A Framework for Modular Linking in OO Languages

Sean McDirmid[1], Wilson C. Hsieh[2], and Matthew Flatt[2]

[1] École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland
sean.mcdirmid@epfl.ch
[2] University of Utah, 84112 Salt Lake City, UT, USA
{wilson, mflatt}@cs.utah.edu

Abstract. The successful assembly of large programs out of software components depends on **modular reasoning**. When the linking of component code is modular, components can be compiled and type checked separately, deployed in binary form, and are easier to reuse. Unfortunately, linking is not modular in many mainstream OO languages such as Java. In this paper we propose an intuitive and formal framework for enhancing a language with modular linking, which is applied to the specific problem of making linking in Java modular. In our proposed framework, the degree to which components can be reasoned about modularly is adversely affected by language features that limit abstraction. We show that most of Java's core language features, such as inheritance, permit a high degree of modular linking even in the presence of cyclic dependencies.

1 Introduction

Reasoning is *modular* if it can be divided into separate reasoning of a system's parts, all of which can be combined into a reasoning of the entire system. General modular reasoning is indispensable in developing large programs out of third-party software components [24], because developers do not need to understand the implementations of the components they reuse. *Modular linking* is a specific kind of modular reasoning where component code can be compiled, linked, and statically type checked separately. Modular linking avoids many "DLL hell" problems related to the link-time binary compatibility [9,18] of components. Although modular linking is a common feature of functional languages because of the elegance and simplicity of functions, modular linking in object-oriented (OO) languages is problematic because of complex language features related to classes and objects.

The degree to which components can be separated by modular reasoning depends on what can be hidden, or "abstracted," between components. Abstraction in Java is complicated by **inheritance**: reasoning about a class defined in a component separate from its inherited superclasses is similar to reasoning about a *mixin* [4,14]. Mixins have well-known type-checking challenges related to **ambiguous methods**, where a subclass may "introduce" a method that conflicts with a method unknowingly provided by a superclass, and **cyclic inheritance**, where mixins are applied recursively. As a result, statically-typed OO languages that support modular linking have done so by restricting inheritance [3], disallowing cyclic dependencies between components [12,17], or by severely limiting abstraction [3,12].

D. Lightfoot and C. Szyperski (Eds.): JMLC 2006, LNCS 4228, pp. 116–135, 2006.

In this paper, we propose a framework that intuitively and formally describes how linking can be made modular independent of a specific language. We apply this framework to the problem of making compilation, linking, and type checking, modular in the Java language, which models our design and implementation of Jiazzi [19]. Jiazzi enhances Java with support for externally linked and separately compiled components based on program units [13].

Issues related to modular linking in OO languages have been previously explored in the areas of managing virtual method namespaces [22,25], merging module systems and OO languages [3,12,17], and reasoning about mixins [4,14]. The work presented in this paper is the first to show how modular linking can be added to a language using a general framework, which directly considers the effects of a language's features, such as OO inheritance. We show that inter-module inheritance restricts abstraction in a way that does not significantly decrease the degree of modular reasoning.

Section 2 motivates modular linking in the Java language. Section 3 describes an intuitive and formal framework for adding modular linking to a language that does not already support it. Section 4 shows how this framework is used to add modular linking to the Java language, and describes how we have dealt with inheritance and cyclic dependencies. Section 5 briefly discusses how our modular linking framework can be used to evaluate language features other than inheritance, such as abstract methods and symmetric multi-methods. Section 6 discusses related work and Section 7 summarizes our conclusions.

2 Motivation

Programs can be assembled out of separately developed and deployed containers of code, which we refer to as *components*. Such components in Java can be physically realized as Java archive (JAR) files that contain compiled classes. The linking of components is important in the development of a program. During linking, type checking is performed to ensure that safety properties are not violated that could result in segmentation faults or circumvent security of the program. When compared to other mainstream languages, Java's support for program linking is advanced: linking always guarantees program type safety, can occur dynamically, and supports laziness. Unfortunately, the linking of components in Java is not modular: the entire code of all components linked into a program is always type checked together. Because the linking of components in Java is not modular, interactions between components can result in errors that can only be debugged by inspecting the source code for all components involved.

In Java, the source code of each component is compiled in an environment where the implementations of all used classes are available, even if those classes are implemented in other components. These classes that originate from other components can differ between compile-time and link-time. For example, the component ***icon*** illustrated in Figure 1 uses the class `Cowboy`, which is provided by the component ***cowboy_cmpl*** when component ***icon*** is compiled but later is provided by the component ***cowboy_lnk*** when component ***icon*** is linked.

The fact that the class `Cowboy` is different between the compile-time and link-time of the component ***icon*** is significant because the class `Icon`, provided by the component

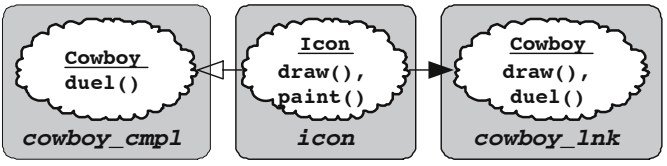

Fig. 1. The components ***icon***, ***cowboy_cmpl***, and ***cowboy_lnk***; throughout this paper, components are illustrated as gray-filled rounded rectangles; classes are clouds whose names are underlined over their methods; clear arrows point from a class to its direct superclass as its containing component is compiled; solid arrows point from a class to its direct superclass as its containing component is linked

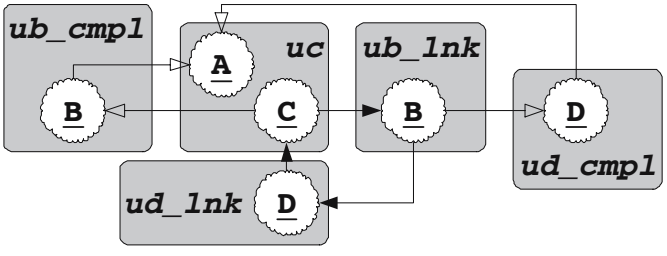

Fig. 2. The components ***uc***, ***ub_cmpl***, ***ub_lnk***, ***ud_cmpl***, and ***ud_lnk***

icon, is a subclass of `Cowboy`. The class `Icon` implements a *draw* method, as in "drawing an icon." When the component ***icon*** is compiled against the component ***cowboy_cmpl***, the class `Cowboy` does not have a *draw* method, but when the component ***icon*** is linked against the component ***cowboy_lnk***, a *draw* method, as in "drawing guns in a cowboy duel," exists in the class `Cowboy`. Assuming both *draw* methods are public and have the same signature, the *draw* method in the class `Icon` should not override the *draw* method in the class `Cowboy`: the *draw* method was not visible when the component ***icon*** was compiled so overriding was not the programmer's intention. However, unintended overriding occurs in Java because unmodular linking disregards compile-time intentions.

Changes in the inheritance hierarchies that occur between compilation and linking can also "break" programs in Java. In Figure 2, the component ***uc*** is compiled against the component ***ub_cmpl***, but it is linked against component ***ub_lnk***, and the component ***ub_lnk*** is compiled against the component ***ud_cmpl***, but linked against the component ***ud_lnk***. When component ***uc*** is compiled against the component ***ub_cmpl***, class `B` appears as a direct subclass of the class `A`, so it is safe for the class `C`, implemented in component ***uc***, to subclass the class `B`. However when the component ***uc*** is linked against ***ub_lnk***, the class `B` is an indirect subclass of the class `C`, so an inheritance cycle is created. Although linking in Java rejects this program, "blame" to one component cannot be assigned for this error. Instead to understand why this error occurs, and thus be able to fix it, the changes that occur from the component ***ub_cmpl*** to ***ub_lnk*** and from the component ***ud_cmpl*** to ***ud_lnk*** must be understood together.

The problems in linking the components illustrated in Figures 1 and 2 are created by inheritance between classes across component boundaries. In Figure 1, the class `Icon` inherits from the class `Cowboy`; in Figure 2, the class `C` inherits from the class `B`. The superclasses, the classes `Cowboy` and `B`, are implemented in components that change between compile-time and link-time. Such changes can lead to the typing problems of *mixins* [4,14], which are classes with explicitly parameterized superclasses. Compared to mixins, the superclasses of classes `Icon` and `C` are implicitly parameterized through linking.

In Java, differences between a component's compile-time and link-time environment are governed by special *binary compatibility* [9,18] guidelines, which specify what changes to components can occur after compile-time that will still allow linking to succeed. Additive changes between link-time and compile-time, such as adding new methods or new superclasses to a class, are generally considered safe according to binary compatibility. In Figure 1 a method is added to the class `Cowboy`, while in Figure 2, the class `D` is added as superclasses of the class `B`.

Even though the changes in Figures 1 and 2 are additive, linking in these examples still breaks. In Figure 1, linking is technically type safe; the components are linked in Java without errors even though programmer intent is not be adhered to. In Figure 2, the components have *cyclic dependencies*: they each use each other's classes and their dependency graph contains a cycle. Binary compatibility primarily accommodates changes to libraries, such as AWT, which cannot have cyclic dependencies with programs. Adding a new superclass to a class is always safe if the containing component does not have any cyclic dependencies with other components in the system, which is not the case in Figure 2.

Perhaps inheritance of classes should be restricted across component boundaries or perhaps cyclic dependencies between components should be disallowed. However, inheritance is an essential mechanism in using the OO paradigm to develop entire programs, not just individual components. The use of cyclic dependencies is the most natural way to codify two-way interactions that commonly occur between classes in different components. Restricting either inheritance or cyclic dependencies disallows the language-supported use of OO design throughout a program. Mixin-style inheritance of classes across component boundaries and cyclic dependencies between components also enables *open classes* [6], which are classes that can be extended with new fields and methods without breaking their existing subtypes.

Modular linking of components can be used to implement open classes with what we call an *open class pattern* [19]. The components illustrated in Figure 3 demonstrate the mechanics of the open class pattern. The class `BButton` is a subclass of the class `BWidget` in the component ***base***, but rather than directly subclass the class `BWidget`, the class `BButton` is a direct subclass of the class `FWidget` from the component ***fixed***. This creates a cyclic dependency between the components ***base*** and ***fixed***, because the class `FWidget` is also an indirect subclass of the class `BWidget`. The class in the inheritance hierarchy between classes `FWidget` and `BWidget` depends on whether the components ***base*** and ***fixed*** are linked with the component ***color_1*** or the components ***color_2*** and ***font***. With the former linking, one new method *`setColor`* is visible in the class `FWidget`, while in the latter linking two new methods *`setColor`*

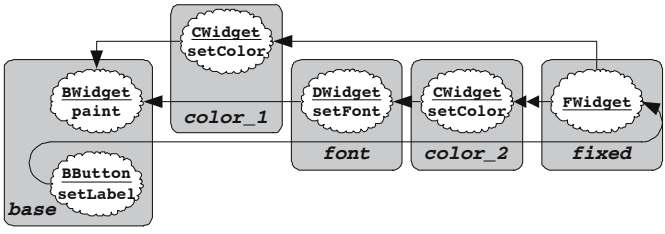

Fig. 3. The components ***base***, ***color_1***, ***color_2***, ***font***, and ***fixed***; double-head arrows point from classes to direct superclasses in an alternative way to link these components together

and `setFont` are visible in the class `FWidget`. Regardless of which linking occurs, the class `BButton` inherits any new methods added because it subclasses the class `FWidget` rather than the class `BWidget`.

In Java, the components illustrated in Figure 3 can be linked together into a valid program. However, the linking is fragile because type checking is not performed in a modular way. Jiazzi, our enhancement to Java that supports modular linking of components, supports the open class pattern with additional renaming mechanisms, not discussed in this paper, so that components can be "mixed-and-matched" to form classes with a desired feature set [19].

3 Modular Linking

When modular reasoning is applied to the linking of component code, type checking is "solved" in two phases: first when the component code is initially compiled, and later when the component code is linked with the code of other components to create a program. Type checking performed during compilation is not duplicated during linking. The benefit of modular linking is that the phases of compilation and linking are truly separate: they occur at separate times and can be performed by different parties.

To better understand modular linking, we propose a formal framework that describes how modular linking can be implemented in an arbitrary language. An intuition of how our framework works is illustrated in Figure 4. Linking that is not modular, which we refer to as *whole linking*, is shown in the top part of Figure 4: the code of components are compiled directly against the code of other components. In our framework, the key to making linking modular is to provide an **abstraction** between the compilation of a component and its linking with other components in a system. Rather than compile a component against other components in a system, a component is compiled against its abstraction, as shown in the bottom part of Figure 4. The abstractions of components are then used during linking to ensure they are compatible. Because components are not compiled directly against each other, as in whole linking, modular linking can reuse the results of compilation to avoid inspecting component code when ensuring static type safety of the system during linking.

A formal description of our framework is expressed over systems written in some language L, where linking of a system ensures the system conforms to the static type

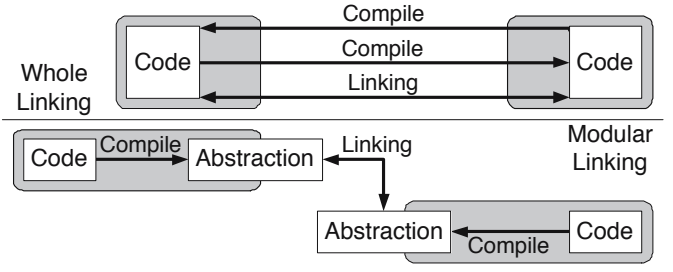

Fig. 4. A comparison between whole linking (top) and modular linking in our framework (bottom); single directed arrows labeled "compile" point from code being compiled to what the code is being compiled against

safety properties of L. Before modular linking is formally described, a corresponding whole linking can be formally described with the definition of the rule **WHOLE-OK**:

$$\frac{\vdash_L \textbf{ENV-OK}\ \overrightarrow{\| \mathtt{c} \|} \qquad \overrightarrow{\| \mathtt{c} \|} \vdash_L \overrightarrow{\textbf{IMPL-OK}\ \mathtt{c}}}{\vdash_L \textbf{WHOLE-OK}\ \mathtt{p} = \overrightarrow{\mathtt{c}}}$$

Notation: Lower-case letters designate instances of constructs, and the same letters (differing only with subscripts) are used to designate instances of the same construct. Rules are in small caps; e.g., **RULE-OK**. Directed overbars are vectors that designate unordered sets; e.g., $\overrightarrow{\mathtt{c}}$ is a set of construct instances designated by $\mathtt{c}$. A rule within a vector applies to all elements of any sets under the same vector, but any elements adjacent to the rule not within the same vector are duplicated as the rule is applied to these elements; e.g., if $\overrightarrow{\mathtt{c}} = \mathtt{c}_0, \mathtt{c}_1, \mathtt{c}_2$ then enforcement of $\overrightarrow{\mathtt{s}}\ \overrightarrow{\textbf{RULE-OK}\ \mathtt{c}}$ expands to enforce $\overrightarrow{\mathtt{s}}$ **RULE-OK** $\mathtt{c}_0$, $\overrightarrow{\mathtt{s}}$ **RULE-OK** $\mathtt{c}_1$, and $\overrightarrow{\mathtt{s}}$ **RULE-OK** $\mathtt{c}_2$.

For the rule **WHOLE-OK**, a system $\mathtt{p}$ consists of syntactically separated, but not modular, parts designated by $\mathtt{c}$ and written in the language L. We refer to these parts as *L-parts*, which have *shapes* that can be used to reason about interactions between L-parts. An L-part shape can be extracted from an L-part using the double bar operator ($\| \mathtt{c} \|$). L-part shapes are combined to form a *typing environment*, which is used to reason about the type correctness of a group of L-parts. A typing environment can be used to ensure the type correctness of L-part implementations, which is enforced by the rule **IMPL-OK**, if it is closed and well-formed, which is enforced by the rule **ENV-OK**. The rules **ENV-OK** and **IMPL-OK** depend on the language L. In Section 4, each L-part models a Java class and definitions for the rules **ENV-OK** and **IMPL-OK** model type checking for the Java language.

In the definition of rule **WHOLE-OK**, only a single "whole" typing environment formed from the shapes of all the system's L-parts ($\overrightarrow{\| \mathtt{c} \|}$) is used to type check the system. Linking according to the rule **WHOLE-OK** is not modular because type checking occurs over a single whole typing environment. For linking to be modular in our framework, the system's L-parts are placed inside special kinds of components referred

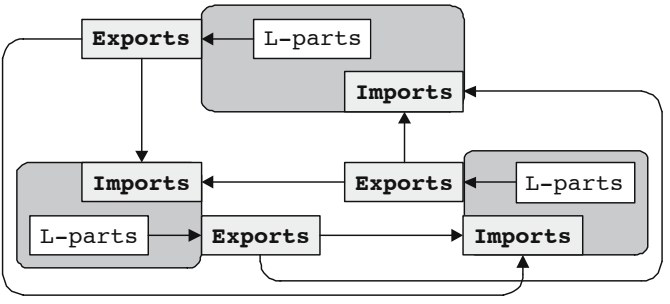

Fig. 5. An illustration of how abstraction enables modular linking in a system of three units; abstraction relationships are solid directed lines from what is being abstracted to the abstraction

to as *unit* constructs, designated by u. Modular linking is then defined in our framework as follows:

$$\frac{\vdash_L \overrightarrow{\textsc{Compile-Ok}\ \mathtt{u}} \qquad \vdash_L \textsc{Link-Ok}\ \overrightarrow{\|\ \mathtt{u}\ \|}}{\vdash_L \textsc{Modular-Ok}\ p = \overrightarrow{\mathtt{u}}}$$

For the rule **Modular-Ok**, a system p consists of a set of units $\overrightarrow{\mathtt{u}}$, where each unit has a *signature*, designated by $\|\ \mathtt{u}\ \|$, which describes the unit's interactions with other units in the system without revealing the unit's implementation. The units of a system each undergo separate *compile-time typing* that is performed by the rule **Compile-Ok**. Compile-time typing can examine the private implementation of a unit, but does not look at how the unit is used in a system. All units of a system also collectively undergo *link-time typing* that is performed by the rule **Link-Ok**. Link-time typing only examines the signatures of the system's units ($\overrightarrow{\|\ \mathtt{u}\ \|}$).

To bridge type checking between the compile-time and link-time typing phases, the signature of each unit abstracts its interactions with other units in the system. The abstraction process is illustrated in Figure 5. A unit signature is divided into two sections: *exports* that abstract the unit's L-parts to other units in the system for use in link-time typing; and *imports* that abstract the exports of other units in the system to the unit for use in its compile-time typing. As a result, the use of a foreign L-part in a unit is abstracted twice: first, when it is exported from its originating unit; and second, when it is imported into the unit.

An L-part c can be described in a signature by an L-part shape designated by s. The format of a unit is $\mathtt{u} = \mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \overrightarrow{\mathtt{c}}$, where U uniquely identifies the unit u in a system, $\overrightarrow{\mathtt{s}_i}$ describes u's imports, $\overrightarrow{\mathtt{s}_e}$ describes u's exports, and $\overrightarrow{\mathtt{c}}$ are the L-parts that make up u's private implementation. The signature of a unit only consists of its identifier, imports, and exports, and does not include its private implementation, so $\|\ \mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \overrightarrow{\mathtt{c}}\ \| = \mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}$. The definitions of the rules **Compile-Ok** and **Link-Ok** are as follows:

$$\frac{\vdash_L \textbf{ENV-OK}\ \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\| \mathtt{c} \|} \qquad \vdash_L \textbf{ENV-OK}\ \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e} \qquad \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\| \mathtt{c} \|} \vdash_L \overrightarrow{\textbf{IMPL-OK}\ c} \quad \vdash_L \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}\ \textbf{ABSTRACTS-OK}\ \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\| \mathtt{c} \|}}{\vdash_L \textbf{COMPILE-OK}\ \mathtt{u} = \mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \overrightarrow{\mathtt{c}}}$$

$$\frac{\vdash \textbf{UNIQUE}\ \overrightarrow{\mathtt{U}} \qquad \vdash_L \textbf{ENV-OK}\ \overrightarrow{\overrightarrow{\mathtt{s}_e}} \qquad \vdash_L \overrightarrow{\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}\ \textbf{ABSTRACTS-OK}\ \overrightarrow{\mathtt{s}_e}}}{\vdash_L \textbf{LINK-OK}\ \overrightarrow{\| \mathtt{u} \| = \mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}}}$$

More notation: Multiple sets can be combined into a larger set with the union ($\cup$) operator. A set of structures is designated by adjacent elements under the same overbar, where an element of the structure can be pulled out to form its own set; e.g., $\overrightarrow{\mathtt{U}\ \overrightarrow{\mathtt{s}_i}}$ can be used to form $\overrightarrow{\mathtt{U}}$ and $\overrightarrow{\overrightarrow{\mathtt{s}_i}}$. The construct $\overrightarrow{\overrightarrow{\mathtt{s}}}$ is not a set of $\mathtt{s}$ sets; it is a single set that is formed by unioning the $\mathtt{s}$ sets together.

The definition of the rule **COMPILE-OK** combines a unit's imports ($\overrightarrow{\mathtt{s}_i}$) with the shapes of the L-parts in its internal implementation ($\overrightarrow{\| \mathtt{c} \|}$) to form a *compile-time typing environment*, which is used to reason about the unit's internal implementation using the rules **ENV-OK** and **IMPL-OK**. These rules are the same as those used in the definition of whole linking (**WHOLE-OK**), whose definitions only depend the language L, and do not depend on whether linking is modular or not. The rule **LINK-OK** combines the exports of all units ($\overrightarrow{\overrightarrow{\mathtt{s}_e}}$) in the system to create an *link-time typing environment*, which is used to type check interactions between units in the system.

The key to modular linking in our framework is an *abstraction relationship* that is symmetrically enforced between compile-time and link-time typing environments. We say that a unit's imports and exports ($\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}$) "abstracts" a typing environment correctly if the two following criteria hold: the imports and exports collectively specify a subset of the abstracted typing environment; and the imports and exports do not hide anything about the abstracted typing environment that could confuse or create ambiguity in modular linking. "Correctly abstracts" is enforced by the rule **ABSTRACTS-OK**, whose definition depends on how type checking can be made modular in language L.

The rule **ABSTRACTS-OK** represents only part of the extra work that must occur to make linking modular; the rest of the extra work is performed by *link reduction*, which transforms a system of units into a linked system of just L-parts. Link reduction rewrites the L-parts of units so that no ambiguities in typing occur over the resulting linked system. Names locally used in a unit must be renamed so that they do not conflict with the names used in other units of the system. In our framework, link reduction allows a modular system to be subjected to program evaluation, and allows us to express a necessary relationship between whole and modular linking as Lemma 1:

Lemma 1 (MODULAR-IMPLIES-WHOLE)

$$\vdash_L \textbf{MODULAR-OK}\ \mathtt{p_u} = \overrightarrow{\mathtt{u}} \qquad \overrightarrow{\| \mathtt{u} \|} \vdash_L \overrightarrow{\mathtt{u} \rightarrow \overrightarrow{\mathtt{c}}} \quad \Rightarrow \quad \vdash_L \textbf{WHOLE-OK}\ \mathtt{p_c} = \overrightarrow{\overrightarrow{\mathtt{c}}}$$

Link reduction occurs with the arrow ($\rightarrow$) operator, and only depends on the signatures of a system's units ($\overrightarrow{\| \mathtt{u} \|}$) when linking each unit. Lemma 1 states that if modular linking ensures that a modular system of units ($\mathtt{p_u}$) is statically type safe, then whole

linking ensures that the corresponding linked system ($\mathtt{p_c}$) is statically type safe. It is possible that the linked system is statically type safe when modular linking has determined that the modular system of units is not; modular linking is more conservative than whole linking.

When Lemma 1 can be proven, modular linking is sound when the corresponding whole linking is sound. A proof of Lemma 1 depends on link reduction and the rule **ABSTRACTS-OK**, where more than one set of definitions may be able to facilitate a proof of Lemma 1. **ABSTRACTS-OK** can be defined in a trivial way that always ensures a proof of Lemma 1:

$$\frac{\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e} == \overrightarrow{\mathtt{s}_r}}{\vdash_L \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e} \textbf{ NO-ABSTRACTS-OK } \overrightarrow{\mathtt{s}_r}} \qquad \overrightarrow{\mathtt{U_a}\ \overrightarrow{\mathtt{s}_{ia}}\ \overrightarrow{\mathtt{s}_{ea}}} \vdash_L \mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \overrightarrow{\mathtt{c}} \rightarrow \overrightarrow{\mathtt{c}}$$

The definition of the rule **-ABSTRACTS-OK** forces a unit's imports and exports to always be equivalent to the unit's compile-time and the system's link-time typing environment, which means no abstraction occurs at all! The entire shape of every L-part in the system would be exposed in the imports and exports of each unit, and any trivial change of any unit in the system would invalidate linking of the entire system. Modular linking is only useful if a sufficient amount of abstraction can be supported. How much abstraction can be supported depends on the features of the core language L.

4 MiniJiazzi

The modular linking framework in Section 3 can be applied to the task of modular linking of programs written in a small Java-like core language, which models the addition of modular linking into Java. Language L is bound to language J, where J is our small Java-like language called core MiniJiazzi. We refer to the resulting language enhanced with modular linking as MiniJiazzi because it models Jiazzi [19]; Jiazzi enhances the full Java language with modular linking. Our experience with Jiazzi is the primary basis for our modular linking framework.

4.1 Core language

Core MiniJiazzi is similar to other small Java-like languages; e.g., ClassicJava [14] or Featherweight Java [16]. The syntax and type-checking rules of core MiniJiazzi are shown in Figure 6. So that we can focus our discussion on how modular type checking must deal with inheritance and virtual methods, core MiniJiazzi does not support fields, constructors, or casting. Besides class implementations (`c`) and class shapes (`s`) (the L-parts and L-part shapes in our modular linking framework), the syntax of core MiniJiazzi also defines method implementations (`m`), method shapes (`n`), types (`t`), and expressions in methods (`e`). Expressions can instantiate classes, access arguments and `this`, and call virtual methods on objects. The extends operator ($\lhd$) is used to specify a class's apparent direct super type, which is either another class or the root type `Object`.

To aid in our reasoning, we have added two features to the MiniJiazzi core language that do not have equivalent support in the Java language but are easily derived from conventional Java class definitions. First, a class shape describes only the "fresh" methods

$$\begin{array}{rlrl} \mathtt{s} = & \mathtt{C[U]} \lhd \mathtt{t}_{super}\ \overrightarrow{\mathtt{n}_{fresh}} & \mathtt{n} = & \mathtt{t}_{return}\ \mathtt{M[U]}(\overrightarrow{\mathtt{t}_{arg}\ \mathtt{x}}) \\ \mathtt{c} = & \mathtt{s}_{sig}\ \overrightarrow{\mathtt{m}_{impl}} & \mathtt{m} = & \mathtt{n}_{sig}\ \{\ \mathtt{e}_{return}\ \} \end{array}$$

$$\mathtt{t} = \mathtt{Object} \mid \mathtt{C[U]}$$

$$\mathtt{e} = \mathtt{new}\ \mathtt{C[U]} \mid \mathtt{x} \mid \mathtt{this} \mid \mathtt{e}.\mathtt{M[U_a]}(\overrightarrow{\mathtt{e}_x})$$

$$\frac{\vdash \textbf{UNIQUE}\ \overrightarrow{\mathtt{C[U]}} \quad \vdash \textbf{UNIQUE}\ \overrightarrow{\mathtt{M[U_a]}} \quad \overrightarrow{\mathtt{s}} \vdash_m \overrightarrow{\mathtt{M[U_a]} \notin \mathtt{t}_s} \quad \overrightarrow{\mathtt{s}} \vdash_t \mathtt{t}_s \cup \overrightarrow{\mathtt{t}_{ret}} \cup \overrightarrow{\mathtt{t}_x} \quad \overrightarrow{\mathtt{s}} \vdash_t \overrightarrow{\mathtt{C[U]} < \mathtt{Object}}}{\vdash_J \textbf{ENV-OK}\ \mathtt{s} = \mathtt{C[U]} \lhd \mathtt{t}_s\ \overrightarrow{\mathtt{n} = \mathtt{t}_{ret}\ \mathtt{M[U_a]}(\overrightarrow{\mathtt{t}_x\ \mathtt{x}})}}$$

$$\frac{\overrightarrow{\mathtt{s}} \vdash_m \mathtt{t}_r\ \overrightarrow{\mathtt{M[U_a]}(\overrightarrow{\mathtt{t}_x\ \mathtt{x}})} \in \mathtt{C[U]} \quad \overrightarrow{\mathtt{n}_{fresh}} \subseteq \overrightarrow{\mathtt{t}_r\ \mathtt{M[U_a]}(\overrightarrow{\mathtt{t}_x\ \mathtt{x}})} \quad \overrightarrow{\overrightarrow{\Gamma(x) = \mathtt{t}_x} \quad \overrightarrow{\Gamma(\mathtt{this})} = \mathtt{C[U]} \quad \overrightarrow{\mathtt{s}}, \Gamma \vdash_e \overrightarrow{\mathtt{e}_r \in \mathtt{t}} \quad \overrightarrow{\mathtt{s}} \vdash_t \overrightarrow{\mathtt{t} \leq \mathtt{t}_r}}}{\overrightarrow{\mathtt{s}} \vdash_J \textbf{IMPL-OK}\ \mathtt{C[U]} \lhd \mathtt{t}_s\ \overrightarrow{\mathtt{n}_{fresh}}\ \overrightarrow{\mathtt{m} = \mathtt{t}_r\ \mathtt{M[U_a]}(\overrightarrow{\mathtt{t}_x\ \mathtt{x}})\ \{\ \mathtt{e}_r\ \}}}$$

$$\frac{\overrightarrow{\mathtt{s}} \vdash_t \mathtt{C[U]} < \mathtt{Object}}{\overrightarrow{\mathtt{s}}, \Gamma \vdash_e \mathtt{new}\ \mathtt{C[U]} \in \mathtt{C[U]}} \qquad \overrightarrow{\mathtt{s}}, \Gamma \vdash_e \mathtt{x} \in \Gamma(x) \qquad \overrightarrow{\mathtt{s}}, \Gamma \vdash_e \mathtt{this} \in \Gamma(\mathtt{this})$$

$$\frac{\overrightarrow{\mathtt{s}}, \Gamma \vdash_e \mathtt{e} \in \mathtt{C_b[U_b]} \quad \overrightarrow{\mathtt{s}} \vdash_m \mathtt{t}_r\ \mathtt{M[U_a]}(\overrightarrow{\mathtt{t}_x\ \mathtt{x}}) \in \mathtt{C_b[U_b]} \quad \overrightarrow{\mathtt{s}}, \Gamma \vdash_e \overrightarrow{\mathtt{e}_x \in \mathtt{t}_y} \quad \overrightarrow{\mathtt{s}} \vdash_t \overrightarrow{\mathtt{t}_y \leq \mathtt{t}_x}}{\overrightarrow{\mathtt{s}}, \Gamma \vdash_e \mathtt{e}.\mathtt{M[U_a]}(\overrightarrow{\mathtt{e}_x}) \in \mathtt{t}_r}$$

Fig. 6. The syntax and type-checking rules of core MiniJiazzi; evaluation reduction is not shown

of a class, which must not exist in the class's superclass. Next, to accommodate link reduction, class and method names are enhanced with *linking offsets*. Linking offsets are significant parts of class and method names; e.g., $\mathtt{C[U_a]}$ and $\mathtt{C[U_b]}$ identify different classes when $\mathtt{U_a} \neq \mathtt{U_b}$. Linking offsets are used during link reduction to distinguish names that may clash after linking.

The type-checking rules of core MiniJiazzi consists of definitions for the rules **ENV-OK** and **IMPL-OK**. The rule **ENV-OK** ensures the following (in the top part of the judgment that defines **ENV-OK**; from left to right, top to bottom):

1. The names of all classes, taking into account their linking offsets, are unique in the typing environment;
2. The shapes of fresh methods are unique in each class;
3. No fresh methods of a class exist in the class's superclasses, which ensures that a method can always be unambiguously referred to in a class by its name;
4. All types referred to in a class shape are defined in the typing environment;
5. Each class is a subtype of `Object`, which ensures there are no inheritance cycles in the typing environment.

For simplicity, the definition of rule **ENV-OK** does not allow for method overloading; overloading can always be handled through renaming. Typing relationships ($\vdash_t$) and method relationship ($\vdash_m$) are used in the definition of **ENV-OK** but are not defined in Figure 6. They have their traditional meanings: subtyping ($\leq$) is reflexive and transitive ($\leq$ adds associative); the $\in$ operator queries whether or not a method is visible in a class. Both typing relationships and method relationships depend only on a typing environment.

The definition for rule **IMPL-OK** ensures the following (again top to bottom, left to right): all methods implemented by a class are declared by the class or one of its superclasses; all fresh methods of the class are implemented; and all implemented methods are well-typed; $\Gamma(\ldots) = \ldots$ defines an expression typing environment.

Expression-level typing ($\vdash_e$) is a judgment over recursive expression structures that determines the static compile-time types of expressions. It takes the form $\overrightarrow{\mathtt{s}}, \mathtt{C[U]}, \Gamma \vdash_e \mathtt{e} \in \mathtt{t}$. The typing of expressions is standard; e.g., typing of method calls at the bottom of Figure 6 only ensures that the method exists in the statically determined type of the calling expression, and that the argument expressions are typed as subtypes of the argument types.

The evaluation reduction rules and the proof that shows that whole type checking in core MiniJiazzi is sound are similar to those in other ClassicJava [14] and Featherweight Java [16]. In this paper, we concentrate on a proof of Lemma 1 from Section 3. To do this, we apply our modular linking framework to core MiniJiazzi by defining the rule **ABSTRACTS-OK** and link reduction.

4.2 Modular Linking

MiniJiazzi is a small model of Jiazzi, which enhances Java with *program units* [13]. We focus on how Jiazzi units make linking in Java modular and not the novel features of Jiazzi units, such as externally-specified linking and hierarchical structuring.

Modular linking is implemented in MiniJiazzi as follows. MiniJiazzi specifies an abstraction relationship that allows for the modular detection inheritance cycles and name clashes, but also allows for enough hiding to permit expressive linking organizations, such as those that use the open class pattern. In MiniJiazzi, potential name clashes (method and class name ambiguity) that could occur with modular type checking are prevented through link reduction.

The structure of units is described in Section 3 as $\mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \overrightarrow{\mathtt{c}}$. The class shapes described by a unit and the classes in a unit's private implementation initially have empty linking offsets ($[\circ]$). Linking offsets will not be specified until link reduction of the modular system occurs. The fact that linking offsets are empty does not have any effect on the rules **ENV-OK** or **IMPL-OK**.

At minimum, abstraction in MiniJiazzi must ensure that the signature of a unit is a "subset" of the compile-time and link-time typing environments of its unit. This criteria is described by the definition for weak abstraction:

$$\frac{\overrightarrow{\mathtt{s}_i \cup \mathtt{s}_e} = \overrightarrow{\mathtt{C}[\circ] \lhd \mathtt{t}_s\ \mathtt{n}} \qquad \overrightarrow{\mathtt{C}[\circ]} \subseteq \overrightarrow{|\mathtt{s}_r|} \qquad \overrightarrow{\mathtt{s}_r} \vdash_m \overrightarrow{\mathtt{n} \in \mathtt{C}[\circ]} \qquad \overrightarrow{\mathtt{s}_r} \vdash_t \overrightarrow{\mathtt{C}[\circ] < \mathtt{t}_s}}{\vdash_J \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}\ \textbf{WEAK-ABSTRACTS-OK}\ \overrightarrow{\mathtt{s}_r}}$$

The unit's typing environment abstraction is designated by $\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}$, which is the imports and exports of a unit. The abstracted typing environment is designated by $\overrightarrow{\mathtt{s}_r}$, which is either the compile-time typing environment of the unit or a link-time typing environment of a system. The single bar operator takes a class shape and returns its class identifier; e.g., $\overrightarrow{|\mathtt{s}_r|}$ is a set of class identifiers for classes described in the abstracted typing environment $\overrightarrow{\mathtt{s}_r}$.

The definition of the rule **WEAK-ABSTRACTS-OK** ensures the following (top to bottom, left to right):

1. Every class described by the abstraction is described by a same-named class in the abstracted typing environment;
2. The methods of each class described by the abstraction exists in a class with the same identifier or its superclass of the abstracted typing environment;
3. The superclass of each class described by the abstraction is a superclass of a class with the same identifier in the abstracted typing environment.

The definition of the rule **WEAK-ABSTRACTS-OK** does not enable modular type checking to detect irresolvable method ambiguity or inheritance cycles, and so cannot be used in a proof of Lemma 1. The problem is that the abstracted typing environment contains the superclass relationships specified by the abstraction ($\overrightarrow{\mathtt{s}_r} \vdash_t \overrightarrow{\mathtt{C}[\circ] < \mathtt{t}_s}$), but the abstraction is free to hide any superclass relationships expressed by the abstracted typing environment. Take as an example the components illustrated in Figure 2 from Section 2, where an inheritance cycle occurs because the fact that class `B` is a subclass of class `C` at link-time can be hidden from component ***uc*** during its compile-time. A stronger definition of abstraction, which does not allow for the hiding of subclassing relationships, is as follows:

$$\frac{\overrightarrow{\mathtt{s}_i \cup \mathtt{s}_e} = \overrightarrow{\mathtt{C}[\circ] \lhd \mathtt{t}_s\ \overrightarrow{\mathtt{n}}} \qquad \overrightarrow{\mathtt{C}[\circ]} \subseteq \overrightarrow{|\mathtt{s}_r|} \qquad \overrightarrow{\mathtt{s}_r} \vdash_m \overrightarrow{\mathtt{n} \in \mathtt{C}[\circ]} \qquad \overrightarrow{\mathtt{C}[\circ] \lhd \mathtt{t}_s\ \ldots} \subseteq \overrightarrow{\mathtt{s}_r}}{\vdash_J \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}\ \textbf{STRONG-ABSTRACTS-OK}\ \overrightarrow{\mathtt{s}_r}}$$

The definition of the rule **STRONG-ABSTRACTS-OK** differs from **WEAK-ABSTRACTS-OK** only in that it prevents the hiding of superclass relationships in the abstracted typing environment by the abstraction with $\overrightarrow{\mathtt{C}[\circ] \lhd \mathtt{t}_s\ \ldots} \subseteq \overrightarrow{\mathtt{s}_r}$. While this stronger abstraction does enable modular type checking—it rejects the linking in Figure 2 and can be used in a proof of Lemma 1—it does not permit enough information hiding through abstraction; e.g., it prevents useful applications of the open class pattern.

In Figure 3 from Section 2, the class `DWidget` may or may not be one of the superclasses of class `FWidget`, depending on whether or not the components ***color_2*** and ***font*** are linked with the components ***base*** and ***fixed*** in a system. With the rule **STRONG-ABSTRACTS-OK**, the fact that the class `DWidget` is a superclass of class `FWidget`, and indirectly a superclass of class `BButton`, would have to be apparent during the compile-time of the components ***base*** and ***fixed***. Unfortunately, this also eliminates the possibility of linking components ***base*** and ***fixed*** with the component ***color_1*** instead, where the class `DWidget` is not provided.

We have discovered an effective compromise between the weakest and strongest abstraction relationships: superclass relationships are hidden by the abstraction of a

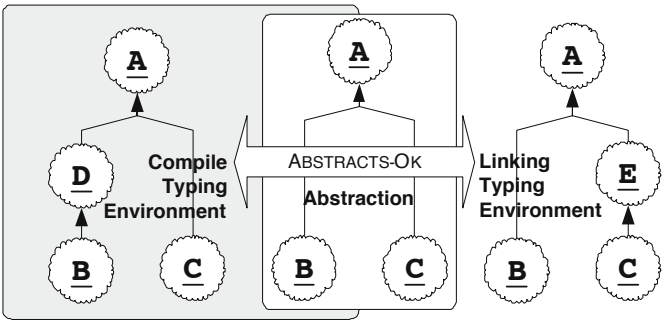

Fig. 7. An example of how the inheritance graph of an abstraction can hide inheritance relationships of classes in compile-time and link-time typing environments not visible in the abstractions

unit if and only if the classes involved are hidden by the abstraction. With help from an auxiliary rule **SUPER-OK**, this best definition of the rule **ABSTRACTS-OK** is as follows:

$$\frac{\mathtt{C}[\circ] \lhd \mathtt{t}_s \ \ldots \in \overrightarrow{\mathtt{s}_r}}{\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}, \overrightarrow{\mathtt{s}_r} \vdash_J \mathtt{C}[\circ] \ \textbf{SUPER-OK}\ \mathtt{t}_s} \qquad \frac{\mathtt{C}[\circ] \lhd \mathtt{C_a}[\circ] \ \ldots \in \overrightarrow{\mathtt{s}_r} \qquad \mathtt{C_a}[\circ] \notin \overrightarrow{|\mathtt{s}_i|} \cup \overrightarrow{|\mathtt{s}_e|} \quad \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}, \overrightarrow{\mathtt{s}_r} \vdash_J \mathtt{C_a}[\circ] \ \textbf{SUPER-OK}\ \mathtt{t}_s}{\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}, \overrightarrow{\mathtt{s}_r} \vdash_J \mathtt{C}[\circ] \ \textbf{SUPER-OK}\ \mathtt{t}_s}$$

$$\frac{\overrightarrow{\mathtt{s}_i \cup \mathtt{s}_e} = \overrightarrow{\mathtt{C}[\circ] \lhd \mathtt{t}_s\ \overrightarrow{\mathtt{n}}} \qquad \overrightarrow{\mathtt{C}[\circ]} \subseteq \overrightarrow{|\mathtt{s}_r|} \qquad \overrightarrow{\mathtt{s}_r} \vdash_m \overrightarrow{\mathtt{n} \in \mathtt{C}[\circ]} \qquad \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}, \overrightarrow{\mathtt{s}_r} \vdash_J \overrightarrow{\mathtt{C}[\circ] \ \textbf{SUPER-OK}\ \mathtt{t}_s}}{\vdash_J \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e} \ \textbf{ABSTRACTS-OK}\ \overrightarrow{\mathtt{s}_r}}$$

This definition of rule **ABSTRACTS-OK** permits the hiding of classes between units even if the classes occur in the middle of the inheritance hierarchy of two classes that are visible in the unit; e.g., class `DWidget` can be hidden from the components ***base*** and ***fixed*** in Figure 3. However, it also prevents the hiding of subtyping relationships between visible classes; e.g., a subtyping relationship between the `Cowboy` and `Icon` classes cannot be hidden if these classes are visible in the same scope.

The rule **SUPER-OK** ensures that a unit's abstraction ($\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}$) expresses every direct and indirect inheritance relationship in the abstracted typing environment ($\overrightarrow{\mathtt{s}_r}$) for classes visible in the abstraction. This relationship is illustrated in Figure 7, where the inheritance graph of the abstraction (center) expresses the proper inheritance relationships of classes `A`, `B`, and `C`, but ignores classes `D` and `E`, which are not visible in the abstraction. The relationship just discussed in English is expressed as Lemma 2, whose proof follows directly from the definition of rule **SUPER-OK**:

Lemma 2 (SUPER-ABSTRACTION)

$$\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e} = \overrightarrow{\mathtt{C}[\circ] \lhd \mathtt{t}_s \ \ldots} \quad \overrightarrow{\mathtt{C}[\circ]} \subseteq \overrightarrow{|\mathtt{s}_r|} \quad \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}, \overrightarrow{\mathtt{s}_r} \vdash_J \overrightarrow{\mathtt{C}[\circ] \ \textbf{SUPER-OK}\ \mathtt{t}_s} \quad \Rightarrow$$
$$\forall \mathtt{C_a}[\circ] \in \overrightarrow{|\mathtt{s}_i|} \cup \overrightarrow{|\mathtt{s}_e|}, \forall \mathtt{C_b}[\circ] \in \overrightarrow{|\mathtt{s}_i|} \cup \overrightarrow{|\mathtt{s}_e|} \ .\ \overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e} \vdash_t \mathtt{C_a}[\circ] < \mathtt{C_b}[\circ] \leftrightarrow \overrightarrow{\mathtt{s}_r} \vdash_t \mathtt{C_a}[\circ] < \mathtt{C_b}[\circ]$$

Our chosen definition of the rule **ABSTRACTS-OK** can be used in a proof of Lemma 1 in Section 3 and allows for a sufficient amount of abstraction to make modular type checking useful.

When units are combined together into a linked program, linking must ensure that name clashes between unit implementations do not cause ambiguities during evaluation of the program. 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 the names involved are exposed in unit signatures, MiniJiazzi's abstraction relationship permits hiding between units. Classes and methods hidden within a unit's implementation should not clash with classes and methods from other units. MiniJiazzi's definition of link reduction ensures that references to hidden classes and methods are disambiguated during linking.

MiniJiazzi achieves disambiguation with *linking offsets*. Every class and method reference within a unit implementation is qualified with a linking offset that is used disambiguate these references. Method implementations also have linking offsets to ensure that they implement or override the appropriate method from a superclass. Linking offsets are always treated as parts of class and method names. Unlike names, however, linking offsets are not provided by the programmer; they are assigned during link reduction, 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. Link reduction then binds linking offsets according to the unit that the class or method originates from.

The core judgments of link reduction specify class and method linking offsets. Other judgments are merely used to traverse the structure of a unit and are not shown in this paper. The following pair of judgments determines how linking offsets are bound for class references:

$$\frac{\mathtt{C}[\circ] \notin |\overrightarrow{\mathtt{s}_{ia}}|}{\mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}, \mathtt{U_a}\ \overrightarrow{\mathtt{s}_{ia}}\ \overrightarrow{\mathtt{s}_{ea}}\ \overrightarrow{\mathtt{c}_a} \vdash_J \mathtt{C}[\circ] \ \rightarrow\ \mathtt{C}[\mathtt{U_a}]} \qquad \frac{\mathtt{C}[\circ] \in |\overrightarrow{\mathtt{s}_{ia}}| \qquad \mathtt{C}[\circ] \in \overrightarrow{\mathtt{s}_{eb}} \qquad \mathtt{U_b}\ \overrightarrow{\mathtt{s}_{eb}} \in \mathtt{U}\ \overrightarrow{\mathtt{s}_e}}{\mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}, \mathtt{U_a}\ \overrightarrow{\mathtt{s}_{ia}}\ \overrightarrow{\mathtt{s}_{ea}}\ \overrightarrow{\mathtt{c}_a} \vdash_J \mathtt{C}[\circ] \ \rightarrow\ \mathtt{C}[\mathtt{U_b}]}$$

These judgments are used to specify linking offsets of any class referred to in a unit. References to classes that are not imported into the unit must be to classes that originate in the unit, so such references are assigned the linking offset of the referring unit, which is identified in the judgments as $\mathtt{U_a}$. References to imported classes are resolved to the unit that exports those classes in a system. Linking offsets for method reference are bound using a similar but more complicated pair of judgments:

$$\frac{\overrightarrow{\mathtt{s}_{ia}} \cup \overrightarrow{\|\mathtt{c}_a\|} \vdash_m \mathtt{M}[\circ] \in_{fresh} \mathtt{C_b}[\circ] \qquad \overrightarrow{\mathtt{s}_{ia}} \cup \overrightarrow{\|\mathtt{c}_a\|} \vdash_t \mathtt{C}[\circ] \leq \mathtt{C_b}[\circ] \qquad \mathtt{C_b}[\circ] \notin \overrightarrow{|\mathtt{s}_{ia}|}}{\mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}, \mathtt{U_a}\ \overrightarrow{\mathtt{s}_{ia}}\ \overrightarrow{\mathtt{s}_{ea}}\ \overrightarrow{\mathtt{c}_a} \vdash_J \mathtt{C}[\circ]{:}\mathtt{M}[\circ] \ \rightarrow\ \mathtt{M}[\mathtt{U_a}]}$$

$$\frac{\begin{array}{c}\overrightarrow{\mathtt{s}_{ia}} \cup \overrightarrow{\|\mathtt{c}_a\|} \vdash_m \mathtt{M}[\circ] \in_{fresh} \mathtt{C_a}[\circ] \qquad \overrightarrow{\mathtt{s}_{ia}} \cup \overrightarrow{\|\mathtt{c}_a\|} \vdash_t \mathtt{C}[\circ] \leq \mathtt{C_a}[\circ] \qquad \mathtt{C_a}[\circ] \in \overrightarrow{|\mathtt{s}_{ia}|} \\ \mathtt{U_b}\ \overrightarrow{\mathtt{s}_{eb}} \in \mathtt{U}\ \overrightarrow{\mathtt{s}_e} \qquad \overrightarrow{\mathtt{s}_e} \vdash_m \mathtt{M}[\circ] \in_{fresh} \mathtt{C_b}[\circ] \qquad \mathtt{C_b}[\circ] \in \overrightarrow{|\mathtt{s}_{eb}|} \qquad \overrightarrow{\mathtt{s}_e} \vdash_t \mathtt{C_a}[\circ] \leq \mathtt{C_b}[\circ]\end{array}}{\mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}, \mathtt{U_a}\ \overrightarrow{\mathtt{s}_{ia}}\ \overrightarrow{\mathtt{s}_{ea}}\ \overrightarrow{\mathtt{c}_a} \vdash_J \mathtt{C}[\circ]{:}\mathtt{M}[\circ] \ \rightarrow\ \mathtt{M}[\mathtt{U_b}]}$$

Each method reference ($\mathtt{C}[\circ]{:}\mathtt{M}[\circ]$) is associated with a class ($\mathtt{C}[\circ]$) that must implement the method being referred to. To determine the method's linking offset, the class that introduces the method must be first found in the compile-time typing environment of

```
unit cowboy {
 import
 export class Cowboy[o] ◁ Object
 { Object draw[o](), Object dual[o]() }
} {
 class Cowboy[cowboy] ◁ Object
 { Object draw[cowboy]()  { /* a gun duel */ },
   Object dual[cowboy]()  { this.draw[cowboy]() }  }
}
unit icon {
 import class Cowboy[o] ◁ Object {  }
 export class Icon[o] ◁ Cowboy { Object paint[o]()  }
} {
 class Icon[icon] ◁ Cowboy[cowboy]
 { Object paint[icon]()  { this.draw[icon]() },
   Object draw[icon]()  { .../* draw icon */ } }
}
unit main {
 import class Icon[o] ◁ Object
 { Object paint[o](), Object draw[o]() }
 export class Main[o] ◁ Object { Object main[o]()  }
} {
 class Main[main] ◁ Object
 { Object main[main]()  { this.draw[cowboy]() } }

}
```

Fig. 8. The units ***cowboy***, ***icon***, and ***main*** that are linked together into a system; linking offsets that result from link reduction are shown

the unit. In the top judgment, the introducing class is not an import of the unit, so the method must originate from the referring unit (U_a). In the bottom judgment, the introducing class is an import, so the class that introduces the method must be found with respect to the link-time typing environment. The unit that the introducing class is exported from is used as the method's linking offset.

The MiniJiazzi units shown in Figure 8, based on the illustration of Figure 1 from Section 2, 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, link reduction binds the linking offset of the call to method *draw* in class `Icon` to the unit ***icon***. Since the unit ***cowboy*** exports the method *draw*, whereas the unit ***icon*** does not, link 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 package-only), ambiguity between the *draw* methods could not be avoided and the Java source compiler would even reject such a construction. In this MiniJiazzi program

a source compiler error does not occur: the scopes of the *draw* methods are separated by abstraction and the unintended ambiguity is avoided through link reduction.

Rather than use linking offsets, other approaches [22,23,25] disambiguate between methods using dictionaries that are based on unit-like scopes. With dictionaries, a link reduction phase is not necessary; rather dictionaries are queried during evaluation. The advantage of using link reduction is the ability to easily express Lemma 1, which would be much more complicated if the evaluation of a modular system were different from the evaluation of a non-modular system.

Link reduction only binds linking offsets and does not need to otherwise change the structure of class expressions within a unit. The abstraction relationship ensures that referenced classes and methods exist within the program typing environment and that method implementations override methods correctly. The only nontrivial aspect of proving Lemma 1 is showing that the typing environment formed by the shapes of the resulting link-reduced classes is well-formed. We express this as Lemma 3:

Lemma 3 (LINK-REDUCED-ENV-OK)

$$p = \overrightarrow{\mathtt{u} = \mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \overrightarrow{\mathtt{c}_x}} \quad \vdash_J \textsc{Env-Ok}\ \overrightarrow{\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\| \mathtt{c}_x \|}}, \quad \overrightarrow{\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}}, \quad \overrightarrow{\overrightarrow{\mathtt{s}_e}}$$

$$\vdash_J \overrightarrow{\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}}\ \textsc{Abstracts-Ok}\ \overrightarrow{\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\| \mathtt{c}_x \|}} \quad \vdash_J \overrightarrow{\overrightarrow{\mathtt{s}_i} \cup \overrightarrow{\mathtt{s}_e}}\ \textsc{Abstracts-Ok}\ \overrightarrow{\overrightarrow{\mathtt{s}_e}}$$

$$\vdash \textsc{Unique}\ \overrightarrow{\mathtt{U}} \quad \overrightarrow{\| \mathtt{u} \|} \vdash_J \overrightarrow{\mathtt{u} \ \rightarrow\ \overrightarrow{\mathtt{c}_y}} \quad \Rightarrow \quad \vdash_J \textsc{Env-Ok}\ \overrightarrow{\overrightarrow{\| \mathtt{c}_y \|}}$$

The proof of Lemma 3 primarily depends on using Lemma 2 to show that the signature of each unit abstracts the inheritance graph of the link-reduced classes. That is, the linking typing environment of the modular system forms an inheritance graph that abstracts the inheritance graph formed by the non-modular system ($\overrightarrow{\overrightarrow{\| \mathtt{c}_y \|}}$), which is expressed as Lemma 4 that has the same antecedents as Lemma 3:

Lemma 4 (MODULAR-WHOLE-SUBTYPING)

$$\ldots \quad \Rightarrow \quad \forall \mathtt{U_a}\ \overrightarrow{\mathtt{s}_{ia}}\ \overrightarrow{\mathtt{s}_{ea}}\ \overrightarrow{\mathtt{c}_a}, \mathtt{U_b}\ \overrightarrow{\mathtt{s}_{ib}}\ \overrightarrow{\mathtt{s}_{eb}}\ \overrightarrow{\mathtt{c}_b} \in \overrightarrow{\mathtt{U}\ \overrightarrow{\mathtt{s}_i}\ \overrightarrow{\mathtt{s}_e}\ \overrightarrow{\mathtt{c}_x}}, \forall \mathtt{C_a} \in |\overrightarrow{\mathtt{s}_{ea}}|, \forall \mathtt{C_b} \in |\overrightarrow{\mathtt{s}_{eb}}|.$$

$$\overrightarrow{\mathtt{s}_e} \vdash_t \mathtt{C_a}[\circ] < \mathtt{C_b}[\circ] \leftrightarrow \overrightarrow{\overrightarrow{\| \mathtt{c}_y \|}} \vdash_t \mathtt{C_a}[\mathtt{U_a}] < \mathtt{C_b}[\mathtt{U_b}]$$

The last consequent of Lemma 4 states that all subtyping relationships in the linking environment between pre-linked classes ($\mathtt{C_a}[\circ]$ and $\mathtt{C_b}[\circ]$) must be preserved in the post-linked classes ($\mathtt{C_a}[\mathtt{U_b}]$ and $\mathtt{C_a}[\mathtt{U_b}]$). Lemma 4 represents the core of our proof of Lemma 1 for MiniJiazzi.

Proof sketch: Our proof of Lemma 4 proceeds by using induction and showing that contradictions necessarily occur if this consequent does not hold. Our inductive base case is based on the fact that segments in the inheritance hierarchy remain unchanged between pre-linking and post-linking as long as they do not contain imported or exported classes. As a result, the following two implications always hold:

$$\overrightarrow{\overrightarrow{\mathtt{s}_e}} \vdash_t \mathtt{C_a}[\circ] \lhd \mathtt{C_b}[\circ] \rightarrow \overrightarrow{\overrightarrow{\| \mathtt{c}_y \|}} \vdash_t \mathtt{C_a}[\mathtt{U_a}] < \mathtt{C_b}[\mathtt{U_b}]$$

$$\overrightarrow{\overrightarrow{\| \mathtt{c}_y \|}} \vdash_t \mathtt{C_a}[\mathtt{U_a}] \not< \mathtt{C_b}[\mathtt{U_b}] \rightarrow \overrightarrow{\overrightarrow{\mathtt{s}_e}} \vdash_t \mathtt{C_a}[\circ] \not\lhd \mathtt{C_b}[\circ]$$

These base cases will be reached reached as long as the the post-linked inheritance graph is acyclic, which is given by **ENV-OK** already being enfrorced on the linking environment.

4.3 Comparisons with Jiazzi

MiniJiazzi formally describes Jiazzi linking system that was introduced in [19]. Jiazzi contains many features that make linking more convenient. Jiazzi supports *package signatures* that describe the shapes for a package of classes and can be reused between units. Package signatures are used to generate import class stubs that enable unit compilation to occur with a standard Java complier. When implementing a unit from scratch, package signatures can also be used to generate skeletons for exported classes. Alternatively, package signatures can be inferred automatically from existing Java classes by assuming public and protected classes and methods should appear in the signatures. Linking occurs after compilation with Java bytecode rewriting of method and class names to implement linking offsets. Unit imports and exports are supported with extension that simplifies usage of the open class pattern that was described in Section 2. Using this mechanism, a unit can extended a package of classes without creating new subtypes. For this purpose, the abstraction we have shown to be safe and possible with MiniJiazzi is very essential as it makes the open class pattern possible.

5 Beyond Inheritance

When considering modular linking, inheritance in Java permits a sufficient amount of abstraction; e.g., the open class pattern as illustrated in Figure 3 from Section 2 can be expressed. Abstract methods in Java, however, cannot be abstracted, despite their name: abstract methods can never be hidden in visible classes and Java interfaces, because modular type checking must ensure concrete subclasses implement all abstract methods. This becomes a significant expressiveness problem when abstract methods are used aggressively in "framework classes," or when components evolve to provide new functionality that require adding new abstract methods to classes.

Binary compatibility in Java allows new abstract methods to be added to library classes. It also allows concrete classes to have abstract methods that are not implemented [18]; e.g., the AWT class `Graphics` is often enhanced with new abstract methods as the AWT library evolves. This is only safe when there are assurances that unimplemented abstract methods are never invoked; e.g., a program compiled against the old AWT library will not cause the new abstract methods added to the class `Graphics` to ever be called. Such assurances cannot be automatically verified with static type checking, so Java uses run-time type checking to detect and reject attempts to invoke unimplemented abstract methods.

Our proposed modular linking framework can be used to reason about modular linking in other languages, OO or otherwise. It can also be used as a metric for new experimental language features. Consider symmetric multi-methods, which support dispatch over arbitrary arguments. Compilation and linking of symmetric multi-methods require the detection of cases where a multi-method is overridden ambiguously [21], that is, where two or more specializations of the multi-method are equally applicable

to a combination of argument bindings. When this is enforced in the most direct way, multi-methods cannot support much abstraction under our modular linking framework because any class used in the specialization of a visible multi-method can never be hidden within a component. However, like inheritance, there is probably a middle ground similar to the rule **SUPER-OK**, where class hiding can be permitted if ambiguous multi-method overriding is type checked more conservatively.

6 Related Work

Separate compilation and modular linking are explored extensively by Cardelli [5] with linksets. In comparison, the work presented in this paper tackles modular linking from a different direction: rather than build modular linking into a newly designed language, we show how modular linking can be added to an existing language after it has been designed. Cardelli also does not address mutually-dependent modules or language features such as inheritance. On the other hand, Cardelli's correctness criteria are more rigorous including issues such as non-termination. In another direction, MTAL (modular typed-assembly language) [15] explores the low-level implications of binaries and modular linking. In contrast, our framework does not deal with low-level details such as Java bytecode, and instead focuses on high-level language abstractions. Effective modular systems must deal with both issues. Our modular linking framework is based on program units, which initially were conceived for Scheme and ML [10], and have been considered in OO extensions of Scheme [10]. With Jiazzi, we have shown how program units can be added to statically-typed OO languages.

Drossopoulou et al. extensively explore and formalize binary compatibility [9] and linking [8] in Java. Jiazzi eschews Java's binary compatibility in favor of modular linking, which we believe is more appropriate for component software. Both modular linking in Jiazzi and binary compatibility in Java address the technical "fragile base class problem." Ancona et al. [1] use a notion of a compilation schema to explore separate source code compilation and runtime linking in Java. The task of separate compilation in Jiazzi, as modeled by MiniJiazzi, is simplified because compile-time and link-time typing environments are explicitly separated by our modular linking framework.

The language JavaMod [3] explores adding a module system to the Java language, while the ML-like language Moby [11,12] explores modular linking for a class-based core language. Unlike Jiazzi, neither JavaMod nor Moby support mixin-style inheritance with abstraction. 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. Like most ML-like languages, Moby does not support modules with cyclic dependencies. MiniJiazzi is the first formalization of a statically-typed module system that supports cyclic dependencies, full mixin-style inheritance, and abstraction for an OO language. The language SmartJavaMod [2] enhances JavaMod with a form of signature inferencing and class overriding. It is an open question whether signature inferencing is feasible in a MiniJiazzi-like system because of abstraction.

The languages Dubious [21] and EML [20] explores modular type checking of symmetric multi-methods, while MultiJava [6,7] explores how symmetric multi-methods can be added to Java in a modular way. The hiding of abstract methods is also restricted

in these languages to preserve modular type checking. Imports are not expressed in the module interfaces of these languages, which leads to a different definition of abstraction than we model in our modular linking framework. Even without multi-methods in the Java language, Jiazzi can implement the open class idiom that multi-methods enable, as illustrated in Figure 3 from Section 2.

Enforcing the privacy of methods in OO languages has been explored extensively in the literature. Riecke and Stone [22] formally explore method privacy in structurally typed OO languages by using method dictionaries, while this work is extended by Stone [23] and Vouillon [25] in the context of class and mixin-based languages. Mini-Jiazzi differs by using linking offsets rather than dictionaries to enforce method scopes and disambiguate between method namespaces.

7 Conclusion and Future Work

We have shown how modular linking can be added to statically-typed OO languages such as Java while allowing several expressive features: cyclic dependencies between components, inheritance across component boundaries, and non-trivial abstraction between components. Our modular linking framework provides the intuitive and formal foundation for our work, and we have used this framework to formally reason about how modular linking can be added to Java with MiniJiazzi. MiniJiazzi models Jiazzi, which is an enhancement of Java whose implementation is available for download:
`http://www.cs.utah.edu/plt/jiazzi`

Although we have shown that inheritance is a modular language feature because it permits an adequate amount of abstraction, abstract methods are problematic, while multi-methods are still an open question. Future work should explore how our modular linking framework and language features such as abstract methods and multi-methods can be made to support more abstraction.

Modular reasoning is what makes developing software out of components possible. This reasoning goes beyond linking, compilation, and type checking to also include execution, testing, debugging, semantic correctness, and so on. Future work should explore how these other kinds of reasoning can be made modular through a more general modular reasoning framework.

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, G. Lagorio, and E. Zucca. Smart modules for Java-like languages. Submitted for publication, Dec. 2005.
3. D. Ancona and E. Zucca. True modules for Java classes. In *Proc. of ECOOP*, pages 354–380, June 2001.
4. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of OOPSLA*, pages 303–311, Oct. 1990.
5. L. Cardelli. Program fragments, linking and modularization. In *Proc. of POPL*, pages 266–277, Jan. 1997.

6. 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.
7. C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. Technical Report 04-01b, Iowa State University, Dept. of Computer Science, Dec. 2004. Accepted for publication, pending revision.
8. S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus: Torwards a model of separate compilation, linking and binary compatibility. In *Proc. of Logic in Computer Science*, July 1999.
9. S. Drossopoulou, D. Wragg, and S. Eisenbach. What is Java binary compatibility? In *Proc. of OOPLSA*, pages 341–361, Oct. 1998.
10. R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. of ICFP*, pages 98–104, Sept. 1998.
11. K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *Proc. of PLDI*, pages 37–49, May 1999.
12. K. Fisher and J. Reppy. Extending Moby with inheritance-based subtyping. In *Proc. of ECOOP*, pages 83–107, June 2000.
13. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. of PLDI*, pages 236–248, May 1998.
14. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. of POPL*, pages 171–183, Jan. 1999.
15. N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Proc. of POPL*, Jan. 1999.
16. 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.
17. 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/.
18. S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In *Proc. of OOPSLA*, Oct. 1998.
19. S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of OOPSLA*, pages 211–222, Oct. 2001.
20. T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *Proc. of ICFP*, Oct. 2002.
21. T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proc. of ECOOP*, pages 279–303, July 1999.
22. J. G. Riecke and C. A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999.
23. C. Stone. Extensible objects without labels. In *Proc. of FOOL*, Jan. 2002.
24. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 1998.
25. J. Vouillon. Combining subsumption and binary methods: An object calculus with views. In *Proc. of POPL*, pages 290–303, Jan. 2001.