt{T}$, then $\mathtt{U}$ must be of the form $\mathtt{P}$ or $\mathtt{P.C}$, so $\mathtt{U}@\mathtt{S} = \mathtt{U}@\mathtt{T} = \mathtt{U}$. (2) uses the fact that, if $\mathtt{S}$ is a relative path type, then either $\mathtt{S} = \mathtt{T}$ or $\mathtt{T} = \mathtt{Object}$. (3) and (4) are easily shown by case analysis on $\mathtt{S}$ and $\mathtt{T}$. □

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

The next two lemmas state that if one class inherits from another, then the field/method types from both classes indeed agree. We write $\mathtt{A}$ <# $\mathtt{B}$ if either (1) $\mathtt{A} = \mathtt{C}$, $\mathtt{B} = \mathtt{D}$, and $\vdash \mathtt{C} <: \mathtt{D}$, or (2) $\mathtt{A} = \mathtt{C.E}$, $\mathtt{B} = \mathtt{D.E}$, and $\vdash \mathtt{C} <: \mathtt{D}$.

Lemma A.3
If $\mathtt{A}$ <# $\mathtt{B}$ and $\mathit{fields}(\mathtt{B}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, then $\mathit{fields}(\mathtt{A}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}, \overline{\mathtt{S}}\ \overline{\mathtt{g}}$ for some $\overline{\mathtt{S}}\ \overline{\mathtt{g}}$.

Proof
By induction on $\Delta \vdash \mathtt{C} <: \mathtt{D}$, where $\mathtt{C}$ and $\mathtt{D}$ are the top-level class name (prefix) of $\mathtt{A}$ and $\mathtt{B}$, respectively. □

Lemma A.4
If $\mathtt{A}$ <# $\mathtt{B}$ and $\mathit{mtype}(\mathtt{m}, \mathtt{B}) = <\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}>\overline{\mathtt{U}} \to \mathtt{U}_0$, then $\mathit{mtype}(\mathtt{m}, \mathtt{A}) = <\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}>\overline{\mathtt{U}} \to \mathtt{U}_0$.

Proof
Similar to Lemma A.3 □

Lemma A.5
If $\Delta; \overline{\mathtt{x}}:\overline{\mathtt{A}}; \mathtt{B} \vdash \mathtt{e}_0 : \mathtt{T}_0$, then $\mathtt{T}_0$ is an absolute class name and $\Delta; \overline{\mathtt{x}}:\overline{\mathtt{A}}; \mathtt{B}' \vdash \mathtt{e}_0 : \mathtt{T}_0$ for any $\mathtt{B}'$.

Proof
By induction on $\Delta; \overline{\mathtt{x}}:\overline{\mathtt{A}}; \mathtt{B} \vdash \mathtt{e}_0 : \mathtt{T}_0$. Note that $\mathtt{A@T} = \mathtt{A@S}$ for any $\mathtt{S}$ and $\mathtt{T}$. □

The following lemmas show that type substitution preserves subtyping, type well-formedness, and typing, and that resolution preserves typing. Then, we prove that substitution of expressions preserves typing.

Lemma A.6 (*Type Substitution Preserves Subtyping*)
If $\Delta, \overline{\mathtt{X}}<:\overline{\mathtt{C}} \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta; \mathtt{A} \vdash \overline{\mathtt{P}}$ ok and $\Delta \vdash \overline{\mathtt{P}}<:\overline{\mathtt{C}}$, then $\Delta \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S} <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}$.

Proof
By induction on $\Delta, \overline{\mathtt{X}}<:\overline{\mathtt{C}} \vdash \mathtt{S} <: \mathtt{T}$. □

Lemma A.7 (*Type Substitution Preserves Type Well-Formedness*)
If $\Delta, \overline{\mathtt{X}}<:\overline{\mathtt{C}}; \mathtt{A} \vdash \mathtt{T}$ ok and $\Delta; \mathtt{A} \vdash \overline{\mathtt{P}}$ ok and $\Delta \vdash \overline{\mathtt{P}}<:\overline{\mathtt{C}}$, then $\Delta; \mathtt{A} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}$ ok.

Proof
By induction on $\Delta, \overline{\mathtt{X}}<:\overline{\mathtt{C}}; \mathtt{A} \vdash \mathtt{T}$ ok. □

Lemma A.8 (*Type Substitution Preserves Typing*)
If $\Delta, \overline{\mathtt{X}}<:\overline{\mathtt{C}}; \Gamma; \mathtt{A} \vdash \mathtt{e} : \mathtt{T}$, and $\Delta; \mathtt{A} \vdash \overline{\mathtt{P}}$ ok and $\Delta \vdash \overline{\mathtt{P}} <: \overline{\mathtt{C}}$, then there exists $\mathtt{S}$ such that $\Delta; [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\Gamma; \mathtt{A} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e} : \mathtt{S}$ and $\Delta \vdash \mathtt{S} <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}$.

Proof
By induction on $\Delta'; \Gamma; \mathtt{A} \vdash \mathtt{e} : \mathtt{T}$, where we let $\Delta' = \Delta, \overline{\mathtt{X}}<:\overline{\mathtt{C}}$, with case analysis on the last rule used.

Case T-VAR**:**

Immediate.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Case T-FIELD: $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i$ $\quad \Delta';\Gamma;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0$
$\mathit{fields}(\widehat{\Delta}'(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ $\quad \mathtt{T} = \mathtt{T}_i@\mathtt{T}_0$

By the induction hypothesis, there exists $\mathtt{T}_0'$ such that $\Delta;[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\Gamma;\mathtt{A} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0 : \mathtt{T}_0'$ and $\Delta \vdash \mathtt{T}_0' <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0$. By Lemma A.3 and the fact that $\widehat{\Delta}(([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0)@\mathtt{A}) <\# \widehat{\Delta}'(\mathtt{T}_0@\mathtt{A})$, we have $\mathit{fields}(\widehat{\Delta}(\mathtt{T}_0'@\mathtt{A})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}, \overline{\mathtt{U}}\ \overline{\mathtt{g}}$ for some $\overline{\mathtt{U}}\ \overline{\mathtt{g}}$. By Lemma A.2(1), $\Delta \vdash \mathtt{T}_i@\mathtt{T}_0' <: \mathtt{T}_i@[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0$. Finally, Lemma A.2(4) gives $\mathtt{T}_i@([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0) = [\overline{\mathtt{P}}/\overline{\mathtt{X}}](\mathtt{T}_i@\mathtt{T}_0)$ (note that $\mathtt{T}_i$ does not contain $\overline{\mathtt{X}}$).

Case T-INVK: $\mathtt{e} = \mathtt{e}_0.\mathtt{<}\overline{\mathtt{Q}}\mathtt{>m}(\overline{\mathtt{e}})$ $\quad \Delta';\Gamma;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0$
$\mathit{mtype}(\mathtt{m}, \widehat{\Delta}'(\mathtt{T}_0@\mathtt{A})) = \mathtt{<}\overline{\mathtt{Y}} \mathtt{\triangleleft} \overline{\mathtt{D}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0$ $\quad \Delta';\mathtt{A} \vdash \overline{\mathtt{Q}}$ ok
$\Delta' \vdash \overline{\mathtt{Q}} <: \overline{\mathtt{D}}$ $\quad \Delta';\Gamma;\mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$
$\Delta' \vdash \overline{\mathtt{T}} <: ([\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}})@\mathtt{T}_0$ $\quad \mathtt{T} = ([\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@\mathtt{T}_0$

By the induction hypothesis, there exists $\mathtt{T}_0'$ such that $\Delta;[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\Gamma;\mathtt{A} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0 : \mathtt{T}_0'$ and $\Delta \vdash \mathtt{T}_0' <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0$ and there exist $\overline{\mathtt{T}}'$ such that $\Delta;[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\Gamma;\mathtt{A} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{e}} : \overline{\mathtt{T}}'$ and $\Delta \vdash \overline{\mathtt{T}}' <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}$. Then, by Lemma A.4, $\mathit{mtype}(\mathtt{m}, \widehat{\Delta}(\mathtt{T}_0'@\mathtt{A})) = \mathtt{<}\overline{\mathtt{Y}} \mathtt{\triangleleft} \overline{\mathtt{D}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0$. By Lemmas A.6 and A.7 and S-TRANS, $\Delta;\mathtt{A} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}$ ok and $\Delta \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}} <: \overline{\mathtt{D}}$ and

$$\begin{array}{lll} \Delta \vdash \overline{\mathtt{T}}' <: & [\overline{\mathtt{P}}/\overline{\mathtt{X}}](([\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}})@\mathtt{T}_0) & \\ = & (([[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}})@[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0) & \text{(by Lemma A.2(4))} \\ = & (([[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}})@\mathtt{T}'_0) & \text{(by Lemma A.2(1)).} \end{array}$$

The rule T-INVK finishes the case.

Case T-NEW: $\mathtt{e} = \mathtt{new}\ \mathtt{A}_0(\overline{\mathtt{e}})$ $\quad \mathit{fields}(\mathtt{A}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ $\quad \Delta';\Gamma;\mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}}$
$\Delta' \vdash \overline{\mathtt{U}} <: \overline{\mathtt{T}}@\mathtt{A}_0$

By the induction hypothesis, there exist $\overline{\mathtt{U}}'$ such that $\Delta;[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\Gamma;\mathtt{A} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{e}} : \overline{\mathtt{U}}'$ and $\Delta \vdash \overline{\mathtt{U}}' <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}$. By Lemmas A.6 and A.2(4), $\Delta \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}} <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}](\overline{\mathtt{T}}@\mathtt{A}_0) = \overline{\mathtt{T}}@\mathtt{A}_0$. Finally, S-TRANS and T-NEW finish the case. □

Lemma A.9 (*Resolution Preserves Typing*)
If $\Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{B} \vdash \mathtt{e} : \mathtt{T}$ and $\mathtt{A} <\# \mathtt{B}$, then $\Delta;\overline{\mathtt{x}} : \overline{\mathtt{T}}@\mathtt{A};\mathtt{B} \vdash \mathtt{e} : \mathtt{T}@\mathtt{A}$.

Proof
By induction on $\Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{B} \vdash \mathtt{e} : \mathtt{T}$ with case analysis on the last rule used.

Case T-VAR:

Trivial.

Case T-FIELD: $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i$ $\quad \Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{B} \vdash \mathtt{e}_0 : \mathtt{T}_0$
$\mathit{fields}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{B})) = \overline{\mathtt{S}}\ \overline{\mathtt{f}}$ $\quad \mathtt{T} = \mathtt{S}_i@\mathtt{T}_0$

By the induction hypothesis, we have $\Delta;\overline{\mathtt{x}} : \overline{\mathtt{T}}@\mathtt{A};\mathtt{B} \vdash \mathtt{e}_0 : \mathtt{T}_0@\mathtt{A}$. By Lemma A.3, $\mathit{fields}(\widehat{\Delta}((\mathtt{T}_0@\mathtt{A})@\mathtt{B})) = \mathit{fields}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{S}}\ \overline{\mathtt{f}}, \overline{\mathtt{U}}\ \overline{\mathtt{g}}$ for some $\overline{\mathtt{U}}\ \overline{\mathtt{g}}$. Then, by T-FIELD, $\Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}}@\mathtt{A};\mathtt{B} \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{S}_i@(\mathtt{T}_0@\mathtt{A})$. Finally, Lemma A.2(3) gives $\mathtt{S}_i@(\mathtt{T}_0@\mathtt{A}) = (\mathtt{S}_i@\mathtt{T}_0)@\mathtt{A}$.

Case T-INVK: $\mathtt{e} = \mathtt{e}_0.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m}(\overline{\mathtt{e}})$ $\quad \Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{B} \vdash \mathtt{e}_0 : \mathtt{T}_0$
$\mathit{mtype}(\mathtt{m}, \widehat{\Delta}(\mathtt{T}_0@\mathtt{B})) = \mathtt{<}\overline{\mathtt{X}} \mathtt{\triangleleft} \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0$ $\quad \Delta;\mathtt{B} \vdash \overline{\mathtt{P}}$ ok
$\Delta \vdash \overline{\mathtt{P}} <: \overline{\mathtt{C}}$ $\quad \Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{B} \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}$
$\Delta \vdash \overline{\mathtt{S}} <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@\mathtt{T}_0$ $\quad \mathtt{T} = ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U}_0)@\mathtt{T}_0$

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

By the induction hypothesis, $\Delta;\overline{\mathtt{x}} : \overline{\mathtt{T}}@\mathtt{A};\mathtt{B} \vdash \mathtt{e}_0 : \mathtt{T}_0@\mathtt{A}$ and $\Delta;\overline{\mathtt{x}} : \overline{\mathtt{T}}@\mathtt{A};\mathtt{B} \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}@\mathtt{A}$. By Lemma A.4, *mtype*($\mathtt{m}$, $\widehat{\Delta}((\mathtt{T}_0@\mathtt{A})@\mathtt{B})) = \mathit{mtype}(\mathtt{m}, \widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = <\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}>\overline{\mathtt{U}}\to\mathtt{U}_0$. By Lemmas A.2(2) and A.2(3), $\Delta \vdash \overline{\mathtt{S}}@\mathtt{A} <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@(\mathtt{T}_0@\mathtt{A})$. By T-Invk, $\Delta;\overline{\mathtt{x}} : \overline{\mathtt{T}}@\mathtt{A};\mathtt{B} \vdash \mathtt{e}_0.<\overline{\mathtt{P}}>\mathtt{m}(\overline{\mathtt{e}}) : ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U}_0)@(\mathtt{T}_0@\mathtt{A})$. Finally, Lemma A.2(3) gives $([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U}_0)@(\mathtt{T}_0@\mathtt{A}) = (([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U}_0)@\mathtt{T}_0)@\mathtt{A}$.

Case T-New: $\mathtt{e} = \mathtt{new}\ \mathtt{A}_0(\overline{\mathtt{e}})$ $\quad \mathit{fields}(\mathtt{A}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ $\quad \Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{B} \vdash \overline{\mathtt{e}}:\overline{\mathtt{U}}$
$\Delta \vdash \overline{\mathtt{U}} <: \overline{\mathtt{T}}@\mathtt{A}_0$ $\quad \mathtt{T} = \mathtt{A}_0$

By the induction hypothesis, $\Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}}@\mathtt{A};\mathtt{B} \vdash \overline{\mathtt{e}}:\overline{\mathtt{U}}@\mathtt{A}$. By Lemma A.2(2) and the fact that $(\overline{\mathtt{T}}@\mathtt{A}_0)@\mathtt{A} = \overline{\mathtt{T}}@\mathtt{A}_0$, we have $\Delta \vdash \overline{\mathtt{U}}@\mathtt{A} <: \overline{\mathtt{T}}@\mathtt{A}_0$. Finally, since $\mathtt{A}_0@\mathtt{A} = \mathtt{A}_0$, by T-New, $\Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}}@\mathtt{A};\mathtt{B} \vdash \mathtt{new}\ \mathtt{A}_0(\overline{\mathtt{e}}):\mathtt{A}_0$. □

Lemma A.10 (*Expression Substitution Preserves Typing*)
If $\Delta;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{A} \vdash \mathtt{e} : \mathtt{T}$ and $\Delta;\Gamma;\mathtt{A} \vdash \overline{\mathtt{d}} : \overline{\mathtt{S}}$ and $\Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$, then there exists $\mathtt{S}$ such that $\Delta;\Gamma;\mathtt{A} \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{S}$ and $\Delta \vdash \mathtt{S} <: \mathtt{T}$.

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

Case T-Var:

If $\mathtt{e} = \mathtt{x}_i$, then $[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x}_i = \mathtt{d}_i$. By assumption, we have $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{d}_i : \mathtt{S}_i$ and $\Delta \vdash \mathtt{S}_i <: \mathtt{T}_i$. If $\mathtt{e} = \mathtt{y} \in \mathit{dom}(\Gamma)$, then $[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{y} = \mathtt{y}$ and we have $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{y} : \Gamma(\mathtt{y})$ by T-Var.

Case T-Field: $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i$ $\quad \Delta;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0$
$\mathit{fields}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{A}_0)) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$ $\quad \mathtt{T} = \mathtt{U}_i@\mathtt{T}_0$

By the induction hypothesis, there exists $\mathtt{U}_0$ such that $\Delta;\Gamma;\mathtt{A} \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{U}_0$ and $\Delta \vdash \mathtt{U}_0 <: \mathtt{T}_0$. By Lemma A.3, we have $\mathit{fields}(\widehat{\Delta}(\mathtt{U}_0@\mathtt{A})) = \overline{\mathtt{U}}\ \overline{\mathtt{f}},\overline{\mathtt{U}'}\ \overline{\mathtt{f}'}$ for some $\overline{\mathtt{U}'}\ \overline{\mathtt{f}'}$. Then, T-Field and Lemma A.2(1), $\Delta;\Gamma;\mathtt{A} \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0.\mathtt{f}_i : \mathtt{U}_i@\mathtt{T}_0 (= \mathtt{U}_i@\mathtt{U}_0)$.

Case T-Invk: $\mathtt{e} = \mathtt{e}_0.<\overline{\mathtt{P}}>\mathtt{m}(\overline{\mathtt{e}})$ $\quad \Delta;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0$
$\mathit{mtype}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = <\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}>\overline{\mathtt{U}}\to\mathtt{U}$ $\quad \Delta;\mathtt{A} \vdash \overline{\mathtt{P}}\ \mathtt{ok}$
$\Delta \vdash \overline{\mathtt{P}} <: \overline{\mathtt{C}}$ $\quad \Delta;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}}$
$\Delta \vdash \overline{\mathtt{V}} <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@\mathtt{T}_0$ $\quad \mathtt{T} = ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U})@\mathtt{T}_0$

By the induction hypothesis, there exist $\mathtt{U}_0$ and $\overline{\mathtt{W}}$ such that

$$\Delta;\Gamma;\mathtt{A} \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{U}_0 \qquad \Delta \vdash \mathtt{U}_0 <: \mathtt{T}_0$$
$$\Delta;\Gamma;\mathtt{A} \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} : \overline{\mathtt{W}} \qquad \Delta \vdash \overline{\mathtt{W}} <: \overline{\mathtt{V}}$$

By Lemma A.4, *mtype*($\mathtt{m}$, $\widehat{\Delta}(\mathtt{U}_0@\mathtt{A})) = <\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}>\overline{\mathtt{U}}\to\mathtt{U}$. By Lemma A.2(1), $\Delta \vdash \overline{\mathtt{V}} <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@\mathtt{U}_0$. By S-Trans, $\Delta \vdash \overline{\mathtt{W}} <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@\mathtt{U}_0$. By T-Invk and Lemma A.2(1), we have $\Delta;\Gamma;\mathtt{A} \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0.<\overline{\mathtt{P}}>\mathtt{m}([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}) : ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U})@\mathtt{U}_0 (= ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U})@\mathtt{T}_0)$.

Case T-New: $\mathtt{e} = \mathtt{new}\ \mathtt{A}_0(\overline{\mathtt{e}})$ $\quad \mathit{fields}(\mathtt{A}_0) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$ $\quad \Delta;\Gamma,\overline{\mathtt{x}}:\overline{\mathtt{T}};\mathtt{A} \vdash \overline{\mathtt{e}}:\overline{\mathtt{V}}$
$\Delta \vdash \overline{\mathtt{V}} <: (\overline{\mathtt{U}}@\mathtt{A}_0)$

By the induction hypothesis, there exist $\overline{\mathtt{W}}$ such that $\Delta;\Gamma;\mathtt{A} \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} : \overline{\mathtt{W}}$ and $\Delta \vdash \overline{\mathtt{W}} <: \overline{\mathtt{V}}$. By transitivity, $\Delta \vdash \overline{\mathtt{W}} <: (\overline{\mathtt{U}}@\mathtt{A}_0)$. By T-New, $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{new}\ \mathtt{A}_0([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}) : \mathtt{A}_0$.
□

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Lemma A.11
If $mtype(\mathtt{m}, \mathtt{A}) = \langle \overline{\mathtt{X}} \lhd \overline{\mathtt{C}} \rangle \overline{\mathtt{S}} \to \mathtt{S}_0$ and $mbody(\mathtt{m}\langle\overline{\mathtt{P}}\rangle, \mathtt{A}) = \overline{\mathtt{x}}.\mathtt{e}$ and $\Delta \vdash \overline{\mathtt{P}} <: \overline{\mathtt{C}}$, then there exist $\mathtt{B}$ and $\mathtt{T}_0$ such that $\mathtt{A} <\# \mathtt{B}$ and $\Delta \vdash \mathtt{T}_0 <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S}_0$ and $\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}, \mathtt{this} : thistype(\mathtt{B}); \mathtt{B} \vdash \mathtt{e} : \mathtt{T}_0$.

Proof
By induction on the derivation of $mbody(\mathtt{m}\langle\overline{\mathtt{P}}\rangle, \mathtt{A}) = \overline{\mathtt{x}}.\mathtt{e}$ with case analysis on the last rule used.

Case MB-TClass**:** $\mathtt{A} = \mathtt{C}$ $\quad$ $\mathtt{class\ C} \lhd \mathtt{D}\ \{.. \overline{\mathtt{M}} ..\}$
$\langle \overline{\mathtt{X}} \lhd \overline{\mathtt{C}} \rangle \mathtt{S}_0\ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}_0;\ \} \in \overline{\mathtt{M}}$

By T-Class and T-Method, there exists $\mathtt{T}'$ such that $\overline{\mathtt{X}} <: \overline{\mathtt{C}}; \overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{this} : \mathtt{C}; \mathtt{C} \vdash \mathtt{e} : \mathtt{T}'$ and $\overline{\mathtt{X}} <: \overline{\mathtt{C}} \vdash \mathtt{T}' <: \mathtt{S}_0$. By Lemmas A.1 and A.8, there exists $\mathtt{T}''$ such that $\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}, \mathtt{this} : \mathtt{C}; \mathtt{C} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0 : \mathtt{T}''$ and $\Delta \vdash \mathtt{T}'' <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}'$. By Lemma A.6 and S-Trans, $\Delta \vdash \mathtt{T}'' <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S}_0$.

Case MB-TSuper**:** $\mathtt{A} = \mathtt{C}$ $\quad$ $\mathtt{class\ C} \lhd \mathtt{D}\ \{.. \overline{\mathtt{M}} ..\}$ $\quad$ $\mathtt{m} \notin \overline{\mathtt{M}}$
$mbody(\mathtt{m}\langle\overline{\mathtt{P}}\rangle, \mathtt{D}) = \overline{\mathtt{x}}.\mathtt{e}$

By the induction hypothesis, there exist $\mathtt{E}$ and $\mathtt{T}'$ such that $\Delta \vdash \mathtt{D} <: \mathtt{E}$ and $\Delta \vdash \mathtt{T}' <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S}$ and $\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}, \mathtt{this} : \mathtt{E}; \mathtt{E} \vdash \mathtt{e} : \mathtt{T}'$. By S-Trans, $\Delta \vdash \mathtt{C} <: \mathtt{E}$.

Case MB-NClass**:** $\mathtt{A} = \mathtt{C.E}$ $\quad$ $\mathtt{class\ C} \lhd \mathtt{D}\ \{..\ \mathtt{class\ E}\ \{.. \overline{\mathtt{M}}\} ..\}$
$\langle \overline{\mathtt{X}} \lhd \overline{\mathtt{C}} \rangle \mathtt{S}_0\ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}_0;\ \} \in \overline{\mathtt{M}}$

By T-TClass and T-Method, there exists $\mathtt{T}'$ such that $\overline{\mathtt{X}} <: \overline{\mathtt{C}}; \overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{this} : .\mathtt{E}; \mathtt{C.E} \vdash \mathtt{e} : \mathtt{T}'$ and $\overline{\mathtt{X}} <: \overline{\mathtt{C}} \vdash \mathtt{T}' <: \mathtt{S}_0$. By Lemmas A.1 and A.8, there exists $\mathtt{T}''$ such that $\Delta; \mathtt{x} : [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}, \mathtt{this} : .\mathtt{E}; \mathtt{C.E} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0 : \mathtt{T}''$ and $\Delta \vdash \mathtt{T}'' <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}'$. By Lemma A.6 and S-Trans, $\Delta \vdash \mathtt{T}'' <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S}_0$.

Case MB-NSuper1, MB-NSuper2**:**

Similar to the case for MB-TSuper above. □

Proof of Theorem 3.1
By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$ with case analysis on the last rule used.

Case R-Field**:** $\mathtt{e} = \mathtt{new\ A}_0(\overline{\mathtt{e}}).\mathtt{f}_i$ $\quad$ $fields(\mathtt{A}_0) = \overline{\mathtt{S}}\ \overline{\mathtt{f}}$ $\quad$ $\mathtt{e}' = \mathtt{e}_i$

By the typing rules T-Field and T-New, we also have

$$\mathtt{T} = \mathtt{S}_i@\mathtt{A}_0 \qquad fields(\mathtt{A}_0) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \qquad \Delta; \Gamma; \mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta \vdash \overline{\mathtt{U}} <: (\overline{\mathtt{S}}@\mathtt{A}_0)$$

In particular, we have $\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e}_i : \mathtt{U}_i$ and $\Delta \vdash \mathtt{U}_i <: \mathtt{S}_i@\mathtt{A}_0$.

Case R-Invk**:** $\mathtt{e} = \mathtt{new\ A}_0(\overline{\mathtt{e}}).\langle\overline{\mathtt{P}}\rangle\mathtt{m}(\overline{\mathtt{d}})$ $\quad$ $mbody(\mathtt{m}\langle\overline{\mathtt{P}}\rangle, \mathtt{A}_0) = \overline{\mathtt{x}}.\mathtt{e}_0$
$\mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ A}_0(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e}_0$

We also have by T-Invk

$$\begin{array}{lll} \Delta; \Gamma; \mathtt{A} \vdash \mathtt{new\ A}_0(\overline{\mathtt{e}}) : \mathtt{A}_0 & mtype(\mathtt{m}, \mathtt{A}_0) = \langle \overline{\mathtt{X}} \lhd \overline{\mathtt{C}} \rangle \overline{\mathtt{S}} \to \mathtt{S} & \Delta; \mathtt{A} \vdash \overline{\mathtt{P}}\ \mathrm{ok} \\ \Delta \vdash \overline{\mathtt{P}} <: \overline{\mathtt{C}} & \Delta; \Gamma; \mathtt{A} \vdash \overline{\mathtt{d}} : \overline{\mathtt{U}} & \\ \Delta \vdash \overline{\mathtt{U}} <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{S}})@\mathtt{A}_0 & \mathtt{T} = ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S})@\mathtt{A}_0 & \end{array}$$

By Lemma A.11, there exist $\mathtt{U}$ and $\mathtt{B}_0$ such that

$$\mathtt{A}_0 <\# \mathtt{B}_0 \qquad \Delta; \overline{\mathtt{x}} : [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}, \mathtt{this} : thistype(\mathtt{B}_0); \mathtt{B}_0 \vdash \mathtt{e}_0 : \mathtt{U} \qquad \Delta \vdash \mathtt{U} <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S}\ .$$

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Then, by Lemma A.9, $\Delta;\overline{\mathtt{x}}:([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{S}})@\mathtt{A}_0,\mathtt{this}:\mathtt{A}_0;\mathtt{B}_0 \vdash \mathtt{e}_0 : \mathtt{U}@\mathtt{A}_0$. By Lemma A.5, $\Delta;\overline{\mathtt{x}}:([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{S}})@\mathtt{A}_0,\mathtt{this}:\mathtt{A}_0;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{U}@\mathtt{A}_0$. By Lemmas A.1 and A.10, there exists $\mathtt{U}'$ such that $\Delta;\Gamma;\mathtt{A} \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{A}_0(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e}_0 : \mathtt{U}'$ and $\Delta \vdash \mathtt{U}' <: \mathtt{U}@\mathtt{A}_0$. Finally, by the fact that $\Delta \vdash \mathtt{U} <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S}$ and Lemma A.2(2), $\Delta \vdash \mathtt{U}@\mathtt{A}_0 <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S})@\mathtt{A}_0$ and, by S-TRANS, $\Delta \vdash \mathtt{U}' <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{S})@\mathtt{A}_0$.

Case RC-FIELD,RC-INV-RECV,RC-INV-ARG,RC-NEW-ARG**:**

Easy. □

Appendix B

Proof of Theorem 4.1

Throughout the proofs here, we assume that the underlying class table is ok. Then, the erasure of the class table will be well defined; therefore, subtyping $<:_{\text{FJ}}$ and the lookup functions $\mathit{fields}_{\text{FJ}}$, $\mathit{mtype}_{\text{FJ}}$ and $\mathit{mbody}_{\text{FJ}}$ will be, too. We write $\mathtt{C} \vdash_{\text{FJ}} \mathtt{M}$ and $\vdash_{\text{FJ}} \mathtt{L}$ for FJ typing judgments for methods and classes, respectively.

We begin with proving a number of properties about erasure.

Lemma B.1 (*Properties of type erasure*)

1. If $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta;\mathtt{A} \vdash \mathtt{S},\mathtt{T}$ ok, then $|\mathtt{S}|_{\Delta,\mathtt{A}} <:_{\text{FJ}} |\mathtt{T}|_{\Delta,\mathtt{A}}$.
2. $|\mathtt{S}@\mathtt{T}|_{\Delta,\mathtt{A}} = |\mathtt{S}|_{\Delta,\widehat{\Delta}(\mathtt{T}@\mathtt{A})}$.
3. If $\mathtt{class}\ \mathtt{C} \lhd \mathtt{D}\ \{..\}$ and $\Delta;\mathtt{D}.\mathtt{E} \vdash \mathtt{T}$ ok, then $|\mathtt{T}|_{\Delta,\mathtt{C}.\mathtt{E}} <:_{\text{FJ}} |\mathtt{T}|_{\Delta,\mathtt{D}.\mathtt{E}}$.
4. If $\Delta;\mathtt{A} \vdash \mathtt{T}$ ok, then $|\widehat{\Delta}(\mathtt{T}@\mathtt{A})| = |\mathtt{T}|_{\Delta,\mathtt{A}}$.
5. If $\Delta;\mathtt{A} \vdash \overline{\mathtt{P}}$ ok and $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{C}};\mathtt{A} \vdash \mathtt{T}$ ok and $\Delta \vdash \overline{\mathtt{P}}<:\overline{\mathtt{C}}$, then $|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}|_{\Delta,\mathtt{A}} <: |\mathtt{T}|_{(\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{C}}),\mathtt{A}}$.
6. If $\mathtt{A}\ <\#\ \mathtt{B}$, then $|\mathtt{A}| <:_{\text{FJ}} |\mathtt{B}|$.

Proof
(1) is by induction on the derivation of $\Delta \vdash \mathtt{S}<:\mathtt{T}$. (2) is by case analysis on S and T. (3) is easy. (4) and (5) are by case analysis on T. (6) is easy. □

Lemma B.2
If $\mathit{fields}(\mathtt{A}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, then $\mathit{fieldsmax}(\mathtt{A}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ and $|\overline{\mathtt{T}}|_{\mathtt{A}} <:_{\text{FJ}} \overline{\mathtt{C}}$.

Proof
By induction on the derivation of $\mathit{fields}(\mathtt{A})$. □

Lemma B.3
If $\mathit{mtype}(\mathtt{m},\mathtt{A}) = <\overline{\mathtt{X}}\lhd\overline{\mathtt{C}}>\overline{\mathtt{T}}\to\mathtt{T}_0$, then $\mathit{mtypemax}(\mathtt{m},\mathtt{A}) = \overline{\mathtt{D}}\to\mathtt{D}_0$, $|\overline{\mathtt{T}}|_{\Delta,\mathtt{A}} <:_{\text{FJ}} \overline{\mathtt{D}}$ and $|\mathtt{T}_0|_{\Delta,\mathtt{A}} <:_{\text{FJ}} \mathtt{D}_0$, where $\Delta = \overline{\mathtt{X}}<:\overline{\mathtt{C}}$.

Proof
By induction on the derivation of $\mathit{mtype}(\mathtt{m},\mathtt{A}) = <\overline{\mathtt{X}}\lhd\overline{\mathtt{C}}>\overline{\mathtt{T}}\to\mathtt{T}_0$. □

Lemma B.4
$\mathit{fields}_{\text{FJ}}(|\mathtt{A}|) = \mathit{fieldsmax}(\mathtt{A})$. Similarly, $\mathit{mtype}_{\text{FJ}}(\mathtt{m},|\mathtt{A}|) = \mathit{mtypemax}(\mathtt{m},\mathtt{A})$.

Proof
By induction on the derivation of $\mathit{fieldsmax}(\mathtt{A})$ and $\mathit{mtypemax}(\mathtt{m},\mathtt{A})$. □

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Lemma B.5

If $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e} : \mathtt{T}$, then $\Delta;\mathtt{A} \vdash \mathtt{T}$ ok.

Proof

By induction on the derivation of $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}:\mathtt{T}$. □

Lemma B.6

If $\overline{\mathtt{N}} = \mathit{nestedclasses}(\mathtt{C})$, then $\mathtt{C} \vdash \overline{\mathtt{N}}$ ok

Proof

By induction of the derivation of $\mathit{nestedclasses}(\mathtt{C})$. □

Proof of Theorem 4.1

We prove the theorem in two steps: first it is shown that if $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}:\mathtt{T}$ then $|\Gamma|_{\Delta,\mathtt{A}} \vdash_{\mathrm{FJ}} |\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} : |\mathtt{T}|_{\Delta,\mathtt{A}}$; and second, we show $|CT|$ is ok.

The first part is proved by induction on the derivation of $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}:\mathtt{T}$ with a case analysis on the last rule used.

Case T-FIELD: $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i \quad \Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0 \quad \mathit{fields}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$
$\mathtt{T} = \mathtt{T}_i@\mathtt{T}_0$

By the induction hypothesis, we have $|\Gamma|_{\Delta,\mathtt{A}} \vdash_{\mathrm{FJ}} |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}} : |\mathtt{T}_0|_{\Delta,\mathtt{A}}$. By Lemma B.5, $\Delta;\mathtt{A} \vdash \mathtt{T}_0$ ok. By Lemmas B.1(4), B.2, and B.4,

$$\mathit{fields}_{\mathrm{FJ}}(|\mathtt{T}_0|_{\Delta,\mathtt{A}}) = \mathit{fieldsmax}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$$
$$|\overline{\mathtt{T}}|_{\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})} <:_{\mathrm{FJ}} \overline{\mathtt{C}}\ .$$

By the rule T-FIELD, we have $|\Gamma|_{\Delta,\mathtt{A}} \vdash_{\mathrm{FJ}} |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}.\mathtt{f}_i : \mathtt{C}_i$.

Since $\overline{\mathtt{T}}$ do not contain any type variables, by Lemma B.1(2),

$$|\overline{\mathtt{T}}|_{\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})} = |\overline{\mathtt{T}}|_{\Delta,\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})} = |\overline{\mathtt{T}}@\mathtt{T}_0|_{\Delta,\mathtt{A}} <:_{\mathrm{FJ}} \overline{\mathtt{C}}.$$

If $|\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}} = \mathtt{C}_i$, then $|\mathtt{e}_0.\mathtt{f}_i|_{\Delta,\Gamma,\mathtt{A}} = |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}.\mathtt{f}_i$ by E-FIELD, finishing the case. On the other hand, if $|\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}} \neq \mathtt{C}_i$, then

$$|\mathtt{e}_0.\mathtt{f}_i|_{\Delta,\Gamma,\mathtt{A}} = (|\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}})|\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}.\mathtt{f}_i$$

by the rule E-FIELD-CAST and $|\Gamma|_{\Delta,\mathtt{A}} \vdash_{\mathrm{FJ}} (|\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}})|\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}.\mathtt{f}_i : |\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}}$ by the rule T-DCAST, finishing the case.

Case T-INVK:

Similar to the case above.

Case T-VAR, T-NEW:

Easy.

The second part ($|CT|$ is ok) follows from the first part with examination of the rules T-METHOD, T-NCLASS, and T-TCLASS. We show that if $\mathtt{A} \vdash \mathtt{M}$ ok then $|\mathtt{A}| \vdash_{\mathrm{FJ}} |\mathtt{M}|_{\mathtt{A}}$ ok. Let $\mathtt{M} = \mathtt{<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{C}}\mathtt{>}\mathtt{T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e}_0\mathtt{;\ \}}$. Then, by E-METHOD, we

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

have

$$\begin{array}{l} \mathit{mtypemax}(\mathtt{m}, \mathtt{A}) = \overline{\mathtt{D}}{\to}\mathtt{D}_0 \\ |\mathtt{M}|_{\mathtt{A}} = \mathtt{D}_0\ \mathtt{m}(\overline{\mathtt{D}}\ \overline{\mathtt{x}}')\{\ \mathtt{return}\ \mathtt{e}_0'\mathtt{;}\ \} \\ \Gamma = \overline{\mathtt{x}}{:}\overline{\mathtt{T}}, \mathtt{this}{:}\mathit{thistype}(\mathtt{A}) \qquad \Delta = \overline{\mathtt{X}}{<:}\overline{\mathtt{C}} \\ \mathtt{e}_i = \begin{cases} \mathtt{x}_i' & \text{if } \mathtt{D}_i = |\mathtt{T}_i|_{\Delta,\mathtt{A}} \\ (|\mathtt{T}_i|_{\Delta,\mathtt{A}})\mathtt{x}_i' & \text{otherwise} \end{cases} \\ \mathtt{e}_0' = [\overline{\mathtt{e}}/\overline{\mathtt{x}}](|\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}})\ . \end{array}$$

By the rule T-METHOD, we have

$$\begin{array}{l} \Delta; \mathtt{A} \vdash \overline{\mathtt{T}}, \mathtt{T}_0, \overline{\mathtt{C}}\ \mathrm{ok} \qquad \Delta; \Gamma; \mathtt{A} \vdash \mathtt{e}_0 : \mathtt{S} \qquad \Delta \vdash \mathtt{S} <: \mathtt{T}_0 \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathit{superclass}(\mathtt{A})) = {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{E}}{>}\overline{\mathtt{U}}{\to}\mathtt{U}_0, \text{ then } (\overline{\mathtt{Y}}, \overline{\mathtt{E}}, \overline{\mathtt{U}}, \mathtt{U}_0) = (\overline{\mathtt{X}}, \overline{\mathtt{C}}, \overline{\mathtt{T}}, \mathtt{T}_0)\ . \end{array}$$

We must show that

$$\begin{array}{l} \overline{\mathtt{x}}' : \overline{\mathtt{D}}, \mathtt{this} : |\mathtt{A}| \vdash_{\mathrm{FJ}} \mathtt{e}_0' : \mathtt{E}_0 \qquad \mathtt{E}_0 <:_{\mathrm{FJ}} \mathtt{D}_0 \\ \text{if } \mathit{mtype}_{\mathrm{FJ}}(\mathtt{m}, |\mathit{superclass}(\mathtt{A})|) = \overline{\mathtt{D}'}{\to}\mathtt{D}_0', \text{ then } \overline{\mathtt{D}'} = \overline{\mathtt{D}} \text{ and } \mathtt{D}_0' = \mathtt{D}_0 \end{array}$$

for some $\mathtt{E}_0$. By the result of the first part, $\overline{\mathtt{x}} : |\overline{\mathtt{T}}|_{\Delta,\mathtt{A}}, \mathtt{this}{:}|\mathit{thistype}(\mathtt{A})|_{\Delta,\mathtt{A}} \vdash_{\mathrm{FJ}} |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}} : |\mathtt{S}|_{\Delta,\mathtt{A}}$. Since, by Lemma B.3, $|\mathtt{T}_i|_{\Delta,\mathtt{A}} <:_{\mathrm{FJ}} \mathtt{D}_i$, we have $\mathtt{x}_i'{:}\mathtt{D}_i \vdash_{\mathrm{FJ}} \mathtt{e}_i{:}|\mathtt{T}_i|_{\Delta,\mathtt{A}}$ for any $0 \leqslant i \leqslant \#(\overline{\mathtt{x}}')$. By Lemma A.10 and the fact that $|\mathit{thistype}(\mathtt{A})|_{\Delta,\mathtt{A}} = |\mathtt{A}|$, we have

$$\overline{\mathtt{x}}'{:}\overline{\mathtt{D}}, \mathtt{this} : |\mathtt{A}| \vdash_{\mathrm{FJ}} \mathtt{e}_0' : \mathtt{C}_0$$

for some $\mathtt{C}_0$ where $\mathtt{C}_0 <:_{\mathrm{FJ}} |\mathtt{S}|_{\Delta,\mathtt{A}}$. On the other hand, by Lemma B.3, $|\mathtt{T}_0|_{\Delta,\mathtt{A}} <:_{\mathrm{FJ}} \mathtt{D}_0$. Since $|\mathtt{S}|_{\Delta,\mathtt{A}}{<:}|\mathtt{T}_0|_{\Delta,\mathtt{A}}$ by Lemma B.1(1), we have $\mathtt{C}_0 <:_{\mathrm{FJ}} \mathtt{D}_0$ by S-TRANS. Let $\mathtt{E}_0$ be $\mathtt{C}_0$. Finally, if $\mathit{mtypemax}(\mathtt{m}, \mathit{superclass}(\mathtt{A}))$ is well defined, then it is easy to show that $\mathit{mtype}_{\mathrm{FJ}}(\mathtt{m}, |\mathit{superclass}(\mathtt{A})|)$ is also well defined. By Lemma B.4,

$$\mathit{mtypemax}(\mathtt{m}, \mathit{superclass}(\mathtt{A})) = \mathit{mtype}_{\mathrm{FJ}}(\mathtt{m}, |\mathit{superclass}(\mathtt{A})|) = \overline{\mathtt{D}}{\to}\mathtt{D}_0\ .$$

It is also straightforward to show that if $\mathtt{C} \vdash \mathtt{N}$ ok then $\vdash_{\mathrm{FJ}} |\mathtt{N}|_{\mathtt{C}}$ ok, and if $\vdash \mathtt{L}$ ok then $\vdash_{\mathrm{FJ}} |\mathtt{L}|$ ok. □

Appendix C

Proof of Theorems 4.2, 4.3, and 4.4

A first important lemma is Lemma C.5, which gives the correspondence between the method body in the erased class table and the method body erased under the context in which the program is running.

Lemma C.1
If $\Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{C}} \vdash \mathtt{e} \overset{\mathrm{exp}}{\Longrightarrow} \mathtt{e}'$ and $\Gamma \vdash_{\mathrm{FJ}} \overline{\mathtt{d}}{:}\overline{\mathtt{D}}$ where $\overline{\mathtt{D}} <:_{\mathrm{FJ}} \overline{\mathtt{C}}$, then $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \overset{\mathrm{exp}}{\Longrightarrow} [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}'$.

Proof
By induction on the derivation of $\Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{C}} \vdash_{\mathrm{FJ}} \mathtt{e}{:}\mathtt{C}$. □

Lemma C.2
If $\Delta', \overline{\mathtt{X}}{<:}\overline{\mathtt{C}}; \Gamma; \mathtt{A} \vdash \mathtt{e}{:}\mathtt{T}$ and $\Delta' \vdash \overline{\mathtt{P}} <: \overline{\mathtt{C}}$ and $\Delta'; \mathtt{A} \vdash \overline{\mathtt{P}}$ ok and Γ' is a type environment such that $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma)$ and $\Delta' \vdash \Gamma'(\mathtt{x}) <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\Gamma(\mathtt{x}))@\mathtt{A}$ for all $\mathtt{x} \in \mathit{dom}(\Gamma)$, then

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

$|\mathtt{e}|_{(\Delta',\overline{\mathtt{X}}<:\overline{\mathtt{C}}),\Gamma,\mathtt{A}}$ is obtained from $|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta',\Gamma',\mathtt{A}}$ by some combination of replacements of some synthetic casts `(D)` with `(C)` where $\mathtt{D} <: \mathtt{C}$, or removals of some casts.

Proof

By induction on the derivation of $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}:\mathtt{T}$, in which we let $\Delta = \Delta', \overline{\mathtt{X}}<:\overline{\mathtt{C}}$, with case analysis on the last rule used.

Case T-VAR**:**

Trivial.

Case T-FIELD**:** $\mathtt{e} = \mathtt{e}_0.\mathtt{f}$ $\quad \Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0$ $\quad \mathit{fields}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ $\quad \mathtt{T} = \mathtt{T}_i@\mathtt{T}_0$

By the induction hypothesis, $|\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}$ is obtained from $|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta',\Gamma',\mathtt{A}}$ by some combination of replacements of some casts `(D)` with `(C)` where $\mathtt{D} <:_{\mathrm{FJ}} \mathtt{C}$, or removals of some casts. By Theorem 4.1, $|\Gamma|_{\Delta,\mathtt{A}} \vdash_{\mathrm{FJ}} |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}} : |\mathtt{T}_0|_{\Delta,\mathtt{A}}$. By Lemmas B.2 and B.1(2), $\mathit{fieldsmax}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$ and $|\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}} = |\mathtt{T}_i|_{\Delta,\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})} <:_{\mathrm{FJ}} \mathtt{D}_i$.

We have two subcases.

Subcase: $|\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}} \neq \mathtt{D}_i$

By the rule E-FIELD-CAST,

$$|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} = (|\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}})|\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}.\mathtt{f}_i.$$

Then, we must show that $|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta',\Gamma',\mathtt{A}} = (\mathtt{D})|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta',\Gamma',\mathtt{A}}.\mathtt{f}_i$ for some $\mathtt{D} <:_{\mathrm{FJ}} |\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}}$. By Lemmas A.9, A.8, and A.10,

$$\Delta';\Gamma';\mathtt{A} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0 : \mathtt{S}_0 \qquad \Delta' \vdash \mathtt{S}_0 <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0@\mathtt{A}$$

for some $\mathtt{S}_0$. Then, by Lemmas A.3 and A.2(2),

$$\mathit{fields}(\widehat{\Delta}'(\mathtt{S}_0@\mathtt{A})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}, \overline{\mathtt{T}}'\ \overline{\mathtt{g}}$$

for some $\overline{\mathtt{T}}'\ \overline{\mathtt{g}}$. By Lemmas A.2(1), A.2(3), and A.2(4), and the fact that $\mathtt{T}_i$ does not contain $\overline{\mathtt{X}}$,

$$\begin{aligned} \mathtt{T}_i@\mathtt{S}_0 &= \mathtt{T}_i@([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0@\mathtt{A}) \\ &= (\mathtt{T}_i@([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0))@\mathtt{A} \\ &= ([\overline{\mathtt{P}}/\overline{\mathtt{X}}](\mathtt{T}_i@\mathtt{T}_0))@\mathtt{A} \end{aligned}$$

and, by Lemmas B.1(2) and B.1(5),

$$|\mathtt{T}_i@\mathtt{S}_0|_{\Delta',\mathtt{A}} = |([\overline{\mathtt{P}}/\overline{\mathtt{X}}](\mathtt{T}_i@\mathtt{T}_0))@\mathtt{A}|_{\Delta',\mathtt{A}} = |[\overline{\mathtt{P}}/\overline{\mathtt{X}}](\mathtt{T}_i@\mathtt{T}_0)|_{\Delta',\mathtt{A}} <:_{\mathrm{FJ}} |\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}}.$$

On the other hand,

$$\mathit{fieldsmax}(\Delta'(\mathtt{S}_0@\mathtt{A})) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}, \overline{\mathtt{D}}'\ \overline{\mathtt{g}}$$

for some $\overline{\mathtt{D}}'\ \overline{\mathtt{g}}$. Therefore, by the rule E-FIELD-CAST,

$$|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta',\Gamma',\mathtt{A}} = (|\mathtt{T}_i@\mathtt{S}_0|_{\Delta',\mathtt{A}})|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta',\Gamma',\mathtt{A}}.\mathtt{f}_i$$

and taking $|\mathtt{T}_i@\mathtt{S}_0|_{\Delta',\mathtt{A}}$ as $\mathtt{D}$ finishes the case.

Subcase: $|\mathtt{T}_i@\mathtt{T}_0|_{\Delta,\mathtt{A}} = \mathtt{D}_i$

Similar to the subcase above.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Case T-INVK: $\mathtt{e} = \mathtt{e}_0.\langle\overline{\mathtt{Q}}\rangle\mathtt{m}(\overline{\mathtt{e}})$ $\quad \Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0$
$mtype(\mathtt{m}, \widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \langle\overline{\mathtt{Y}} \lhd \overline{\mathtt{D}}\rangle\overline{\mathtt{U}} \to \mathtt{U}_0$ $\quad \Delta;\mathtt{A} \vdash \overline{\mathtt{Q}}$ ok
$\Delta \vdash \overline{\mathtt{Q}} <: \overline{\mathtt{D}}$ $\quad \Delta;\Gamma;\mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$ $\quad \Delta \vdash \overline{\mathtt{T}} <: ([\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}})@\mathtt{T}_0$
$\mathtt{T} = ([\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@\mathtt{T}_0$

By the induction hypothesis, $|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}$ are obtained from $|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{e}}|_{\Delta',\Gamma',\mathtt{A}}$ by some combination of replacements of some casts $(\mathtt{D})$ with $(\mathtt{C})$ where $\mathtt{D} <:_{\mathrm{FJ}} \mathtt{C}$, or removals of some casts. By Theorem 4.1, $|\Gamma|_{\Delta,\mathtt{A}} \vdash_{\mathrm{FJ}} |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}} : |\mathtt{T}_0|_{\Delta,\mathtt{A}}$. Since, by Lemmas B.1(2), B.1(5), and B.3, $mtypemax(\mathtt{m}, \widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{E}} \to \mathtt{E}_0$ and $|\mathtt{T}|_{\Delta,\mathtt{A}} = |[\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0|_{\Delta,\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})} <:_{\mathrm{FJ}} |\mathtt{U}_0|_{(\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{D}}),\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})} <:_{\mathrm{FJ}} \mathtt{E}_0$.

Now we have two subcases.

Subcase: $|\mathtt{T}|_{\Delta,\mathtt{A}} \neq \mathtt{E}_0$

By the rule E-INVK-CAST,

$$|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} = (|\mathtt{T}|_{\Delta,\mathtt{A}})|\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}.\mathtt{m}(|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}})$$

Now, we must show that

$$|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta',\Gamma',\mathtt{A}} = (\mathtt{D})|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta',\Gamma',\mathtt{A}}.\mathtt{m}(|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{e}}|_{\Delta',\Gamma',\mathtt{A}})$$

for some $\mathtt{D} <:_{\mathrm{FJ}} |\mathtt{T}|_{\Delta,\mathtt{A}}$. By Lemmas A.9, A.8, and A.10,

$$\Delta';\Gamma';\mathtt{A} \vdash [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0 : \mathtt{S}_0 \qquad \Delta' \vdash \mathtt{S}_0 <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0@\mathtt{A}\ .$$

By Lemma A.4,

$$mtype(\mathtt{m}, \widehat{\Delta}'(\mathtt{S}_0@\mathtt{A})) = \langle\overline{\mathtt{Y}} \lhd \overline{\mathtt{D}}\rangle\overline{\mathtt{U}} \to \mathtt{U}_0$$

By Lemmas A.2(1), A.2(3), and A.2(4), and the fact that $\mathtt{U}_0$ does not contain any type variables in $\overline{\mathtt{X}}$,

$$\begin{aligned}([[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@\mathtt{S}_0 &= ([[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0@\mathtt{A}) \\ &= (([[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{T}_0))@\mathtt{A} \\ &= ([\overline{\mathtt{P}}/\overline{\mathtt{X}}](([\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@\mathtt{T}_0))@\mathtt{A}.\end{aligned}$$

Then, by Lemmas B.1(2) and B.1(5),

$$\begin{aligned}|([[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@\mathtt{S}_0|_{\Delta',\mathtt{A}} &= |([\overline{\mathtt{P}}/\overline{\mathtt{X}}](([\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@\mathtt{T}_0))@\mathtt{A}|_{\Delta',\mathtt{A}} \\ &= |[\overline{\mathtt{P}}/\overline{\mathtt{X}}](([\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@\mathtt{T}_0)|_{\Delta',\mathtt{A}} <:_{\mathrm{FJ}} |([\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@\mathtt{T}_0|_{\Delta,\mathtt{A}}\ .\end{aligned}$$

On the other hand, it is easy to show that

$$mtypemax(\widehat{\Delta}'(\mathtt{S}_0@\mathtt{A})) = mtypemax(\widehat{\Delta}(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{E}} \to \mathtt{E}_0\ .$$

Then, by the rule E-INVK-CAST,

$$|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}|_{\Delta',\Gamma',\mathtt{A}} = (|([[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@\mathtt{S}_0|_{\Delta',\mathtt{A}})|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta',\Gamma',\mathtt{A}}.\mathtt{m}(|[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{e}}|_{\Delta',\Gamma',\mathtt{A}})$$

and taking $|([[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}/\overline{\mathtt{Y}}]\mathtt{U}_0)@\mathtt{S}_0|_{\Delta,\mathtt{A}}$ as $\mathtt{D}$ finishes the subcase.

Subcase: $|\mathtt{T}|_{\Delta,\mathtt{A}} = \mathtt{E}_0$

Similar to the subcase above.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Case T-NEW**:**

Immediate from the induction hypothesis. □

Lemma C.3
If $\Delta;\Gamma;\mathtt{D.E} \vdash \mathtt{e} : \mathtt{T}$ and $\Delta \vdash \mathtt{C} <: \mathtt{D}$, then $\Delta;\Gamma;\mathtt{C.E} \vdash \mathtt{e} : \mathtt{T}$.

Proof
By straightforward induction on $\Delta;\Gamma;\mathtt{D.E} \vdash \mathtt{e} : \mathtt{T}$, with Lemmas A.3 and A.4. □

Lemma C.4
If $\Delta;\Gamma;\mathtt{D.E} \vdash \mathtt{e} : \mathtt{T}$ and $\Delta \vdash \mathtt{C} <: \mathtt{D}$, then $|\Gamma|_{\Delta,\mathtt{C.E}} \vdash |\mathtt{e}|_{\Delta,\Gamma,\mathtt{C.E}} \overset{\text{exp}}{\Longrightarrow} |\mathtt{e}|_{\Delta,\Gamma,\mathtt{D.E}}$.

Proof
By induction on $\Delta;\Gamma;\mathtt{D.E} \vdash \mathtt{e} : \mathtt{T}$. We show only the case for T-FIELD since the other cases are either easy or similar to this main case.

Case T-FIELD**:** $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_i \quad \Delta;\Gamma;\mathtt{D.E} \vdash \mathtt{e}_0 : \mathtt{T}_0 \quad \mathit{fields}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{D.E})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$
$\mathtt{T} = \mathtt{T}_i@\mathtt{T}_0$

By Lemma B.5, $\Delta;\mathtt{D.E} \vdash \mathtt{T}$ ok. By the induction hypothesis, we have $|\Gamma|_{\Delta,\mathtt{C.E}} \vdash |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{C.E}} \overset{\text{exp}}{\Longrightarrow} |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{D.E}}$. By Lemma C.3,

$$\Delta;\Gamma;\mathtt{C.E} \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta;\Gamma;\mathtt{C.E} \vdash \mathtt{e} : \mathtt{T}\ .$$

Let $\mathtt{C} = \mathit{fieldsmax}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{C.E}))(\mathtt{f}_i)$. It is easy to check that $\widehat{\Delta}(\mathtt{T}_0@\mathtt{C.E}) <\!\#\ \widehat{\Delta}(\mathtt{T}_0@\mathtt{D.E})$ and $\mathit{fieldsmax}(\widehat{\Delta}(\mathtt{T}_0@\mathtt{D.E}))(\mathtt{f}_i) = \mathtt{C}$. By Lemmas B.1(3) and B.2, $|\mathtt{T}|_{\Delta,\mathtt{C.E}} <:_{\text{FJ}} |\mathtt{T}|_{\Delta,\mathtt{D.E}} <:_{\text{FJ}} \mathtt{C}$. We have four subcases depending on whether these three types are equal or not. If $|\mathtt{T}|_{\Delta,\mathtt{C.E}} \neq |\mathtt{T}|_{\Delta,\mathtt{D.E}} \neq \mathtt{C}$, then

$$\begin{aligned} |\mathtt{e}|_{\Delta,\Gamma,\mathtt{C.E}} &= (|\mathtt{T}|_{\Delta,\mathtt{C.E}})|\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{C.E}}.\mathtt{f}_i \\ |\mathtt{e}|_{\Delta,\Gamma,\mathtt{D.E}} &= (|\mathtt{T}|_{\Delta,\mathtt{D.E}})|\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{D.E}}.\mathtt{f}_i\ . \end{aligned}$$

by E-FIELD-CAST. Thus, $|\Gamma|_{\Delta,\mathtt{C.E}} \vdash |\mathtt{e}|_{\Delta,\Gamma,\mathtt{C.E}} \overset{\text{exp}}{\Longrightarrow} |\mathtt{e}|_{\Delta,\Gamma,\mathtt{D.E}}$. The other three subcases are similar. □

Lemma C.5
If $\mathit{mbody}(\mathtt{m}<\overline{\mathtt{P}}>,\ \mathtt{A}) = \overline{\mathtt{x}}.\mathtt{e}$ and $\mathit{mtype}(\mathtt{m},\ \mathtt{A}) = <\overline{\mathtt{X}} \lhd \overline{\mathtt{C}}> \overline{\mathtt{T}} \to \mathtt{T}_0$ and $\Delta;\mathtt{B} \vdash \overline{\mathtt{P}}$ ok and $\Delta \vdash \overline{\mathtt{P}} <: \overline{\mathtt{C}}$ and $\Delta \vdash \overline{\mathtt{U}} <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{T}})@\mathtt{A}$, then $\mathit{mbody}_{\text{FJ}}(\mathtt{m},\ |\mathtt{A}|) = \overline{\mathtt{x}}.\mathtt{e}'$ and $|\overline{\mathtt{x}} : \overline{\mathtt{U}},\ \mathtt{this} : \mathit{thistype}(\mathtt{A})|_{\Delta,\mathtt{A}} \vdash |\mathtt{e}|_{\Delta,\ (\overline{\mathtt{x}}:\overline{\mathtt{U}},\mathtt{this}:\mathit{thistype}(\mathtt{A})),\ \mathtt{A}} \overset{\text{exp}}{\Longrightarrow} \mathtt{e}'$.

Proof
By induction on the derivation of $\mathit{mbody}(\mathtt{m}<\overline{\mathtt{P}}>, \mathtt{A})$ with a case analysis on the last rule used.

Case MB-NCLASS**:** $\mathtt{A} = \mathtt{C.E} \quad \mathtt{class\ C} \lhd \mathtt{D}\{\ ..\ \mathtt{class\ E}\{\ ..\ \overline{\mathtt{M}}\ \}\ ..\ \}$
$<\overline{\mathtt{X}} \lhd \overline{\mathtt{C}}>\mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}_0;\ \} \in \overline{\mathtt{M}} \quad \mathtt{e} = [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{e}_0$

Let $\Delta' = \overline{\mathtt{X}} <: \overline{\mathtt{C}}$ and $\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{.E}$. By T-METHOD, we have

$$\Delta';\Gamma;\mathtt{C.E} \vdash \mathtt{e}_0 : \mathtt{S}_0 \qquad \Delta' \vdash \mathtt{S}_0 <: \mathtt{T}_0\ .$$

Then, by Lemma C.2, $|\mathtt{e}_0|_{\Delta',\Gamma,\mathtt{C.E}}$ is obtained from $|\mathtt{e}|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{U}},\mathtt{this}:\mathtt{.E}),\mathtt{C.E}}$ by some combination of replacements of some casts `(C)` with `(D)` where $\mathtt{C} <:_{\text{FJ}} \mathtt{D}$. By Theorem 4.1

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

and Lemma B.1(1),

$$|\Gamma|_{\Delta',\mathtt{C.E}} \vdash_{\mathrm{FJ}} |\mathtt{e}_0|_{\Delta',\Gamma,\mathtt{C.E}} : |\mathtt{S}_0|_{\Delta',\mathtt{C.E}} \qquad |\mathtt{S}_0|_{\Delta',\mathtt{C.E}} <:_{\mathrm{FJ}} |\mathtt{T}_0|_{\Delta',\mathtt{C.E}} .$$

Now, by Lemma B.3, $mtypemax(\mathtt{m},\mathtt{C.E}) = \overline{\mathtt{D}}\to\mathtt{D}_0$ for some $\overline{\mathtt{D}}$ and $\mathtt{D}_0$, and, by E-Method, $mtype_{\mathrm{FJ}}(\mathtt{m},|\mathtt{C.E}|) = \overline{\mathtt{x}}.[\overline{\mathtt{e}}/\overline{\mathtt{x}}](|\mathtt{e}_0|_{\Delta',\Gamma,\mathtt{C.E}})$, where

$$\mathtt{e}_i = \begin{cases} \mathtt{x}_i & \text{if } \mathtt{D}_i = |\mathtt{T}_i|_{\Delta',\mathtt{C.E}} \\ (|\mathtt{T}_i|_{\Delta',\mathtt{C.E}})\mathtt{x}_i & \text{otherwise} \end{cases}$$

for $i = 1,\ldots,\#(\overline{\mathtt{x}})$. By Lemmas B.1(1), B.1(2), and B.1(5),

$$|\overline{\mathtt{U}}|_{\Delta,\mathtt{C.E}} <:_{\mathrm{FJ}} |([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{T}})@\mathtt{C.E}|_{\Delta,\mathtt{C.E}} <:_{\mathrm{FJ}} |[\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}|_{\Delta,\mathtt{C.E}} <:_{\mathrm{FJ}} |\overline{\mathtt{T}}|_{(\Delta,\Delta'),\mathtt{C.E}} = |\overline{\mathtt{T}}|_{\Delta',\mathtt{C.E}} .$$

Thus, each $\mathtt{e}_i$ is either a variable or a variable with an upcast under the environment $|\overline{\mathtt{x}}{:}\overline{\mathtt{U}},\mathtt{this}{:}.\mathtt{E}|_{\Delta,\mathtt{C.E}}$. Then, by Lemma A.10, we have

$$|\overline{\mathtt{x}}{:}\overline{\mathtt{U}},\mathtt{this}{:}.\mathtt{E}|_{\Delta,\mathtt{C.E}} \vdash_{\mathrm{FJ}} [\overline{\mathtt{e}}/\overline{\mathtt{x}}](|\mathtt{e}_0|_{\Delta',\Gamma,\mathtt{C.E}}) : \mathtt{E}_0$$

for some $\mathtt{E}_0$ such that $\mathtt{E}_0 <:_{\mathrm{FJ}} |\mathtt{S}_0|_{\Delta',\mathtt{C.E}}$. Therefore, we have

$$|\overline{\mathtt{x}}{:}\overline{\mathtt{U}},\mathtt{this}{:}.\mathtt{E}|_{\Delta,\mathtt{C.E}} \vdash |\mathtt{e}|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{U}},\mathtt{this}:.\mathtt{E}),\mathtt{C.E}} \overset{\mathrm{exp}}{\Longrightarrow} [\overline{\mathtt{e}}/\overline{\mathtt{x}}](|\mathtt{e}_0|_{\Delta',\Gamma,\mathtt{C.E}}),$$

finishing the case.

Case MB-NSuper1: $\mathtt{A} = \mathtt{C.E}$ $\quad \mathtt{class\ C} \lhd \mathtt{D}\{\ ..\ \overline{\mathtt{N}}\}$ $\quad \mathtt{E} \notin \overline{\mathtt{N}}$
$mbody(\mathtt{m}\langle\overline{\mathtt{P}}\rangle,\mathtt{D.E}) = \overline{\mathtt{x}}.\mathtt{e}$

It must be the case that $mtype(\mathtt{m},\mathtt{D.E}) = \langle\overline{\mathtt{X}}\lhd\overline{\mathtt{C}}\rangle\overline{\mathtt{T}}\to\mathtt{T}_0$. By the induction hypothesis and the definition of erasure, we have $mbody_{\mathrm{FJ}}(\mathtt{m},|\mathtt{C.E}|) = mbody_{\mathrm{FJ}}(\mathtt{m},|\mathtt{D.E}|) = \overline{\mathtt{x}}.\mathtt{e}'$ and

$$|\overline{\mathtt{x}}{:}\overline{\mathtt{U}},\mathtt{this}{:}.\mathtt{E}|_{\Delta,\mathtt{A}} \vdash |\mathtt{e}|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{U}},\mathtt{this}:.\mathtt{E}),\mathtt{D.E}} \overset{\mathrm{exp}}{\Longrightarrow} \mathtt{e}' .$$

By Lemma C.4,

$$|\overline{\mathtt{x}}{:}\overline{\mathtt{U}},\mathtt{this}{:}.\mathtt{E}|_{\Delta,\mathtt{A}} \vdash |\mathtt{e}|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{U}},\mathtt{this}:.\mathtt{E}),\mathtt{C.E}} \overset{\mathrm{exp}}{\Longrightarrow} |\mathtt{e}|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{U}},\mathtt{this}:.\mathtt{E}),\mathtt{D.E}} .$$

Transitivity of $\overset{\mathrm{exp}}{\Longrightarrow}$ finishes the case.

Case MB-NSuper2:

Similarly to the case above.

Case MB-TClass, MB-TSuper:

Easy. □

Proof of Theorem 4.2

By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$ with a case analysis on the last reduction rule used.

Case R-Field: $\mathtt{e} = \mathtt{new\ A_0}(\overline{\mathtt{e}}).\mathtt{f}_i$ $\quad fields(\mathtt{A}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ $\quad \mathtt{e}' = \mathtt{e}_i$

We have two subcases depending on the last erasure rule used.

Subcase E-Field-Cast: $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} = (\mathtt{D})\mathtt{new}\ |\mathtt{A}_0|(|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}).\mathtt{f}_i$
$fieldsmax(\mathtt{A}_0)(\mathtt{f}_i) \neq |\mathtt{T}|_{\Delta,\mathtt{A}} = \mathtt{D}$

By Lemma B.4, $fields_{\mathrm{FJ}}(|\mathtt{A}_0|) = \ldots,\mathtt{C}\ \mathtt{f}_i,\ldots$ for some $\mathtt{C}$, we have $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} \longrightarrow_{\mathrm{FJ}}$

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

$\texttt{(D)}|\texttt{e}_i|_{\Delta,\Gamma,\texttt{A}}$. On the other hand, by Theorem 3.1, $\Delta;\Gamma;\texttt{A} \vdash \texttt{e}_i : \texttt{T}_i$ such that $\Delta \vdash \texttt{T}_i <: \texttt{T}$. By Theorem 4.1, $|\Gamma|_{\Delta,\texttt{A}} \vdash_{\text{FJ}} |\texttt{e}_i|_{\Delta,\Gamma,\texttt{A}} : |\texttt{T}_i|_{\Delta,\texttt{A}}$. Since $|\texttt{T}_i|_{\Delta,\texttt{A}} <:_{\text{FJ}} |\texttt{T}|_{\Delta,\texttt{A}}$ by Lemma B.1(1), $\texttt{(D)}|\texttt{e}_i|_{\Delta,\Gamma,\texttt{A}}$ is obtained by adding an upcast to $|\texttt{e}_i|_{\Delta,\Gamma,\texttt{A}}$.

Subcase E-FIELD: $|\texttt{e}|_{\Delta,\Gamma,\texttt{A}} = \texttt{new}\ |\texttt{A}_0|(|\overline{\texttt{e}}|_{\Delta,\Gamma,\texttt{A}})\texttt{.f}_i$

Similarly to the subcase above.

Case R-INVK: $\texttt{e} = \texttt{new}\ \texttt{A}_0(\overline{\texttt{e}})\texttt{.<}\overline{\texttt{P}}\texttt{>m}(\overline{\texttt{d}})$ $\quad mbody(\texttt{m<}\overline{\texttt{P}}\texttt{>}, \texttt{A}_0) = \overline{\texttt{x}}\texttt{.e}_0$
$\texttt{e}' = [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new}\ \texttt{A}_0(\overline{\texttt{e}})/\texttt{this}]\texttt{e}_0$

By T-INVK, we have

$mtype(\texttt{m}, \texttt{A}_0) = \texttt{<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{C}}\texttt{>}\overline{\texttt{T}}\texttt{→T}_0$ $\quad \Delta;\texttt{A} \vdash \overline{\texttt{P}}\ \text{ok}$ $\quad \Delta \vdash \overline{\texttt{P}} <: \overline{\texttt{C}}$
$\Delta;\Gamma;\texttt{A} \vdash \overline{\texttt{d}} : \overline{\texttt{U}}$ $\quad \Delta \vdash \overline{\texttt{U}} <: ([\overline{\texttt{P}}/\overline{\texttt{X}}]\overline{\texttt{T}})@\texttt{A}_0$ $\quad \texttt{T} = ([\overline{\texttt{P}}/\overline{\texttt{X}}]\texttt{T}_0)@\texttt{A}_0$.

We have two subcases depending on the last erasure rule used.

Subcase E-INVK-CAST: $|\texttt{e}|_{\Delta,\Gamma,\texttt{A}} = \texttt{(D)new}\ |\texttt{A}_0|(|\overline{\texttt{e}}|_{\Delta,\Gamma,\texttt{A}})\texttt{.m}(|\overline{\texttt{d}}|_{\Delta,\Gamma,\texttt{A}})$
$mtypemax(\texttt{m}, \texttt{A}_0) = \overline{\texttt{E}}\texttt{→E}_0 \quad \texttt{E}_0 \neq |\texttt{T}|_{\Delta,\texttt{A}} = \texttt{D}$

Let $\Gamma' = \overline{\texttt{x}}\texttt{:}\overline{\texttt{U}}, \texttt{this} : thistype(\texttt{A}_0)$. Then, by Lemma C.5 and $mbody_{\text{FJ}}(\texttt{m}, |\texttt{A}_0|) = \overline{\texttt{x}}\texttt{.e}_0'$,

$$|\Gamma'|_{\Delta,\texttt{A}_0} \vdash |\texttt{e}_0|_{\Delta,\Gamma',\texttt{A}_0} \overset{\text{exp}}{\Longrightarrow} \texttt{e}_0'$$

By Lemma C.1 and the fact that $|\texttt{e}'|_{\Delta,\Gamma,\texttt{A}} = [|\overline{\texttt{d}}|_{\Delta,\Gamma,\texttt{A}}/\overline{\texttt{x}}, |\texttt{new}\ \texttt{A}_0(\overline{\texttt{e}})|_{\Delta,\Gamma,\texttt{A}}/\texttt{this}]|\texttt{e}_0|_{\Delta,\Gamma',\texttt{A}_0}$,

$$|\Gamma|_{\Delta,\texttt{A}} \vdash |\texttt{e}'|_{\Delta,\Gamma,\texttt{A}} \overset{\text{exp}}{\Longrightarrow} [|\overline{\texttt{d}}|_{\Delta,\Gamma,\texttt{A}}/\overline{\texttt{x}}, |\texttt{new}\ \texttt{A}_0(\overline{\texttt{e}})|_{\Delta,\Gamma,\texttt{A}}/\texttt{this}]\texttt{e}_0'.$$

By Theorems 3.1 and 4.1, $|\Gamma|_{\Delta,\texttt{A}} \vdash_{\text{FJ}} |\texttt{e}'|_{\Delta,\Gamma,\texttt{A}} : |\texttt{T}'|_{\Delta,\texttt{A}}$ for some $\texttt{T}'$ such that $\Delta \vdash \texttt{T}' <: \texttt{T}$. By Lemma B.1(1), $|\texttt{T}'|_{\Delta,\texttt{A}} <:_{\text{FJ}} |\texttt{T}|_{\Delta,\texttt{A}} = \texttt{D}$. Thus,

$$|\Gamma|_{\Delta,\texttt{A}} \vdash |\texttt{e}'|_{\Delta,\Gamma,\texttt{A}} \overset{\text{exp}}{\Longrightarrow} \texttt{(D)}|\texttt{e}'|_{\Delta,\Gamma,\texttt{A}}.$$

Finally,

$$|\Gamma|_{\Delta,\texttt{A}} \vdash |\texttt{e}'|_{\Delta,\Gamma,\texttt{A}} \overset{\text{exp}}{\Longrightarrow} \texttt{(D)}[|\overline{\texttt{d}}|_{\Delta,\Gamma,\texttt{A}}/\overline{\texttt{x}}, |\texttt{new}\ \texttt{A}_0(\overline{\texttt{e}})|_{\Delta,\Gamma,\texttt{A}}/\texttt{this}]\texttt{e}_0'.$$

Subcase E-INVK:

Similarly to the subcase above. □

Now, we prove Theorem 4.3 after the following lemmas. The first lemma shows how one execution step in FJ is reflected in .FJ and the next shows expansion and FJ reduction can commute. Then, Theorem 4.3 is an easy corollary of them.

Lemma C.6
Suppose $\Delta;\Gamma;\texttt{A} \vdash \texttt{e} : \texttt{T}$. If $|\texttt{e}|_{\Delta,\Gamma,\texttt{A}} \longrightarrow_{\text{FJ}} \texttt{d}$, then $\texttt{e} \longrightarrow \texttt{e}'$ for some $\texttt{e}'$ and $|\Gamma|_{\Delta,\texttt{A}} \vdash |\texttt{e}'|_{\Delta,\Gamma,\texttt{A}} \overset{\text{exp}}{\Longrightarrow} \texttt{d}$. In other words, the diagram in Figure C.1 commutes.

Proof
By induction on the derivation of $|\texttt{e}|_{\Delta,\Gamma,\texttt{A}} \longrightarrow_{\text{FJ}} \texttt{d}$ with a case analysis by the last rule used. We show only a few main cases.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

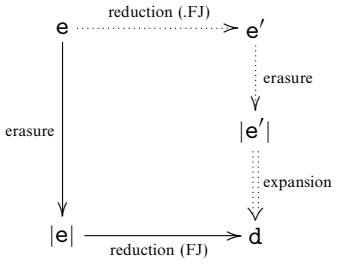

Fig. C.1. Commuting diagram of Lemma C.6.

Case RC-CAST: $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} = \mathtt{(C)e}_0 \qquad \mathtt{e}_0 \longrightarrow_{\mathrm{FJ}} \mathtt{d}_0 \qquad \mathtt{d} = \mathtt{(C)d}_0$

Both $\mathtt{e}$ and $\mathtt{e}_0$ must be either a field access or a method invocation. We have another case analysis with the last reduction rule for the derivation of $\mathtt{e}_0 \longrightarrow_{\mathrm{FJ}} \mathtt{d}_0$. The cases for RC-FIELD, RC-INVK-RECV, and RC-INVK-ARG are omitted, since the conclusion easily follows from the induction hypothesis.

Subcase R-FIELD: $\mathtt{e}_0 = \mathtt{new\ D}(\overline{\mathtt{e}})\mathtt{.f}_i \qquad \mathit{fields}_{\mathrm{FJ}}(\mathtt{D}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}} \qquad \mathtt{d}_0 = \mathtt{e}_i$

By inspecting the derivation of $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}}$, it must be the case that

$$\mathtt{e} = \mathtt{new\ B}(\overline{\mathtt{e}}')\mathtt{.f}_i \qquad |\mathtt{B}| = \mathtt{D} \qquad |\overline{\mathtt{e}}'|_{\Delta,\Gamma,\mathtt{A}} = \overline{\mathtt{e}} \qquad \mathit{fieldsmax}(\mathtt{B}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$$
$$|\mathtt{T}|_{\Delta,\mathtt{A}} = \mathtt{C} \neq \mathtt{C}_i\ .$$

By Theorems 3.2 and 3.1 and Lemma B.4, we have $\mathtt{e} \longrightarrow \mathtt{e}_i'$ and $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_i' : \mathtt{S}$ and $\Delta \vdash \mathtt{S} <: \mathtt{T}$. By Theorem 4.1, $|\Gamma|_{\Delta,\mathtt{A}} \vdash_{\mathrm{FJ}} |\mathtt{e}_i'|_{\Delta,\Gamma,\mathtt{A}} : |\mathtt{S}|_{\Delta,\mathtt{A}}$. By Lemma B.1(1), $|\mathtt{S}|_{\Delta,\mathtt{A}} <:_{\mathrm{FJ}} |\mathtt{T}|_{\Delta,\mathtt{A}}$. Then, $|\Gamma|_{\Delta,\mathtt{A}} \vdash \mathtt{e}_i \overset{\exp}{\Longrightarrow} (|\mathtt{T}|_{\Delta,\mathtt{A}})\mathtt{e}_i$, finishing the case.

Subcase R-INVK: $\mathtt{e}_0 = \mathtt{new\ D}(\overline{\mathtt{d}})\mathtt{.m}(\overline{\mathtt{e}}) \qquad \mathit{mbody}_{\mathrm{FJ}}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{x}}\mathtt{.e}_m$
$\mathtt{d}_0 = [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D}(\overline{\mathtt{d}})/\mathtt{this}]\mathtt{e}_m$

By inspecting the derivation of $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}}$, it must be the case that

$$\mathtt{e} = \mathtt{new\ B}(\overline{\mathtt{d}}')\mathtt{.<}\overline{\mathtt{P}}\mathtt{>m}(\overline{\mathtt{e}}') \qquad |\mathtt{B}| = \mathtt{D} \qquad |\overline{\mathtt{d}}'|_{\Delta,\Gamma,\mathtt{A}} = \overline{\mathtt{d}}$$
$$|\overline{\mathtt{e}}'|_{\Delta,\Gamma,\mathtt{A}} = \overline{\mathtt{e}} \qquad \mathit{mtype}(\mathtt{m},\mathtt{B}) = \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0 \qquad [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U}_0\mathtt{@B} = \mathtt{T}$$
$$\mathit{mtypemax}(\mathtt{m},\mathtt{B}) = \overline{\mathtt{E}}\mathtt{\to}\mathtt{E}_0 \qquad |\mathtt{T}|_{\Delta,\mathtt{A}} = \mathtt{C} \neq \mathtt{E}_0\ .$$

By Theorems 3.2 and 3.1, for some $\mathtt{S}$,

$$\mathit{mbody}(\mathtt{m<}\overline{\mathtt{P}}\mathtt{>},\mathtt{B}) = \overline{\mathtt{x}}\mathtt{.e}_m' \qquad \mathtt{e} \longrightarrow [\overline{\mathtt{e}}'/\overline{\mathtt{x}}, \mathtt{new\ B}(\overline{\mathtt{d}}')/\mathtt{this}]\mathtt{e}_m'$$
$$\Delta;\Gamma;\mathtt{A} \vdash [\overline{\mathtt{e}}'/\overline{\mathtt{x}}, \mathtt{new\ B}(\overline{\mathtt{d}}')/\mathtt{this}]\mathtt{e}_m' : \mathtt{S} \qquad \Delta \vdash \mathtt{S} <: \mathtt{T}\ .$$

By Theorem 4.1 and the fact that

$$|[\overline{\mathtt{e}}'/\mathtt{x}, \mathtt{new\ B}(\overline{\mathtt{d}}')/\mathtt{this}]\mathtt{e}_m'|_{\Delta,\Gamma,\mathtt{A}} = [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D}(\overline{\mathtt{d}})/\mathtt{this}]|\mathtt{e}_m'|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{S}},\mathtt{this}:\mathit{thistype}(\mathtt{B})),\mathtt{B}}$$

where $\overline{\mathtt{S}}$ are the types of $\overline{\mathtt{e}}'$, we have

$$|\Gamma|_{\Delta,\mathtt{A}} \vdash_{\mathrm{FJ}} [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D}(\overline{\mathtt{d}})/\mathtt{this}]|\mathtt{e}_m'|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{S}},\mathtt{this}:\mathit{thistype}(\mathtt{B})),\mathtt{B}} : |\mathtt{S}|_{\Delta,\mathtt{A}}\ .$$

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

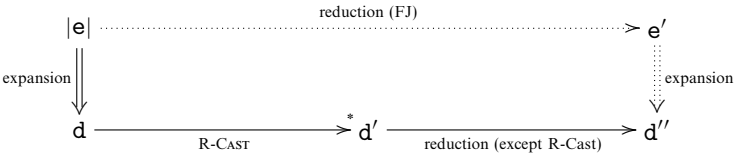

Fig. C.2. Commuting diagram of Lemma C.7.

Since $|\mathtt{S}|_{\Delta,\mathtt{A}} <:_{\mathrm{FJ}} |\mathtt{T}|_{\Delta,\mathtt{A}}$ by Lemma B.1(1),

$$|\Gamma|_{\Delta,\mathtt{A}} \vdash [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D}(\overline{\mathtt{d}})/\mathtt{this}]|\mathtt{e}_m{}'|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{S}},\mathtt{this}:\mathit{thistype}(\mathtt{B})),\mathtt{B}}$$
$$\stackrel{\mathrm{exp}}{\Longrightarrow} (|\mathtt{T}|_{\Delta,\mathtt{A}})[\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D}(\overline{\mathtt{d}})/\mathtt{this}]|\mathtt{e}_m{}'|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{S}},\mathtt{this}:\mathit{thistype}(\mathtt{B})),\mathtt{B}}\ .$$

On the other hand, since $\Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}@\mathtt{B}$, by Lemma C.5,

$$|\overline{\mathtt{x}}:\overline{\mathtt{S}}, \mathtt{this}:\mathit{thistype}(\mathtt{B})|_{\Delta,\mathtt{B}} \vdash |\mathtt{e}_m{}'|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{S}},\mathtt{this}:\mathit{thistype}(\mathtt{B})),\mathtt{B}} \stackrel{\mathrm{exp}}{\Longrightarrow} \mathtt{e}_m\ .$$

By Lemma C.1,

$$|\Gamma|_{\Delta,\mathtt{A}} \vdash [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D}(\overline{\mathtt{d}})/\mathtt{this}]|\mathtt{e}_m{}'|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{S}},\mathtt{this}:\mathit{thistype}(\mathtt{B})),\mathtt{B}} \stackrel{\mathrm{exp}}{\Longrightarrow} [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D}(\overline{\mathtt{d}})/\mathtt{this}]\mathtt{e}_m\ .$$

Then,

$$|\Gamma|_{\Delta,\mathtt{A}} \vdash (|\mathtt{T}|_{\Delta,\mathtt{A}})[\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D}(\overline{\mathtt{d}})/\mathtt{this}]|\mathtt{e}_m{}'|_{\Delta,(\overline{\mathtt{x}}:\overline{\mathtt{S}},\mathtt{this}:\mathit{thistype}(\mathtt{B})),\mathtt{B}}$$
$$\stackrel{\mathrm{exp}}{\Longrightarrow} (|\mathtt{T}|_{\Delta,\mathtt{A}})[\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D}(\overline{\mathtt{d}})/\mathtt{this}]\mathtt{e}_m\ .$$

Finally, by the fact that $\mathtt{C} = |\mathtt{T}|_{\Delta,\mathtt{A}}$ and transitivity of the expansion relation, we have

$$|\Gamma|_{\Delta,\mathtt{A}} \vdash |[\overline{\mathtt{e}}'/\mathtt{x}, \mathtt{new\ B}(\overline{\mathtt{d}}')/\mathtt{this}]\mathtt{e}_m{}'|_{\Delta,\Gamma,\mathtt{A}} \stackrel{\mathrm{exp}}{\Longrightarrow} (\mathtt{C})[\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ D}(\overline{\mathtt{d}})/\mathtt{this}]\mathtt{e}_m\ .$$

Case R-Cast**:**

Cannot happen since no casts are inserted before `new` expressions by erasure.

Case R-Field**:**

Similar to the subcase for R-Field in the case for RC-Cast above.

Case R-Invk**:**

Similar to the subcase for R-Invk in the case for RC-Cast above. The case for R-Cast and the other cases for induction steps are straightforward. □

Lemma C.7

Suppose $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}:\mathtt{T}$ and $|\Gamma|_{\Delta,\mathtt{A}} \vdash |\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} \stackrel{\mathrm{exp}}{\Longrightarrow} \mathtt{d}$. If $\mathtt{d} \longrightarrow_c^* \mathtt{d}'$ and $\mathtt{d}' \longrightarrow_n \mathtt{d}''$, then $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} \longrightarrow_{\mathrm{FJ}} \mathtt{e}'$ and $|\Gamma|_{\Delta,\mathtt{A}} \vdash \mathtt{e}' \stackrel{\mathrm{exp}}{\Longrightarrow} \mathtt{d}''$. In other words, the diagram in Figure C.2 commutes.

Proof

By induction on the derivation of the last reduction step with a case analysis by the last rule used. Refer to Igarashi *et al.* (2001) for details. □

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

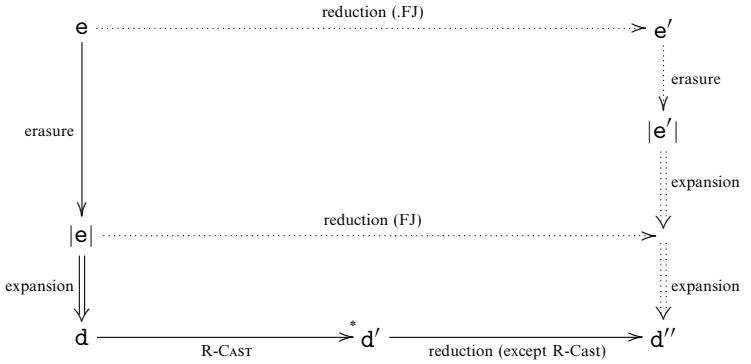

Fig. C.3. Commuting diagram of Theorem 4.3.

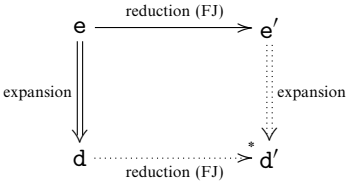

Fig. C.4. Commuting diagram of Lemma C.8.

Proof of Theorem 4.3
Follows from Lemmas C.6 and C.7. See Figure C.3. □

Finally, Theorem 4.4 is proved after the following lemma.

Lemma C.8
If $\Gamma \vdash_{\mathrm{FJ}} \mathtt{e}:\mathtt{C}$ and $\mathtt{e} \longrightarrow_{\mathrm{FJ}} \mathtt{e}'$ and $\Gamma \vdash \mathtt{e} \overset{\mathrm{exp}}{\Longrightarrow} \mathtt{d}$, then there exists some FJ expression $\mathtt{d}'$ such that $\Gamma \vdash \mathtt{e}' \overset{\mathrm{exp}}{\Longrightarrow} \mathtt{d}'$ and $\mathtt{d} \longrightarrow_{\mathrm{FJ}}{}^{*} \mathtt{d}'$. In other words, the diagram shown in Figure C.4 commutes.

Proof
By induction on the derivation of $\mathtt{e} \longrightarrow_{\mathrm{FJ}} \mathtt{e}'$ with a case analysis on the last reduction rule used. See Igarashi *et al.* (2001) for details. □

Proof of Theorem 4.4
For the first part, we first show that if $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}:\mathtt{T}$ and $\mathtt{e} \longrightarrow^{*} \mathtt{e}'$, then there exists some FJ expression $\mathtt{d}'$ such that $|\Gamma|_{\Delta,\mathtt{A}} \vdash |\mathtt{e}'|_{\Delta,\Gamma,\mathtt{A}} \overset{\mathrm{exp}}{\Longrightarrow} \mathtt{d}'$ and $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} \longrightarrow_{\mathrm{FJ}}{}^{*} \mathtt{d}'$, by induction on the length n of reduction sequence $\mathtt{e} \longrightarrow^{*} \mathtt{e}'$. The base case is trivial. As for the induction step, we have the commuting diagram shown in Figure C.5, in which commutation (1) is proved by Theorem 4.2, (2) by the induction hypothesis, and (3) by Lemma C.8.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

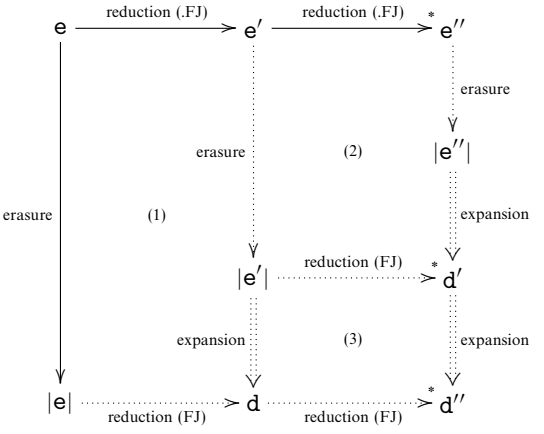

Fig. C.5. Commuting diagram in the proof of Theorem 4.4.

When e' is a value, $|\mathsf{e}'|_{\Delta,\Gamma,\mathtt{A}}$ is also a value and d' is obtained only by inserting some upcasts to $|\mathsf{e}'|_{\Delta,\Gamma,\mathtt{A}}$. So, $|\mathsf{e}'|_{\Delta,\Gamma,\mathtt{A}} \longrightarrow_{\mathrm{FJ}}{}^{*} \mathsf{d}'$.

The second part is proved as follows. First, $|\mathsf{e}|_{\Delta,\Gamma,\mathtt{A}} \longrightarrow_{\mathrm{FJ}}{}^{*} \mathtt{v}$ can be rewritten as

$$|\mathsf{e}|_{\Delta,\Gamma,\mathtt{A}} \longrightarrow_c^* \longrightarrow_n \longrightarrow_c^* \cdots \longrightarrow_n \mathsf{e}'' \longrightarrow_c^* \mathtt{v}\ .$$

By repeatedly applying Theorem 4.3, there exists a .FJ expression e' such that $\mathsf{e} \longrightarrow^{*} \mathsf{e}'$ and $|\Gamma|_{\Delta,\mathtt{A}} \vdash |\mathsf{e}'|_{\Delta,\Gamma,\mathtt{A}} \overset{\exp}{\Longrightarrow} \mathsf{e}''$. By the fact that e'' is an expansion (of $|\mathsf{e}'|_{\Delta,\Gamma,\mathtt{A}}$) and is obtained from $\mathtt{v}$ by inserting upcasts, e' must be a .FJ value $\mathtt{v}'$ and $|\mathsf{e}'|_{\Delta,\Gamma,\mathtt{A}}$ is necessarily equal to $\mathtt{v}$. □

References

Aspinall, D. & Hofmann, M. (2005) Dependent types. In *Advanced Topics in Types and Programming Languages*, Pierce, B. C. (ed), The MIT Press, pp. 45–86.

Bracha, G., Odersky, M., Stoutamire, D. & Wadler, P. (1998, October). Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pp. 103–200.

Bruce, K. B. (2003) Some challenging typing issues in object-oriented languages. In *Proceedings of Workshop on Object-Oriented Development (WOOD'03)*. Electronic Notes in Theoretical Computer Science, vol. 82, no. 8.

Bruce, K. B. & Foster, J. N. (2004) LOOJ: Weaving LOOM into Java. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2004)*. Lecture Notes on Computer Science, vol. 3086. Oslo, Norway: Springer-Verlag.

Bruce, K. B. & Vanderwaart, J. C. (1999) Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Proceedings of 15th Conference on the Mathematical Foundations of Programming Semantics (MFPS XV)*. Electronic Notes in Theoretical Computer Science, vol. 20. New Orleans, LA: Elsevier. Available at: `http://www.elsevier.nl/locate/entcs/volume20.html`.

https://doi.org/10.1017/S0956796807006405 Published online by Cambridge University Press

Bruce, K. B., Cardelli, L., Castagna, G., The Hopkins Objects Group, Leavens, G. T. & Pierce, B. (1996) On binary method. *Theory Pract. Object Systems.* **1**(3), 221–242.

Bruce, K. B., Petersen, L. & Fiech, A. (1997) Subtyping is not a good "match" for object-oriented languages. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97)*. Lecture Notes on Computer Science, vol. 1241. Jyväskylä, Finland: Springer-Verlag, pp. 104–127.

Bruce, K. B., Odersky, M. & Wadler, P. (1998) A statically safe alternative to virtual types. *Proceedings of 12th European Conference on Object-Oriented Programming (ECOOP'98)*. Lecture Notes on Computer Science, vol. 1445. Brussels, Belgium: Springer-Verlag, pp.

Canning, P., Cook, W., Hill, W., Olthoff, W. & Mitchell, J. C. (1989) F-bounded polymorphism for object-oriented programming. In *Proceedings of ACM Conference on Functional Programming and Computer Architecture (FPCA'89)*. London, England: ACM Press, pp. 273–280.

Clarke, D., Drossopoulou, S., Noble, J. & Wrigstad, T. (2007, March). Tribe: A simple virtual class calculus. In *Proceedings of International Conference on Aspect-Oriented Software Design (AOSD'07)*, pp. 121–134.

Ernst, E. (1999 June) *gbeta—A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance.* Ph.D. thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark.

Ernst, E. (2001) Family polymorphism. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2001)*. Lecture Notes on Computer Science, vol. 2072. Budapest, Hungary: Springer-Verlag, pp. 303–326.

Ernst, E. (2003). Higher-order hierarchies. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2003)*. Lecture Notes on Computer Science, vol. 2743. Darmstadt, Germany: Springer-Verlag, pp. 303–328.

Ernst, E., Ostermann, K. & Cook, W. R. (2006, January). A virtual class calculus. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL2006)*, pp. 270–282.

Flatt, M., Krishnamurthi, S. & Felleisen, M. (1998, January). Classes and mixins. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pp. 171–183.

Igarashi, A. & Viroli, M. (2007, January). Variant path types for scalable extensibility. In *Proceedings of the International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD 2007)*. Available at: `http://foolwood07.cs.uchicago.edu/`.

Igarashi, A., Pierce, B. C. & Wadler, P. (2001). Featherweight Java: A minimal core calculus for Java and GJ. *ACM transactions on programming languages and systems*, **23**(3), 396–450. A preliminary summary appeared in *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, ACM SIGPLAN Notices, vol. 34, no. 10, pp. 132–146, October 1999.

Igarashi, A., Saito, C. & Viroli, M. (2005). Lightweight family polymorphism. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS2005)*. Lecture Notes in Computer Science, vol. 3780. Tsukuba, Japan: Springer-Verlag, pp. 101–177.

Jolly, P., Drossopoulou, S., Anderson, C. & Ostermann, K. (2004, June). Simple dependent types: Concord. In *Proceedings of 6th ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP2004)*.

Madsen, O. L. & Møller-Pedersen, B. (1989, October). Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*, pp. 397–406.

Nystrom, N., Chong, S. & Myers, A. C. (2004, October) Scalable extensibility via nested inheritance. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*.

Odersky, M. (2002, January) *Inferred type instantiation for GJ*. Available at: `http://lampwww.epfl.ch/~odersky/papers/localti02.html`.

Odersky, M., Cremet, V., Röckl, C. & Zenger, M. (2003). A nominal theory of objects with dependent types. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*. Lecture Notes on Computer Science, vol. 2743. Darmstadt, Germany: Springer-Verlag, pp. 201–224.

Thorup, K. K. & Torgersen, M. (1999) Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *Proceedings of 13th European Conference on Object-Oriented Programming (ECOOP'99)*. Lecture Notes on Computer Science, vol. 1628. Lisbon, Portugal: Springer-Verlag, pp. 186–204.

Torgersen, M. (2004, June) The expression problem revisited: Four new solutions using generics. *Proceedings of European Conference on Object-Oriented Programming (ECOOP2004)*. Lecture Notes on Computer Science, vol. 3086, pp. 123–146.

Wright, A. K. & Felleisen, M. (1994). A syntactic approach to type soundness. *Inform. and Comput*. **115**(1), 38–94.