Expressive Modular Linking for Object-Oriented Languages

UUCS-02-014

Sean McDirmid, Wilson C. Hsieh, Matthew Flatt
School of Computing
University of Utah
{mcdirmid,mflatt,wilson}@cs.utah.edu

October 11, 2002

In this paper we show how modular linking of program fragments can be added to statically typed, object-oriented (OO) languages. Programs are being assembled out of separately developed software components deployed in binary form. Unfortunately, mainstream OO languages (such as Java) still do not provide support for true modular linking. Modular linking means that program fragments can be separately compiled and type checked, and that linking can ensure global program type correctness without analyzing program fragment implementations.

Supporting modular linking in OO languages is complicated by two expressive features that current OO languages do not support together: mixin-style inheritance across program fragment boundaries, and cyclic dependencies between program fragments. In a previous paper at OOPSLA 2001, we have demonstrated the practical uses for such expressiveness. When such expressiveness is permitted, link-time type checking rules must ensure that method collisions and inheritance cycles do not occur after program fragments are linked into a program. In this paper, we show how modular linking with both cyclic linking and mixin-style inheritance can be supported using a type-checking architecture that can be added on top of existing OO languages, such as Java.

Expressive Modular Linking for Object-Oriented Languages

Sean McDirmid, Wilson C. Hsieh, Matthew Flatt

Abstract

In this paper we show how modular linking of program fragments can be added to statically typed, object-oriented (OO) languages. Programs are being assembled out of separately developed software components deployed in binary form. Unfortunately, mainstream OO languages (such as Java) still do not provide support for true modular linking. Modular linking means that program fragments can be separately compiled and type checked, and that linking can ensure global program type correctness without analyzing program fragment implementations.

Supporting modular linking in OO languages is complicated by two expressive features that current OO languages do not support together: mixin-style inheritance across program fragment boundaries, and cyclic dependencies between program fragments. In a previous paper at OOPSLA 2001, we have demonstrated the practical uses for such expressiveness. When such expressiveness is permitted, link-time type checking rules must ensure that method collisions and inheritance cycles do not occur after program fragments are linked into a program. In this paper, we show how modular linking with both cyclic linking and mixin-style inheritance can be supported using a type-checking architecture that can be added on top of existing OO languages, such as Java.

1 Introduction

Linking is the process of turning a collection of program fragments into a complete program. **Modular** linking is important, because programs are being assembled out of separately developed components deployed in binary form. Modular linking requires that the implementation of a program fragment (fragment, for short) be compiled and type-checked separately from foreign fragments [4]. Modular linking has many benefits: it simplifies dependency management, makes program compilation and linking more efficient, promotes local reasoning of fragments, makes dynamic linking of fragments more straightforward, and makes compilation and linking errors easier for users to reason about because the errors are not related to hidden implementation details. Modular linking has been extensively studied in the context of functional languages, where function constructs especially facilitate modular linking, but modular linking for object-oriented (OO) languages is still an active area of research.

Classes in OO languages encapsulate and permit the reuse of object implementations using instantiation and inheritance. Classes and fragments address different development concerns and so should be supported with distinct constructs [21], where a fragment can conceptually be a set of classes. Unlike fragments that contain functions, modular linking of fragments that contain classes is difficult: class inheritance can create complex dependencies when used between classes in different fragments. Such a use of class inheritance is similar to *mixin-style inheritance* [3, 12]: superclass implementations are not known until linking. Mixins have well-known type-checking challenges related to ambiguous methods: a subclass may “introduce” a method that conflicts with a method already provided by a superclass at link-time.

Existing OO languages that support modular linking do so by restricting mixin-style inheritance, disallowing cyclic dependencies between fragments, or by severely limiting what can be hidden between fragments [2, 10, 16]. In this paper, we show how separate compilation and modular linking can be supported in an OO language without such restrictions on expressiveness. In previous work [18] we have demonstrated the value of removing these restrictions. For this paper, we use a small language called MiniJiazzi with a Java-like core language and a linking language based on program units [11]. MiniJiazzi forms the theoretical foundation for our previous work on Jiazzi [18], which enhances Java’s linking model with support for separately compiled and externally linked fragments.

We have included 5 pages of appendices that contain type rules and proof sketches, which the referees should feel free to ignore. Ignoring those pages, the paper consists of less than 11 pages of text and figures and 1 page of bibliography.

```
fragment base {
 class BaseWidget  extends  Object { Object paint() { return ...  } }
 class BaseButton  extends  Widget {...}
}
fragment win32 {
 class Widget  extends  BaseWidget {
  Object winpaint(int x) { return ...  }
  Object paint() { return ...  }
 }
 class Button  extends  BaseButton { Object winpaint(int x) { return ...  }}
}
fragment motif {
 class Widget  extends  BaseWidget {
  Object motifpaint(int x) { return ...  }
  Object paint() { return ...  }
 }
 class Button  extends  BaseButton { Object motifpaint(int x) { return ...  }}
}
```

Figure 1: A set of fragments that implement either Win32 or Motif UI widget classes, depending on whether the fragment `base` is linked against the `win32` or `motif` fragments.

Issues related to the combination of modular linking and object-oriented languages have been previously explored in the areas of managing virtual method namespaces [19, 22], merging module systems and OO languages [2, 10, 16], and reasoning about mixin-style inheritance [3, 12]. The work presented in this paper is the first to show how to add modular linking to an OO language and support both cyclic linking and expressive mixin-style inheritance. Our type-checking architecture is also novel. Modular type checking in MiniJiazzi ensures that compile-time type checking and link-time type checking environments are **congruent** with an intermediate type checking environment that does not change between compile-time and link-time. Defining such congruence is key to the correctness and expressiveness of the linking language.

Section 2 briefly motivates modular linking for an OO language and describes the challenges that must be overcome when adding modular linking to an OO language. Section 3 introduces the MiniJiazzi language and describes our type checking methodology for adding modular linking to an OO language. Section 4 shows how we deal with cyclic dependencies between fragments in the presence of mixin-style inheritance. Section 5 discusses related work and Section 6 summarizes our conclusions.

2 Motivation

We consider a *fragment* to be a collection of classes compiled together into a single binary. Linking combines multiple fragments into a program that is a collection of all of the classes in the fragments. In OO languages like Java, there is no explicit support for fragments; instead, they are defined indirectly by a developer using constructs such as JAR files, codebases, and packages. In this section, we use examples from Java and explicitly define logical fragment boundaries. For the rest of this paper, we use Java as our prototypical OO language.

Fragments in Java can be linked with fragments that they were not compiled with, subject to certain binary compatibility [7, 13] rules. Binary compatibility supports fragment evolution: it allows an older application to be linked with a newer version of a class library, and it also supports component software development. Consider the three fragments in Figure 1 that implement parts of user interface (UI) widget classes. The `base` fragment provides a base implementation of UI widget classes while the `win32` and `motif` fragments add platform-specific implementations to the UI widget classes. Linking the `base` fragment with either the `win32` or `motif` fragments (but not both) yields a configuration of the UI widget classes that is appropriate for a specific platform.

The example in Figure 1 demonstrates the utility of mixin-style inheritance across fragment boundaries and cyclic linking of fragments. A special inheritance pattern is used in the fragment `base` to ensure that the button class is a subclass of all the widget classes. Rather than having the class `BaseButton` inherit directly from class `BaseWidget`, it in-

```
fragment person_v1 {
 class Person  extends  Object {}
}
fragment person_v2 {
 class Person  extends  Icon {}
}
fragment cowboyicon {
 class Cowboy  extends  Person {
  Object duel() { return this.draw(); }
  Object draw() { return ... /* cowboys draw guns */ }
 }
 class Icon  extends  Object {
  Object paint() { return this.draw(); }
  Object draw() { return ... /* draw an icon */ }
 }
}
```

Figure 2: The program fragment `cowboyicon` is originally compiled against $\texttt{person}_{\texttt{v1}}$ but is actually linked against program fragment $\texttt{person}_{\texttt{v2}}$.

herits directly from the class `Widget` provided by one of the platform-specific fragments, which in turn inherits from the class `BaseWidget`—an exported class inherits from an imported class! This technique allows enhancements to the widget class to also enhance the button class by using a cyclic dependency between the `base` fragment and either of the platform-specific fragments. The fragments in Figure 1 allow methods and fields to be modularly added to existing classes. This usage of fragments is a static alternative to design patterns such as abstract factories [14], that support dynamic configuration of feature sets. Our previous work [18] discusses the benefits of and mechanics behind this usage of fragments in Jiazzi at greater length.

In normal Java, the linking in Figure 1 can be implemented by creating a custom classloader to implement the required linking logic. However, such linking is fragile, because Java does not support modular linking; it depends on global type checking after link-time to ensure global program type correctness [7, 17]. Global type checking complicates dynamic loading and makes it difficult to debug linking errors. Java's global checking also cannot detect method name ambiguities, because its class signatures do not contain enough information.

Cyclic linking of fragments and mixin-style inheritance of classes between fragments complicates modular type checking: in particular detection of method name ambiguities. When a method in a superclass has the same name as a "new" method in a subclass, a linker could either resolve the ambiguity automatically or reject the mixin instantiation as a method collision. Automatically resolving the ambiguity can cause separate development of components to go awry, as shown by the three fragments in Figure 2. The fragment `cowboyicon` is originally compiled against the fragment $\texttt{person}_{\texttt{v1}}$, but it is actually linked into a program with fragment $\texttt{person}_{\texttt{v2}}$. When the programmer developed the fragment `cowboyicon`, the class `Cowboy` was not a subclass of the class `Icon`, so the method *`draw`* in class `Cowboy` is not intended to override the method *`draw`* in class `Icon`. However, an inheritance relationship between the classes `Person` and `Icon` is created in $\texttt{person}_{\texttt{v2}}$ that did not exist in $\texttt{person}_{\texttt{v1}}$, and as a result `Cowboy` becomes a subclass of `Icon`, creating cowboy icons that draw their guns when painted! Although the program created by linking the fragments `cowboyicon` and $\texttt{person}_{\texttt{v2}}$ is correctly typed, the ambiguity in Figure 2 cannot be detected in Java, because Java is not designed to detect (and reject) method collisions of this sort.

3 MiniJiazzi

MiniJiazzi consists of a small Java-like core language, which is used to implement the classes in fragments, and a separate linking language, which is used to link fragments together into a program. The MiniJiazzi core language is independent from the linking language with its own separate properties, which allows us to modularize our reasoning and soundness proof of the entire MiniJiazzi language. The separation of the core language and linking language is also reflected in Jiazzi [18]; Jiazzi is implemented as an enhancement to Java and does not require changes to the Java core language or its core type checking rules. We forgo an detailed discussion of the MiniJiazzi core language so that the abstract relationship between the linking language and the core language can be emphasized. A detailed description of the MiniJiazzi core language, whose syntax, type checking, and evaluation are similar to other small Java-like languages [12, 15], is given in Appendix A.

Notation: Uppercase letters identify constructs: e.g., A, B, and C identify classes and U, V, and W identify fragments; lowercase letters (possibly subscripted) designate constructs: e.g., $\mathtt{c}_a$ designates a class expression and $\mathtt{s}_b$ designates a class signature. Rules are in small caps, e.g., RULE-OK. Directed overbars ($\overrightarrow{\ldots}$) are vectors that designate a sequence, e.g., $\overrightarrow{\mathtt{A}}$ is a sequence of class identifiers and $\overrightarrow{\mathtt{u}}$ is a sequence of unit expressions. Multiple sequences can be joined using the union operator ($\cup$). A rule within a vector applies to all elements specified by the vector. Terms adjacent to the rule but outside of the vector are copied as the vector is expanded. For example, if $\overrightarrow{\mathtt{A}} = \mathtt{B}\ \mathtt{C}$ then D $\overrightarrow{\textbf{Rule-Ok}\ \mathtt{A}}$ expands to D **RULE-OK** B and D **RULE-OK** C.

3.1 Core Language

The MiniJiazzi core language is used to construct class expressions (designated by "c"), which are similar to Java class definitions: they express subclassing relationships with other classes and implement methods (which possibly override methods defined in super classes). Compilation and type checking of a class expression requires a typing environment, which describes the structure of the class expression's dependencies. While these dependencies are other class expressions, a typing environment is not formed directly from class expressions. Instead a typing environment can be formed from *class signatures* (designated by "s"), which describe the subclassing relationships and available methods of corresponding class expressions without exposing implementation details. Finally, class signatures must declare what "new" methods they introduce, so that mixin constructions can be checked appropriately.

Type checking of a class expression ensures that a class expression is well-formed with respect to a well-formed typing environment. A typing environment is well-formed if the following criteria are ensured: first, a typing environment must be closed: no classes can be referenced by the typing environment unless they are described in the typing environment; second, there are no inheritance cycles expressed within the typing environment: all classes are direct or indirect subclasses of Object; third, there are no method collisions within the typing environment: no method is introduced by a subclass if that method is already introduced by a superclass.

A class expression is well-formed with respect to a typing environment if its methods are implemented correctly according to the typing environment: e.g., instantiated classes and invoked methods must be described by the typing environment. The rule ENVIRONMENT-OK ensures that a typing environment is well-formed, and the rule EXPRESSION-OK ensures that a class expression is well-formed in the presence of a typing environment. Without considering the linking language, a program in the MiniJiazzi core language is a sequence of class expressions ($\overrightarrow{\mathtt{c}}$) that can be type checked together to ensure program type correctness using the rule GLOBAL-OK, which is defined in the following judgement:

$$\frac{\vdash_{tc} \textbf{Environment-Ok}\ \overrightarrow{\|\ \mathtt{c}\ \|} \qquad \overrightarrow{\|\ \mathtt{c}\ \|} \vdash_{tc} \overrightarrow{\textbf{Expression-Ok}\ \mathtt{c}}}{\vdash_{tc} \textbf{Global-Ok}\ \overrightarrow{\mathtt{c}}}$$

The typing environment is formed by combining the class signatures from all class expressions of the program; the signature extraction operator returns the signature of a class (where $\|\ \mathtt{c}\ \| = \mathtt{s}$). GLOBAL-OK is not a modular rule, and is only presented here as a point of comparison. We assume that type checking described by the rule GLOBAL-OK and standard evaluation rules are sound, where a soundness proof is similar to other Java-like small languages [12, 15].

3.2 Linking Language

In order for MiniJiazzi to support modular linking, it must support fragments that are subject to deployment and composition into a program. In MiniJiazzi, a fragment is realized as a *unit expression* (designated by "u"). A unit expression has a *signature* (designated by "v"), which describes the unit's interface with other units, and an *implementation*, which is a sequence of class expression. A unit signature consists of the signatures of imported classes that are provided by other units and the signatures of exported classes that are provided to other units. Here is the syntax of a unit, and the definition of the signature extraction operator ($\| \mathtt{u} \| = \mathtt{v}$).

$$\mathtt{u} ::= \mathtt{v}_{sig} \; \{ \; \overrightarrow{\mathtt{c}_{impl}} \; \} \qquad \mathtt{v} ::= \mathtt{U} \; \{ \; \overrightarrow{\mathtt{s}_{import}} \; \overrightarrow{\mathtt{s}_{export}} \; \} \qquad \| \, \mathtt{u} = \mathtt{v}_{sig} \; \{ \; \overrightarrow{\mathtt{c}_{impl}} \; \} \, \| = \mathtt{v}_{sig}$$

We use `U`, `V`, and `W` as unit identifiers. If type checking using the MiniJiazzi linking language is to be modular, type checking must be separated into two phases. *Compile-time* type checking ensures that a unit is internally consistent; it only examines one unit at a time. *Link-time* type checking ensures that units are externally consistent with each other: it ensures that all classes imported into linked units are properly satisfied by classes exported from linked units. Imported and exported classes are expressed in unit signatures, so unit implementations do not need to be examined during link-time type checking. A program in MiniJiazzi is a sequence of unit expressions ($\overrightarrow{\mathtt{u}}$) that is type-checked to ensure program type correctness using the rule MODULAR-OK:

$$\frac{\vdash_{tc} \overrightarrow{\textbf{COMPILE-OK}\ \mathtt{u}} \quad \vdash_{tc} \textbf{LINK-OK}\ \overrightarrow{\| \mathtt{u} \|}}{\vdash_{tc} \textbf{MODULAR-OK}\ \overrightarrow{\mathtt{u}}}$$

Where the rule COMPILE-OK implements compile-time type checking and the rule LINK-OK implements link-time type checking. Given a definition of *link reduction*, which is the process of transforming classes from their unlinked form within a unit to their linked form within a program, we can compare rules MODULAR-OK and GLOBAL-OK with Lemma 1:

Lemma 1 (MODULAR-IMPLIES-GLOBAL) *If a program of units is type correct using modular type-checking rules, then the link-reduced units, which together yield a program of classes, are type-correct using global type checking rules.*

$$\vdash_{tc} \textbf{MODULAR-OK}\ \overrightarrow{\mathtt{u}} \quad \overrightarrow{\| \mathtt{u} \| \vdash_l \mathtt{u} \rightarrow \overrightarrow{\mathtt{c}}} \quad \Rightarrow \quad \vdash_{tc} \textbf{GLOBAL-OK}\ \overrightarrow{\overrightarrow{\mathtt{c}}}$$

Link reduction is performed using the $\vdash_l$ judgement from a unit to its link-reduced classes. Lemma 1 can be proven given the appropriate definitions of the type-checking rules COMPILE-OK and LINK-OK, described in Section 3.3, and link reduction, described in Section 3.4.

3.3 Congruence

Global type checking of a monolithic program with the rule GLOBAL-OK uses only one typing environment; modular type checking of a program composed out of units consists of three kinds of typing environments. A *compilation typing environment* is used to perform compile-time type checking of a unit's implementation. A *linkage typing environment* is used to perform link-time type checking between units as they are linked into a complete program. Finally, an *intermediate typing environment* acts as a bridge between the compilation and linkage typing environments, so modular type checking can ensure global program type correctness.

Separate compilation is enabled in Jiazzi by having unit signatures contain the signatures of imported classes. To form the compilation typing environment of a unit, the class signatures extracted from the class expressions of the unit's implementation are combined with imported class signatures as $\overrightarrow{\| \mathtt{c}_{impl} \|} \cup \overrightarrow{\mathtt{s}_{import}}$. Since units are compiled separately, linking a program in MiniJiazzi must ensure that a unit's imports are adequately satisfied by classes exported from other unit's of the program. Unit signatures also embed the signatures of exported classes, which avoids examining unit implementations during linking. A program need only be relinked when the signature of a linked unit changes. The linkage typing environment is formed from the exported class signatures of every unit as $\overrightarrow{\overrightarrow{\mathtt{s}_{export}}}$.

Compile-time type checking of a unit is implemented in the rule COMPILE-OK, which ensures that each class expression contained within the unit is well-formed (using the core language type checking rule EXPRESSION-OK with respect to the unit's compilation typing environment). Link-time type checking of a unit is implemented in the rule LINK-OK. The intermediate typing environment is formed from the signatures of a unit's imported and exported classes as $\overrightarrow{\texttt{s}_{import}} \cup \overrightarrow{\texttt{s}_{import}}$. All typing environments must be well-formed, which is ensured using the core language type checking rule ENVIRONMENT-OK described earlier.

The key to making type checking modular in MiniJiazzi is ensuring that the compilation typing environments of each linked unit and linkage typing environment are compatible. This can be achieved by ensuring that for each unit, the unit's compilation typing environment and the linkage typing environment are **congruent** with the unit's intermediate typing environment. The CONGRUENT rule enforces this relationship, and is used to define the rules COMPILE-OK and LINK-OK:

$$\frac{\begin{array}{cc} \vdash_{tc} \textbf{Environment-Ok}\ \overrightarrow{\texttt{s}_i} \cup \overrightarrow{\|\ \texttt{c}\ \|} & \overrightarrow{\texttt{s}_i} \cup \overrightarrow{\|\ \texttt{c}\ \|} \vdash_{tc} \overrightarrow{\textbf{Expression-Ok}\ \texttt{c}} \\ \vdash_{tc} \textbf{Environment-Ok}\ \overrightarrow{\texttt{s}_i} \cup \overrightarrow{\texttt{s}_e} & \vdash_{tc} \overrightarrow{\texttt{s}_i} \cup \overrightarrow{\|\ \texttt{c}\ \|}\ \textbf{Congruent}\ \overrightarrow{\texttt{s}_i} \cup \overrightarrow{\texttt{s}_e} \end{array}}{\vdash_{tc} \textbf{Compile-Ok}\ \texttt{u} = \texttt{U}\ \{\ \overrightarrow{\texttt{s}_i}\ \overrightarrow{\texttt{s}_e}\ \}\ \{\ \overrightarrow{\texttt{c}}\ \}} \qquad \frac{\begin{array}{c} \vdash \textbf{Unique}\ \overrightarrow{\texttt{U}} \\ \vdash_{tc} \textbf{Environment-Ok}\ \overrightarrow{\overrightarrow{\texttt{s}_e}} \\ \vdash_{tc} \overrightarrow{\overrightarrow{\texttt{s}_e}}\ \overrightarrow{\textbf{Congruent}\ \overrightarrow{\texttt{s}_e} \cup \overrightarrow{\texttt{s}_i}} \end{array}}{\vdash_{tc} \textbf{Link-Ok}\ \overrightarrow{\texttt{v} = \texttt{U}\ \{\ \overrightarrow{\texttt{s}_i}\ \overrightarrow{\texttt{s}_e}\ \}}}$$

Congruence ensures that the typing environment to the left of the rule CONGRUENT, known as the *source typing environment*, is substitutable with the typing environment on the right of the rule CONGRUENT, known as the *sink typing environment*. This substitutability ensures that modular type checking can ensure that the resulting *program typing environment*, which is formed from the results of all link-reduced units, is also well-formed. We state this as Lemma 2:

Lemma 2 (PROGRAM-ENVIRONMENT-WELL-FORMED) *If compilation, intermediate, and linkage typing environments are well-formed, compilation typing environments are congruent with intermediate typing environments, and the linkage typing environment is congruent with all intermediate typing environments, then the linked reduced classes of a unit program together constitute a well-formed program typing environment.*

$$\begin{array}{c} \overrightarrow{\texttt{u} = \texttt{v}\ \{\ \overrightarrow{\texttt{c}_x}\ \}} \quad \overrightarrow{\texttt{v} = \texttt{U}\ \{\ \overrightarrow{\texttt{s}_i}\ \overrightarrow{\texttt{s}_e}\ \}} \quad \vdash_{tc} \textbf{Environment-Ok}\ \overrightarrow{\overrightarrow{\texttt{s}_i} \cup \overrightarrow{\|\ \texttt{c}_x\ \|}} \quad \vdash_{tc} \overrightarrow{\overrightarrow{\texttt{s}_i} \cup \overrightarrow{\|\ \texttt{c}_x\ \|}\ \textbf{Congruent}\ \overrightarrow{\texttt{s}_i} \cup \overrightarrow{\texttt{s}_e}} \\ \vdash_{tc} \textbf{Environment-Ok}\ \overrightarrow{\overrightarrow{\texttt{s}_i} \cup \overrightarrow{\texttt{s}_e}} \quad \vdash_{tc} \overrightarrow{\overrightarrow{\texttt{s}_e}}\ \overrightarrow{\textbf{Congruent}\ \overrightarrow{\texttt{s}_i} \cup \overrightarrow{\texttt{s}_e}} \quad \vdash_{tc} \textbf{Environment-Ok}\ \overrightarrow{\overrightarrow{\texttt{s}_e}} \\ \vdash \textbf{Unique}\ \overrightarrow{\texttt{U}} \quad \overrightarrow{\texttt{v}} \vdash_l \overrightarrow{\texttt{u} \ \to\ \overrightarrow{\texttt{c}_y}} \quad \Rightarrow \quad \vdash_{tc} \textbf{Environment-Ok}\ \overrightarrow{\overrightarrow{\|\ \texttt{c}_y\ \|}} \end{array}$$

Proving Lemma 2 is a large part of the proof for Lemma 1. The rule CONGRUENT must at least ensure that the source typing environment provides every class, method, and subclassing relationship that exists in the sink typing environment. However, this weakest definition of congruence would not be sufficient to detect inheritance cycles and method collisions in the presence of cyclic linking. The strongest definition of CONGRUENT would ensure that the source typing environment is equivalent to the sink typing environment—that the source typing environment does not provide any class, method, or subclassing relationship that does not exist in the sink typing environment. While this strongest definition of congruence would be safe, it is too restrictive: every class, method, and subclassing relationship of every unit would have to be visible. Such a requirement would defeat the purpose of separate compilation and would prevent a unit from being reused in multiple programs. In Section 4 we define congruence in a way that is safe and also allows for sufficient information hiding.

3.4 Link Reduction

When units are combined together into a linked program, linking must ensure that name clashes between unit implementations do not cause ambiguities at runtime. Name clashes in OO languages result from classes from different units with the same name, or distinct methods with the same names that are defined in compatible classes that originate from different units. While modular type checking can detect name clashes between units if classes and methods are exposed in unit signatures, all but the strongest congruence conditions permit some hiding between units. Finally, classes and methods hidden within a unit's implementation should not clash with classes and methods from other units. Link reduction ensures that references to hidden classes and methods are disambiguated during linkage.

MiniJiazzi achieves disambiguation with *linking offsets*. Every class and method reference within a unit implementation is qualified with a linking offset that is utilized during evaluation to disambiguate references. Method expressions also have linking offsets to ensure that they implement or override the appropriate method from a superclass. Linking offsets are not provided by the programmer; they are assigned by the linker, in much the same way as branch offsets are rewritten when a dynamically-linked library (DLL) is loaded into memory. Before link reduction occurs, all linking offsets of a unit's implementation are empty (type checking does not use linking offsets). Link reduction then binds linking offsets according to the unit that the class or method originates from. The following are judgments that determine how linking offsets are bound for class references:

$$\frac{\mathtt{A}[\circ] \in \overrightarrow{|\ \mathtt{s}_{iu}\ |} \quad \mathtt{A}[\circ] \in \overrightarrow{|\ \mathtt{s}_{ew}\ |} \quad \mathtt{W}\ \overrightarrow{\mathtt{s}_{ew}} \in \overrightarrow{\mathtt{V}\ \mathtt{s}_e}}{\overrightarrow{\mathtt{v} = \mathtt{V}\ \{\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \}}, \mathtt{u} = \mathtt{U}\ \{\ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}}\ \}\ \{\ \overrightarrow{\mathtt{c}}\ \} \vdash_l \mathtt{A} \mapsto \mathtt{W}} \qquad \frac{\mathtt{A}[\circ] \notin \overrightarrow{|\ \mathtt{s}_{iu}\ |}}{\overrightarrow{\mathtt{v} = \mathtt{V}\ \{\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \}}, \mathtt{u} = \mathtt{U}\ \{\ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}}\ \}\ \{\ \overrightarrow{\mathtt{c}}\ \} \vdash_l \mathtt{A} \mapsto \mathtt{U}}$$

Classes are always identified by the terms A, B, C, and D and an identifier extraction operator returns the name of a class from a class signature ($|\ \mathtt{s}\ | = \mathtt{A}[\mathtt{W}]$). The symbol $[\circ]$ indicates an empty linking offset and linking offsets are always in brackets ($[\mathtt{U}]$). The map operator ($\mapsto$) is used to find the appropriate linking offset for a class reference in a unit u. For a reference to an imported class, the linking environment is used to determine which unit an imported class is provided by, using that unit as the linking offset. The linking offsets of references to classes that are not imported into the referring unit are bound to the referring unit as these references are to classes contained within the unit. Method reference linking offsets are bound using a similar set of judgments:

$$\frac{\begin{array}{c}\overrightarrow{\mathtt{s}_{iu}} \cup \overrightarrow{\|\ \mathtt{c}\ \|} \vdash_m \langle \mathtt{A}[\circ].\mathtt{M}[\circ] \rangle = \mathtt{B}[\circ] \quad \mathtt{B}[\circ] \in \overrightarrow{|\ \mathtt{s}_{iu}\ |} \\ \overrightarrow{\mathtt{s}_e} \vdash_m \langle \mathtt{B}[\circ].\mathtt{M}[\circ] \rangle = \mathtt{C}[\circ] \quad \overrightarrow{\mathtt{s}}, \mathtt{u} \vdash_l \mathtt{C} \mapsto \mathtt{W}\end{array}}{\overrightarrow{\mathtt{v} = \mathtt{V}\ \{\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \}}, \mathtt{u} = \mathtt{U}\ \{\ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}}\ \}\ \{\ \overrightarrow{\mathtt{c}}\ \} \vdash_l \mathtt{A.M} \mapsto \mathtt{W}} \qquad \frac{\overrightarrow{\mathtt{s}_{iu}} \cup \overrightarrow{\|\ \mathtt{c}\ \|} \vdash_m \langle \mathtt{A}[\circ].\mathtt{M}[\circ] \rangle = \mathtt{B}[\circ] \quad \mathtt{B}[\circ] \notin \overrightarrow{|\ \mathtt{s}_{iu}\ |}}{\overrightarrow{\mathtt{v} = \mathtt{V}\ \{\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \}}, \mathtt{u} = \mathtt{U}\ \{\ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}}\ \}\ \{\ \overrightarrow{\mathtt{c}}\ \} \vdash_l \mathtt{A.M} \mapsto \mathtt{U}}$$

The method introduction operator $\overrightarrow{\mathtt{s}} \vdash_m \langle \mathtt{A}[\mathtt{U}].\mathtt{M}[\mathtt{V}] \rangle = \mathtt{B}[\mathtt{W}]$ determines that the class $\mathtt{B}[\mathtt{W}]$, a super class of $\mathtt{A}[\mathtt{U}]$, introduces the method $\mathtt{M}[\mathtt{V}]$ as expressed in the typing environment $\overrightarrow{\mathtt{s}}$. If, according to the compilation typing environment, the class that introduces a method is not imported, then the linking offset of the method is bound to the referencing unit. Otherwise, the linking offset of a method reference is bound to the unit that exports the superclass that introduces the method, which can be located using the linkage typing environment.

The MiniJiazzi units shown in Figure 3 demonstrate how link reduction resolves method ambiguity. The unit `icon` specifies what it expects from other units through its imports. Since the unit `icon` does not import the method *draw* from the `cowboy` unit, within the implementation of class `Icon` link reduction binds the linking offset of the call to method *draw* to the unit `icon`. Since the unit `cowboy` exports the method *draw*, whereas the unit `icon` does not, linking reduction binds the linking offset of the call to *draw* within the class `Main` to the unit `cowboy`.

In Java, method scope is established by packages and access flags. If the above example was written in normal Java using packages and access flags (the method *draw* in a package `cowboy` would be public, and the method *draw* in a package `icon` would be private), ambiguity between the *draw* methods could not be avoided and the Java source compiler would reject such a construction. Using unit signatures to establish the namespace of a unit makes MiniJiazzi more flexible than Java, because a unit can restrict its imported namespace as well as its exported namespace.

Rather than use linking offsets, other approaches [19, 20, 22] disambiguate between methods using dictionaries that are based on scope. Linking offsets simulate dictionaries by using automatic renaming of class and method references. MiniJiazzi avoids the use of dictionaries during evaluation, which allows us to reason more easily about the differences between the linking language and the pure core language: we do not require a separate set of evaluation rules for programs that use the linking language.

```
unit cowboy {
 import
 export class Cowboy extends Object {  Object draw(), Object dual() }
} {
 class Cowboy[cowboy] extends Object {
  introduces Object draw()
 } {
  Object draw[cowboy]()  { .../* draw guns in a duel */ },
  Object dual[cowboy]()  { this.draw[cowboy]() }
 }
}
unit icon {
 import class Cowboy extends Object {   }
 export class Icon extends Cowboy {  Object paint()  }
} {
 class Icon[icon] extends Cowboy[cowboy] {
  introduces Object paint(), Object draw()
 } {
  Object paint[icon]()  { this.draw[icon]() },
  Object draw[icon]()  { .../* draw an icon */ }
 }
}
unit main {
 import class Icon extends Object {  Object paint(), Object draw() }
 export class Main extends Object {  Object main()  }
} {
 class Main[main] extends Object {
  introduces Object main()
 } {
  Object main[main]()  { this.draw[cowboy]() }
 }
}
```

Figure 3: Three linked units in MiniJiazzi that demonstrate how linking offsets eliminate ambiguity in the *draw* method for classes Cowboy and Icon.

Link reduction only binds class and method reference linking offsets; it does not otherwise change the structure of class expressions within a unit. The weakest congruence conditions ensure that referenced classes and methods exist within the program typing environment and that method implementations override methods correctly. As a result, showing that the classes that result from link-reduced unit expressions are well-formed with respect to the program typing environment is only a matter of structural induction. The only nontrivial aspect of proving Lemma 1 is proving Lemma 2. We concentrate on the proof, which depends on the proper definition of the rule CONGRUENT, in Section 4.

4 Congruence

Congruence between typing environments is complicated by the fact that fragments can be linked in cycles. With cyclic linking, the weakest form of congruence (where a congruent typing environment is only required to have every class, method, and subclassing relationship expressed by the target environment) is insufficient to ensure global program type correctness. The strongest form of congruence is not desirable either, as it prevents any hiding between fragments. Finding a definition of congruence "in the middle" is complicated by the fact that class inheritance can span fragment boundaries: allowing a congruent typing environment to provide too many methods or subclassing relationships could lead to inheritance cycles or method collisions. Supporting cyclic dependencies between fragments is also problematic in many other modular languages (e.g., ML [6]).

For OO languages, we have discovered an effective compromise between the weakest and strongest congruence conditions. In addition to providing every subclassing relationship of the target typing environment, a source typing environment must express only subclassing relationships that do not conflict with subclassing relationships of the sink typing environment. If two classes in the sink typing environment do not have a subclassing relationship, then those

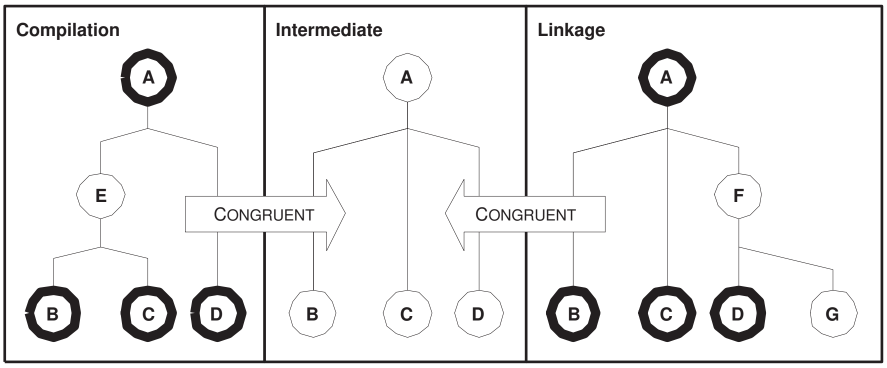

Figure 4: Three class-inheritance hierarchies expressed by compilation, intermediate, and linkage typing environments; classes in the compilation and linkage typing environments that are also in the intermediate typing environment are in bold; congruence conditions are indicated as arrows.

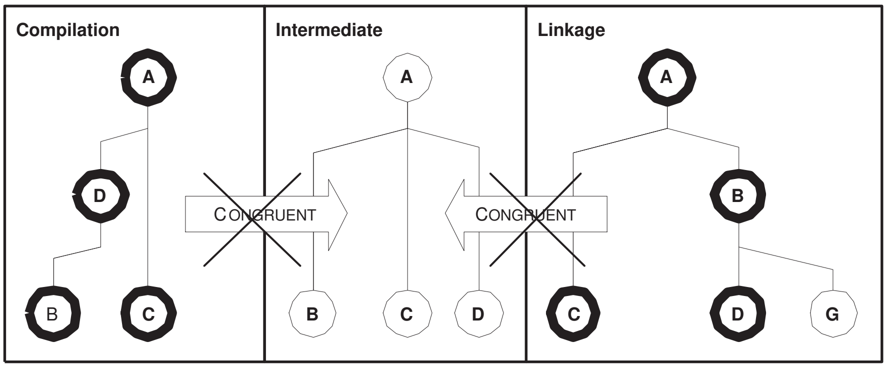

Figure 5: Examples of compilation and linkage typing environments that are not congruent with the intermediate typing environment.

two classes in the source typing environment must also not have a subclassing relationship. This congruence condition allows classes and methods to be hidden between fragments and still allows modular type checking to detect method collisions and inheritance cycles.

In Figure 4, both the compilation and linkage typing environments are congruent with the intermediate typing environment. The intermediate typing environment contains the signatures for the classes `A`, `C`, `D`, and `E`, where `D`, `C`, and `E` are subclasses of `A`. The compilation typing environment contains the signature for class `B` in addition to classes contained in the intermediate typing environment. The compilation typing environment is congruent with the intermediate typing environment because classes `C` and `D` are also subclasses (albeit indirect) of `A` in the compilation typing environment, and no new subclassing relationship between `A`, `C`, `D`, and `E` is expressed in the compilation typing environment that is not already expressed in the intermediate typing environment. A similar argument shows that the linkage typing environment is congruent with the intermediate typing environment.

In Figure 5, neither the compilation nor the linkage typing environments are congruent with the intermediate typing environment. While each typing environment expresses every subclassing relationship expressed by the intermediate typing environment, each establishes a subclassing relationship incompatible with the intermediate typing environment. Both the compilation and linkage typing environments express a subclassing relationship between classes `B` and `D`, while the intermediate typing environment expresses no subclassing relationship between these two classes. As a result, if program linking were permitted, an inheritance cycle would be created between classes `B` and `D`.

In MiniJiazzi, the rule CONGRUENT takes the following form:

$$\frac{\overrightarrow{\mathtt{s}_{source}} \vdash_t \mathtt{A}[\circ] \lhd \mathtt{t}_s}{\overrightarrow{\mathtt{s}_{source}}, \overrightarrow{\mathtt{s}_{sink}} \vdash_{tc} \mathtt{A}\ \textbf{SUPER-OK}\ \mathtt{t}_s}$$

$$\frac{\overrightarrow{\mathtt{s}_{source}} \vdash_t \mathtt{A}[\circ] \lhd \mathtt{B}[\circ] \quad \mathtt{B}[\circ] \notin \overrightarrow{\mid \mathtt{s}_{sink} \mid} \qquad \overrightarrow{\mathtt{s}_{source}}, \overrightarrow{\mathtt{s}_{sink}} \vdash_{tc} \mathtt{B}\ \textbf{SUPER-OK}\ \mathtt{t}_s}{\overrightarrow{\mathtt{s}_{source}}, \overrightarrow{\mathtt{s}_{sink}} \vdash_{tc} \mathtt{A}\ \textbf{SUPER-OK}\ \mathtt{t}_s}$$

$$\frac{\begin{array}{c}\overrightarrow{\mathtt{s}_{source}} \vdash_m \overrightarrow{\langle\langle \mathtt{A}[\circ] \rangle\rangle = \overrightarrow{\mathtt{n}_{source}}} \\ \overrightarrow{\mathtt{s}_{sink}} \vdash_m \overrightarrow{\langle\langle \mathtt{A}[\circ] \rangle\rangle = \overrightarrow{\mathtt{n}_{sink}}} \\ \overrightarrow{\mid \mathtt{s}_{source} \mid} \supseteq \overrightarrow{\mid \mathtt{s}_{sink} \mid} \quad \overrightarrow{\overrightarrow{\mathtt{n}_{source}} \supseteq \overrightarrow{\mathtt{n}_{sink}}} \\ \overrightarrow{\mathtt{s}_{source}}, \overrightarrow{\mathtt{s}_{sink}} \vdash_{tc} \overrightarrow{\mathtt{A}\ \textbf{SUPER-OK}\ \mathtt{t}_s}\end{array}}{\vdash_{tc} \overrightarrow{\mathtt{s}_{source}}\ \textbf{CONGRUENT}\ \overrightarrow{\mathtt{s}_{sink} = \mathtt{A}[\circ] \lhd \mathtt{t}_s \{ \overrightarrow{\mathtt{n}} \}}}$$

The method operator $\overrightarrow{\mathtt{s}} \vdash_m \langle\langle \mathtt{A}[\mathtt{U}] \rangle\rangle = \overrightarrow{\mathtt{n}}$ extracts the sequence of all methods in the class $\mathtt{A}[\mathtt{U}]$ and its superclasses, given the typing environment $\overrightarrow{\mathtt{s}}$. The rule CONGRUENT can enforce the congruent subclassing relationship between a source typing environment and a sink typing environment using a recursively defined rule SUPER-OK. Intuitively, SUPER-OK ensures that direct subclassing relationships in the sink environment are satisfied with (possibly indirect) subclassing relationships in the source environment without first proceeding through any classes also present in the sink environment (otherwise, SUPER-OK is stuck). The rule CONGRUENT permits method hiding and some hiding of subclass relationships, and still enables the detection of method collisions and inheritance cycles.

4.1 Proof Sketch

The SUPER-OK rule ensures that all subclassing relationships visible in a sink typing environment are visible in the source typing environment, and that the source typing environment does not introduce any subclassing relationships that conflict with the sink typing environment. We can express this as Lemma 3:

Lemma 3 (SUPER-OK-EQUIVALENCE) *With respect to classes visible in a sink typing environment, the source typing environment and sink typing environment agree on the subclassing relationships between these classes.*

$$\begin{array}{c}\overrightarrow{\mathtt{s}_{sink}} = \overrightarrow{\mathtt{C}[\circ] \lhd \mathtt{t}_s \ldots} \quad \overrightarrow{\mid \mathtt{s}_{source} \mid} \supseteq \overrightarrow{\mid \mathtt{s}_{sink} \mid} \quad \mathtt{s}_{source}, \mathtt{s}_{sink} \vdash_t \overrightarrow{\mathtt{C}\ \textbf{SUPER-OK}\ \mathtt{t}_s} \\ \Rightarrow \quad \forall \mathtt{A}[\circ], \mathtt{B}[\circ] \in \overrightarrow{\mid \mathtt{s}_{sink} \mid} \quad . \quad \overrightarrow{\mathtt{s}_{sink}} \vdash_t \mathtt{A}[\circ] < \mathtt{B}[\circ] \leftrightarrow \overrightarrow{\mathtt{s}_{source}} \vdash_t \mathtt{A}[\circ] < \mathtt{B}[\circ]\end{array}$$

The proof of Lemma 2 depends primarily on Lemma 4:

Lemma 4 (LOCAL-EQUIVALENCE) *With respect to classes visible in a compilation typing environment, the compilation typing environment and linkage typing environment agree on the subclassing relationships between these classes.*

$$\begin{array}{c}\vdash_{tc} \textbf{MODULAR-OK}\ \overrightarrow{\mathtt{u} = \mathtt{V} \{ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e} \} \{ \overrightarrow{\mathtt{c}_x} \}} \quad \overrightarrow{\| \mathtt{u} \|} \vdash_l \overrightarrow{\mathtt{u} \ \rightarrow \ \overrightarrow{\mathtt{c}_y}} \quad \Rightarrow \\ \forall \mathtt{u}_u \in \overrightarrow{\mathtt{u}} \quad \mathtt{u}_u = \mathtt{U} \{ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}} \} \{ \overrightarrow{\mathtt{c}_{xu}} \}, \quad \forall \mathtt{A}[\circ], \mathtt{B}[\circ] \in \overrightarrow{\mid \mathtt{s}_{iu} \mid} \cup \overrightarrow{\mid \| \mathtt{c}_{xu} \| \mid} \\ \overrightarrow{\| \mathtt{u} \|}, \mathtt{u}_u \vdash_l \mathtt{A} \mapsto \mathtt{V}, \mathtt{B} \mapsto \mathtt{W} \quad . \quad \overrightarrow{\mathtt{s}_{iu}} \cup \overrightarrow{\| \mathtt{c}_{xu} \|} \vdash_t \mathtt{A}[\circ] < \mathtt{B}[\circ] \leftrightarrow \overrightarrow{\overrightarrow{\| \mathtt{c}_y \|}} \vdash_t \mathtt{A}[\mathtt{V}] < \mathtt{B}[\mathtt{W}]\end{array}$$

A proof for Lemma 4 that primarily depends on Lemma 3 is shown in Appendix B. We can prove Lemma 2 by showing that inheritance cycles and method collisions are impossible. The absence of inheritance cycles falls directly out of Lemma 4, as the compilation typing environment of each unit does not have cycles, so the resulting link-reduced classes cannot express any cycles either. Method collisions can only occur between methods introduced in the same unit, because link reduction specifies different linking offsets to methods that originate from different units. If a method collision does occur, it must be created within one unit, where it must have been detectable according to the subclassing relationships ensured by Lemma 4.

5 Related Work

Separate compilation and modular linking are explored by Cardelli [4] in his work with linksets. Program units [8, 11] extend work in this area by considering higher-order core languages, circular dependencies, and dynamically typed classes. MiniJiazzi shows how program units can be added to statically-typed object-oriented languages.

Drossopoulou et al. [7] point out problems in Java's linking model, especially the fact that is not very safe, and propose a safer model for linking in Java. Ancona et al. [1] use a notion of a compilation schema to explore separate source code compilation and runtime linking in Java as it is currently. Our work proposes a new compilation and linking model for OO languages that addresses the deficiencies in Java's existing compilation and linking models.

Ancona and Zucca [2] explore adding a true module to the Java language with a language called JavaMod. Fisher and Reppy [9, 10] explore modular linking for a class-based core language using a ML-like module system called Moby. Unlike MiniJiazzi, neither JavaMod nor Moby support true mixin-style inheritance. In JavaMod, methods hidden in a superclass are not visible in a subclass, while in Moby, methods provided by a superclass can only be invoked by explicitly specifying the superclass. Moby does not support modules with cyclic dependencies. As far as module systems go, MiniJiazzi is the first formalization of a statically-typed module system that supports both cyclic dependencies and full mixin-style inheritance for an OO language.

Enforcing the privacy of methods in OO languages has been explored extensively in the literature. Riecke and Stone [19] formally explore method privacy in structurally typed OO languages by using method dictionaries, while this work is extended by Stone [20] and Vouillon [22] in the context of class and mixin-based languages. MiniJiazzi differs by using linking offsets rather than dictionaries to enforce method scopes and disambiguate between method namespaces.

6 Conclusion

We have shown how modular linking in an OO language can be supported while allowing several expressive features: cyclic dependencies between fragments, inheritance across fragment boundaries, and information hiding between fragments. MiniJiazzi forms the theoretical basis for our implementation of Jiazzi, a component system for the Java language. Jiazzi is more sophisticated than MiniJiazzi, but their type-checking rules are similar. An implementation of Jiazzi is available for download:
`http://www.cs.utah.edu/plt/jiazzi`

Not all features of OO languages facilitate modular linking. We did not have room to describe how abstract methods makes modular linking difficult: they limit information hiding between fragments. If a class has an abstract method, then that abstract method can never be hidden when the class is visible in another fragment. While abstract methods can be used in Jiazzi units, they are not as modular as normal virtual methods. Future work will further explore interactions of modular linking with such OO language features.

References

[1] D. Ancona, G. Lagorio, and E. Zucca. A formal framework for Java separate compilation. In *In Proc. of ECOOP*, pages 609–636, June 2002.

[2] D. Ancona and E. Zucca. True modules for Java classes. In *In Proc. of ECOOP*, pages 354–380, June 2001.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of OOPSLA*, pages 303–311, Oct. 1990.

[4] L. Cardelli. Program fragments, linking and modularization. In *Proc. of POPL*, pages 266–277, Jan. 1997.

[5] C. Clifton, G. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. of OOPSLA*, pages 130–146, Oct. 2000.

[6] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. of PLDI*, pages 50–63, May 1999.

[7] S. Drossopoulou, D. Wragg, and S. Eisenbach. What is Java binary compatibility? In *Proc. of OOPLSA*, pages 341–361, Oct. 1998.

[8] R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. of ICFP*, pages 98–104, Sept. 1998.

[9] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *Proc. of PLDI*, pages 37–49, May 1999.

[10] K. Fisher and J. Reppy. Extending Moby with inheritance-based subtyping. In *Proc. of ECOOP*, pages 83–107, June 2000.

[11] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. of PLDI*, pages 236–248, May 1998.

[12] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. of POPL*, pages 171–183, Jan. 1999.

[13] I. Forman, M. Conner, S. Danforth, and I. Raper. Release-to-release binary compatibility in SOM. In *Proc. of OOPLSA*, Oct. 1995.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[15] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. of OOPSLA*, pages 132–146, Oct. 1999.

[16] X. Leroy, D. Doligez, J. Garrigue, D. R'emy, and J. Vouillon. The Objective CAML system, documentation and user's manual, 2000. http://caml.inria.fr/ocaml/htmlman/.

[17] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In *Proc. of OOPSLA*, Oct. 1998.

[18] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of OOPSLA*, pages 211–222, Oct. 2001.

[19] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999.

[20] C. Stone. Extensible objects without labels. In *Proc. of FOOL*, Jan. 2002.

[21] C. Szyperski. Import is not inheritance: Why we need both: modules and classes. In *Proc. of ECOOP*, pages 19–32, 1992.

[22] J. Vouillon. Combining subsumption and binary methods: An object calculus with views. In *Proc. of POPL*, pages 290–303, Jan. 2001.

A MiniJiazzi Core Language

MiniJiazzi core language syntax

class-identifer — $\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}$ method-identifer — $\mathtt{M}, \mathtt{N}$ fragment-identifer — $\mathtt{U}, \mathtt{V}, \mathtt{W}, \circ$

$$\text{class-signature } \mathtt{s} ::= \mathtt{A[U]} \lhd \mathtt{t}_s \{ \overrightarrow{\mathtt{n}_{fresh}} \} \qquad \text{method-signature } \mathtt{n} ::= \mathtt{t}_{return}\, \mathtt{M[U]}(\overrightarrow{\mathtt{t}_{arg}\ \mathtt{x}})$$

$$\text{class-expression } \mathtt{c} ::= \mathtt{s}_{sig} \{ \overrightarrow{\mathtt{m}_{impl}} \} \qquad \text{method-expression } \mathtt{m} ::= \mathtt{n}_{sig} \{ \mathtt{e}_{\mathtt{return}} \}$$

$$\text{type } \mathtt{t} ::= \mathtt{Object} \mid \mathtt{A[U]}$$

$$\text{method-subexpression } \mathtt{e} ::= \mathtt{new}\ \mathtt{A[U]} \mid (\mathtt{t})\ \mathtt{e} \mid \mathtt{x} \mid \mathtt{this} \mid \mathtt{e}.\mathtt{M[A[U], V]}(\overrightarrow{\mathtt{e}_\mathtt{x}})$$

The syntax of the MiniJiazzi core language is a small Java-like language subset. To emphasize inheritance and virtual methods, the MiniJiazzi core language class expressions only express subclassing relationships and contain methods. The symbol $\lhd$ is used to express a direct subclassing relationship. As mentioned in Section 3.4, link offsets are added to the syntax of the core language even though programmers do not use them directly.

MiniJiazzi core language extraction, subtyping, and method relationships

$$\| \mathtt{c} = \mathtt{s} \{ \overrightarrow{\mathtt{m}_{impl}} \} \| = \mathtt{s} \qquad \| \mathtt{m} = \mathtt{n} \{ \mathtt{e} \} \| = \mathtt{n} \qquad | \mathtt{A[U]} \lhd \mathtt{t}_s \{ \overrightarrow{\mathtt{n}} \} | = \mathtt{A[U]} \qquad | \mathtt{t}_r\ \mathtt{M[U]}(\overrightarrow{\mathtt{t}_x\ \mathtt{x}}) | = \mathtt{M[U]}$$

$$\frac{\mathtt{A[U]} \lhd \mathtt{t}_s \{ \ldots \} \in \overrightarrow{\mathtt{s}}}{\overrightarrow{\mathtt{s}} \vdash_t \mathtt{A[U]}, \mathtt{A[U]} \lhd \mathtt{t}_s, \mathtt{A[U]} < \mathtt{t}_s} \qquad \frac{\overrightarrow{\mathtt{s}} \vdash_t \mathtt{t}_a < \mathtt{t}_b}{\overrightarrow{\mathtt{s}} \vdash_t \mathtt{t}_a \leq \mathtt{t}_b} \qquad \begin{array}{c}\overrightarrow{\mathtt{s}} \vdash_t \mathtt{t}_a \leq \mathtt{t}_a \\ \overrightarrow{\mathtt{s}} \vdash_t \mathtt{Object}\end{array} \qquad \frac{\overrightarrow{\mathtt{s}} \vdash_t \mathtt{t}_a \leq \mathtt{t}_b < \mathtt{t}_c}{\overrightarrow{\mathtt{s}} \vdash_t \mathtt{t}_a < \mathtt{t}_c}$$

$$\overrightarrow{\mathtt{s}} \vdash_m \langle\langle \mathtt{Object} \rangle\rangle = \emptyset$$

$$\frac{\mathtt{A[U]} \lhd \mathtt{t}_s \{ \overrightarrow{\mathtt{n}_x} \} \in \overrightarrow{\mathtt{s}} \quad \overrightarrow{\mathtt{s}} \vdash_m \langle\langle \mathtt{t}_s \rangle\rangle = \overrightarrow{\mathtt{n}_y}}{\overrightarrow{\mathtt{s}} \vdash_m \overrightarrow{\mathtt{n}_x} \cup \langle\langle \mathtt{A[U]} \rangle\rangle = \overrightarrow{\mathtt{n}_y}} \qquad \frac{\begin{array}{c}\mathtt{A[U]} \lhd \mathtt{t}_s \{ \overrightarrow{\mathtt{n}} \} \in \overrightarrow{\mathtt{s}} \\ \overrightarrow{\mathtt{s}} \vdash_m \mathtt{M[U]} \in \overrightarrow{|\,\mathtt{n}\,|}\end{array}}{\overrightarrow{\mathtt{s}} \vdash_m \langle \mathtt{A[U]}.\mathtt{M[U]} \rangle = \mathtt{A[U]}} \qquad \frac{\begin{array}{c}\mathtt{A[U]} \lhd \mathtt{t}_s \{ \overrightarrow{\mathtt{n}} \} \in \overrightarrow{\mathtt{s}} \\ \mathtt{M[W]} \notin \overrightarrow{|\,\mathtt{n}\,|} \quad \overrightarrow{\mathtt{s}} \vdash_t \mathtt{A[U]} \lhd \mathtt{B[V]} \\ \overrightarrow{\mathtt{s}} \vdash_m \langle \mathtt{B[V]}.\mathtt{M[W]} \rangle = \mathtt{C[W]}\end{array}}{\overrightarrow{\mathtt{s}} \vdash_m \langle \mathtt{A[U]}.\mathtt{M[W]} \rangle = \mathtt{C[W]}}$$

MiniJiazzi core language type checking

$$\frac{\begin{array}{c}\overrightarrow{\mathtt{s}} \vdash_{tc} \mathtt{A[U]}\ \overrightarrow{\textsc{Method-Ok}\ \mathtt{m}} \\ \overrightarrow{\mathtt{n}} \subseteq \overrightarrow{\|\,\mathtt{m}\,\|} \quad \vdash \textsc{Unique}\ \overrightarrow{|\,\|\,\mathtt{m}\,\|\,|}\end{array}}{\overrightarrow{\mathtt{s}} \vdash_{tc} \textsc{Expression-Ok}} \qquad \frac{\begin{array}{c}\vdash \textsc{Unique}\ \overrightarrow{\mathtt{A[U]}} \quad \overrightarrow{\mathtt{s}} \vdash_t \overrightarrow{\overrightarrow{\mathtt{t}_x}} \quad \overrightarrow{\mathtt{s}} \vdash_t \overrightarrow{\mathtt{t}_r} \\ \overrightarrow{\vdash \textsc{Unique}\ \overrightarrow{\mathtt{M[U]}}} \quad \overrightarrow{\mathtt{s}} \vdash_t \overrightarrow{\mathtt{t}_s \leq \mathtt{Object}} \quad \overrightarrow{\overrightarrow{\mathtt{s}} \vdash_m \overrightarrow{\mathtt{n}} \cap \langle\langle \mathtt{t}_s \rangle\rangle = \emptyset}\end{array}}{\vdash_{tc} \textsc{Environment-Ok}}$$

$$\mathtt{A[U]} \lhd \mathtt{t}_s \{ \overrightarrow{\mathtt{n}} \} \{ \overrightarrow{\mathtt{m}} \} \qquad \overrightarrow{\mathtt{s} = \mathtt{A[U]} \lhd \mathtt{t}_s \{ \overrightarrow{\mathtt{n} = \mathtt{t}_r\ \mathtt{M[U]}(\overrightarrow{\mathtt{t}_x})} \}}$$

MiniJiazzi method-level type checking

$$\frac{\overrightarrow{s} \vdash_t \overrightarrow{t_x} \quad \overrightarrow{s} \vdash_t t_r \quad \vdash \textsc{Unique}\ \overrightarrow{x} \quad \overrightarrow{\Gamma(x) = t_x} \quad \Gamma(\texttt{this}) = A[U] \quad \Gamma, \overrightarrow{s} \vdash_{tc} e \in t_y \quad \overrightarrow{s} \vdash_t t_y \leq t_r \quad \overrightarrow{s} \vdash_m \in t_r\ M[W](\overrightarrow{t_x}) \in \langle\langle A[U] \rangle\rangle}{\overrightarrow{s} \vdash_{tc} A[U]\ \textbf{Method-Ok}\ t_r\ M[W](\overrightarrow{t_x\ x})\ \{\ e\ \}}$$

$$\Gamma, \overrightarrow{s} \vdash_{tc} x \in \Gamma(x) \qquad \Gamma, \overrightarrow{s} \vdash_{tc} \texttt{this} \in \Gamma(\texttt{this})$$

$$\frac{\overrightarrow{s} \vdash_t A[U]}{\Gamma, \overrightarrow{s} \vdash_{tc} \texttt{new}\ A[U] \in A[U]}$$

$$\frac{\Gamma, \overrightarrow{s} \vdash_{tc} e \in t_b \quad \overrightarrow{s} \vdash_t (t_a \leq t_b \text{ or } t_b < t_a)}{\Gamma, \overrightarrow{s} \vdash_{tc} (t_a)\ e \in t_a} \qquad \frac{\Gamma, \overrightarrow{s} \vdash_{tc} e \in t_b \quad \overrightarrow{s} \vdash_t t_a \not\leq t_b, t_b \not< t_a}{\Gamma, \overrightarrow{s} \vdash_{tc} (t_a)\ e \in \mathit{error}}$$

$$\frac{\Gamma, \overrightarrow{s} \vdash_{tc} e \in t, \overrightarrow{e_x \in t_y} \quad \overrightarrow{s} \vdash_t t \leq A[V], \overrightarrow{t_y \leq t_x} \quad \overrightarrow{s} \vdash_m t_r\ M[W](\overrightarrow{t_x}) \in \langle\langle A[V] \rangle\rangle}{\Gamma, \overrightarrow{s} \vdash_{tc} e.M[A[V], W](\overrightarrow{e_x}) \in t_r}$$

Method invocations specify a class offset in addition to a linking offset. This class offset is used only as a convenience to the linker and is not used during evaluation. Like link offsets, class offsets are not specified directly by the programmer. Rather they can be provided by the type checker during compile-time type checking.

MiniJiazzi core language evaluation

$$\frac{\overrightarrow{c} \vdash_c e \ \rightarrow\ \texttt{new}\ A[V] \quad \overrightarrow{\| c \|} \vdash_t A[V] \leq t}{\overrightarrow{c} \vdash_c (t)\ e \ \rightarrow\ \texttt{new}\ A[V]} \qquad \frac{\overrightarrow{c} \vdash_c e \ \rightarrow\ \texttt{new}\ A[V] \quad \overrightarrow{\| c \|} \vdash_t A[V] \not\leq t}{\overrightarrow{c} \vdash_c (t)\ e \ \rightarrow\ \mathit{downcast\_error}}$$

$$\frac{\overrightarrow{c} \vdash_c e \ \rightarrow\ \texttt{new}\ A[W], \quad \overrightarrow{e_x \ \rightarrow\ \texttt{new}\ t_x} \quad \overrightarrow{c} \vdash_c \lfloor A[W].M[V] \rfloor = (\overrightarrow{x})\ \{\ e\ \}}{\overrightarrow{c} \vdash_c e.M[\ldots, V](\overrightarrow{e_x}) \ \rightarrow\ [\texttt{new}\ t_{this}/\texttt{this}, \overrightarrow{\texttt{new}\ t_x/x}]\ e_y}$$

$$\frac{A[W] \lhd t_s \{\ \ldots \overrightarrow{m}\ \} \in \overrightarrow{c} \quad M[V](\overrightarrow{x})\ \{\ e\ \} \in \overrightarrow{m}}{\overrightarrow{c} \vdash_c \lfloor A[W].M[V] \rfloor = (\overrightarrow{x})\ \{\ e\ \}} \qquad \frac{A[W] \lhd t_s \{\ \ldots \overrightarrow{m}\ \} \in \overrightarrow{c} \quad M[V](\ldots)\ \{\ \ldots\ \} \notin \overrightarrow{m} \quad \overrightarrow{c} \vdash_c \lfloor t_s.M[V] \rfloor = (\overrightarrow{x})\ \{\ e\ \}}{\overrightarrow{c} \vdash_c \lfloor A[W].M[V] \rfloor = (\overrightarrow{x})\ \{\ e\ \}}$$

The method lookup operator $\lfloor A[U].M[V] \rfloor = \overrightarrow{x\ \{\ e\ \}}$ implements basic evaluation virtual method lookup in an OO language, finding the most overriding implementation of method $M[V]$ from a given evaluation type $A[U]$. Soundness can be stated with the addition of the rule Global-Ok defined in Section 3.1:

$$\begin{gathered} \texttt{Main}[\circ] \lhd \texttt{Object}\ \{\ \texttt{Object}\ \texttt{main}()\ \}\{\ \texttt{Object}\ \texttt{main}[\texttt{Main}[\circ], \circ]()\ \{\ e_m\ \}\ \} \in \overrightarrow{c} \\ \vdash_{tc} \textsc{Global-Ok}\ \overrightarrow{c} \quad \Rightarrow \quad \overrightarrow{c} \vdash_e [\texttt{new}\ \texttt{Main}[\circ]/\texttt{this}]e_m \twoheadrightarrow e_n \\ \exists A, V (e_n = \texttt{new}\ A[V] \quad \text{or} \quad e_n = \mathit{downcast\_error} \quad \text{or} \quad e_n \text{ is an infinite loop}) \end{gathered}$$

While core language type checking rules utilize link offsets, these offsets are empty when the rule Modular-Ok is used to perform type checking. We specify type checking rules that examine linking offsets so Global-Ok can be used to compare the type correctness of the resulting link-reduced classes. Link reduction in MiniJiazzi binds every class reference and method reference expressed in a class expressions to an appropriate linking offset according to the definition of $\overrightarrow{s}, u \vdash_l \mapsto$ given in Section 3.4. As such, link reduction is implemented in the following judgments as a traversal through the implementation of a unit expression:

MiniJiazzi link reduction

$$\frac{\vec{\mathtt{s}},\mathtt{u} \vdash_l \overrightarrow{\mathtt{s}_a \rightarrow \mathtt{s}_b} \quad \vec{\mathtt{s}},\mathtt{u} \vdash_l \overrightarrow{|\,\mathtt{s}_a\,|\,\overline{\mathtt{m}_a \rightarrow \mathtt{m}_b},}}{\vec{\mathtt{v}} \vdash_l \mathtt{u} = \ldots_{\{\,\overrightarrow{\mathtt{s}_a\,\{\,\overrightarrow{\mathtt{m}_a}\,\}}\,\}} \rightarrow \mathtt{s}_{b\,\{\,\overrightarrow{\mathtt{m}_b}\,\}}} \text{ (u-link)}
\qquad
\frac{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{A[\circ]}\,\overrightarrow{\mathtt{n}_a \rightarrow \mathtt{n}_b} \quad \vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{t}_a \rightarrow \mathtt{t}_b \quad \vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{A} \mapsto \mathtt{W}}{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{A[\circ]} \lhd \mathtt{t}_{a\{\,\overrightarrow{\mathtt{n}_a}\,\}} \rightarrow \mathtt{A[W]} \lhd \mathtt{t}_{b\{\,\overrightarrow{\mathtt{n}_b}\,\}}} \text{ (s-link)}$$

$$\frac{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{A[\circ]}\;\mathtt{t}_r\;\mathtt{M[\circ]}(\mathtt{t}_x) \rightarrow \mathtt{t}_v\;\mathtt{M[W]}(\mathtt{t}_y) \quad \vec{\mathtt{s}},\mathtt{u} \vdash_l \mathsf{e_a} \rightarrow \mathsf{e_b}}{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{A[\circ]}\;\mathtt{t}_r\;\mathtt{M[\circ]}(\overrightarrow{\mathtt{t}_x\;\mathtt{x}})_{\{\,\mathsf{e_a}\,\}} \rightarrow \mathtt{t}_v\;\mathtt{M[W]}(\overrightarrow{\mathtt{t}_y\;\mathtt{x}})_{\{\,\mathsf{e_b}\,\}}} \text{ (m-link)}
\qquad
\frac{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{t}_r \rightarrow \mathtt{t}_v \quad \vec{\mathtt{s}},\mathtt{u} \vdash_l \overrightarrow{\mathtt{t}_x \rightarrow \mathtt{t}_y} \quad \vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{A.M} \mapsto \mathtt{W}}{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{A[\circ]}\;\mathtt{t}_r\;\mathtt{M[\circ]}(\overrightarrow{\mathtt{t}_x}) \rightarrow \mathtt{t}_v\;\mathtt{M[W]}(\overrightarrow{\mathtt{t}_y})} \text{ (n-link)}$$

$$\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{Object} \rightarrow \mathtt{Object} \text{ (t-link)}
\qquad
\frac{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{A} \mapsto \mathtt{W}}{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{A[\circ]} \rightarrow \mathtt{A[W]}} \text{ (t-link)}$$

$$\frac{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{t}_a \rightarrow \mathtt{t}_b}{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{new}\;\mathtt{t}_a \rightarrow \mathtt{new}\;\mathtt{t}_b} \text{ (e-link)}
\qquad
\frac{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{t}_a \rightarrow \mathtt{t}_b \quad \vec{\mathtt{s}},\mathtt{u} \vdash_l \mathsf{e_a} \rightarrow \mathsf{e_b}}{\vec{\mathtt{s}},\mathtt{u} \vdash_l (\mathtt{t}_a)\;\mathsf{e_a} \rightarrow (\mathtt{t}_b)\;\mathsf{e_b}} \text{ (e-link)}
\qquad
\frac{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathsf{e_a} \rightarrow \mathsf{e_b} \quad \vec{\mathtt{s}},\mathtt{u} \vdash_l \overrightarrow{\mathsf{e_x} \rightarrow \mathsf{e_y}} \quad \vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{B.M} \mapsto \mathtt{W} \quad \vec{\mathtt{s}},\mathtt{u} \vdash_l \mathtt{B} \mapsto \mathtt{V}}{\vec{\mathtt{s}},\mathtt{u} \vdash_l \mathsf{e_a}.\mathtt{M[B[\circ],\circ]}(\overrightarrow{\mathsf{e_x}}) \rightarrow \mathsf{e_b}.\mathtt{M[B[V],W]}(\overrightarrow{\mathsf{e_y}})} \text{ (e-link)}$$

B Proof of Lemma 4

In MiniJiazzi, the sink and source typing environments used in a congruence check always overlap. The compilation typing environment and the intermediate typing environment of a unit are both formed using the unit's imported class signatures, and the intermediate typing environment of a unit and the program's linkage typing environment are both formed using the unit's exported class signatures. A consequence of source/sink typing environments that express equivalent subtyping relationships is that we can reason about the uncommon parts of the source and sink typing environments with Lemma 5:

Lemma 5 (PARTIAL-SUBTYPING) *For source and sink typing environments that have equivalent subtyping relationships with respect to sink classes, the uncommon parts of the source and sink typing environments, express equivalent subtyping relationships from classes the sink typing environment.*

$$\frac{\overrightarrow{|\,\mathtt{s}_{src}\,|} \supseteq \overrightarrow{|\,\mathtt{s}_{snk}\,|} \quad \vdash \textbf{UNIQUE}\,\overrightarrow{|\,\mathtt{s}_{cmn}\,|} \cup \overrightarrow{|\,\mathtt{s}_{snk}\,|} \quad \vdash \textbf{UNIQUE}\,\overrightarrow{|\,\mathtt{s}_{cmn}\,|} \cup \overrightarrow{|\,\mathtt{s}_{src}\,|} \\ \forall \mathtt{A[V]} \in \overrightarrow{|\,\mathtt{s}_{snk}\,|} \cup \overrightarrow{|\,\mathtt{s}_{cmn}\,|}, \forall \mathtt{B[W]} \in \overrightarrow{|\,\mathtt{s}_{snk}\,|} \cup \overrightarrow{|\,\mathtt{s}_{cmn}\,|}\,.\; \overrightarrow{\mathtt{s}_{snk}} \cup \overrightarrow{\mathtt{s}_{cmn}} \vdash_t \mathtt{A[V]} < \mathtt{B[W]} \leftrightarrow \overrightarrow{\mathtt{s}_{src}} \cup \overrightarrow{\mathtt{s}_{cmn}} \vdash_t \mathtt{A[V]} < \mathtt{B[W]}}{\forall \mathtt{A[V]} \in \overrightarrow{|\,\mathtt{s}_{snk}\,|}, \forall \mathtt{B[W]} \in \overrightarrow{|\,\mathtt{s}_{cmn}\,|} \cup \overrightarrow{|\,\mathtt{s}_{snk}\,|}\,.\; \overrightarrow{\mathtt{s}_{src}} \vdash_t \mathtt{A[V]} < \mathtt{B[W]} \leftrightarrow \overrightarrow{\mathtt{s}_{snk}} \vdash_t \mathtt{A[V]} < \mathtt{B[W]}}$$

We can take a partial typing environment equivalence relationship and combine them with other partial typing environment equivalent relationships as long as the class signatures involved do not overlap, or if the class signatures overlap completely. We can express this as Lemma 6 and Lemma 7:

Lemma 6 (COMBINE-PARTIAL-1) *Two partial typing environments subtyping equivalence relationships with dis-*

joint sink class signatures can be combined.

$$\frac{\begin{array}{c}\forall \mathtt{A[U]} \in \overrightarrow{\mathtt{A[U]}} \quad \forall \mathtt{B[V]} \in \overrightarrow{\mathtt{B[V]}}\ .\ \vec{\mathtt{s}}_a \vdash_t \mathtt{A[\circ]} < \mathtt{B[\circ]} \leftrightarrow \vec{\mathtt{s}}_b \vdash_t \mathtt{A[U]} < \mathtt{B[V]} \quad \vdash \textbf{UNIQUE}\overrightarrow{|\ \mathtt{s}_a\ |} \cup \overrightarrow{|\ \mathtt{s}_c\ |} \\ \forall \mathtt{C[W]} \in \overrightarrow{\mathtt{C[W]}} \quad \forall \mathtt{D[Z]} \in \overrightarrow{\mathtt{D[Z]}}\ .\ \vec{\mathtt{s}}_c \vdash_t \mathtt{C[\circ]} < \mathtt{D[\circ]} \leftrightarrow \vec{\mathtt{s}}_d \vdash_t \mathtt{C[W]} < \mathtt{D[Z]} \quad \vdash \textbf{UNIQUE}\overrightarrow{|\ \mathtt{s}_b\ |} \cup \overrightarrow{|\ \mathtt{s}_d\ |} \\ \overrightarrow{\mathtt{A[\circ]}} \subseteq \overrightarrow{|\ \mathtt{s}_a\ |} \quad \overrightarrow{\mathtt{A[U]}} \subseteq \overrightarrow{|\ \mathtt{s}_b\ |} \quad \overrightarrow{\mathtt{C[\circ]}} \subseteq \overrightarrow{|\ \mathtt{s}_c\ |} \quad \overrightarrow{\mathtt{C[W]}} \subseteq \overrightarrow{|\ \mathtt{s}_d\ |}\end{array}}{\forall \mathtt{A[U]} \in \overrightarrow{\mathtt{A[U]}} \cup \overrightarrow{\mathtt{C[W]}} \quad \forall \mathtt{B[V]} \in \overrightarrow{\mathtt{B[V]}} \cup \overrightarrow{\mathtt{D[Z]}}\ .\ \vec{\mathtt{s}}_a \cup \vec{\mathtt{s}}_c \vdash_t \mathtt{A[\circ]} < \mathtt{B[\circ]} \ \leftrightarrow\ \vec{\mathtt{s}}_b \cup \vec{\mathtt{s}}_d \vdash_t \mathtt{A[U]} < \mathtt{B[V]}}$$

Lemma 7 (COMBINE-PARTIAL-2) *A partial typing environment subtyping equivalence relationship can be combined with another partial typing environment subtyping equivalence relationship that contains all of its sink class signatures.*

$$\frac{\begin{array}{c}\forall \mathtt{A[U]} \in \overrightarrow{\mathtt{A[U]}} \quad \forall \mathtt{B[V]} \in \overrightarrow{\mathtt{B[V]}}\ .\ \vec{\mathtt{s}}_a \vdash_t \mathtt{A[\circ]} < \mathtt{B[\circ]} \leftrightarrow \vec{\mathtt{s}}_b \vdash_t \mathtt{A[U]} < \mathtt{B[V]} \quad \vdash \textbf{UNIQUE}\overrightarrow{|\ \mathtt{s}_a\ |} \cup \overrightarrow{|\ \mathtt{s}_c\ |} \\ \forall \mathtt{C[W]} \in \overrightarrow{\mathtt{C[W]}} \quad \forall \mathtt{D[Z]} \in \overrightarrow{\mathtt{D[Z]}}\ .\ \vec{\mathtt{s}}_a \cup \vec{\mathtt{s}}_c \vdash_t \mathtt{C[\circ]} < \mathtt{D[\circ]} \leftrightarrow \vec{\mathtt{s}}_b \cup \vec{\mathtt{s}}_d \vdash_t \mathtt{C[W]} < \mathtt{D[Z]} \quad \vdash \textbf{UNIQUE}\overrightarrow{|\ \mathtt{s}_b\ |} \cup \overrightarrow{|\ \mathtt{s}_d\ |} \\ \overrightarrow{\mathtt{A[\circ]}} \subseteq \overrightarrow{|\ \mathtt{s}_a\ |} \quad \overrightarrow{\mathtt{A[U]}} \subseteq \overrightarrow{|\ \mathtt{s}_b\ |} \quad \overrightarrow{\mathtt{C[\circ]}} \subseteq \overrightarrow{|\ \mathtt{s}_a\ |} \cup \overrightarrow{|\ \mathtt{s}_c\ |} \quad \overrightarrow{\mathtt{C[W]}} \subseteq \overrightarrow{|\ \mathtt{s}_b\ |} \cup \overrightarrow{|\ \mathtt{s}_d\ |}\end{array}}{\forall \mathtt{A[U]} \in \overrightarrow{\mathtt{A[U]}} \cup \overrightarrow{\mathtt{C[W]}} \quad \forall \mathtt{B[V]} \in \overrightarrow{\mathtt{B[V]}} \cup \overrightarrow{\mathtt{D[Z]}}\ .\ \vec{\mathtt{s}}_a \cup \vec{\mathtt{s}}_c \vdash_t \mathtt{A[\circ]} < \mathtt{B[\circ]} \ \leftrightarrow\ \vec{\mathtt{s}}_b \cup \vec{\mathtt{s}}_d \vdash_t \mathtt{A[U]} < \mathtt{B[V]}}$$

Proof of Lemma 4: Given a type correct linked program, we can use Lemma 3 and transitivity that a linked unit's compilation typing environment and the program's linkage typing have equivalent subclassing relationships with respect to all classes in unit intermediate typing environments:

$$\begin{array}{c}\forall \mathtt{A[\circ]} \in \overrightarrow{|\ \mathtt{s}_{iu}\ |} \cup \overrightarrow{|\ \mathtt{s}_{eu}\ |}, \forall \mathtt{B[\circ]} \in \overrightarrow{|\ \mathtt{s}_{iu}\ |} \cup \overrightarrow{|\ \mathtt{s}_{eu}\ |}\ .\ \mathtt{U}\ \{\ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}}\ \}\ \{\ \overrightarrow{\mathtt{c}_{xu}}\ \} \in \vec{\mathtt{u}} \\ \overrightarrow{\mathtt{s}_{iu}} \cup \overrightarrow{\mathtt{c}_{xu}} \vdash_t \mathtt{A[\circ]} < \mathtt{B[\circ]} \leftrightarrow \vec{\vec{\mathtt{s}}}_e \vdash_t \mathtt{A[\circ]} < \mathtt{B[\circ]}\end{array}$$

Use Lemma 5 to show that the subclassing relationships expressed by a unit's exported class signatures are also expressed in the unit's implementation class expressions:

$$\begin{array}{c}\forall \mathtt{A[\circ]} \in \overrightarrow{|\ \mathtt{s}_{eu}\ |}, \forall \mathtt{B[\circ]} \in \overrightarrow{|\ \mathtt{s}_{iu}\ |} \cup \overrightarrow{|\ \mathtt{s}_{eu}\ |}\ .\ \mathtt{U}\ \{\ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}}\ \}\ \{\ \overrightarrow{\mathtt{c}_{xu}}\ \} \in \vec{\mathtt{u}} \\ \overrightarrow{\|\ \mathtt{c}_{xu}\ \|} \vdash_t \mathtt{A[\circ]} < \mathtt{B[\circ]} \leftrightarrow \overrightarrow{\mathtt{s}_{eu}} \vdash_t \mathtt{A[\circ]} < \mathtt{B[\circ]}\end{array}$$

Relate to the linked reduced versions of a unit's implementation classes:

$$\begin{array}{c}\forall \mathtt{U} \in \vec{\vec{\mathtt{V}}}, \forall \mathtt{A[\circ]} \in \overrightarrow{|\ \mathtt{s}_{eu}\ |}, \forall \mathtt{B[\circ]} \in \overrightarrow{|\ \mathtt{s}_{iu}\ |} \cup \overrightarrow{|\ \mathtt{s}_{eu}\ |}\ .\ \mathtt{u} = \mathtt{U}\ \{\ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}}\ \}\ \{\ \overrightarrow{\mathtt{c}_{xu}}\ \} \in \vec{\mathtt{u}} \\ \vec{\vec{\mathtt{s}}}, \mathtt{u} \vdash_l \mathtt{B} \mapsto \mathtt{W} \quad \vec{\vec{\mathtt{s}}}, \mathtt{u} \vdash_l \overrightarrow{\mathtt{c}_{xu}} \ \to\ \overrightarrow{\mathtt{c}_{yu}} \quad \overrightarrow{\mathtt{c}_{yu}} \vdash_t \mathtt{A[U]} < \mathtt{B[W]} \leftrightarrow \overrightarrow{\mathtt{s}_{eu}} \vdash_t \mathtt{A[\circ]} < \mathtt{B[\circ]}\end{array}$$

Use Lemma 6 to recombine the partial typing environments equivalent subclassing relationships into complete typing environment equivalent subclassing relationships:

$$\forall \mathtt{A[\circ]} \in \overrightarrow{|\ \mathtt{s}_{eu}\ |}, \forall \mathtt{B[\circ]} \in \overrightarrow{|\ \mathtt{s}_{ew}\ |}\ . \quad \mathtt{U}\ \overrightarrow{\mathtt{s}_{eu}} \in \overrightarrow{\mathtt{V}\ \vec{\vec{\mathtt{s}}}_e} \quad \mathtt{W}\ \overrightarrow{\mathtt{s}_{ew}} \in \overrightarrow{\mathtt{V}\ \vec{\vec{\mathtt{s}}}_e} \quad \overrightarrow{\overrightarrow{\|\ \mathtt{c}_y\ \|}} \vdash_t \mathtt{A[U]} < \mathtt{B[W]} \leftrightarrow \vec{\vec{\mathtt{s}}}_e \vdash_t \mathtt{A[\circ]} < \mathtt{B[\circ]}$$

Consider each unit one at a time, and only its imported and exported classes, from the previous step we can derive the following:

$$\forall \mathtt{U} \in \overrightarrow{\mathtt{V}}, \forall \mathtt{A}[\circ] \in \overrightarrow{\lceil \mathtt{s}_{iu} \rceil} \cup \overrightarrow{\lceil \mathtt{s}_{eu} \rceil}, \forall \mathtt{B}[\circ] \in \overrightarrow{\lceil \mathtt{s}_{iu} \rceil} \cup \overrightarrow{\lceil \mathtt{s}_{eu} \rceil}. \quad \mathtt{u} = \mathtt{U}_{\{\ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}}\ \}\ \{\ \overrightarrow{\mathtt{c}_{xu}}\ \}} \in \overrightarrow{\mathtt{u}}$$

$$\overrightarrow{\mathtt{s}}, \mathtt{u} \vdash_l \mathtt{A} \mapsto \mathtt{V} \quad \overrightarrow{\mathtt{s}}, \mathtt{u} \vdash_l \mathtt{B} \mapsto \mathtt{W} \quad \overrightarrow{\mathtt{s}_{iu}} \cup \overrightarrow{\|\ \mathtt{c}_{xu}\ \|} \vdash_t \mathtt{A}[\circ] < \mathtt{B}[\circ] \leftrightarrow \overrightarrow{\overrightarrow{\|\ \mathtt{c}_y\ \|}} \vdash_t \mathtt{A}[\mathtt{V}] < \mathtt{B}[\mathtt{W}]$$

We can use transitivity to relate equivalent subclassing relationships between the link-reduced typing environment and compilation typing environments, but only for classes in the intermediate typing environment:

$$\forall \mathtt{U} \in \overrightarrow{\mathtt{V}}, \forall \mathtt{A}[\circ] \in \overrightarrow{\lceil \|\ \mathtt{c}_{xu}\ \| \rceil}, \forall \mathtt{B}[\circ] \in \overrightarrow{\lceil \mathtt{s}_{iu} \rceil} \cup \overrightarrow{\lceil \|\ \mathtt{c}_{xu}\ \| \rceil}. \quad \mathtt{u} = \mathtt{U}_{\{\ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}}\ \}\ \{\ \overrightarrow{\mathtt{c}_{xu}}\ \}} \in \overrightarrow{\mathtt{u}}$$

$$\overrightarrow{\mathtt{s}}, \mathtt{u} \vdash_l \mathtt{B} \mapsto \mathtt{W} \quad \overrightarrow{\mathtt{s}}, \mathtt{u} \vdash_l \overrightarrow{\mathtt{c}_{xu} \ \rightarrow\ \mathtt{c}_{yu}} \quad \overrightarrow{\|\ \mathtt{c}_{xu}\ \|} \vdash_t \mathtt{A}[\circ] < \mathtt{B}[\circ] \leftrightarrow \overrightarrow{\|\ \mathtt{c}_{yu}\ \|} \vdash_t \mathtt{A}[\mathtt{U}] < \mathtt{B}[\mathtt{W}]$$

And we can use Lemma 7 to combine this result with the previous result to complete the proof of Lemma 4:

$$\forall \mathtt{U} \in \overrightarrow{\mathtt{V}}, \forall \mathtt{A}[\circ] \in \overrightarrow{\lceil \mathtt{s}_{iu} \rceil} \cup \overrightarrow{\lceil \|\ \mathtt{c}_{xu}\ \| \rceil}, \forall \mathtt{B}[\circ] \in \overrightarrow{\lceil \mathtt{s}_{iu} \rceil} \cup \overrightarrow{\lceil \|\ \mathtt{c}_{xu}\ \| \rceil}. \quad \mathtt{u} = \mathtt{U}_{\{\ \overrightarrow{\mathtt{s}_{iu}}\ \overrightarrow{\mathtt{s}_{eu}}\ \}\ \{\ \overrightarrow{\mathtt{c}_{xu}}\ \}} \in \overrightarrow{\mathtt{u}}$$

$$\overrightarrow{\mathtt{s}}, \mathtt{u} \vdash_l \mathtt{B} \mapsto \mathtt{W} \quad \overrightarrow{\mathtt{s}_{iu}} \cup \overrightarrow{\|\ \mathtt{c}_{xu}\ \|} \vdash_t \mathtt{A}[\circ] < \mathtt{B}[\circ] \leftrightarrow \overrightarrow{\overrightarrow{\|\ \mathtt{c}_y\ \|}} \vdash_t \mathtt{A}[\mathtt{U}] < \mathtt{B}[\mathtt{W}]$$