Java Binary Compatibility is Almost Correct Version 2α

David Wragg and Sophia Drossopoulou and Susan Eisenbach
Department of Computing
Imperial College of Science, Technology and Medicine
email: dpw, sd and se @doc.ic.ac.uk

February 6, 1998

Abstract

The Java language description is unusual in that it defines the effect of interleaving separate compilation and source code modifications. In Java, certain source code modifications, such as adding a method to a class, are defined as *binary compatible*. The Java language description does not require the re-compilation of programs importing classes or interfaces which were modified in binary compatible ways, and it claims that successful linking and execution of the altered program is guaranteed. In this paper we show that Java binary compatibility does not actually guarantee successful linking and execution. We then suggest a framework in which we formalize the requirement of safe linking and execution without re-compilation and we propose a more modest definition of binary compatibility. We prove for a substantial subset of Java, that our definition guarantees safe linking and execution.

1 Introduction

Separate compilation, already introduced in the seventies [13, 2], enables programmers to develop and compile separately, parts of a larger program. Provided that they respect the interfaces to the rest of the program, one expects to be able to link and run without type errors. Separate compilation is supported by most language implementations, although it is under-specified in many language descriptions. Recently, Cardelli [1] proposed a framework to describe separate compilation and safe linking of modules.

Java [8] is unusual in that it defines the effect of interleaving of separate compilation and source code modifications, and imposes weaker requirements on re-compilation. Typically, source code modifications require re-compilation of any programs importing the modified classes or interfaces. In Java however, certain source code modifications, such as adding a method to a class, are defined as *binary compatible*, based on ideas suggested in [7]. The Java language

description does not require the re-compilation of programs importing classes or interfaces which were modified in binary compatible ways. It claims that successful linking and execution of the altered program is guaranteed.

Not only do binary compatible changes not require re-compilation of other classes, but in fact such re-compilations may not be possible: a binary compatible change to the source code for one class may cause the source code for other classes to no longer be type correct. Yet the guarantee of successful linking and execution still holds since only the binaries are consulted during these steps.

It has already been demonstrated in [4, 5, 3] that loopholes in the definition and implementation of binary compatibility provide opportunities for breaking the Java security mechanism. In this paper we are concerned with a more mundane, and more direct problem, namely that Java binary compatibility *does not guarantee* successful linking and execution. It is not difficult to put together a sequence of program modifications, each of which is binary compatible in the sense of Java, which results in link errors.

In this paper we suggest a framework for formalizing the requirement of safe linking and execution without re-compilation, propose a more modest definition of binary compatibility, and prove for a substantial subset of Java, that our definition guarantees safe linking and execution.

The Java language specification defines the term *binary compatibility* both in an "extensional" manner, expressing the requirements for binary compatible changes

> A change to a type is binary compatible with ... pre-existing binaries if pre-existing binaries that previously linked without error ... will continue to link without error ... with the binary of the modified ...

and in an "intensional" manner, namely stating *which* changes are binary compatible

> ...adding a class, adding an instance variable to a class, ...

and it seems to claim that the intensional definition implies the extensional.

In order to be able to prove such a claim, we need to clarify and distinguish the two meanings, and we therefore introduce two new terms: We consider source code modifications to be *link compatible*, if the types (*i.e.* classes and interfaces) that successfully linked with the original program will also successfully link with the modified program. We call our more restricted list of allowable changes (*e.g.* we do not allow the addition of methods to interfaces), *safe binary compatible*. Finally, we prove that safe binary compatibility implies link compatibility.

The terminology we use is based on our previous work formalizing the semantics of Java [6], and makes use of the soundness theorem proven there. However, it could be based on another formalization of Java, as long as it gave a sensible meaning to type checking, and distinguished source code from compiled code [12].

The remainder of this paper is organized as follows: In section 2 we summarize the meaning of binary compatibility from Java, demonstrate why it works

in most cases, and show an example where it does not. In section 3 we summarize the formalization concepts from [6] needed for the current discussion. In section 4 we give the definitions of link compatibility and safe binary compatibility and demonstrate that the latter implies the former. In section 5 we discuss other possible definitions for link compatibility. Finally in section 6 we draw conclusions and outline further work.

2 Binary Compatibility in Java

The following example demonstrates some of the issues around binary compatibility, and will serve as running example throughout the paper. We start with the classes `Student` and `CStudent`, and the commented code is ignored. For simplicity we ignore the issue of access restrictions (*e.g.* `private`, `public`, `import`). The class `CStudent` inherits the instance variable `grade` of type `int`. In the class `C100`, `DonKnuth`, of class `CStudent`, is assigned the grade `1`. This code fragment is well formed, and can be compiled, producing the files `Student.class`, `CStudent.class` and `C100.class`.

```
    class Student
       {   int grade; }
    class CStudent extends Student
       {   /* char grade; */ }

    CStudent DonKnuth;

    class C100
       {   void f() { DonKnuth.grade=1;  } }
/*  class Marker
       {   void g() { DonKnuth.grade='A';  } }  */
```

The comment is then removed from class `CStudent`, *i.e.* we add the instance variable `grade` of type `char` to class `CStudent`. The class `CStudent` is re-compiled, producing, say `CStudent'.class`. The addition of an instance variable is a binary compatible change [8], and therefore does not require further re-compilation. Then, we remove the comments around class `Marker`, thus defining a new class, `Marker`. In the body of its method `g()`, we assign the grade `'A'` to `DonKnuth`.

The class `Marker` is type correct in the new, augmented environment, and thus it can be compiled producing the file, say, `Marker.class`. The two changes, *i.e.* the addition of the instance variable `grade` in class `CStudent`, and the creation of the class `Marker`, are binary compatible changes, so the old classes do not need to be re-compiled. The binaries (*i.e.* `Student.class`, `CStudent'.class`, `C100.class` and `Marker.class`), can safely be linked together.

It is worth noting that the old sources are *not* type correct any more. If, for example, we re-compiled the class `C100`, we would obtain a type error for the

assignment `DonKnuth.grade=1`. Also, it is worth noticing that in the binary `C100.class` the expression `DonKnuth.grade` refers to an integer, whereas in the binary `Marker.class` it refers to a character. The two occurrences of the expression refer to different parts of `DonKnuth`; this is reflected in the code produced, and is also shown in section 3.

2.1 A problem with binary compatibility

The next example demonstrates that the definition of binary compatibility in [8] needs to be less permissive. In particular, it considers the addition of methods to interfaces to be a binary compatible change, and as a result it does not prevent values of a particular interface type referring to objects of classes which do not fully implement that interface. This problem is known to JavaSoft [11].

```
interface I
   {   void meth1(); /* void meth2(); */}
class C implements I
   {   void meth1()
       {System.out.println("C::meth1() called");}}
class D
   {   void meth3() {I anI = new C;  /*anI.meth2();*/}}
```

Consider compiling interface `I` with `meth2()` commented out, then class `C`, and then class `D` with the comments around the expression `anI.meth2();`. Compilation will be successful.

Next the interface `I` is compiled again, this time after having removed the comments, so the method `meth2()` is added. This is a binary compatible change in Java, and does not require further re-compilation. So far, the application cannot actually invoke the methods added to `I`. But by making another binary compatible change, such as removing the comments from `meth3` in class `D`, code is incorporated which invokes the method added to `I`. The language specification [8] lists this as a binary compatible change, and does not require re-compilation. However, some kind of error should occur, yet what this error should be and when it should be detected is not specified.

The behaviour of Java implementations differs: After the first change and re-compilation, running the code from an application or an applet viewer, using the Linux port of the JDK version 1.1.3, gives:

```
java.lang.IllegalAccessError: Unimplemented interface method
```

After the first change and re-compilation, the code runs successfully as an applet with Netscape Communicator version 4.03. However, after the second change and re-compilation, the following error occurs:

```
java.lang.IncompatibleClassChangeError: interface method
meth2()V not implemented in C.
```

The best solution seems to be to consider adding methods (directly or by adding super-interfaces) to interfaces as a binary incompatible change.

3 Formalization of the Java Semantics

In this paper we shall use some of the formalization of type checking, compiling and executing of Java programs found in [6].

$$\begin{array}{ccccccc} \text{Java} & \supset & \text{Java}_s & \xrightarrow[\mathcal{C}]{} & \text{Java}_{se} & \rightsquigarrow_{\mathtt{p}} & \text{Java}_{se} \\ & & \downarrow & & \downarrow & & \downarrow \\ & & \text{Type} & = & \text{Type} & \geq_{wdn} & \text{Type} \end{array}$$

We have defined Java_s, a subset of Java, describing primitive types, classes and inheritance, instance variables and instance methods, interfaces, shadowing of instance variables, dynamic method binding, the `null` value, arrays, exceptions and exception handling. We do *not* include object and array creation yet, because their current description in [6] is not abstract enough for use in this work.

The syntax of Java_s can be found in the appendix. For convenience, we split the typing information and the evaluation information into an environment, usually denoted by a Γ, and a program, usually denoted by a `p`. An environment is a sequence of class declarations, interface declarations, and variable declarations. A program is a sequence of class bodies. A class body names the super-class of the class, and contains the method bodies for the class.

For the computer science students example, the original fragment would be represented in Java_s by the environment Γ, and the program $\mathtt{p}_s$. Γ consists of $\Gamma_{\mathtt{st}}\,\Gamma_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}$, where

$\Gamma_{\mathtt{st}}$ = `Student ext Object{grade : int}, DonKnuth : CStudent`
$\Gamma_{\mathtt{cst}}$ = `CStudent ext Student{}`
$\Gamma_{\mathtt{c100}}$ = `C100 ext Object{f :→ void}`

The program $\mathtt{p}_s$ consists of the three class bodies $\mathtt{p}_{\mathtt{st}}$, $\mathtt{p}_{\mathtt{cst}}$, $\mathtt{p}_{\mathtt{c100}}$, where

$\mathtt{p}_{\mathtt{st}}$ = `Student ext Object{}`
$\mathtt{p}_{\mathtt{cst}}$ = `CStudent ext Student{}`
$\mathtt{p}_{\mathtt{c100}}$ = `C100 ext Object{f is{DonKnuth.grade = 1}}`

Classes(Γ), *Interfaces*(Γ), and *Vars*(Γ) are the sets of names of classes, interfaces and variables declared in an environment Γ respectively. Similarly, *Classes*(`p`) are all the classes that appear in program `p`. Also:

- Γ(`C`) is the class declaration for a class `C` in Γ, or `Undef` if `C` $\notin$ *Classes*(Γ).
- Γ(`I`) is the interface declaration for an interface `I` in Γ, or `Undef` if `I` $\notin$ *Interfaces*(Γ).
- Γ(`C`) is the type of a variable `x` declared in Γ, or `Undef` in `x` $\notin$ *Vars*(Γ)

We use the terms *program fragments* and *environment fragments* to refer to entities which, while syntactically correct programs or environments respectively, are not intended to be self contained. We denote concatenation of program fragments or environment fragments by juxtaposition, *e.g.* $\Gamma\,\Gamma'$ is the environment composed of Γ and Γ'.

The assertions to describe the inheritance relationships for classes and interfaces, and the widening relationship, in an environment Γ are:

- $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$ indicates that $\mathtt{C}$ is a direct or indirect sub-class of $\mathtt{C}'$.
  $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}$ is equivalent to $\mathtt{C} \in \mathit{Classes}(\Gamma)$
- $\Gamma \vdash \mathtt{I} \leq \mathtt{I}'$ indicates that $\mathtt{I}$ is a direct or indirect sub-interface of $\mathtt{I}'$
  $\Gamma \vdash \mathtt{I} \leq \mathtt{I}$ is equivalent to $\mathtt{I} \in \mathit{Interfaces}(\Gamma)$
- $\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$ indicates that the type $\mathtt{T}$ *widens to* $\mathtt{T}'$, *i.e.* a value of type $\mathtt{T}$ can be assigned to a variable of type $\mathtt{T}'$ without any kind of run-time check.

We indicate by $\Gamma \vdash \diamond$, that the declarations in environment Γ are well-formed, *e.g.* that every identifier has a unique declaration, that instance variables are unique in a class, *etc.* Provided that $\Gamma \vdash \diamond$, Java$_s$ programs can be type checked in terms of a type inference system, parts of which appear in the appendix. The assertion $\Gamma \vdash \mathtt{t} : \mathtt{T}$ signifies that the term $\mathtt{t}$ has type $\mathtt{T}$ for the environment Γ, and the assertion $\Gamma \vdash \mathtt{p}\ \diamond$ signifies that the program $\mathtt{p}$ is well-typed in the environment Γ, *i.e.* that the class bodies contain type correct function bodies which return values of the expected types. The assertion $\Gamma \vdash \mathtt{p} \Diamond\!\!\!\!\Diamond$ signifies that $\mathtt{p}$ is complete, *i.e.* that it is well-typed and contains a class body for each class in Γ. Note that the notation $\Gamma \vdash \mathtt{p} \Diamond\!\!\!\!\Diamond$ corresponds to $\Gamma \vdash \mathtt{p}\ \diamond$ in [6]. That paper did not discuss program fragments, and so only needed to express the requirement that a program be well-typed *and* complete.

When Java$_s$ programs are type checked, they are enriched with type information. This is necessary for function calls and instance variable selection. The enriched language is Java$_{se}$, and enriching is performed by a type preserving mapping $\mathcal{C}$, which can also be understood as an abstraction of the compilation from Java to binary code. The syntax of Java$_{se}$ is an extension of the Java$_s$ syntax and is given in the appendix.

The Java$_{se}$ version of the original computer science students program consists of the compilation of each of the class bodies,

$$
\begin{array}{lll}
\mathtt{p}_{se} & = \mathcal{C}\{\!\{\mathtt{p}_s, \Gamma\}\!\} & = \mathtt{p}^{se}_{\mathtt{st}}\,\mathtt{p}^{se}_{\mathtt{cst}}\,\mathtt{p}^{se}_{\mathtt{c100}}, \qquad \text{where} \\
\mathtt{p}^{se}_{\mathtt{st}} & = \mathcal{C}\{\!\{\mathtt{p}_{\mathtt{st}}, \Gamma\}\!\} & = \mathtt{Student\ ext\ Object\{\}}, \\
\mathtt{p}^{se}_{\mathtt{cst}} & = \mathcal{C}\{\!\{\mathtt{p}_{\mathtt{cst}}, \Gamma\}\!\} & = \mathtt{CStudent\ ext\ Student\{\}}, \qquad \text{and} \\
\mathtt{p}^{se}_{\mathtt{c100}} & = \mathcal{C}\{\!\{\mathtt{p}_{\mathtt{c100}}, \Gamma\}\!\} & = \mathtt{C100\ ext\ Object} \\
 & & \quad \mathtt{\{f\ is\{DonKnuth[Student].grade = 1\}\}}
\end{array}
$$

Notice, that in $\mathtt{p}^{se}_{\mathtt{c100}}$ the instance variable access for `grade` has been enriched by the class from which `grade` is inherited.

Java$_{se}$ terms also have types. For a Java$_{se}$ term $\mathtt{t}$, the assertion $\Gamma \vdash_{se} \mathtt{t} : \mathtt{T}$ signifies that $\mathtt{t}$ has the type $\mathtt{T}$. For a Java$_{se}$ program $\mathtt{p}$, the assertion $\Gamma \vdash_{se} \mathtt{p} \diamond$ signifies that $\mathtt{p}$ is well-typed, whereas $\Gamma \vdash_{se} \mathtt{p} \circledast$ signifies that $\mathtt{p}$ is well-typed and complete.

In [6] we do not distinguish between assertions in Java$_s$, and those in Java$_{se}$; they all have the form ... $\vdash$ .. : .. or ... $\vdash$ ... $\diamond$. In this paper, we distinguish them through the suffix $_{se}$, for additional clarity.

The type system for Java$_{se}$ is an extension of that for Java$_s$, and some of it appears in the appendix. For type checking Java$_{se}$ instance variable access, the class from which the instance variable was inherited is taken into account. Similarly, for a Java$_{se}$ method call, the statically determined argument types are taken into account. These properties are crucial for the proof of the soundness theorem, and for the proofs of the lemmas in section 4.

The following lemma says that the enriching step preserves the type of the expression.

Lemma 1 *For types* $\mathtt{T}$, $\mathtt{T}'$ *any state* σ, *and Java*$_s$ *term* $\mathtt{t}$:

$$\mathit{If} \quad \Gamma \vdash \mathtt{t} : \mathtt{T} \quad \mathit{then} \quad \Gamma \vdash_{se} \mathcal{C}\{\!\{\mathtt{t}, \Gamma\}\!\} : \mathtt{T}$$

A term rewrite system $\rightsquigarrow_{\mathtt{p}}$ describes the Java$_{se}$ operational semantics for a particular program. The subject reduction theorem proves that one evaluation step preserves the types up to sub-classes/sub-interfaces. Finally, in [6] we prove a soundness theorem, which we shall need in the next section in order to prove the properties of the suggested safe binary compatibility:

Theorem 1 Soundness *For any Java*$_s$ *term* $\mathtt{t}$, *well-formed environment* Γ, *any type* $\mathtt{T}$ *with* $\Gamma \vdash \mathtt{t} : \mathtt{T}$, *any Java*$_{se}$ *program* $\mathtt{p}$ *with* $\Gamma \vdash_{se} \mathtt{p} \circledast$, *any state* σ *conforming to* Γ, *there exists a unique Java*$_{se}$ *term* $\mathtt{t}'$, *and a state* σ', *such that:*

- $\mathtt{T} \neq \mathtt{void}$, $\langle \mathcal{C}\{\!\{\mathtt{t}, \Gamma\}\!\}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'}{}^{*} \langle \mathtt{t}', \sigma' \rangle$, $\mathtt{t}'$ *is ground,* $\exists \mathtt{T}' :$ $\Gamma, \sigma' \vdash_{se} \mathtt{t}' : \mathtt{T}'$, $\Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}$ *and* σ' *conforms to* Γ *or*
- $\mathtt{T} = \mathtt{void}$, *and* $\langle \mathcal{C}\{\!\{\mathtt{t}, \Gamma\}\!\}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'}{}^{*} \langle \sigma' \rangle$ *and* σ' *conforms to* Γ *or*
- $\langle \mathcal{C}\{\!\{\mathtt{t}, \Gamma\}\!\}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'}{}^{*}$ *does not terminate or raises an exception*

Similar soundness theorems are proven in [12, 9] as well. Note that the exceptions here may be divide-by-zero, null access, *etc*, or user-defined exceptions, but they may not be linker exceptions. Thus, the Soundness Theorem suggests that the requirement $\Gamma \vdash_{se} \mathtt{p} \circledast$ indicates that $\mathtt{p}$ is a complete successfully-linked Java$_{se}$ program, and the requirement $\Gamma \vdash_{se} \mathtt{p} \diamond$ indicates that $\mathtt{p}$ is a "linkable" Java$_{se}$ program fragment.

Note, that in [6] we state the soundness theorem slightly differently. Namely we say that for any Java$_s$ program $\mathtt{p}$ with $\Gamma \vdash \mathtt{p} \diamond$, for the Java$_{se}$ program $\mathtt{p}_{se}$, $\mathtt{p}_{se} = \mathcal{C}\{\!\{\mathtt{p}, \Gamma\}\!\}$, the execution of $\mathtt{p}_{se}$ leads to a well-typed, ground term. However, this is so because the theorem in the [6] form is more interesting for language soundness. The soundness theorem in the above form is needed for the proof of theorem 2, and it can easily be proven, following the same technique as the original theorem.

4 Making Binary Compatibility Safe

4.1 Link Compatibility

The term link compatibility aims to capture the intuition underlying binary compatibility. It places some requirements on source code modifications.

Java source code modifications are reflected as environment modifications (modifications of the class hierarchy, the types of instance variables, methods *etc*) and as program code modifications (modifications of method bodies *etc*). We shall consider a pair $(\Gamma, \mathtt{p})$ describing the original environment and Java$_{se}$ program, and modify it with $(\Gamma'_1, \mathtt{p}'_1)$, the environment fragment and Java$_s$ program fragment to be incorporated, to give a new environment and program: the new environment consists of Γ'_1 and the part of Γ which is not superseded by Γ'_1 (denoted by Γ_0), and the new program consists of the compilation of $\mathtt{p}'_1$ in the new environment and the part of $\mathtt{p}$ which is not superseded by $\mathtt{p}'_1$ (denoted by $\mathtt{p}_0$), as shown in Figure 1.

Definition 1 *Given an environment* Γ*, a Java*$_{se}$ *program* $\mathtt{p}$*, an environment fragment* Γ'_1*, and a Java*$_s$ *program fragment* $\mathtt{p}'_1$*, such that Classes*$(\Gamma) =$ *Classes*$(\mathtt{p})$ *and Classes*$(\Gamma'_1) =$ *Classes*$(\mathtt{p}'_1)$*,* $(\Gamma, \mathtt{p}) \leftarrow_{\mathcal{C}} (\Gamma'_1, \mathtt{p}'_1)$ *is the result of* compiling Γ'_1, $\mathtt{p}'_1$ into Γ, $\mathtt{p}$.

$$(\Gamma, \mathtt{p}) \leftarrow_{\mathcal{C}} (\Gamma'_1, \mathtt{p}'_1) = (\Gamma_0\, \Gamma'_1, \mathtt{p}_0\, \mathcal{C}\, \{\mathtt{p}'_1, \Gamma_0\, \Gamma'_1\})$$

where Γ_0 *and* $\mathtt{p}_0$ *are unambiguously defined by:*

- $\Gamma = \Gamma_0\, \Gamma_1$
- $\mathtt{p} = \mathtt{p}_0\, \mathtt{p}_1$
- *Classes*$(\Gamma_0) =$ *Classes*$(\mathtt{p}_0)$
- *Classes*$(\Gamma_0) \cap$ *Classes*$(\Gamma'_1) = \emptyset =$ *Interfaces*$(\Gamma_0) \cap$ *Interfaces*(Γ'_1)
- *Classes*$(\mathtt{p}_1) =$ *Classes*$(\Gamma_1) \subseteq$ *Classes*(Γ'_1), *Interfaces*$(\Gamma_1) \subseteq$ *Interfaces*(Γ'_1)

In the above definition the split of Γ into Γ_0 and Γ_1 is unambiguously defined through Γ'_1. Specifically, any class from Γ that is also defined in Γ'_1 may not appear in Γ_0, and therefore it must appear in Γ_1. Any class from Γ that is not defined in Γ'_1 may not appear in Γ_1, and therefore must appear in Γ_0. Similarly for interfaces in Γ_0 and Γ_1, and for the split of $\mathtt{p}$ into $\mathtt{p}_0$ and $\mathtt{p}_1$.

In terms of the computing students example, the addition of the instance variable `grade` in the class `CStudent` corresponds to a new environment fragment, $\Gamma'_{\mathtt{cst}}$ = `CStudent ext Student{ grade : char }`, without affecting the corresponding Java$_s$ program fragment $\mathtt{p}_{\mathtt{cst}}$. Compiling $(\Gamma'_{\mathtt{cst}}, \mathtt{p}_{\mathtt{cst}})$ into $(\Gamma_{\mathtt{st}}\, \Gamma_{\mathtt{cst}}\, \Gamma_{\mathtt{c100}}, \mathtt{p}^{se}_{\mathtt{st}}\, \mathtt{p}^{se}_{\mathtt{cst}}\, \mathtt{p}^{se}_{\mathtt{c100}})$ gives

$$\begin{aligned}(\Gamma_{\mathtt{st}}\, \Gamma_{\mathtt{cst}}\, \Gamma_{\mathtt{c100}}, \mathtt{p}^{se}_{\mathtt{st}}\, \mathtt{p}^{se}_{\mathtt{cst}}\, \mathtt{p}^{se}_{\mathtt{c100}}) &\leftarrow_{\mathcal{C}} (\Gamma'_{\mathtt{cst}}, \mathtt{p}_{\mathtt{cst}}) \\ &= (\Gamma_{\mathtt{st}}\, \Gamma'_{\mathtt{cst}}\, \Gamma_{\mathtt{c100}}, \mathtt{p}^{se}_{\mathtt{st}}\, \mathtt{p}^{se}_{\mathtt{cst}}\, \mathtt{p}^{se}_{\mathtt{c100}})\end{aligned}$$

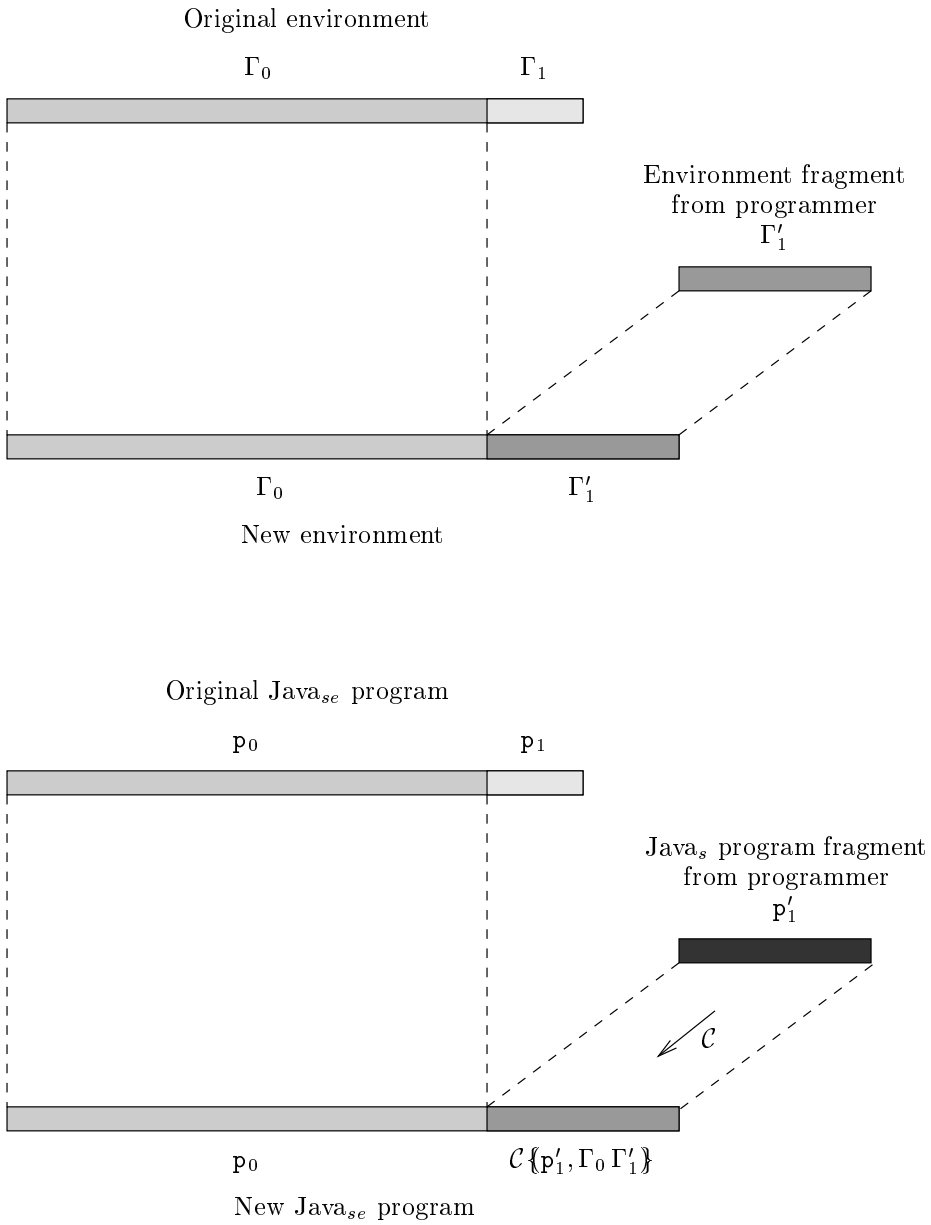

Figure 1: The separate compilation of Γ'_1, $\mathtt{p}'_1$ into Γ, $\mathtt{p}$

The next change, the addition of the class `Marker`, corresponds to a new environment fragment, $\Gamma'_{\mathtt{m}}$, and a new program fragment, $\mathtt{p_m}'$, where $\Gamma'_{\mathtt{m}} =$ `Marker ext Object {g: → void }`, and $\mathtt{p_m}' =$ `Marker ext Object {g is {DonKnuth.grade='A' }}`. These fragments are compiled into the result of the last change, $(\Gamma_{\mathtt{st}}\,\Gamma'_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}, \mathtt{p}^{se}_{\mathtt{st}}\,\mathtt{p}^{se}_{\mathtt{cst}}\,\mathtt{p}^{se}_{\mathtt{c100}})$, giving:

$$\begin{array}{c}(\Gamma_{\mathtt{st}}\,\Gamma'_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}, \mathtt{p}^{se}_{\mathtt{st}}\,\mathtt{p}^{se}_{\mathtt{cst}}\,\mathtt{p}^{se}_{\mathtt{c100}}) \leftarrow_{\mathcal{C}} (\Gamma'_{\mathtt{m}}, \mathtt{p_m}') \\ = (\Gamma_{\mathtt{st}}\,\Gamma'_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}\,\Gamma'_{\mathtt{m}}, \mathtt{p}^{se}_{\mathtt{st}}\,\mathtt{p}^{se}_{\mathtt{cst}}\,\mathtt{p}^{se}_{\mathtt{c100}}\,\mathcal{C}\{\mathtt{p_m}', \Gamma_{\mathtt{st}}\,\Gamma'_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}\,\Gamma'_{\mathtt{m}}\}).\end{array}$$

For such modifications to be link compatible, we require the following: The old and the new environments should be well-formed. Any Java$_{se}$ expression with type `T` in the old environment should also have type `T` in the new environment. The original program `p` should be well-formed and complete in the original environment Γ. The modified part of the program, $\mathtt{p}'_1$, should be well-formed in the new environment Γ''.

Definition 2 *An environment* Γ' *is* environment compatible *with environment* Γ *iff*

- $\Gamma \vdash \diamond$
- $\Gamma' \vdash \diamond$
- *For all Java*$_{se}$ *expressions* `e`*, types* `T` : $\quad \Gamma \vdash_{se} \mathtt{e} : \mathtt{T} \implies \Gamma' \vdash_{se} \mathtt{e} : \mathtt{T}$

For an environment Γ*, and Java*$_{se}$ *program* `p`*, an environment fragment* Γ'_1*, and a Java*$_s$ *program fragment* $\mathtt{p}'_1$*,* $(\Gamma'_1, \mathtt{p}'_1)$ *is* link compatible *with* $(\Gamma, \mathtt{p})$ *iff:*

- $\Gamma \vdash_{se} \mathtt{p} \Lsh\!\Rsh$
- Γ'' *is environment compatible with* Γ
- $\Gamma'' \vdash \mathtt{p}'_1 \diamond$

where $(\Gamma'', \mathtt{p}'') = (\Gamma, \mathtt{p}) \leftarrow_{\mathcal{C}} (\Gamma'_1, \mathtt{p}'_1)$.

In terms of the computing students example, one can see that $(\Gamma'_{\mathtt{cst}}, \mathtt{p_{cst}})$ is link compatible with $(\Gamma_{\mathtt{st}}\,\Gamma_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}, \mathtt{p}^{se}_{\mathtt{st}}\,\mathtt{p}^{se}_{\mathtt{cst}}\,\mathtt{p}^{se}_{\mathtt{c100}})$. Also

$$\Gamma_{\mathtt{st}}\,\Gamma'_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}\,\Gamma'_{\mathtt{m}} \vdash_{se} \mathtt{DonKnuth[Student].grade} = 1 : \mathtt{void},$$

and therefore it is the case that

$$\Gamma_{\mathtt{st}}\,\Gamma'_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}\,\Gamma'_{\mathtt{m}} \vdash_{se} \mathtt{p}^{se}_{\mathtt{st}}\,\mathtt{p}^{se}_{\mathtt{cst}}\,\mathtt{p}^{se}_{\mathtt{c100}} \diamond,$$

so the unmodified part of the program, as compiled originally with the old environment $\Gamma_{\mathtt{st}}\,\Gamma_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}$, is type correct in the new environment . On the other hand, `DonKnuth.grade=1` is type incorrect in the new environment, and so it is *not* the case that $\Gamma_{\mathtt{st}}\,\Gamma'_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}\,\Gamma'_{\mathtt{m}} \vdash \mathtt{p_{c100}} \diamond$.

It is straightforward to show that environment compatibility is transitive and reflexive. Despite its name, environment compatibility it is not a symmetric relationship (we chose the term "compatibility" for consistency with the terminology in the Java language specification).

On the other hand, the concepts of transitivity and reflexivity are not applicable to the link compatibility relationship, because the domain and range of that relation do not match. Instead, one might consider the following "naïve transitivity property": For non-overlapping Γ_1', Γ_2' if $(\Gamma_1', \mathtt{p}_1')$ is link compatible with $(\Gamma, \mathtt{p})$, and $(\Gamma_2', \mathtt{p}_2')$ is link compatible with $(\Gamma, \mathtt{p}) \leftarrow_{\mathcal{C}} (\Gamma_1', \mathtt{p}_1')$, then $(\Gamma_1'\,\Gamma_2', \mathtt{p}_1'\,\mathtt{p}_2')$ is link compatible with $(\Gamma, \mathtt{p})$. But $\mathtt{p}_1'$ may not be well-typed in the environment component of $(\Gamma, \mathtt{p}) \leftarrow_{\mathcal{C}} (\Gamma_1'\,\Gamma_2', \mathtt{p}_1'\,\mathtt{p}_2')$, and therefore property does not hold.

For example, consider an original Java$_{se}$ program and environment given by compiling the classes `Student` and `CStudent`. As the first change, the class `Marker` is compiled into the program. As the second change, the modified class `CStudent`$'$ (where the instance variable `grade` of type `char` has been added to `CStudent`) is compiled into the result of the first change. It is the case that both of the changes are link compatible. However, the change formed by naïvely composing the two steps, *i.e.* compiling `Marker` *and* `CStudent`$'$ into the original program, is not a link compatible change, since the Java$_s$ class definition of `Marker` is not well-typed in an environment featuring the class declaration from `CStudent`$'$.

The lack of a "naïve transitivity property" for link compatibility does not diminish the applicability of that concept. Although a sequence of link compatible changes cannot be folded into a single large link compatible change, such sequences do preserve an important property, namely that the resulting Java$_{se}$ programs remain well-typed and complete, as shown in Theorem 2.

For the proof of that theorem we use the following lemma which says that if a new environment is environment compatible with an old environment, then any Java$_{se}$ class body that is well-formed in the old environment will be well-formed in the new environment, as long as the corresponding class declaration is the same in both environments.

Lemma 2 *For environments* Γ *and* Γ'*, if*

- Γ' *is environment compatible with* Γ

then

- $\forall \mathtt{cBody} = \mathtt{C\ ext}\ ...\{...\}$
  $[\ \Gamma(\mathtt{C}) = \Gamma'(\mathtt{C})\ \ \textit{and}\ \ \Gamma \vdash_{se} \mathtt{cBody}\ \diamond\ ] \implies \Gamma' \vdash_{se} \mathtt{cBody}\ \diamond$

The proof of lemma 2 proceeds as follows: The class definition `cBody` consists of methods $\mathtt{m}_1, ...\mathtt{m}_n$, and the body of each method is a Java$_{se}$ term. If $\Gamma \vdash_{se} \mathtt{cBody}\ \diamond$, then the method body for $\mathtt{m}_i$ in `cBody` must be type correct and have the return type according to the declaration of $\mathtt{m}_i$ in $\Gamma(\mathtt{C})$. But because of environment compatibility the method body for $\mathtt{m}_i$ will also be type correct in

Γ' and have the same return type as in Γ. And if $\Gamma'(\mathtt{C}) = \Gamma(\mathtt{C})$, the signature for $\mathtt{m_i}$ in $\Gamma'(\mathtt{C})$ will be identical to its signature in $\Gamma'(\mathtt{C})$. Thus each method body is type correct in Γ' and conforms to its signature in $\Gamma'(\mathtt{C})$, and therefore $\Gamma' \vdash_{se} \mathtt{cBody} \diamond$.

The following theorem demonstrates that if we apply a sequence of link compatible changes to $(\Gamma, \mathtt{p})$ obtaining $(\Gamma'', \mathtt{p}'')$, then we can evaluate terms in the context of $\mathtt{p}''$ and execution will then produce a value of the expected type, or loop forever, or produce a run-time exception. None of these outcomes corresponds to a link error.

Theorem 2 *Given an environment Γ, Java$_{se}$ program $\mathtt{p}$, a sequence of environment fragments $\Gamma'_1, ...\Gamma'_n$, and a sequence of Java$_s$ program fragments $\mathtt{p}'_1, ...\mathtt{p}'_n$, if*

- *for all i, $1 \leq i \leq n$,*
  $(\Gamma'_i, \mathtt{p}'_i)$ is link compatible with $((\Gamma, \mathtt{p}) \leftarrow_{\mathcal{C}} (\Gamma'_1, \mathtt{p}'_1))... \leftarrow_{\mathcal{C}} (\Gamma'_{i-1}, \mathtt{p}'_{i-1})$

then

- *$\Gamma^{new} \vdash_{se} \mathtt{p}^{new} \ll$ where*
  $(\Gamma^{new}, \mathtt{p}^{new}) = ((\Gamma, \mathtt{p}) \leftarrow_{\mathcal{C}} (\Gamma'_1, \mathtt{p}'_1))... \leftarrow_{\mathcal{C}} (\Gamma'_n, \mathtt{p}'_n)$

Furthermore, by the Soundness theorem, the evaluation of a term in the context of $\mathtt{p}''$ will produce a value of the expected type, or loop forever, or produce a run-time exception.

Proof 4.1 *We first show that for any Java$_{se}$ program $\mathtt{p}$, environment Γ, Java$_s$ program fragment $\mathtt{p}'$ and environment fragment Γ', if $(\Gamma', \mathtt{p}')$ is link compatible with $(\Gamma, \mathtt{p})$ then for $(\Gamma'', \mathtt{p}'') = (\Gamma, \mathtt{p}) \leftarrow_{\mathcal{C}} (\Gamma', \mathtt{p}')$ it holds that $\Gamma'' \vdash_{se} \mathtt{p}'' \ll$:*

By definition 2, Γ'' is environment compatible with Γ, and so $\Gamma'' \vdash \diamond$.

For $\mathtt{p}'' = \mathtt{p}_0\, \mathcal{C}\{\!|\mathtt{p}', \Gamma'|\!\}$, where $\mathtt{p}_0$ has the same meaning as in definition 1 (i.e. the unmodified part of the program), by definition 2 we obtain $\Gamma \vdash_{se} \mathtt{p} \ll$, which implies $\Gamma \vdash_{se} \mathtt{p}_0 \diamond$, and so by lemma 2 gives $\Gamma'' \vdash_{se} \mathtt{p}_0 \diamond$. Link compatibility also gives $\Gamma'' \vdash \mathtt{p}' \diamond$, which by lemma 1 implies that $\Gamma'' \vdash_{se} \mathcal{C}\{\!|\mathtt{p}', \Gamma''|\!\} \diamond$. Therefore, $\Gamma'' \vdash_{se} \mathtt{p}'' \diamond$.

Furthermore, from definition 1 we have Classes$(\mathtt{p}'')$ = Classes(Γ'') and thus $\Gamma'' \vdash \mathtt{p}'' \ll$.

By application of this result n times, we obtain that $\Gamma^{new} \vdash_{se} \mathtt{p}^{new} \ll$.

Note that $\mathtt{p}^{new}$ is constructed by compilation of n program fragments with n versions of the environment, *i.e.* for $\Gamma_1, ..., \Gamma_n$ non-overlapping:

$$\begin{aligned} \mathtt{p}^{new} = \mathtt{p}_0\, &\mathcal{C}\{\!|\mathtt{p}'_1, \Gamma_0\, \Gamma'_1\, \Gamma_2 ... \Gamma_n|\!\} \\ &\mathcal{C}\{\!|\mathtt{p}'_2, \Gamma_0\, \Gamma'_1\, \Gamma'_2 ... \Gamma_n|\!\} \\ &\vdots \\ &\mathcal{C}\{\!|\mathtt{p}'_n, \Gamma_0\, \Gamma'_1\, \Gamma'_2 ... \Gamma'_n|\!\} \end{aligned}$$

The environments in which these compilations take place are crucial; note that $\mathcal{C}\{\!|\mathtt{p}'_1, \Gamma_0\, \Gamma'_1\, \Gamma_2 ... \Gamma_n|\!\}$ is not necessarily equal to $\mathcal{C}\{\!|\mathtt{p}'_1, \Gamma_0\, \Gamma'_1\, \Gamma'_2 ... \Gamma'_n|\!\}$. Indeed, the former may be defined but the latter undefined.

4.2 Weak Environment Compatibility

We have thus established that link compatibility is a desirable property. From definition 2 we see that link compatibility requires the new environment to be environment compatible with the old, and the new Java$_s$ program fragment to be type correct in the new environment. The latter requirement is very easy to establish, and corresponds to a successful local compilation step. Nevertheless, the first requirement, namely environment compatibility, is not obviously straightforward to establish, since it requires the two environments to give the same types to *all* Java$_{se}$ expressions.

In some cases, environment compatibility can be established easily. If only method bodies are changed in a Java$_s$ class definition, then the corresponding class declaration in the environment is unaffected, and as environment compatibility is reflexive, the relevant requirement for link compatibility holds. This confirms the fact that modifications of method bodies without modification of their signatures are binary compatible changes [8].

In this chapter we consider less trivial cases, and describe properties of changes to environments which can be established in a practical way, yet which imply environment compatibility.

Definition 3 *Given two well-formed environments,* Γ *and* Γ'*,* Γ' *is* weakly environment compatible *with* Γ *iff all the following hold:*

- $\forall \mathtt{C} \ : \ \Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C} \implies \Gamma' \vdash \mathtt{C} \sqsubseteq \mathtt{C}$
- $\forall \mathtt{I} \ : \ \Gamma \vdash \mathtt{I} \leq \mathtt{I} \implies \Gamma' \vdash \mathtt{I} \leq \mathtt{I}$
- $\forall \mathtt{T}, \mathtt{T}' \ : \ \Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}' \implies \Gamma' \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$
- $\forall \mathtt{C}, \mathtt{f} \ : \ \mathit{FDec}(\Gamma, \mathtt{C}, \mathtt{f}) = (\mathtt{C}, \mathtt{T}) \implies \mathit{FDec}(\Gamma', \mathtt{C}, \mathtt{f}) = (\mathtt{C}, \mathtt{T})$
- $\forall \mathtt{T}_1, \mathtt{m}, \mathtt{AT} \ : \ (\mathtt{T}, \mathtt{MT}) \in \mathit{MDecs}(\Gamma, \mathtt{T}_1, \mathtt{m}) \implies (\mathtt{T}', \mathtt{MT}) \in \mathit{MDecs}(\Gamma', \mathtt{T}_1, \mathtt{m})$
- $\mathit{Vars}(\Gamma) \subseteq \mathit{Vars}(\Gamma')$*, and* $\forall \mathtt{x} \in \mathit{Vars}(\Gamma) \ : \ \Gamma \vdash \mathtt{x} : \mathtt{T} \implies \Gamma' \vdash \mathtt{x} : \mathtt{T}$

Weak environment compatibility is a reflexive, transitive and antisymmetric relation. The following lemma says that, as desired, weak environment compatibility implies environment compatibility.

Lemma 3 *For environments* Γ *and* Γ'*, if* Γ' *is weakly environment compatible with* Γ*, then* Γ' *is environment compatible with* Γ*.*

This is proved by structural induction over the typing of Java$_{se}$ expressions.

4.3 Safe Binary Compatibility

The *safe binary compatible changes* are those changes described in [8], which apply to the language Java$_s$, *and* can be demonstrated to be safe.

The safe binary compatible changes are:

- adding a new class C or interface I to a program, as long as the name of the new type is not the same as that of any existing type;
- changing which is the direct super-class of a class C, as long as all direct or indirect super-classes continue to be direct or indirect super-classes;
- changing which are the direct super-interfaces of a class C, as long as all direct or indirect super-interfaces continue to be direct or indirect super-interfaces;
- adding a field to a class C;
- adding a method to a class C;
- changing a method body in class C, or changing the names (but not the types) of the formal parameters of a method.

These are formally described by the corresponding effect on an environment:

Definition 4 *An environment* Γ' *is the* result of *the application of a* safe binary compatible change *to another environment* Γ*, iff one of the following holds:*

- $\Gamma' = \Gamma, \mathtt{C\ ext\ C'\ impl\ I_1, ...I_n\ \{\ fDecls,\ mDecls\ \}}$
  $\mathtt{C} \notin \mathit{Classes}(\Gamma)$
- $\Gamma' = \Gamma, \mathtt{I\ ext\ I_1, ...I_n\ \{\ mDecls\ \}}$
  $\mathtt{I} \notin \mathit{Interfaces}(\Gamma)$
- $\Gamma(\mathtt{C}) = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_n\ \{\ fDecls,\ mDecls\ \}}$
  $\Gamma'(\mathtt{C}) = \mathtt{C\ ext\ C''\ impl\ I_1, ...I_n\ \{\ fDecls,\ mDecls\ \}}$
  $\Gamma' \vdash \mathtt{C''} \leq_{wdn} \mathtt{C'}$
  $\forall \mathtt{X} \neq \mathtt{C}\ :\ \Gamma'(\mathtt{X}) = \Gamma(\mathtt{X})$
- $\Gamma(\mathtt{C}) = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_n\ \{\ fDecls,\ mDecls\ \}}$
  $\Gamma'(\mathtt{C}) = \mathtt{C\ ext\ C'\ impl\ I'_1, ...I'_k\ \{\ fDecls,\ mDecls\ \}}$
  $\forall \mathtt{i} \in \{\mathtt{1...n}\}\ :\ \exists \mathtt{j} \in \{\mathtt{1...k}\}\ :\ \Gamma' \vdash \mathtt{I'_j} \leq_{wdn} \mathtt{I_i}$
  $\forall \mathtt{X} \neq \mathtt{C}\ :\ \Gamma'(\mathtt{X}) = \Gamma(\mathtt{X})$
- $\Gamma(\mathtt{C}) = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_m\ \{v_1 : T_1, ...v_n : T_n,\ mDecls\}}$
  $\Gamma'(\mathtt{C}) = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_m\ \{v_1 : T_1, ...v_n : T_n, v_{n+1} : T_{n+1},\ mDecls\}}$
  $\forall \mathtt{X} \neq \mathtt{C}\ :\ \Gamma'(\mathtt{X}) = \Gamma(\mathtt{X})$
- $\Gamma(\mathtt{C}) = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_m\ \{fDecls,\ m_1 : MT_1, ...\ m_n : MT_n\}}$
  $\Gamma'(\mathtt{C}) = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_m\ \{fDecls,\ m_1 : MT_1, ...\ m_n : MT_n, m_{n+1} : MT_{n+1}\}}$
  $\forall \mathtt{X} \neq \mathtt{C}\ :\ \Gamma'(\mathtt{X}) = \Gamma(\mathtt{X})$
- $\Gamma = \Gamma'$

The following lemma says that safe binary compatible changes on an environment and local changes to method bodies, that type check in the new environment are binary compatible with the old environment program pair.

Lemma 4 *Given environments* Γ, Γ', *if* Γ' *is the result of the application of a sequence of safe binary compatible changes to* Γ, *then* Γ' *is weakly environment compatible with* Γ.

It can be shown that each kind of safe binary compatible change preserves weak environment compatibility. Since weak environment compatibility is reflexive and transitive, a sequence of safe binary compatible changes will preserve weak environment compatibility.

Thus these safe binary compatible changes can be used as a conservative approximation to link compatibility, and by definition 2 can be used by programmers to determine whether source code they produce is link compatible with the existing program, and so determine if recompilation of the entire program can be avoided.

5 The Concept of Link Compatibility

Central to our paper is the concept of link compatibility, a term which does not appear as such in the language description, but which aims to reflect the purpose of binary compatibility. In our definition, a new environment fragment, Java_s program fragment pair is link compatible with an original environment and Java_{se} program. Therefore, the context of the modification (*i.e.* features of the entire original environment) may be relevant.

This contrasts with the approach in [7], which considers a relationship between the fragment of the program that is replaced and its replacement, without reference to the context of the change (*i.e.* features of the whole program); it requires that *any* programs that linked with the original fragment should link with the modified fragment. This could be expressed by the following definition:

Definition 5 *Consider environment fragments* Γ' *and* Γ'', *and* Java_s *program fragments* $\mathtt{p}'$ *and* $\mathtt{p}''$, *such that* $\mathit{Classes}(\Gamma') = \mathit{Classes}(\mathtt{p}')$ *and* $\mathit{Classes}(\Gamma'') = \mathit{Classes}(\mathtt{p}'')$.
The pair $(\Gamma'', \mathtt{p}'')$ *is* strongly link compatible *with* $(\Gamma', \mathtt{p}')$, *iff for all environments* Γ *and* Java_{se} *programs* $\mathtt{p}$:

$$(\Gamma', \mathtt{p}') \textit{ link compatible with } (\Gamma, \mathtt{p}) \quad \Longrightarrow \quad (\Gamma'', \mathtt{p}'') \textit{ link compatible with } (\Gamma, \mathtt{p})$$

It is trivial to show that strong link compatibility is a partial order.

An interesting research direction would follow the route of strong link compatibility. In particular, some of the Java binary compatible changes preserve strong link compatibility (*e.g.* addition of instance variables into classes), whereas others don't because they make explicit mention of the remaining program (*e.g.* changing the direct super-class of a class, provided that all direct or indirect super-classes continue to be direct or indirect super-classes).

6 Conclusions and Further Work

The contribution of this paper is, we believe, twofold

- We suggest a terminology and formal framework with which to describe the effects and properties of binary compatibility.
- We define safe binary compatibility, a restricted form of that defined by the language specification, and prove for a substantial subset of Java, that safe binary compatible changes do not require re-compilation and guarantee successful linking.

We expect that better formalizations will be found, but feel that our approach is adequate, as it allows us to argue that successful linking can be guaranteed without the re-compilation of whole programs.

Our work makes use of the formal description and the soundness results in [6]. However, this is purely a matter of convenience. It could be based on any other formal description of the Java semantics which addresses the following features: type checking, compilation into a rich enough form to describe execution of method calls, instance variable access, array and object creation, type checking the intermediate form, operational semantics and a type soundness theorem. Alternatively, it might be possible to recast some of the work in terms of a formal description of the Java bytecode and bytecode verifier (such as [10]).

We intend to extend Java$_s$ to encompass a larger subset of Java, and then extend safe binary compatibility to include access restrictions, static variables and methods *etc.* A more direct requirement is the description of constructors in a way that reflects the language semantics *and* would allow the preservation of link compatibility.

Other further work includes refining the description of separate compilation to consider compilation in partial environments, rather than in the environment for the whole program. For example, in the computing students example the classes do not have to be compiled in the complete environment:

$$\begin{array}{lll} \mathtt{p}^{se}_{\mathtt{st}} & = \mathcal{C}\{\!|\mathtt{p}_{\mathtt{st}}, \Gamma|\!\} & = \mathcal{C}\{\!|\mathtt{p}_{\mathtt{st}}, \Gamma_{\mathtt{st}}|\!\} \\ \mathtt{p}^{se}_{\mathtt{cst}} & = \mathcal{C}\{\!|\mathtt{p}_{\mathtt{cst}}, \Gamma|\!\} & = \mathcal{C}\{\!|\mathtt{p}_{\mathtt{cst}}, \Gamma_{\mathtt{st}}\,\Gamma_{\mathtt{cst}}|\!\} \\ \mathtt{p}^{se}_{\mathtt{c100}} & = \mathcal{C}\{\!|\mathtt{p}_{\mathtt{c100}}, \Gamma|\!\} & = \mathcal{C}\{\!|\mathtt{p}_{\mathtt{c100}}, \Gamma_{\mathtt{st}}\,\Gamma_{\mathtt{cst}}\,\Gamma_{\mathtt{c100}}|\!\} \end{array}$$

It is necessary to consider compilation in partial environments in order to realistically describe libraries distributed in binary form, since the library author does not have access to the environments for the programs in which the library may be used. Also, we have so far only considered a linear sequence of changes to a program, but in cases with libraries, changes may occur to the library in parallel with changes to the program using the library, culminating in a merge of the most recent binaries for the library and the most recent binaries for the main program [11].

Another interesting line of investigation would be the application of our formalism to the approach described in [7], with its stronger requirement for

binary compatible changes, as already outlined by the strong link compatibility definition in section 5.

Finally, a more distant and ambitious task remains the formalization of the dynamic linker/loader, and an approach to the associated security issues.

7 Acknowledgements

We are grateful to to Guy Steele for feedback on the concept of binary compatibility, and to Gabrielle Sinnadurai and David von Oheimb for suggestions on the presentation.

References

[1] L. Cardelli. Program Fragments, Linking, and Modularization. In *POPL'97 Proceedings*, January 1997.

[2] M. Dausmann, S. Drossopoulou, G. Persch, and G. Winterstein. A Separate Compilation System for Ada. In *Proc. GI Tagung: Werkzeuge der Programmiertechnik*. Springer Verlag Lecture Notes in Computer Science, 1981.

[3] Drew Dean. The Security of Static Typing with Dynamic Linking. In *Fourth ACM Conference on Computer and Communication Security*, 1997. Revised version Tech Report number SRI CSL 9704.

[4] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From Hotjava to Netscape and beyond. In *Security and Privacy'96 Proceedings*, May 1996.

[5] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[6] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is Java Sound? *Theory and Practice of Object Systems*, 1998. to appear, available at http://www-dse.doc.ic.ac.uk/projects/slurp/.

[7] Ira Forman, Michael Conner, Scott Danforth, and Larry Raper. Release-to-Release Binary Compatibility in SOM. In *OOPSLA'95 Proceedings*, 1995.

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

[9] Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe — definitely. In *POPL'98 Proceedings*, January 1998.

[10] Raymie Stata and Martin Abadi. A Type System For Java Bytecode Subroutines. In *POPL'98 Proceedings*, January 1998.

[11] Guy Steele. Private Conversation, January 1998.

[12] Donald Syme. Proving Java Type Sound. Technical Report 427, Cambridge University, June 1997. to appear in Formal Syntax and Semantics of Java^{tm}, edited by Jim Alves Foss, Springer, LNCS.

[13] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.

Appendix

A1 The Syntax of Java$_s$

Program	::=	(*ClassBody*)*
ClassBody	::=	`ClassId ext ClassName {` (*MethBody*)* `}`
MethBody	::=	`MethId is` (λ `ParId :` *VarType*.)*
		`{`*Stmts* `; return` [*Expr*] `}`
Stmts	::=	ϵ | *Stmts* `;` *Stmt*
Stmt	::=	`if` *Expr* `then` *Stmts* `else` *Stmts*
	|	*Var* `=` *Expr* | *Expr* | `throw` *Expr*
	|	`try` *Stmts* (`catch ClassName Id` *Stmts*)* `finally` *Stmts*
	|	`try` *Stmts* (`catch ClassName Id` *Stmts*)$^+$
Expr	::=	*Value* | *Var* |
		Expr.`MethName (` *Expr**) (`[` *Expr* `]`)$^+$(`[ ]`)*
Var	::=	`Name` | *Var*.`VarName` | *Var*`[`*Expr*`]` | `this`
Value	::=	*PrimValue* | `null`
PrimValue	::=	*intValue* | *charValue* | *byteValue* | ...
VarType	::=	*SimpleType* | *ArrayType*
SimpleType	::=	*PrimType* | `ClassName` | `InterfaceName`
ArrayType	::=	*SimpleType*`[ ]` | *ArrayType*`[ ]`
	|	`InterfaceName`
PrimType	::=	`bool` | `char` | `int` | ...
Type	::=	*VarType* | `void` | `nil`
Env	::=	*StandardEnv* | *Env* `;` *Decl*
StandardEnv	::=	`Exception ext Object`...`NullPE ext Exception`...`;` ...
Decl	::=	`ClassId ext ClassName impl` (`InterfName`)*
		`{`(`VarId` : *VarType*)* (`MethId` : *MethType*)*`}`
	|	`InterfId ext InterfName`*{(`MethId` : *MethType*)*`}`
	|	`VarId` : *VarType*
MethType	::=	*ArgType* $\rightarrow$ (*VarType* | `void`)
ArgType	::=	[*VarType* ($\times VarType$)*]

A2 Some of the Java$_s$ Type Checking Rules

$\mathtt{i}$ is integer, $\mathtt{c}$ is character, $\mathtt{x}$ is identifier

---

$\Gamma \vdash \mathtt{null} : \mathtt{nil}$, $\Gamma \vdash \mathtt{true} : \mathtt{bool}$, $\Gamma \vdash \mathtt{false} : \mathtt{bool}$,
$\Gamma \vdash \mathtt{i} : \mathtt{int}$, $\Gamma \vdash \mathtt{c} : \mathtt{char}$, $\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})$
$\mathcal{C}\{\!|\mathtt{z}, \Gamma|\!\} = \mathtt{z}$ if $\mathtt{z}$ is integer, character, identifier, `null`, `true`, or `false`

$$\frac{\Gamma \vdash \mathtt{v} : \mathtt{T} \quad \Gamma \vdash \mathtt{e} : \mathtt{T}' \quad \Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}}{\Gamma \vdash \mathtt{v} := \mathtt{e} : \mathtt{void} \quad \mathcal{C}\{\!|\mathtt{v} := \mathtt{e}, \Gamma|\!\} = \mathcal{C}\{\!|\mathtt{v}, \Gamma|\!\} := \mathcal{C}\{\!|\mathtt{e}, \Gamma|\!\}}$$

$$\frac{}{\Gamma \vdash \mathtt{return} : \mathtt{void} \quad \mathcal{C}\{\!|\mathtt{return}, \Gamma|\!\} = \mathtt{return}}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{bool} \quad \Gamma \vdash \mathtt{stmts} : \mathtt{void} \quad \Gamma \vdash \mathtt{stmt} : \mathtt{T} \quad \Gamma \vdash \mathtt{stmts}' : \mathtt{T}'}{\begin{array}{l}\Gamma \vdash \mathtt{stmts\ ;\ stmt} : \mathtt{T} \\ \mathcal{C}\{\!|\mathtt{stmts\ ;\ stmt}, \Gamma|\!\} = \mathcal{C}\{\!|\mathtt{stmts}, \Gamma|\!\}\ ;\ \mathcal{C}\{\!|\mathtt{stmt}, \Gamma|\!\} \\ \Gamma \vdash\ \mathtt{if\ e\ then\ stmts\ else\ stmts}' : \mathtt{void} \\ \mathcal{C}\{\!|\ \mathtt{if\ e\ then\ stmts\ else\ stmts}', \Gamma|\!\} = \\ \quad \mathtt{if}\ \mathcal{C}\{\!|\mathtt{e}, \Gamma|\!\}\ \mathtt{then}\ \mathcal{C}\{\!|\mathtt{stmts}, \Gamma|\!\}\ \mathtt{else}\ \mathcal{C}\{\!|\mathtt{stmts}', \Gamma|\!\}\end{array}}$$

$$\frac{\Gamma \vdash \mathtt{v} : \mathtt{T}[] \quad \Gamma \vdash \mathtt{e} : \mathtt{int}}{\Gamma \vdash \mathtt{v[e]} : \mathtt{T} \quad \mathcal{C}\{\!|\mathtt{v[e]}, \Gamma|\!\} = \mathcal{C}\{\!|\mathtt{v}, \Gamma|\!\}[\mathcal{C}\{\!|\mathtt{e}, \Gamma|\!\}]}$$

$$\frac{\Gamma \vdash \mathtt{e_i} : \mathtt{T_i}\ \ \mathtt{i} \in \{1...\mathtt{n}\}, \mathtt{n} \geq 1 \quad \mathit{MostSpec}(\Gamma, \mathtt{m}, \mathtt{T_1}, \mathtt{T_2} \times ... \times \mathtt{T_n}) = \{(\mathtt{T}, \mathtt{MT})\}}{\begin{array}{l}\Gamma \vdash \mathtt{e_1.m(e_2...e_n)} : \mathit{Res}(\mathtt{MT}) \\ \mathcal{C}\{\!|\mathtt{e_1.m(e_2...e_n)}, \Gamma|\!\} = \\ \quad \mathcal{C}\{\!|\mathtt{e_1}, \Gamma|\!\}.[\mathit{Args}(\mathtt{MT})]\mathtt{m}(\mathcal{C}\{\!|\mathtt{e_2}, \Gamma|\!\}...\mathcal{C}\{\!|\mathtt{e_n}, \Gamma|\!\})\end{array}}$$

$$\frac{\Gamma \vdash \mathtt{v} : \mathtt{T} \quad \mathit{FDec}(\Gamma, \mathtt{T}, \mathtt{f}) = (\mathtt{C}, \mathtt{T}')}{\Gamma \vdash \mathtt{v.f} : \mathtt{T}' \quad \mathcal{C}\{\!|\mathtt{v.f}, \Gamma|\!\} = \mathcal{C}\{\!|\mathtt{v}, \Gamma|\!\}.[\mathtt{C}]\mathtt{f}}$$

$$\frac{\begin{array}{l}\mathtt{mBody} = \mathtt{m\ is}\ \lambda \mathtt{x_1} : \mathtt{T_1}...\lambda \mathtt{x_n} : \mathtt{T_n}.\{\mathtt{stmts}\} \\ \mathtt{x_i} \neq \mathtt{this} \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \mathtt{z_1}, ..., \mathtt{z_n} \text{ are new variables in } \Gamma \\ \mathtt{stmts}' = \mathtt{stmts}[\mathtt{z_1}/\mathtt{x_1}, ..., \mathtt{z_n}/\mathtt{x_n}] \\ \Gamma, \mathtt{z_1} : \mathtt{T_1}...\mathtt{z_n} : \mathtt{T_n} \vdash \mathtt{stmts}' : \mathtt{T}' \\ \Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{mBody} : \mathtt{T_1} \times ... \times \mathtt{T_n} \to \mathtt{T} \\ \mathcal{C}\{\!|\mathtt{mBody}, \Gamma|\!\} = \\ \quad \mathtt{m\ is}\ \lambda \mathtt{x_1} : \mathtt{T_1}...\lambda \mathtt{x_n} : \mathtt{T_n}.\{\mathcal{C}\{\!|\mathtt{stmts}, \Gamma|\!\}\}\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0, \mathtt{k} \geq 0, \mathtt{m} \geq 0, \Gamma \vdash \Gamma\ \diamond \\ \Gamma(\mathtt{C}) = \mathtt{C\ ext\ C'\ \ impl\ I_1 ... I_n} \\ \quad \{\mathtt{v_1 : T_1 ... v_k : T_k, m_1 : MT_1 ... m_l : MT_l}\} \\ \mathtt{cBody = C\ ext\ C'\ \{mBody_1, ... mBody_l\}} \\ \Gamma(\mathtt{this}) = \mathtt{Undef} \\ \mathtt{mBody_i = m_i\ is\ mPrsSts_i} \quad \mathtt{i} \in \{1...l\} \\ \Gamma, \mathtt{this : C} \vdash \mathtt{mBody_i : MT_i} \quad \mathtt{i} \in \{1...l\}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{cBody} : \Gamma(\mathtt{C}) \\ \mathcal{C}\{\!|\mathtt{cBody}, \Gamma|\!\} = \mathtt{C\ \ ext\ C'}\ \{\mathcal{C}\{\!|\mathtt{mBody_1}, \Gamma|\!\} ... \mathcal{C}\{\!|\mathtt{mBody_l}, \Gamma|\!\}\}\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0 \\ \mathtt{p = cBody_1, ... cBody_n} \\ \mathtt{cBody_i = C_i\ ext\ ...\{...\}} \quad \text{for } \mathtt{i} \in \{1...n\} \\ \Gamma \vdash \mathtt{cBody_i} : \Gamma(\mathtt{C_i}) \quad \mathtt{i} \in \{1...n\}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{p}\ \diamond \\ \mathcal{C}\{\!|\mathtt{p}, \Gamma|\!\} = \mathcal{C}\{\!|\mathtt{cBody_1}, \Gamma, \mathtt{this : C}|\!\} ... \mathcal{C}\{\!|\mathtt{cBody_n}, \Gamma, \mathtt{this : C}|\!\}\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{p = p_1 p_2} \implies \\ \quad \mathit{Classes}(\mathtt{p_1}) \cap \mathit{Classes}(\mathtt{p_2}) = \emptyset \\ \mathit{Classes}(\Gamma) = \mathit{Classes}(\mathtt{p}) \\ \Gamma \vdash \mathtt{p}\ \diamond\end{array}}{\Gamma \vdash \mathtt{p}\ \diamond\!\!\diamond}$$

A3 Altering the Syntax of Java$_s$ to Obtain Java$_{se}$ Syntax

Type	::=	... | `ClassName-Thrn`	
Expr	::=	...	
	|	*Expr*.[*ArgType*]`MethName`(*Expr**)	*replaces Expr*.`MethName`(*Expr**)
	|	*Stmts*	
Var	::=	...	
	|	*Var*.[`ClassName`]`VarName`	*replaces Var*.`VarName`
	|	$\iota_{\mathtt{i}}$.[`ClassName`]`VarName` | $\iota_{\mathtt{i}}$[*Expr*]	`i` *an integer*
	|	`null`.[`ClassName`]`VarName`	
	|	`null`[*Expr*]	
Value	::=	... | *RefValue*	
RefValue	::=	$\iota_{\mathtt{i}}$	`i` *an integer*

A4 Some Java$_{se}$ Types

$$\frac{\sigma(\iota_{\mathtt{i}}) = \ll ... \gg^{\mathtt{C}}}{\Gamma, \sigma \vdash_{se} \iota_{\mathtt{i}} : \mathtt{C}} \qquad \frac{\sigma(\iota_{\mathtt{i}}) = [\![...]\!]^{\mathtt{T}[]_1 ... []_n}}{\Gamma, \sigma \vdash_{se} \iota_{\mathtt{i}} : \mathtt{T}[]_1 ... []_n} \qquad \frac{\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{Object}}{\Gamma, \sigma \vdash_{se} \mathtt{null} : \mathtt{T}}$$

$$\frac{\begin{array}{l}\Gamma, \sigma \vdash_{se} \mathtt{v} : \mathtt{T} \\ \Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{C} \\ \mathit{FDec}(\Gamma, \mathtt{C}, \mathtt{f}) = (\ \mathtt{C}, \mathtt{T'})\end{array}}{\Gamma, \sigma \vdash_{se} \mathtt{v}.[\mathtt{C}]\mathtt{f} : \mathtt{T'}} \qquad \frac{\begin{array}{l}\Gamma, \sigma \vdash_{se} \mathtt{e_i} : \mathtt{T'_i} \quad \mathtt{i} \in \{1...n\}, \mathtt{n} \geq 0 \\ \Gamma \vdash \mathtt{T'_i} \leq_{wdn} \mathtt{T_i} \quad \mathtt{i} \in \{2...n\} \\ \mathit{FirstFit}(\Gamma, \mathtt{m}, \mathtt{T'_1}, \mathtt{T_2} \times ... \times \mathtt{T_n}) = \{(\mathtt{T}, \mathtt{MT})\}\end{array}}{\Gamma, \sigma \vdash_{se} \mathtt{e_1}.[\mathtt{T_2} \times ... \times \mathtt{T_n}\ ]\mathtt{m}(\mathtt{e_2 ... e_n}) : \mathit{Res}(\mathtt{MT})}$$