What *is* Java Binary Compatibility?

Sophia Drossopoulou, David Wragg, Susan Eisenbach
Department of Computing
Imperial College
sd{dpw,se}@doc.ac.ac.uk

Abstract

Separate compilation allows the decomposition of programs into units that may be compiled separately, and linked into an executable. Traditionally, separate compilation was equivalent to the compilation of all units together, and modification and re-compilation of one unit required re-compilation of all importing units.

Java suggests a more flexible framework, in which the linker checks the integrity of the binaries to be combined. Certain source code modifications, such as addition of methods to classes, are defined as *binary compatible.* The language description guarantees that binaries of types (*i.e.* classes or interfaces) modified in binary compatible ways may be re-compiled and linked with the binaries of types that imported and were compiled using the earlier versions of the modified types.

However, this is not always the case: some of the changes considered by Java as binary compatible do *not* guarantee successful linking and execution. In this paper we study the concepts around binary compatibility. We suggest a formalization of the requirement of safe linking and execution without re-compilation, investigate alternatives, demonstrate several of its properties, and propose a more restricted definition of binary compatible changes. Finally, we prove for a substantial subset of Java, that this restricted definition guarantees error-free linking and execution.

1 Introduction

Module systems [19, 18], introduced in the seventies, support the decomposition of large programs into small, more manageable units (modules, classes, clusters, packages). Traditionally, separate compilation [3] allowed these units to be compiled one at a time using only the signature (*i.e.* type) information from imported units. The object code of such separately compiled units would be combined by a linker into an executable. If each unit were compiled after any unit it imported, each unit compiled successfully, and all units were present, then linking would be successful. The compiler had to check that units respected imported units' signatures, whereas the linker had to reconcile external references, and to check the order of compilation, typically using time stamps in the object code. Therefore, separate compilation was equivalent to the compilation of all units together.

Because of the intended support for loading and executing remotely produced code, Java has a different approach to separate compilation and linking. As before, classes may be compiled separately – even on different machines, and the compiler has to check that units respect imported units' signatures. Also, if each unit compiles successfully, and it is compiled *after* any unit it imported, then linking will be successful. However, the remit of the linker has been extended: Not only does it have to resolve external references, it also has to ensure that binaries are structurally correct (verification), and that they respect the types of entities they import from other binaries (resolution).

In the traditional approach, when the signature of a unit is modified and re-compiled, all importing units have to be re-compiled as well. In Java however, re-compilation of importing units cannot always be enforced. It is the task of the linker to ensure that the binaries respect each others' exported signatures, independently of the order of compilation. Certain source code modifications, such as adding a method to a class, are defined as *binary compatible* [8]. The Java language description does not require the re-compilation of units importing units which were modified in binary compatible ways, and 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 such re-compilations *may not be possible*: a binary compatible change to the source code for one class may cause the source code of other classes no longer to be type correct. Yet the guarantee of successful linking and execution still holds since only the binaries are consulted during these steps. In particular, it is possible to link successfully and execute binaries corresponding to type-incorrect source code. Separate compilation is no longer equivalent to compilation of all units together.

This is a deliberate feature and constitutes a crucial ingredient of the Java approach [11]. It allows the modification (usually through extension) of libraries, without requiring re-compilation of software using these libraries.

Binary compatibility is a powerful but immature language feature; although supported in previous forms by some language implementations, Java is the first case we know of where it is explicitly described in the language definition. We feel that its exact meaning and properties are not fully understood. This is unfortunate, since [5, 4] demonstrate that loopholes in the definition and implementation of binary compatibility provide opportunities to break Java security.

The Java language specification [10] devotes a whole chapter to binary compatibility, giving examples, and pointing out possible interplay of features. However, it does not give an exact definition, and uses the term *binary compatibility* in two senses. It lists the changes considered to be binary compatible, *e.g.* on p.237:

> "*...a list of some important binary compatible changes that Java supports: re-implementing existing methods, ..., adding new fields to an existing class or interface, ..., adding a class, ...*"

and describes the guarantee of such changes, p.240:

> "*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.* "

So, from the Java description we have

modifications		*guarantee*
list of binary compatible changes	$\Longrightarrow$	no re-compilation, linking without errors, safe execution

There is no appropriate precedent for a terminology in this area: Corresponding to the guarantee we define *link compatible changes* as source code modifications for which all types (*i.e.* classes and interfaces) that successfully linked with the original binaries will also successfully link with the binaries obtained after modification and re-compilation. *Safe changes* are those changes that can be proven to preserve the guarantee; they include most changes listed in [10] *e.g.* adding instance variables to classes, modifying method bodies. They do *not* include the addition of methods to interfaces, because, as we shall see, this does not preserve the property of linking without errors:

modifications		*guarantee*
list of binary compatible changes	$\Longrightarrow$	no re-compilation, linking without errors, safe execution
formalized as $\downarrow$		*formalized as* $\downarrow$
list of safe changes	$\Longrightarrow$	link compatibility

Based on the above formalization we were able to distinguish nuances in the concept of binary compatibility, and to formulate and prove composability properties:

- The definition of link compatibility allows application of the term to binaries that are not standalone. This is a common situation for libraries importing further libraries.
- We argue that the exact definition of link compatibility should cater for the possibility of linking with further, yet unknown binaries, *i.e.* it should say: "A change is binary compatible with pre-existing binaries if *any further pre-existing binaries* that link without error with the former pre-existing binaries continue to do so after the change to the former pre-existing binaries."
- We show that applying a sequence of link compatible changes to a binary preserves all the linking capabilities of the original binary.
- We show that link compatible changes applied to different, but possibly mutually dependent binaries, preserve all the linking capabilities of the original program consisting of the original binaries. This caters for the case where programmers develop different interdependent libraries, and says that binary compatible changes do not alter the linking capabilities of the overall system.
- We demonstrate that two consecutive link compatible changes usually cannot be folded into one; and that two different link compatible changes applied to the same binary usually cannot be reconciled.

We build on some of our previous work formalizing the semantics of Java [6, 7], but we could have used any formalization that gives meaning to type checking and distinguishes source code from compiled code, *e.g.* [17].

The remainder of this paper is organized as follows: In section 2 we examine the motivation and some subtleties of binary compatibility, and demonstrate these in terms of examples. In section 3 we summarize the formalization from [7] needed for the current discussion. In section 4 we formalize compilation and linking of fragments. In sections 5-6 we define link compatibility, prove its composition properties, define safe changes and prove that they are link compatible. In appendix A we justify our approach and discuss alternatives. Finally, in section 7 we draw conclusions and outline further work.

2 Binary compatibility in Java

The motivation for the concept of binary compatibility in Java is the intention to support large scale re-use of software available on the Internet [11].

In particular, Java avoids the *fragile base class problem*, found, in most C++ implementations, where an instance variable (data member) access is compiled into an offset from the beginning of the object, fixed at compile-time. If new instance variables are added and the class is re-compiled, then offsets may change, and object code previously compiled using the original definition of the class may not execute safely together with the object code of the modified class. Similar problems arise with virtual function calls. The term "fragile base class problem" is also used in a wider sense, to describe the problems arising in separately developed systems using inheritance for code re-use [13].

C++ development environments usually attempt to compensate by automatically re-compiling all files importing the modified class. Although Java development environments do the same, there are realistic cases where this strategy would be too restrictive. For instance, if one developed a local program `P`, which imported a library `L1`, the source for `L1` was not available, `L1` imported library `L2`, and `L2` was modified, then re-compilation of `L1` would not be possible. Any further development of `P` would therefore be impossible.

In contrast, Java promises that if the modification to `L2` were binary compatible, then the binaries of the modified `L2`, the original `L1` and the current `P` can be linked without error. This is possible, because Java binaries carry more type information than object code usually does.

Interestingly, it is possible to modify types in binary *incompatible* ways, and to still be able to link without errors with the binaries of some importing types. Still, other binaries will exist, which linked without errors with the type, but no longer link without errors with the binary of the modified type.

1st phase

```
class Student {  int grade;  }
class CStudent extends Student {  }
class Lab {
   CStudent guy;
   void f(){ guy.grade=100; }
    }
```

2nd phase

```
class CStudent extends Student {
   char grade;
   }
```

3rd phase

```
class Marker {
   CStudent guy;
   void g(){ guy.grade='A'; }
   }
```

Figure 1: Students and computing students - code

2.1 An example

The example from figure 1 demonstrates some of the issues connected with binary compatibility. It consists of three phases.

In the first phase we create the classes `Student`, `CStudent`, and `Lab`. 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 `Lab`, the field `guy`, of class `CStudent`, is assigned the grade `1`. This program is well-formed, and can be compiled, producing three binary files `Student.class`, `CStudent.class` and `Lab.class`. In the second phase we add the field `grade` of type `char` to class `CStudent`, and re-compile `CStudent`, producing `CStudent′.class`. In the third phase we define a new class, `Marker`. In the body of its method `g()`, we assign the grade `'A'` to `guy`. The class `Marker` is type correct, and thus it can be compiled to produce the file `Marker.class`.

The two changes, *i.e.* the addition of field `grade` in class `CStudent`, and the creation of class `Marker`, are binary compatible changes. So, the corresponding binaries, *i.e.* `Student.class`, `CStudent′.class`, `Lab.class` and `Marker.class`, can safely be linked together.

The sources are *not* type correct any more. An attempt to re-compile the class `Lab` would flag a type error for the assignment `guy.grade=100`, since the expression `guy.grade` now refers to the field in class `CStudent` which is of type `char`. Also, the compiled form of the expression `guy.grade` in the binary `Lab.class` refers to an integer, whereas the compiled form of the same

```
1st phase
 interface I {
    void meth1();
    }
 class C implements I { void meth1(){... } }
 class D {
    void meth3() { I anI = new C(); }
    }
2nd phase
 interface I {
    void meth1();
    void meth2();
     }
3rd phase
 class D {
    void meth3()
       { I anI = new C(); anI.meth2(); }
    }
```

Figure 2: Adding a method to an interface

expression in the binary `Marker.class` refers to a character. The two compiled forms exist at the same time, and refer to different fields of a `CStudent` object. An implementation of Java has to reflect this in the code produced; in our formalization in section 3 we describe this in terms of different Java$_{se}$ intermediate code. Similar situations can arise for method calls.

2.2 A problem with binary compatibility

The example in figure 2 demonstrates that the list of binary compatible changes given in [10] is too permissive and so fails to fulfil the guarantee. 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 [16].

In the first phase consider compiling interface `I`, and classes `C`, `D`. Compilation will be successful. In the second phase method `meth2()` is added to interface `I`, and `I` is re-compiled. This is listed as a binary compatible change [10]. In the third phase, code invoking `anI.meth2()` is added to the body of `meth3` in class `D` and then `D` is re-compiled. Since the new method body is type correct, this is a binary compatible change as well, [10]. According to the guarantee of binary compatibility, the binaries for `I`′, `C` and `D`′ should link and run successfully. But they cannot, as there is no implementation of `meth2()`.

Thus, although addition of methods to interfaces is listed as a binary compatible change in [10], it does not uphold the promise of safe linking and execution.

1st phase

$$\begin{array}{lcl}
\Gamma^{\mathtt{st}} & = & \mathtt{Student\,ext\,Object} \\
 & & \quad \{\ \mathtt{grade} : \mathtt{int}\ \} \\
\Gamma^{\mathtt{cs}} & = & \mathtt{CStudent\ ext\ Student}\ \{\ \} \\
\Gamma^{\mathtt{lab}} & = & \mathtt{Lab\ ext\ Object} \\
 & & \quad \{\ \mathtt{guy} : \mathtt{CStudent},\ \mathtt{f} :\rightarrow \mathtt{void}\ \}
\end{array}$$

2nd phase

$$\begin{array}{lcl}
\Gamma^{\mathtt{cs}'} & = & \mathtt{Student\ ext\ Student} \\
 & & \quad \{\ \mathtt{grade} : \mathtt{char}\ \}
\end{array}$$

3rd phase

$$\begin{array}{lcl}
\Gamma^{\mathtt{m}} & = & \mathtt{Marker\ ext\ Object} \\
 & & \quad \{\ \mathtt{guy} : \mathtt{CStudent},\ \mathtt{g} :\rightarrow \mathtt{void}\ \}
\end{array}$$

Figure 3: Environment for computing students

3 Formalization of the Java semantics

This section summarizes material from [7] needed for the formalization of separate compilation and binary compatibility. In [7] we describe the semantics of a substantial subset of Java encompassing primitive types, classes, interfaces, inheritance, fields, methods, interfaces, shadowing, dynamic method binding, the value `null`, arrays, exceptions and exception handling. We distinguish between three languages: Java$_s$ is our subset of Java, Java$_{se}$ is an enriched version of Java$_s$ containing compile-time information necessary for execution, Java$_r$ is an extension of Java$_{se}$ supporting run-time constructs such as addresses.

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

We give type systems for Java$_s$, Java$_{se}$ and Java$_r$. The two latter are slight modifications of the former. We prove that a well-typed Java$_s$ term retains its type when transformed to the corresponding Java$_{se}$ or Java$_r$ term. The operational semantics, $\rightsquigarrow_{\mathtt{p}}$, describes the execution of Java$_r$ terms for a particular Java$_{se}$ program `p`. We prove a subject reduction theorem, stating that execution of Java$_r$ terms preserves types up to subclasses/subinterfaces. In the remainder of this section we discuss these concepts in more depth.

A Java$_s$ program consists of an environment, usually denoted by a Γ, and Java$_s$ body, usually denoted by a `p`. The syntax of environments can be found in appendix B, that of Java$_s$ bodies can be found in appendix C. The first phase of the computing students example corresponds to environment $\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}}$, as given in figure 3, and body $\mathtt{p}^{\mathtt{st}}\,\mathtt{p}^{\mathtt{cs}}\,\mathtt{p}^{\mathtt{lab}}$, as given in figure 4.

The order of declarations and definitions is not significant, therefore $\Gamma\,\Gamma' = \Gamma'\,\Gamma$, and $\mathtt{p}\,\mathtt{p}' = \mathtt{p}'\,\mathtt{p}$. The sets $\mathcal{C}l(\Gamma)$, $\mathcal{C}l(\mathtt{p})$, $\mathcal{I}t(\Gamma)$, and $\mathcal{V}r(\Gamma)$ contain the names of all classes, interfaces or variables declared in environment Γ or program p respectively. The set $\mathcal{D}(\_)$ is the union of the previous sets. For example, $\mathcal{D}(\mathtt{p^{cs}\,p^{lab}}) = \mathcal{D}(\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}}) = \{\mathtt{CStudent}, \mathtt{Lab}\}$.

The assertion $\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$ indicates that in environment Γ, type T *widens to* type $\mathtt{T}'$, *i.e.* values of type T can be assigned to variables of type $\mathtt{T}'$ without any run-time checks.

1st phase		
$\mathtt{p^{st}}$	=	`Student ext Object { }`
$\mathtt{p^{cs}}$	=	`CStudent ext Student { }`
$\mathtt{p^{lab}}$	=	`Lab ext Object { f is{ guy.grade = 100; } }`
2nd phase		
$\mathtt{p^{cs'}}$	=	`CStudent ext Student { }` = $\mathtt{p^{cs}}$
3rd phase		
$\mathtt{p^{m}}$	=	`Marker ext Object { g is{ guy.grade = 'A'; } }`

Figure 4: Java$_s$ class bodies for computing students

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 fields are unique in a class, *etc.* Provided that $\Gamma \vdash \diamond$, Java$_s$ terms can be type checked in terms of a type inference system, part of which appears in appendix D. The assertion $\Gamma \vdash \mathtt{t} : \mathtt{T}$ signifies that term t has type T for environment Γ; the assertion $\Gamma \vdash \mathtt{p} \diamond$ signifies that program body p is well-typed in environment Γ, *i.e.* the class bodies contain type correct function bodies which return values of the expected types. The assertion $\Gamma \vdash \mathtt{p} \otimes$ signifies that p is complete, *i.e.* that it is well-typed and contains a class body for each class in Γ.

To support execution of method calls and field access, Java$_s$ is enriched with type information. The enriched language is called Java$_{se}$; enriching is performed by the mapping $\mathcal{C}$, which can be understood as an abstraction of compilation from Java source code to binary code. Only type correct terms are mapped, *i.e.* $\mathcal{C}\{\!\{\Gamma, \mathtt{t}\}\!\}$ is defined only iff there exists a type T with $\Gamma \vdash \mathtt{t} : \mathtt{T}$. Furthermore, if $\Gamma \vdash \mathtt{t} : \mathtt{T}$, and $\Gamma\,\Gamma' \vdash \diamond$ (*i.e.* Γ' does not "affect" Γ), then $\Gamma\,\Gamma' \vdash \mathtt{t} : \mathtt{T}$ and $\mathcal{C}\{\!\{\Gamma, \mathtt{t}\}\!\} = \mathcal{C}\{\!\{\Gamma\,\Gamma', \mathtt{t}\}\!\}$. The syntax of Java$_{se}$ is an extension of the Java$_s$ syntax and is given in appendix E.

The Java$_{se}$ version of the students class bodies is given in figure 5. In $\mathtt{p}^{\mathtt{lab}}_{se}$ the field access `guy.grade` has been enriched by the class from which `grade` is inherited, and is compiled to `guy[Student].grade`, whereas in $\mathtt{p}^{\mathtt{m}}_{se}$ it is compiled to `guy[CStudent].grade`.

Java$_{se}$ terms also have types, indicated by assertions $\Gamma \vdash_{se} \mathtt{t} : \mathtt{T}$. For a Java$_{se}$ program body p, $\Gamma \vdash_{se} \mathtt{p} \diamond$ means that p is well-typed, whereas $\Gamma \vdash_{se} \mathtt{p} \otimes$ signifies

1st phase		
$\mathtt{p}^{\mathtt{st}}_{se}$	=	$\mathcal{C}\{\!\{\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}}, \mathtt{p^{st}}\}\!\}$
	=	`Student ext Object { }`
$\mathtt{p}^{\mathtt{cs}}_{se}$	=	$\mathcal{C}\{\!\{\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}}, \mathtt{p^{cs}}\}\!\}$
	=	`CStudent ext Student { }`
$\mathtt{p}^{\mathtt{lab}}_{se}$	=	$\mathcal{C}\{\!\{\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}}, \mathtt{p^{lab}}\}\!\}$
	=	`Lab ext Object { f is{ guy[Student].grade = 100 } }`
2nd phase		
$\mathtt{p}^{\mathtt{cs'}}_{se}$	=	$\mathcal{C}\{\!\{\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs'}}\,\Gamma^{\mathtt{lab}}, \mathtt{p^{cs'}}\}\!\}$
	=	`CStudent ext Student { }` = $\mathtt{p}^{\mathtt{cs}}_{se}$
3rd phase		
$\mathtt{p}^{\mathtt{m}}_{se}$	=	$\mathcal{C}\{\!\{\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs'}}\,\Gamma^{\mathtt{lab}}\,\Gamma^{\mathtt{m}}, \mathtt{p^{m}}\}\!\}$
	=	`Marker ext Object { g is{ guy[CStudent].grade = 'A' } }`

Figure 5: Java$_{se}$ class bodies for computing students

that p is well-typed and complete. The type system for Java$_{se}$ is identical to that of Java$_s$ except for the two cases where the Java$_{se}$ syntax differs from that of Java$_s$; these appear in appendix F. When type checking Java$_{se}$ field access expressions, the parent class containing the field declaration is taken into account. Similarly, the statically determined argument types are taken into account when type checking Java$_{se}$ method calls. These properties of the Java$_{se}$ types reflect, at a higher level, checks performed by the byte-code verifier [15, 12], and are crucial for proving the lemmas in section 5. The following lemma says that $\mathcal{C}$ preserves types:

Lemma 1 *For types* T, $\mathtt{T}'$ *Java$_s$ term* t:
$$\Gamma \vdash \mathtt{t} : \mathtt{T} \quad \Longrightarrow \quad \Gamma \vdash_{se} \mathcal{C}\{\!\{\Gamma, \mathtt{t}\}\!\} : \mathtt{T}$$

Java$_r$ is an extension of Java$_{se}$ describing *run-time* terms, such as addresses, or `null`-values in field access or method calls. For Java$_{se}$ program body p, Java$_r$ terms are executed according to rewrite system $\leadsto_{\mathtt{p}}$.

The subject reduction theorem proven in [7] (and similarly in [17, 14]) states that for any well-typed, non-ground Java$_r$ term and any Java$_{se}$ body p with $\vdash_{se} \mathtt{p} \otimes$, there exists a rewrite step which either terminates, or produces a new, well-typed Java$_r$ term, or contains an exception. The exception may be a language defined exception, such as divide-by-zero, null-pointer-access *etc*, or any of the user-defined exceptions, but not one of the linker exceptions. In particular, because the subject reduction theorem ensures the existence of a rewrite step, it also guarantees that all required method bodies and fields will be present. Absence of fields or method bodies is the kind of thing that would throw a linker exception [12].

The subject reduction theorem thus suggests that the assertion $\Gamma \vdash_{se} \mathtt{p} \otimes$ means that p is a complete successfully linked Java$_{se}$ program body. The assertion $\Gamma \vdash_{se} \mathtt{p} \Diamond\!\!\Diamond$ can be established by proving that $\Gamma \vdash_{se} \mathtt{p} \Diamond$ and that $\mathcal{C}l(\mathtt{p}) = \mathcal{C}l(\Gamma)$. The latter requirement is usually a last step and is straightforward to establish. However, the requirement $\Gamma \vdash_{se} \mathtt{p} \Diamond$ is not that easy; in general it requires full type checking.

Therefore, we consider the preservation of the property $\Gamma \vdash_{se} \mathtt{p} \Diamond$ to be an appropriate approximation of the guarantee of binary compatibility. For notational convenience, we use the notation $\vdash_{se}(\Gamma, \mathtt{p}) \Diamond$ as a synonym for $\Gamma \vdash_{se} \mathtt{p} \Diamond$.

4 Concatenating and compiling fragments

We shall call a pair $\mathtt{F} = (\Gamma, \mathtt{p})$, a *fragment*, where Γ is an environment and $\mathtt{p}$ is one or more class bodies. If $\mathtt{p}$ is a Java$_s$ body then $\mathtt{F}$ will be a Java$_s$ fragment, otherwise it will be a Java$_{se}$ fragment. Fragments consist of the declaration and body of one or more classes; they represent parts of programs, or libraries, and they need *not* be self-contained.

In this section we introduce operators to describe concatenation and compilation of fragments. In some cases we expect the constituent environments and bodies to be *disjoint*, as defined in:

Definition 1 *For environments* Γ, Γ' *and bodies* $\mathtt{p}$, $\mathtt{p}'$:

- Γ, Γ' *are* disjoint *iff* $\mathcal{D}(\Gamma) \cap \mathcal{D}(\Gamma') = \emptyset$.
- $\mathtt{p}$, $\mathtt{p}'$ *are* disjoint *iff* $\mathcal{D}(\mathtt{p}) \cap \mathcal{D}(\mathtt{p}') = \emptyset$.
- $(\Gamma, \mathtt{p})$ *and* $(\Gamma', \mathtt{p}')$ *are* disjoint, *iff* Γ, Γ' *and* $\mathtt{p}$, $\mathtt{p}'$ *are disjoint.*

For example, $\Gamma^{\mathtt{cs}'}$ and $\Gamma^{\mathtt{m}}$ are disjoint, whereas $\Gamma^{\mathtt{cs}'}$ and $\Gamma^{\mathtt{m}}\,\Gamma^{\mathtt{cs}}$ are not. The parts of well formed environments or programs are disjoint, *e.g.* $\Gamma\,\Gamma' \vdash \Diamond$ implies that Γ, Γ' are disjoint.

The operator $\_ \circ \_$ represents *concatenation* of fragments through juxtaposition, without performing any checks.

Definition 2 *For fragments* $\mathtt{F} = (\Gamma, \mathtt{p})$, $\mathtt{F}' = (\Gamma', \mathtt{p}')$:

- $\mathtt{F} \circ \mathtt{F}' = (\Gamma\,\Gamma', \mathtt{p}\,\mathtt{p}')$

Concatenation is associative and commutative. If $\mathtt{F}$ and $\mathtt{F}'$ are disjoint, then $\vdash \mathtt{F} \Diamond$ and $\vdash \mathtt{F}' \Diamond$ implies $\vdash \mathtt{F} \circ \mathtt{F}' \Diamond$. Also, $\vdash \mathtt{F} \circ \mathtt{F}' \Diamond$ implies that $\mathtt{F}$ and $\mathtt{F}'$ are disjoint.

The operator $\_ \oplus \_$ describes *updating* the first argument by the declarations/bodies from the second, whereby any class or interface in both will be taken from the second:

Definition 3 *For environments* Γ, Γ' *and bodies* $\mathtt{p}$, $\mathtt{p}'$ *fragments* $\mathtt{F} = (\Gamma, \mathtt{p})$, $\mathtt{F}' = (\Gamma', \mathtt{p}')$:

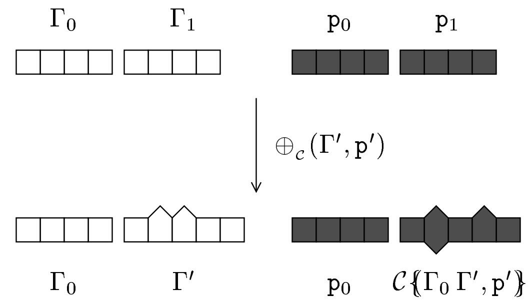

Figure 6: $(\Gamma_0\,\Gamma_1, \mathtt{p}_0\,\mathtt{p}_1) \oplus_c (\Gamma', \mathtt{p}')$

- $\Gamma \oplus \Gamma' = \Gamma_0\,\Gamma'$, *where* Γ_0 *such that* $\Gamma = \Gamma_0\,\Gamma_1$, $\mathcal{D}(\Gamma_1) \subseteq \mathcal{D}(\Gamma')$, *and* Γ_0, Γ' *disjoint.*
- $\mathtt{p} \oplus \mathtt{p}' = \mathtt{p}_0\,\mathtt{p}'$, *where* $\mathtt{p}_0$ *such that* $\mathtt{p} = \mathtt{p}_0\,\mathtt{p}_1$, $\mathcal{D}(\mathtt{p}_1) \subseteq \mathcal{D}(\mathtt{p}')$, *and* $\mathtt{p}_0$, $\mathtt{p}'$ *disjoint.*
- $\mathtt{F} \oplus \mathtt{F}' = (\Gamma \oplus \Gamma', \mathtt{p} \oplus \mathtt{p}')$

Updating is associative but not commutative. For disjoint fragments $\mathtt{F}$, $\mathtt{F}'$ updating is equivalent to concatenation, and also $\mathtt{F} \circ (\mathtt{F}'' \oplus \mathtt{F}') = (\mathtt{F} \circ \mathtt{F}'') \oplus \mathtt{F}$.

The operation $\mathcal{C}\{\!\{\mathtt{F}, \mathtt{F}'\}\!\}$ describes the compilation of a fragment $\mathtt{F}'$ in the context of $\mathtt{F}$, *i.e.* compilation using the environment provided by both $\mathtt{F}$ and $\mathtt{F}'$.

Definition 4 *For fragment* $\mathtt{F} = (\Gamma, \mathtt{p})$, *and Java$_s$ fragment* $\mathtt{F}' = (\Gamma', \mathtt{p}')$:

- $\mathcal{C}\{\!\{\mathtt{F}, \mathtt{F}'\}\!\} = (\Gamma', \mathcal{C}\{\!\{\Gamma \oplus \Gamma', \mathtt{p}'\}\!\})$

Thus, $\mathcal{C}\{\!\{(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}, \mathtt{p}^{\mathtt{st}}\,\mathtt{p}^{\mathtt{cs}}), (\Gamma^{\mathtt{cs}'}, \mathtt{p}^{\mathtt{cs}'})\}\!\} = (\Gamma^{\mathtt{cs}'}, \mathtt{p}^{\mathtt{cs}'}_{se}) = \mathcal{C}\{\!\{(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}, \mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}), (\Gamma^{\mathtt{cs}'}, \mathtt{p}^{\mathtt{cs}'})\}\!\}$.

The operation $\mathtt{F} \oplus_c \mathtt{F}'$ describes the *effect* of the compilation of a Java$_s$ fragment $\mathtt{F}'$ *on* an existing Java$_{se}$ fragment $\mathtt{F}$. The original Java$_{se}$ fragment $\mathtt{F}$ is updated by the compilation of $\mathtt{F}'$ in the context of $\mathtt{F}$.

Definition 5 *For Java$_{se}$ fragment* $\mathtt{F}$, *and Java$_s$ fragment* $\mathtt{F}'$:

- $\mathtt{F} \oplus_c \mathtt{F}' = \mathtt{F} \oplus \mathcal{C}\{\!\{\mathtt{F}, \mathtt{F}'\}\!\}$

So, $(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}, \mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}) \oplus_c (\Gamma^{\mathtt{cs}'}, \mathtt{p}^{\mathtt{cs}'}) = (\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}'}, \mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}'}_{se})$. Figure 6 describes the compilation of the Java$_s$ fragment $(\Gamma', \mathtt{p}')$ into existing Java$_{se}$ fragment $(\Gamma_0\,\Gamma_1, \mathtt{p}_0\,\mathtt{p}_1)$. The ensuing environment, $\Gamma \oplus \Gamma'$, consists of Γ' and Γ_0, the part of Γ which is not superseded by Γ'. The new program body, $\mathtt{p} \oplus \mathcal{C}\{\!\{\Gamma \oplus \Gamma', \mathtt{p}'\}\!\}$, consists of the compilation of $\mathtt{p}'$ in the new environment and $\mathtt{p}_0$, the part of $\mathtt{p}$ which is not superseded by $\mathtt{p}'$.

In general, $\mathcal{C}\{\!\{\mathtt{F}, \mathtt{F}\}\!\} \oplus_c \mathtt{F}' \neq \mathcal{C}\{\!\{\mathtt{F} \oplus \mathtt{F}', \mathtt{F} \oplus \mathtt{F}'\}\!\}$. The left hand side represents separate compilation of fragments whereas the right hand side represents compilation of all fragments together. As we mentioned earlier, in Java these are different, and it is possible for the first to be defined, and the latter to be undefined.

Because the arguments of $\_\oplus_c\_$ come from different domains, the concepts of commutativity and associativity do not apply. We shall use $\oplus_c$ implicitly in a left-associative manner. For fragments $\mathtt{F}_0$, $\mathtt{F}{=}(\Gamma,\mathtt{p})$, $\mathtt{F}'{=}(\Gamma',\mathtt{p}')$, such that $\mathcal{D}(\Gamma)=\mathcal{D}(\Gamma')$ and $\mathtt{p}=\mathtt{p}'$, the equality $(\mathtt{F}_0\oplus\mathtt{F}')\oplus_c(\Gamma',\epsilon)=(\mathtt{F}_0\oplus\mathtt{F}')\oplus_c(\Gamma',\mathtt{p}')$ holds, where ϵ describes the empty environment or program body.

The second phase of the students example compiles $(\Gamma^{\mathtt{cs}'},\mathtt{p}^{\mathtt{cs}})$ into $(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}},\mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}\,\mathtt{p}^{\mathtt{lab}}_{se})$, giving:

$$\begin{aligned}&(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}},\mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}\,\mathtt{p}^{\mathtt{lab}}_{se})\oplus_c(\Gamma^{\mathtt{cs}'},\mathtt{p}^{\mathtt{cs}})\\ =\ &(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}},\mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}\,\mathtt{p}^{\mathtt{lab}}_{se})\oplus_c(\Gamma^{\mathtt{cs}'},\epsilon)\\ =\ &(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}'}\,\Gamma^{\mathtt{lab}},\mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}\,\mathtt{p}^{\mathtt{lab}}_{se})\end{aligned}$$

In the third phase we compile the new fragment $(\Gamma^{\mathtt{m}},\mathtt{p}^{\mathtt{m}})$ into the result of the previous change, giving:

$$\begin{aligned}&(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}'}\,\Gamma^{\mathtt{lab}},\mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}\,\mathtt{p}^{\mathtt{lab}}_{se})\oplus_c(\Gamma^{\mathtt{m}},\mathtt{p}^{\mathtt{m}})\\ =\ &(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}'}\,\Gamma^{\mathtt{lab}}\,\Gamma^{\mathtt{m}},\mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}\,\mathtt{p}^{\mathtt{lab}}_{se}\,\mathcal{C}\{\!\!\{\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}'}\,\Gamma^{\mathtt{lab}}\,\Gamma^{\mathtt{m}},\mathtt{p}^{\mathtt{m}}\}\!\!\})\\ =\ &(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}'}\,\Gamma^{\mathtt{lab}}\,\Gamma^{\mathtt{m}},\mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}\,\mathtt{p}^{\mathtt{lab}}_{se}\mathtt{p}^{\mathtt{m}}_{se})\end{aligned}$$

The following lemma, used to prove lemma 5, describes the result of compiling fragment $\mathtt{F}''$ into $\mathtt{F}\circ\mathtt{F}'$. If $\mathcal{C}\{\!\!\{\mathtt{F}',\mathtt{F}''\}\!\!\}$ is defined, *i.e.* compilation of $\mathtt{F}''$ does not need information from $\mathtt{F}$, then $\mathtt{F}$ remains unaffected, and is not taken into account for compilation of $\mathtt{F}''$. If $\mathtt{F}$ and $\mathtt{F}''$ are disjoint, then $\mathtt{F}$ remains unaffected but may be taken into account for compilation of $\mathtt{F}''$.

Lemma 2 *For fragments* $\mathtt{F}$, $\mathtt{F}'$, $\mathtt{F}''$, *with* $\mathtt{F}$ *and* $\mathtt{F}'$ *disjoint*:

- $\mathcal{C}\{\!\!\{\mathtt{F}',\mathtt{F}''\}\!\!\}$ *defined* $\Longrightarrow$ $(\mathtt{F}\circ\mathtt{F}')\oplus_c\mathtt{F}''=\mathtt{F}\circ(\mathtt{F}'\oplus_c\mathtt{F}'')$
- $\mathtt{F}$ *and* $\mathtt{F}''$ *disjoint* $\Longrightarrow$ $(\mathtt{F}\circ\mathtt{F}')\oplus_c\mathtt{F}''=\mathtt{F}\circ(\mathtt{F}'\oplus\mathcal{C}\{\!\!\{\mathtt{F}\circ\mathtt{F}',\mathtt{F}''\}\!\!\})$

5 Link compatibility

The term *link compatibility* aims to capture the guarantee given by binary compatibility. It restricts source code modifications in terms of the properties of the resulting compilation. As we argued in section 3, well-formedness, expressed by the assertion $\vdash_{se}\mathtt{F}\ \diamond$, should be preserved throughout binary compatible changes.

We consider $\mathtt{F}'$ a *link compatible change* of a fragment $\mathtt{F}$, if all fragments $\mathtt{F}_0$ that successfully linked with $\mathtt{F}$ continue to do so after compilation of $\mathtt{F}'$ into $\mathtt{F}$.

Definition 6 *A Java$_s$ fragment* $\mathtt{F}'$, *is a* link compatible change *of a Java$_{se}$ fragment* $\mathtt{F}$, *iff*
For all $\mathtt{F}_0$ *disjoint with* $\mathtt{F}'$:

$$\vdash_{se}\mathtt{F}_0\circ\mathtt{F}\ \diamond\qquad\Longrightarrow\qquad\vdash_{se}(\mathtt{F}_0\circ\mathtt{F})\oplus_c\mathtt{F}'\ \diamond$$

For example, $(\Gamma^{\mathtt{cs}'},\mathtt{p}^{\mathtt{cs}})$ is a link compatible change of $(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}},\mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}\,\mathtt{p}^{\mathtt{lab}}_{se})$, and $(\Gamma^{\mathtt{cs}'},\epsilon)$ is a link compatible change of $(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}},\mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}\,\mathtt{p}^{\mathtt{lab}}_{se})$. In section 6 we discuss how to prove such statements.

Originally we had defined as link compatible changes $\mathtt{F}'$ those guaranteeing that $\vdash_{se}\mathtt{F}\ \diamond\ \Longrightarrow\ \vdash_{se}\mathtt{F}\oplus_c\mathtt{F}'\ \diamond$, but this definition turned out to be too weak, *c.f.* appendix A where we discuss alternatives. The requirement $\vdash_{se}(\mathtt{F}_0\circ\mathtt{F})\oplus_c\mathtt{F}'\ \diamond$ ensures successful compilation of $\mathtt{F}'$ in the context of both $\mathtt{F}_0$ and $\mathtt{F}$. It is weaker than asking $\vdash_{se}\mathtt{F}_0\circ(\mathtt{F}\oplus_c\mathtt{F}')\ \diamond$, because it is possible for $(\mathtt{F}_0\circ\mathtt{F})\oplus_c\mathtt{F}'$ to be defined and for $\mathtt{F}\oplus_c\mathtt{F}'$ not to be. This subtlety is deliberate. It allows $\mathtt{F}'$ to be considered a link compatible change for a library $\mathtt{F}$, which imports other libraries, and which cannot be compiled in isolation, *i.e.* for which $\vdash_{se}\mathtt{F}\ \diamond$ does not hold. Such a library can only be compiled in the presence of one or more further libraries, represented by the fragment $\mathtt{F}_0$, with which $\vdash_{se}\mathtt{F}_0\circ\mathtt{F}\ \diamond$ holds.

Therefore, the fragment $\mathtt{F}$ does not need to contain *all* the type information necessary to type check $\mathtt{F}'$; it only needs to contain *enough* information to ensure type correct compilation of $\mathtt{F}'$ in the context of all appropriate fragments $\mathtt{F}_0$. Thus, $\mathtt{F}$ acts as a kind of filter for $\mathtt{F}_0$, by requiring that $\vdash_{se}\mathtt{F}_0\circ\mathtt{F}\ \diamond$. Consider, for example:

$$\begin{aligned}\Gamma^{\mathtt{C}}&=\ \mathtt{class\ C\ ext\ Object}\ \{\mathtt{f}:\rightarrow\mathtt{int}\},\\ \Gamma^{\mathtt{D}}&=\ \mathtt{class\ D\ ext\ C}\ \{\mathtt{f}:\rightarrow\mathtt{int}\},\\ \Gamma^{\mathtt{D}'}&=\ \mathtt{class\ D\ ext\ C}\ \{\mathtt{f}:\rightarrow\mathtt{int},\mathtt{x}:\mathtt{char}\},\end{aligned}$$

The fragment $(\Gamma^{\mathtt{D}'},\epsilon)$ is a link compatible change of $(\Gamma^{\mathtt{C}}\,\Gamma^{\mathtt{D}},\epsilon)$, of $(\Gamma^{\mathtt{C}},\epsilon)$, *and* of $(\Gamma^{\mathtt{D}},\epsilon)$. The latter holds, because any Γ_0 with $\Gamma_0\,\Gamma^{\mathtt{D}}\vdash\diamond$ also satisfies $\Gamma_0\,\Gamma^{\mathtt{D}'}\vdash\diamond$.

Our original intuition was, for $\mathtt{F}'$ a link compatible change of $\mathtt{F}$, that $\mathtt{F}$ need only contain the definitions or declarations modified by $\mathtt{F}'$. This was incorrect, because in general these do not hold sufficient information to ensure type correctness in the context of all appropriate fragments $\mathtt{F}_0$. For example, consider the environments:

$$\begin{aligned}\Gamma^{\mathtt{A}}&=\ \mathtt{class\ A\ ext\ Object}\ \{\mathtt{f}:\rightarrow\mathtt{int}\},\\ \Gamma^{\mathtt{A}'}&=\ \mathtt{class\ A\ ext\ Object}\ \{\mathtt{f}:\rightarrow\mathtt{char}\},\\ \Gamma^{\mathtt{B}}&=\ \mathtt{class\ B\ ext\ A}\ \{\ \},\\ \Gamma^{\mathtt{B}'}&=\ \mathtt{class\ B\ ext\ A}\ \{\mathtt{f}:\rightarrow\mathtt{int}\}\end{aligned}$$

The fragment $(\Gamma^{\mathtt{B}'},\epsilon)$ is a link compatible change of $(\Gamma^{\mathtt{A}}\,\Gamma^{\mathtt{B}},\epsilon)$, and of $(\Gamma^{\mathtt{A}},\epsilon)$, but it is *not* a link compatible change of $(\Gamma^{\mathtt{B}},\epsilon)$. Namely, $\vdash_{se}(\Gamma^{\mathtt{A}'},\epsilon)\circ(\Gamma^{\mathtt{B}},\epsilon)\ \diamond$ holds, but $\Gamma^{\mathtt{A}'}\,\Gamma^{\mathtt{B}'}\vdash\diamond$ does not! And so, it is not the case that $\vdash_{se}((\Gamma^{\mathtt{A}'},\epsilon)\circ(\Gamma^{\mathtt{B}},\epsilon))\oplus_c(\Gamma^{\mathtt{B}'},\epsilon)\ \diamond$.

5.1 Properties of link compatible changes

We now discuss and prove the following five properties of link compatible changes:

- **Preservation over larger fragments**: link compatibility is preserved by larger fragments.
- **Preservation over sequences**: a sequence of link compatible changes preserves well-formedness – as shown in figure 7.
- **Preservation over libraries**: several link compatible changes when applied to different fragments preserve well formedness – as shown in figures 8, 9.
- **Lack of diamond property**: for two different link compatible changes applied to the same fragment, there does not necessarily exist a further link compatible change reconciling the two – as shown in figure 11.
- **Lack of folding property**: in general, two link compatible changes cannot be folded into one link compatible change– as shown in figure 10.

These properties are crucial in delineating the exact nature of binary compatibility. In fact, we have been discussing with the Java language developers whether a diamond property and the preservation over libraries *are* satisfied by binary compatibility, and to what extent these properties *should be* satisfied [16]. Thus, a major contribution of this paper lies, we believe, in formulating and distinguishing these properties.

The preservation over larger fragments automatically establishes link compatibility for all fragments that contain a smaller fragment for which this property has already been established. The preservation over sequences guarantees that link compatible steps may be combined, and preserve the linking capabilities – provided that each step is a link compatible change of the result of the application of all previous modifications. The preservation over sequences is not surprising, but the fact that it is satisfied demonstrates that the definition is appropriate.

The lack of folding and diamond properties restrict the ways in which link compatible changes may be combined. The lack of diamond property means that programmers may not apply *independent* link compatible changes to the *same* fragment and expect the linking capabilities to be preserved. However, the preservation over libraries allows programmers to apply *independent* link compatible changes and expect the linking capabilities to be preserved, as long as they were working on *different* fragments. In particular, it means that various libraries may be modified separately, each in link compatibile ways, and still preserve their linking capabilities. This holds, even if these libraries should import each other.

Next we formulate and prove these properties.

Preservation over larger fragments A link compatible change of a given fragment is also a link compatible change of any larger fragment:

Lemma 3 *For fragments* $\mathtt{F}$, $\mathtt{F}'$, $\mathtt{F}''$, *where* $\mathtt{F}'$ *and* $\mathtt{F}''$ *are disjoint*:

$$\mathtt{F}' \textit{ is a link compatible change of } \mathtt{F} \implies \mathtt{F}' \textit{ is a link compatible change of } \mathtt{F}''\circ\mathtt{F}$$

Preservation over sequences As outlined in figure 7, a sequence of link compatible steps, $\mathtt{F}'_1$, ... $\mathtt{F}'_n$, applied to fragment $\mathtt{F}$ preserves the linking capabilities of $\mathtt{F}$. In order to establish that a step is link compatible, we need to know the effect of all prior steps, thus we require that $\mathtt{F}'_{i+1}$ is link compatible for $\mathtt{F}_0\circ\mathtt{F}\oplus_c\mathtt{F}'_1 ...\oplus_c\mathtt{F}'_i$.

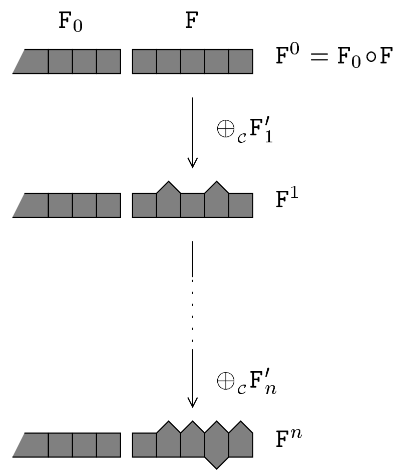

Figure 7: Preservation over sequences

Lemma 4 *For Java*$_{se}$ *fragments* $\mathtt{F}$, $\mathtt{F}_0$, *a sequence of Java*$_s$ *fragments* $\mathtt{F}'_1$, ... $\mathtt{F}'_n$, $\mathtt{F}_0$ *disjoint* $\mathtt{F}'_i$, *if*

- *for all* i, $1 \leq i \leq n$:
  $\mathtt{F}^i$ *defined* $\implies \mathtt{F}'_{i+1}$ *link compatible change of* $\mathtt{F}^i$
  where $\mathtt{F}^i = \mathtt{F}_0\circ\mathtt{F}\oplus_c\mathtt{F}'_1 ...\oplus_c\mathtt{F}'_i$

then

- $\vdash_{\overline{s}e}\mathtt{F}_0\circ\mathtt{F}\ \diamond \implies \vdash_{\overline{s}e}(\mathtt{F}_0\circ\mathtt{F})\oplus_c\mathtt{F}'_1 ...\oplus_c\mathtt{F}'_n\ \diamond$

Proof by induction on k; using that $\mathtt{F}^0{=}\mathtt{F}_0\circ\mathtt{F}$ and $\mathtt{F}^{k+1}{=}\mathtt{F}^k\oplus_c\mathtt{F}'_k$, prove that $\vdash_{\overline{s}e}\mathtt{F}^k\ \diamond$ for all k. Also, $\mathtt{F}^n = (\mathtt{F}_0\circ\mathtt{F})\oplus_c\mathtt{F}'_1 ...\oplus_c\mathtt{F}'_n$. □

Preservation over libraries Link compatible modifications $\mathtt{F}'_i$ applied to fragments $\mathtt{F}_i$ which are parts of a program $\mathtt{F}\circ\mathtt{F}_1\circ ...\circ\mathtt{F}_n$, preserve the linking capabilities of that program, provided that the modifications are link compatible for the particular fragments only – *i.e.* require $\mathtt{F}'_i$ is a link compatible change of $\mathtt{F}_i$, which is stronger than requiring $\mathtt{F}'_i$ to be a link compatible change of $\mathtt{F}_1 ... \mathtt{F}_n$.

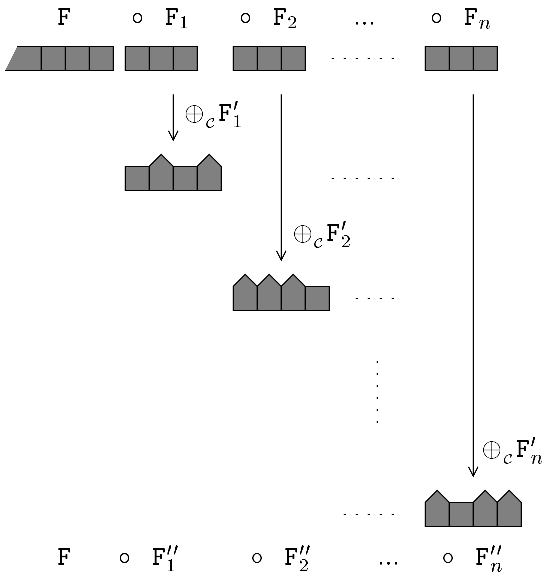

Figure 8: Preservation over libraries where $\mathtt{F}''_k = \mathtt{F}_k \oplus \mathcal{C}\{\!\{\mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_{k-1}\circ\mathtt{F}_k...\circ\mathtt{F}_n, \mathtt{F}'_k\}\!\}$

In contrast to preservation over sequences, we do not need to know the effect of another modification in order to establish that $\mathtt{F}'_i$ is a link compatible change of $\mathtt{F}_i$. However, we may take another modification into account when applying a modification. We distinguish the following two cases: 1) The application of a modification takes into account the effect of the previous modifications, thus $\mathtt{F}_k$ is transformed to $\mathtt{F}''_k$, where $\mathtt{F}''_k = \mathtt{F}_k \oplus \mathcal{C}\{\!\{\mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_{k-1}\circ\mathtt{F}_k...\circ\mathtt{F}_n, \mathtt{F}'_k\}\!\}$; as described in figure 8. 2) The application of a modification does not take into account the effect of any other modifications and compiles in the original context, *i.e.* $\mathtt{F}_k$ is transformed to $\mathtt{F}''_k$, where $\mathtt{F}''_k = \mathtt{F}_k \oplus \mathcal{C}\{\!\{\mathtt{F}\circ \mathtt{F}_1 \; ... \circ \mathtt{F}_n, \mathtt{F}'_k\}\!\}$; as described in figure 9.

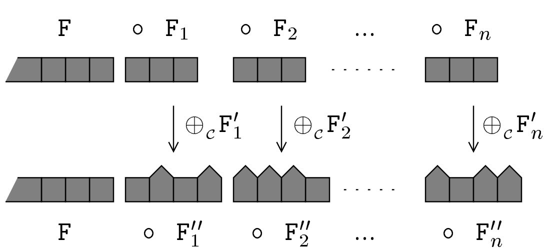

Figure 9: Preservation over libraries where $\mathtt{F}''_k = \mathtt{F}_k \oplus \mathcal{C}\{\!\{\mathtt{F}\circ\mathtt{F}_1...\circ\mathtt{F}_n, \mathtt{F}'_k\}\!\}$

The first case represents the situation where programmers make changes to the particular fragments that belong to them, but *are aware* of each other's actions. The second case corresponds to the situation where programmers take a snapshot of each other's work, and then go on to work on their own fragments *unaware* of each other's activity. In both cases, when all modified fragments are put together, the resulting program $\mathtt{F} \circ \mathtt{F}''_1 \; ...\circ \; \mathtt{F}''_n$ preserves the linking capabilities of the original program. The order of the fragments is immaterial for the current lemma.

Lemma 5 *For Java$_{se}$ fragments* $\mathtt{F}$, $\mathtt{F}_1$, ... $\mathtt{F}_n$, *Java$_s$ fragments* $\mathtt{F}'_1$, ... $\mathtt{F}'_n$, *where* $\mathtt{F}'_i$ *disjoint from* $\mathtt{F}_k$, *from* $\mathtt{F}'_k$ *and from* $\mathtt{F}$ *for all* $i \neq k$, $i, k \in \{1...n\}$, *if*

- $\mathtt{F}'_i$ *is a link compatible change of* $\mathtt{F}_i$ *for* $1 \leq i \leq n$
- $\vdash_{se} \mathtt{F} \circ \mathtt{F}_1 \circ \; ... \circ \mathtt{F}_n \; \diamond$

then

- $\vdash_{se} \mathtt{F} \circ \mathtt{F}''_1 \; ...\circ \; \mathtt{F}''_n \; \diamond$
  where $\mathtt{F}''_k = \mathtt{F}_k \oplus \mathcal{C}\{\!\{\mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_{k-1}\circ\mathtt{F}_k...\circ\mathtt{F}_n, \mathtt{F}'_k\}\!\}$
- $\vdash_{se} \mathtt{F} \circ \mathtt{F}''_1 \; ...\circ \; \mathtt{F}''_n \; \diamond$
  where $\mathtt{F}''_k = \mathtt{F}_k \oplus \mathcal{C}\{\!\{\mathtt{F}\circ \; \mathtt{F}_1 \; ... \circ \mathtt{F}_n, \mathtt{F}'_k\}\!\}$

Proof Because $\vdash_{se} \mathtt{F} \circ \mathtt{F}_1 \circ \; ... \circ \mathtt{F}_n \; \diamond$, we know that $\mathtt{F}_i$ are disjoint from $\mathtt{F}_k$ and from $\mathtt{F}$, for $i \neq k$.
1st Part Define $\mathtt{F}^k = \mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_k\circ\mathtt{F}_{k+1}...\circ\mathtt{F}_n$, where $\mathtt{F}''_k = \mathtt{F}_k \oplus \mathcal{C}\{\!\{\mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_{k-1}\circ\mathtt{F}_k...\circ\mathtt{F}_n, \mathtt{F}'_k\}\!\}$. To show that $\vdash_{se}\mathtt{F}^n \; \diamond$.

For all $k \neq j$, if $\mathtt{F}''_k$ and $\mathtt{F}''_j$ are defined, then $\mathtt{F}''_k$ is disjoint from $\mathtt{F}''_j$, from $\mathtt{F}'_j$, from $\mathtt{F}_j$ and from $\mathtt{F}$.

Show by induction on k that $\mathtt{F}''_k$ and $\mathtt{F}^k$ are defined, and that $\vdash_{se}\mathtt{F}^k \; \diamond$. The case where $k = 0$ follows from the assumptions of the lemma. For the induction step $(k+1 \Rightarrow k+2)$: by induction hypothesis

$\vdash_{se}\mathtt{F}^{k+1} \; \diamond$ by definition of $\mathtt{F}^{k+1}$

$\vdash_{se}\mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_k\circ\mathtt{F}_{k+1}...\circ\mathtt{F}_n \; \diamond$ $\circ$ commutative

$\vdash_{se}(\mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_k\circ\mathtt{F}_{k+2}...\circ\mathtt{F}_n)\circ\mathtt{F}_{k+1} \; \diamond$

$\mathtt{F}'_{k+1}$ link compatible change of $\mathtt{F}_{k+1}$

$\vdash_{se}((\mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_k\circ\mathtt{F}_{k+2}...\circ\mathtt{F}_n)\circ\mathtt{F}_{k+1}) \oplus_c \mathtt{F}'_{k+1} \; \diamond$

lemma 2

$\mathtt{F}''_i$ disj. from $\mathtt{F}''_l$, $\mathtt{F}_l$ for $1 \leq i \neq l \leq k$

$\mathtt{F}_l$ disj. from $\mathtt{F}_j$ for $1 \leq l \neq j \leq n$

$\vdash_{se}(\mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_k\circ\mathtt{F}_{k+2}...\circ\mathtt{F}_n)\circ\mathtt{F}_{k+1} \oplus$
$\mathcal{C}\{\!\{\mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_k\circ\mathtt{F}_{k+2}...\circ\mathtt{F}_n\circ\mathtt{F}_{k+1}, \mathtt{F}'_{k+1}\}\!\} \; \diamond$

definition of $\mathtt{F}''_{k+1}$

$\vdash_{se}(\mathtt{F}\circ\mathtt{F}''_1\circ...\mathtt{F}''_k\circ\mathtt{F}_{k+2}...\circ\mathtt{F}_n)\circ\mathtt{F}''_{k+1} \; \diamond$

definition of $\mathtt{F}^{k+2}$

$\vdash_{se}\mathtt{F}^{k+2} \; \diamond$.

Therefore, $\mathtt{F}''_{k+1}$ is defined and $\vdash_{se}\mathtt{F}^{k+1} \; \diamond$ holds.
2nd Part similar to and easier than 1st part. □

Lack of folding property The concepts of transitivity and reflexivity are not applicable to the link compatibility relationship, because its domain and range do not match. Instead, one might consider the following "folding property", outlined in figure 10:

For disjoint $\mathtt{F}'_1$, $\mathtt{F}'_2$, if $\mathtt{F}'_1$ is a link compatible change of $\mathtt{F}$, and $\mathtt{F}'_2$ is a link compatible change of $(\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_1$, then $\mathtt{F}'_1 \oplus \mathtt{F}'_2$ is a link compatible change of $\mathtt{F}_0 \circ \mathtt{F}$, and $(\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_1 \oplus_c \mathtt{F}'_2. = (\mathtt{F}_0 \circ \mathtt{F}) \oplus_c (\mathtt{F}'_1 \oplus \mathtt{F}'_2)$

$$\begin{tikzcd}
\mathtt{F}_0 \circ \mathtt{F} \arrow[d, "\oplus_c \mathtt{F}'_1"] & & \mathtt{F}_0 \circ \mathtt{F} \arrow[dd, "\oplus_c (\mathtt{F}'_1 \oplus \mathtt{F}'_2)"] \\
(\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_1 \arrow[d, "\oplus_c \mathtt{F}'_2"] & \not\Rightarrow & \\
(\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_1 \oplus_c \mathtt{F}'_2 & & (\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_1 \oplus_c \mathtt{F}'_2
\end{tikzcd}$$

Figure 10: Lack of folding property

Such a property does *not* hold. As a counter-example, consider Java$_{se}$ fragment corresponding to `Student` and `CStudent`, *i.e.* $\mathtt{F} = (\Gamma^{\mathtt{st}}\, \Gamma^{\mathtt{cs}}, \mathtt{p}^{\mathtt{st}}\, \mathtt{p}^{\mathtt{cs}})$. First, the class `Lab` is compiled, *i.e.* $\mathtt{F}'_1 = (\Gamma^{\mathtt{lab}}, \mathtt{p}^{\mathtt{lab}})$. Then, the modified class `CStudent`$'$ is compiled, *i.e.* $\mathtt{F}'_2 = (\Gamma^{\mathtt{cs}'}, \mathtt{p}^{\mathtt{cs}})$. Both changes are link compatible changes, yet the change formed by naïvely composing the two steps, *i.e.* compiling `Lab` *and* `CStudent`$'$ into the original program, is not a link compatible change, since the Java$_s$ class body of `Lab` is not well-typed in an environment featuring the class declaration from `CStudent`$'$.

Lack of diamond property For certain $\mathtt{F}'_1$ and $\mathtt{F}'_2$, link compatible changes of $\mathtt{F}$, there do not exist fragments $\mathtt{F}'_3$ and $\mathtt{F}'_4$, such that $\mathtt{F}'_3$, $\mathtt{F}'_4$ disjoint with $\mathtt{F}'_1$, $\mathtt{F}'_2$, and $\mathtt{F}'_3$ is a link compatible change of $\mathtt{F} \oplus_c \mathtt{F}'_1$, and $\mathtt{F}'_4$ is a link compatible change of $\mathtt{F} \oplus_c \mathtt{F}'_2$, and $\mathtt{F} \oplus_c \mathtt{F}'_1 \oplus_c \mathtt{F}'_3 = \mathtt{F} \oplus_c \mathtt{F}'_2 \oplus_c \mathtt{F}'_4$.

For example, $\mathtt{F}'_1$ might be introducing a method `f` with signature `int` $\rightarrow$ `int` into a class `C`, and $\mathtt{F}'_2$ introducing another method `f` with signature `int` $\rightarrow$ `char` into the same class `C`. The lack of diamond property does not contradict the preservation over libraries, because there we required the modifications to be applied to *disjoint* fragments.

5.2 Type preserving changes

In the previous section we established the power of link compatibility, and argued that it models the guarantee by binary compatibility. However, we have not discussed yet how to prove that a particular modification is link compatible.

$$\begin{tikzcd}
 & \mathtt{F}_0 \circ \mathtt{F} \arrow[dl] \arrow[dr] & \\
(\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_1 & & (\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_2
\end{tikzcd}$$

$$\not\Rightarrow$$

$$\begin{tikzcd}
 & \mathtt{F}_0 \circ \mathtt{F} \arrow[dl] \arrow[dr] & \\
(\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_1 \arrow[dr] & & (\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_2 \arrow[dl] \\
 & (\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_1 \oplus_c \mathtt{F}'_3 = (\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'_2 \oplus_c \mathtt{F}'_4 &
\end{tikzcd}$$

Figure 11: Lack of diamond property

In this section we introduce *type preserving changes*, and prove that type preserving changes are link compatible. In section 6 we shall introduce *safe changes*, which correspond to those changes suggested in the Java specification, which apply to Java$_s$, *and* can be demonstrated to ensure link compatibility, and we shall prove that safe changes are type preserving. Thus, we have:

modifications				*guarantee*
list of safe changes	$\Longrightarrow$	type preserving changes	$\Longrightarrow$	link compatible changes

A *type preserving change of* an environment Γ preserves the types of all Java$_{se}$ expressions `e` given by Γ and context environments Γ_0.

Definition 7 *An environment* Γ' *is a* type preserving change of *environment* Γ *iff for all* Γ_0 *disjoint with* Γ', *for all Java$_{se}$ expressions* `e`, *types* `T`:

$$\Gamma_0\, \Gamma \vdash_{se} \mathtt{e} : \mathtt{T} \Longrightarrow \Gamma_0\, \Gamma \oplus \Gamma' \vdash_{se} \mathtt{e} : \mathtt{T}$$

For example, consider $\Gamma^{\mathtt{A}}$, $\Gamma^{\mathtt{A}'}$, $\Gamma^{\mathtt{B}}$, $\Gamma^{\mathtt{B}'}$ as introduced in the beginning of section 5. Then the environment $\Gamma^{\mathtt{B}'}$ is a type preserving change of $\Gamma^{\mathtt{A}}\, \Gamma^{\mathtt{B}}$, and of $\Gamma^{\mathtt{A}}$, but it is not a type preserving change of $\Gamma^{\mathtt{B}}$. It holds that $\Gamma^{\mathtt{A}'}\, \Gamma^{\mathtt{B}}, \mathtt{x} : \Gamma^{\mathtt{B}} \vdash \mathtt{x}[].\mathtt{f}() : \mathtt{char}$, but it does not hold that $\Gamma^{\mathtt{A}'}\, \Gamma^{\mathtt{B}} \oplus \Gamma^{\mathtt{B}'}, \mathtt{x} : \Gamma^{\mathtt{B}} \vdash \mathtt{x}[].\mathtt{f}() : \mathtt{char}$. In fact, it does not even hold that $\Gamma^{\mathtt{A}'}\, \Gamma^{\mathtt{B}} \oplus \Gamma^{\mathtt{B}'} \vdash \diamond$.

Notice, that Γ might be incomplete in the above definition , *i.e.* it might not satisfy $\Gamma \vdash \diamond$, and it might not have a type for the expression `e`. The requirement that $\Gamma_0\, \Gamma \vdash_{se} \mathtt{e} : \mathtt{T} \Longrightarrow \Gamma_0\, \Gamma \oplus \Gamma' \vdash_{se} \mathtt{e} : \mathtt{T}$ is strictly stronger than $\Gamma \vdash_{se} \mathtt{e} : \mathtt{T} \Longrightarrow \Gamma \oplus \Gamma' \vdash_{se} \mathtt{e} : \mathtt{T}$. For example, $\Gamma^{\mathtt{B}'}$ vacuously satisfies the requirement

$\Gamma^{\mathtt{A}} \vdash_{\mathrm{se}} \mathtt{e} : \mathtt{T} \implies \Gamma^{\mathtt{A}} \oplus \Gamma^{\mathtt{B}'} \vdash_{\mathrm{se}} \mathtt{e} : \mathtt{T}$, since no expression satisfies $\Gamma^{\mathtt{A}} \vdash_{\mathrm{se}} \mathtt{e} : \mathtt{T}$. We expect for Γ with $\Gamma \vdash \diamond$, the requirement $\Gamma \vdash_{se} \mathtt{e} : \mathtt{T} \implies \Gamma \oplus \Gamma' \vdash_{se} \mathtt{e} : \mathtt{T}$ to be equivalent with $\Gamma_0\,\Gamma \vdash_{se} \mathtt{e} : \mathtt{T} \implies \Gamma_0\,\Gamma \oplus \Gamma' \vdash_{se} \mathtt{e} : \mathtt{T}$.

Notice also, that a type preserving change of of an environment does *not* preserve the types of Java$_s$ terms. So, $\Gamma^{\mathrm{st}}\,\Gamma^{\mathrm{cs}}, \mathtt{guy} : \mathtt{CStudent} \vdash \mathtt{guy.grade} : \mathtt{int}$, whereas $(\Gamma^{\mathrm{st}}\,\Gamma^{\mathrm{cs}}, \mathtt{guy} : \mathtt{CStudent}) \oplus \Gamma^{\mathrm{cs}'} \vdash \mathtt{guy.grade} : \mathtt{char}$.

As with link compatibility, in general, if Γ' is a type preserving change of a smaller environment Γ, then it is also a type preserving change of the larger environment $\Gamma\,\Gamma''$.

The following lemma describes how type preserving changes of environments combined with type correct compilations of class bodies produce link compatible modifications. The second requirement, asking that $\Gamma_0\,\Gamma \vdash \diamond \implies \Gamma_0\,\Gamma \oplus \Gamma' \vdash \mathtt{p}'\ \diamond$, allows us to consider modifications which need a context Γ_0 for their compilation. Thus we can have libraries which are not stand alone. That requirement could be replaced by the stronger requirement that $\Gamma \oplus \Gamma' \vdash \mathtt{p}'\ \diamond$. The third requirement ensures that a new class body will be provided for any class in Γ', *i.e.* whose declaration is modified.

Lemma 6 *For environments* Γ, Γ', *Java$_{se}$ program body* $\mathtt{p}$, *Java$_s$ program body* $\mathtt{p}'$, *if*

- Γ' *is type preserving change of* Γ
- $\forall\,\Gamma_0$ *disj. with* Γ': $\Gamma_0\,\Gamma \vdash \diamond \implies \Gamma_0\,\Gamma \oplus \Gamma' \vdash \mathtt{p}'\ \diamond$
- $\mathcal{Cl}(\Gamma') \subseteq \mathcal{Cl}(\mathtt{p}')$

then

- $(\Gamma', \mathtt{p}')$ *is a link compatible change of* $(\Gamma, \mathtt{p})$

Proof through careful application of the definitions and type checking rules.

Let us call $\mathtt{F} = (\Gamma, \mathtt{p})$, $\mathtt{F}' = (\Gamma', \mathtt{p}')$. Take any Java$_{se}$ fragment $\mathtt{F}_0 = (\Gamma_{00}, \mathtt{p}_{00})$, such that $\mathtt{F}_0$ disjoint from $\mathtt{F}'$, and $\vdash_{se} \mathtt{F}_0 \circ \mathtt{F}\ \diamond$. To show that $\vdash_{se} (\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}'\ \diamond$.

Because $\vdash_{se} \mathtt{F}_0 \circ \mathtt{F}\ \diamond$, it also holds that Γ_{00} and Γ are disjoint, and, because of the requirements of the lemma, $\Gamma'' \vdash \mathtt{p}'\ \diamond$, where $\Gamma'' = \Gamma_{00}\,\Gamma \oplus \Gamma'$. Therefore, $\Gamma'' \vdash \diamond$. It remains to prove that $\Gamma'' \vdash_{se} \mathtt{p}''\ \diamond$, where $\mathtt{p}'' = \mathcal{C}\{\!\{\Gamma'', \mathtt{p}_{00}\,\mathtt{p} \oplus \mathtt{p}'\}\!\}$.

Take any Java$_{se}$ class body $\mathtt{cBody}$ from $\mathtt{p}''$. Let $\mathtt{C}$ be the name of the class to which $\mathtt{cBody}$ belongs.

1st Case: $\mathtt{C} \in \mathcal{Cl}(\mathtt{p}')$. Then there exists a Java$_s$ class body $\mathtt{cBody}'$, such that $\mathtt{p}' = \mathtt{cBody}'\,\mathtt{p}'_1$, and that $\mathcal{C}\{\!\{\Gamma'', \mathtt{cBody}'\}\!\} = \mathtt{cBody}$. Because $\Gamma'' \vdash \mathtt{p}'\ \diamond$, we also have that $\Gamma'' \vdash \mathtt{cBody}'\ \diamond$, and with lemma 1, we also get that $\Gamma'' \vdash_{se} \mathtt{cBody}\ \diamond$.

2nd Case: $\mathtt{C} \notin \mathcal{Cl}(\mathtt{p}')$, therefore $\mathtt{cBody}$ stems from $\mathtt{p}_{00}$ or $\mathtt{p}$. Because $\Gamma_{00}\,\Gamma \vdash_{se} \mathtt{p}_{00}\,\mathtt{p}\ \diamond$, it also holds that $\Gamma_{00}\,\Gamma \vdash_{se} \mathtt{cBody}\ \diamond$. Because $\mathcal{Cl}(\Gamma') \subseteq \mathcal{Cl}(\mathtt{p}')$, we also have that $\mathtt{C} \notin \mathcal{Cl}(\Gamma')$. Therefore, $\mathtt{C}$ has the same definition in $\Gamma_{00}\,\Gamma$ and in $\Gamma_{00}\,\Gamma \oplus \Gamma'$. Take any method body $\mathtt{mBody}$ from $\mathtt{cBody}$; because $\mathtt{cBody}$ is type correct, through application of the type rule for class bodies, we obtain: $\Gamma_{00}\,\Gamma, \mathtt{this} : \mathtt{C} \vdash_{se} \mathtt{mBody} : \mathtt{T}_1 \times ...\mathtt{T}_n \to \mathtt{T}$, where $\mathtt{T}_1 \times ...\mathtt{T}_n \to \mathtt{T}$ is a signature of $\mathtt{m}$ in class $\mathtt{C}$ in the environment $\Gamma_{00}\,\Gamma$, and where $\mathtt{mBody}$ has the form $\mathtt{mBody} = \mathtt{m}\ \mathtt{is}\ \lambda\mathtt{x}_1 : \mathtt{T}_1 ... \lambda\mathtt{x}_n : \mathtt{T}_n.\{\mathtt{stmts}\}$. Applying the type rules for method bodies, we obtain: $\Gamma_{00}\,\Gamma, \mathtt{this} : \mathtt{C}, \mathtt{z}_1 : \mathtt{T}_1, ...\mathtt{z}_n : \mathtt{T}_n \vdash_{se} \mathtt{stmts}[\mathtt{z}_1/\mathtt{x}_1, ..., \mathtt{z}_n/\mathtt{x}_n] : \mathtt{T}$, where $\mathtt{z}_1$, ... $\mathtt{z}_n$ are fresh identifiers in $\mathtt{stmts}$ and in $\Gamma_{00}\,\Gamma$. From definition 7, it follows that $\Gamma_{00}\,(\Gamma \oplus \Gamma'), \mathtt{this} : \mathtt{C}, \mathtt{w}_1 : \mathtt{T}_1, ...\mathtt{w}_n : \mathtt{T}_n \vdash_{se} \mathtt{stmts}[\mathtt{w}_1/\mathtt{x}_1, ...\mathtt{w}_n/\mathtt{x}_n] : \mathtt{T}$, where we renamed $\mathtt{z}_1$, ... $\mathtt{z}_n$ to $\mathtt{w}_1$, ... $\mathtt{w}_n$ in order to avoid any name clashes. Therefore, applying the Java$_{se}$ type rule for method bodies, we obtain that $\Gamma_{00}\,\Gamma \oplus \Gamma', \mathtt{this} : \mathtt{C} \vdash_{se} \mathtt{mBody} : \mathtt{T}_1 \times ...\mathtt{T}_n \to \mathtt{T}$, and because the definition of $\mathtt{C}$ in $\Gamma_{00}\,\Gamma$ is identical to that in $\Gamma_{00}\,\Gamma \oplus \Gamma'$, we have that all method bodies in $\mathtt{cBody}$ satisfy their signature in $\Gamma_{00}\,\Gamma \oplus \Gamma'$. So, it holds that $\Gamma_{00}\,\Gamma \oplus \Gamma' \vdash_{se} \mathtt{cBody}\ \diamond$.

Therefore, $\Gamma_{00}\,\Gamma \oplus \Gamma' \vdash_{se} \mathtt{cBody}\ \diamond$ for any $\mathtt{cBody}$ in $\mathtt{p}''$. This, finally, gives that $\vdash_{se} (\Gamma'', \mathtt{p}'')\ \diamond$. □

From lemma 6 we see that link compatibility requires the environment modification to be a type preserving change of the original environment, and the Java$_s$ program body modification to be type correct in the new environment. The latter requirement is very easy to establish, and corresponds to a successful local compilation step. This confirms that "*reimplementing method bodies is a binary compatible change*", [10].

However, the first requirement from lemma 6, namely type preservation, is not obviously straightforward to establish, since it requires that *for all possible environments* Γ_{00}, the two environments should give the same types to *all* Java$_{se}$ expressions.

In the next section we consider restricted modifications to the environment which imply type preservation.

6 Safe changes

Safe changes are those of the changes described in [10], which apply to the language Java$_s$, *and* can be demonstrated to preserve the guarantees of binary compatibility. In particular, they do *not* include the addition of instance methods to interfaces, which was demonstrated to be problematic in section 2. The safe changes are:

- no change at all
- adding a new class $\mathtt{C}$ or interface $\mathtt{I}$ to a program, as long as the name of the new type is not the same as that of any existing type;

- changing 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 the direct super-interfaces of an interface I, 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;

and are formalized in definition 8. Remember that changing method bodies, or the names (but not the types) of the formal parameters of a method, are already considered link compatible changes because of lemma 3; therefore these changes do not need to be defined as safe changes.

Definition 8 *An environment* Γ' *is a* a safe change of *another environment* Γ*, iff:*

- *for all* Γ_0 *disjoint with* Γ':
  $\Gamma_0\,\Gamma \vdash \diamond \implies \Gamma_0\Gamma \oplus \Gamma' \vdash \diamond$

and one of the following holds:

- $\Gamma' = \epsilon$
- $\Gamma' = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_n\ \{\ fDcls,\ mDcls\}}$
  and $\mathtt{C} \notin \mathcal{C}l(\Gamma)$
- $\Gamma' = \mathtt{I\ ext\ I_1, ...I_n\ \{\ mDcls\ \}}$ *and* $\mathtt{I} \notin \mathcal{I}t(\Gamma)$
- $\Gamma' = \mathtt{C\ ext\ C''\ impl\ I_1, ...I_n\ \{\ fDcls,\ mDcls\ \}}$
  $\Gamma\ = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_n\ \{\ fDcls,\ mDcls\ \}}, \Gamma_1$
  and $\Gamma \vdash \mathtt{C''} \leq_{wdn} \mathtt{C'}$
- $\Gamma' = \mathtt{C\ ext\ C'\ impl\ I'_1, ...I'_k\ \{\ fDcls,\ mDcls\ \}}$
  $\Gamma\ = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_n\ \{\ fDcls,\ mDcls\ \}}, \Gamma_1$
  $\forall \mathtt{i} \in \{1...\mathtt{n}\} \exists \mathtt{j} \in \{1...\mathtt{k}\} :\ \Gamma' \vdash \mathtt{I'_j} \leq_{wdn} \mathtt{I_i}$
- $\Gamma' = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_m}$
  $\{\mathtt{v_1 : T_1}, ...\mathtt{v}_n : \mathtt{T}_n, \mathtt{v_{n+1} : T_{n+1}},\ \mathtt{mDcls}\}$
  $\Gamma\ = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_m}$
  $\{\mathtt{v_1 : T_1}, ...\mathtt{v_n : T_n},\ \mathtt{mDcls}\}, \Gamma_1$
- $\Gamma' = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_m}$
  $\{\mathtt{fDcls},\ \mathtt{m_1 : MT_1}, ...\ \mathtt{m_n : MT_n}, \mathtt{m_{n+1} : MT_{n+1}}\}$
  $\Gamma\ = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_m}$
  $\{\mathtt{fDcls},\ \mathtt{m_1 : MT_1}, ...\ \mathtt{m_n : MT_n}\}, \Gamma_1$

Remember that the order of declarations is not significant, therefore $\Gamma\ = \Gamma_1, \mathtt{C\ ext\ C'}...$, only means that Γ contains such a declaration of class C. The requirement $\Gamma_0\,\Gamma \vdash \diamond \implies \Gamma_0\Gamma \oplus \Gamma' \vdash \diamond$, which ensures preservation of well formedness of the environment in all appropriate contexts Γ_0, could be replaced by the stronger requirement $\Gamma\,\Gamma' \vdash \diamond$, which corresponds to requiring succesful compilation in the context of Γ. The original requirement, $\Gamma_0\,\Gamma \vdash \diamond \implies \Gamma_0\Gamma \oplus \Gamma' \vdash \diamond$, is trivially satisfied by the first five cases of definition 8. In the sixth case, which describes the addition a new field, $\mathtt{v_{n+1}}$, to a class, this field must have a different name than any of the other fields in the class, *i.e.* $\mathtt{v_{n+1}} \neq \mathtt{v}_i$ for $1 \leq i \leq n$. The seventh case describes the addition of an instance method $\mathtt{m_{n+1}}$ to a class. The new method, $\mathtt{m}_{n+1}$, may not override any of the methods already in C; if $\mathtt{m}_{n+1}$ overrides any method inherited by C from any of its superclasses, then it must have the same result type as the overriden method. This means, that either one of the superclasses of C must contain a method with identifier $\mathtt{m}_{n+1}$ and signature $\mathtt{MT}_{n+1}$, or all of the superclasses of C must be present in Γ.

The following lemma says that safe changes are type preserving.

Lemma 7 *Given environments* Γ, Γ', *if* Γ' *is a safe change of* Γ*, then* Γ' *is a type preserving change of* Γ.

Proof Take any Γ', safe change of Γ. To show that Γ' is type preserving change of of Γ.
For any environment Γ_0 disjoint from Γ', any Java$_{se}$ expression $\mathtt{e_0}$, and type $\mathtt{T_0}$, $\Gamma_0\,\Gamma \vdash_{se} \mathtt{e_0} : \mathtt{T_0}$ implies that $\Gamma_0\,\Gamma \vdash \diamond$, which implies that Γ_0 and Γ are disjoint.
Take any environment Γ_0 disjoint from Γ'.
Show for any T, T′ that $\Gamma_0\,\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T'}$ implies that $\Gamma_0\,\Gamma \oplus \Gamma' \vdash \mathtt{T} \leq_{wdn} \mathtt{T'}$, using structural induction on the proof of $\Gamma_0\,\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T'}$.
Show for any class C, that if C has in environment $\Gamma_0\,\Gamma$ a declaration of a field v with type T, then class C also has in environment $\Gamma_0\,\Gamma \oplus \Gamma'$ a declaration of field v with type T. Similarly, if class C inherits from another class C′ in environment $\Gamma_0\,\Gamma$ a declaration of a field v with type T, then class C also inherits from the class C′ in environment $\Gamma_0\,\Gamma \oplus \Gamma'$ a declaration of field v with type T. These field declarations must be unique. Any methods declared or inherited by interface I in environment $\Gamma_0\,\Gamma$, are also declared or inherited by interface I in environment $\Gamma_0\,\Gamma \oplus \Gamma'$. Finally, for any method with identifier m with argument type AT and result type T declared or inherited by class C in environment $\Gamma_0\,\Gamma$, there exists a method with identifier m with argument type AT and result type T declared or inherited by class C in environment $\Gamma_0\,\Gamma \oplus \Gamma'$.
Then show, by structural induction on the proof, that $\Gamma_0\,\Gamma \vdash_{se} \mathtt{e} : \mathtt{T}$ implies $\Gamma_0\,\Gamma \oplus \Gamma' \vdash_{se} \mathtt{e} : \mathtt{T}$. For the cases where e is a variable, an instance method call, or an instance variable access one has to apply case analysis on the contents of Γ', according to definition 8. □

In the computing students example $\Gamma^{\mathtt{cs}'}$ adds an instance variable to a class, therefore it is a safe change of $\Gamma^{\mathtt{cs}}$, and so with lemma 7, $\Gamma^{\mathtt{cs}'}$ is a type preserving change of $\Gamma^{\mathtt{cs}}$. Because type preservation automatically applies to larger environments, $\Gamma^{\mathtt{cs}'}$ is a type preserving change of $\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{st}}$. With lemma 6, $(\Gamma^{\mathtt{cs}'}, \mathtt{p}^{\mathtt{cs}}_{se})$ is a link compatible change of $(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}, \mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se})$. Similarly, $\Gamma^{\mathtt{m}}$ adds a class to environment $\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}'}\,\Gamma^{\mathtt{lab}}$, therefore it is a safe change; and so, the pair $(\Gamma^{\mathtt{m}}, \mathtt{p}^{\mathtt{m}})$ is a link compatible change of $(\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}'}\,\Gamma^{\mathtt{lab}}, \mathtt{p}^{\mathtt{st}}_{se}\,\mathtt{p}^{\mathtt{cs}}_{se}\,\mathtt{p}^{\mathtt{lab}}_{se})$.

7 Conclusions and further work

The contributions of this paper are:

- We suggest a terminology and formal framework with which to describe the effects and properties of binary compatibility.
- We define safe changes, a subset of the binary compatible changes listed in the language specification, and prove for a substantial subset of Java, that safe changes guarantee successful linking without re-compilation.
- We identify as the characteristic property of safe changes that they preserve the types of the enriched Java$_{se}$ expressions.
- We have investigated the properties of combinations of binary compatible modifications.

We expect that better formalizations will be found; indeed the formulation suggested in this paper is the result of many discussions and iterations over previous approaches [20], and we continue work in this direction. Some of the outstanding questions are described in chapter A.

Concepts for binary compatibility as proposed in [8] influenced the Java language design. Ours is the only formalization for a concrete language and proof of correctness we know of. In [2] fragments consisting of a signature and a body are used to describe linkable units, and linking consists of a type checking and a substitution phase. Our formalism distinguishes between source code and compiled code, mainly because in Java separate compilation is not equivalent to compilation of all parts together, a fact already pointed out but not pursued in [2].

We shall extend Java$_s$ to encompass a larger subset of Java, and extend safe binary compatibility to include access restrictions, static variables and methods, *etc.* 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 the computing students, *e.g.* , some classes do not need to be compiled in the complete environment, because $\mathcal{C}\{\!\{\Gamma^{\mathtt{st}}\,\Gamma^{\mathtt{cs}}\,\Gamma^{\mathtt{lab}}, \mathtt{p}^{\mathtt{st}}\}\!\} = \mathcal{C}\{\!\{\Gamma^{\mathtt{st}}, \mathtt{p}^{\mathtt{st}}\}\!\}$.

It would be interesting to recast some of this work in terms of a formal description of the Java byte-code and byte-code verifier (such as [15, 9]). The fact that separate compilation of the types is not equivalent to compilation of all types together can be seen as another case of lack of full abstraction property in language translation, which, as shown in [1] may lead to loss of protection. It remains to investigate how far problems with binary compatibility can be understood in these terms.

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

Acknowledgements

We acknowledge the financial support from the EPSRC (Grant Refs: GR/L 76709 and GR/K 73282). We are grateful to Guy Steele for valuable feedback, to Gabrielle Sinnadurai, David von Oheimb and to the anonymous ECOOP and OOPSLA referees, and most particularly to one of them, for useful and detailed suggestions on the presentation.

References

[1] Martin Abadi. Protection in Programming Language Translations. In *ICALP'98 Proceedings*. Springer Verlag, 1998. to appear, also available at: http://gatekeeper.dec.com/pub/DEC/SRC /research-resports/abstracts/src-rr-154.html.

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

[3] 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.

[4] 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.

[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 and Susan Eisenbach. Java is type safe — probably. In *Proceedings of the European Conference on Object-Oriented Programming*, June 1997.

[7] Sophia Drossopoulou and Susan Eisenbach. Towards an Operational Semantics and a Proof of Type Soundness for Java. In Jim Alvez Foss, editor, *Formal Syntax and Semantics of Java*. Springer Verlag Lecture Notes in Computer Science, 1998. to appear, available at http://www-dse.doc.ic.ac.uk/projects/slurp/.

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

[9] Allen Goldberg. A Specification of Java Loading and Bytecode Verification. Technical report, Kestrel Institute, December 1997.

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

[11] James Gosling and H. McGilton. The Java Language Environment A White Paper, http://java.sun.com/docs/white/langenv, 1996.

[12] Tim Lindholm and Frank Yellin. *The Java Virtual Machine*. Addison-Wesley, 1997.

[13] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *ECOOP'98 Proceedings*. Springer Verlag, 1998. to appear.

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

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

[16] Guy Steele. Private Communication, January 1998.

[17] 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.

[18] US Department of Defense. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815 A.

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

[20] David Wragg, Sophia Drossopoulou, and Susan Eisenbach. Java binary compatibility is almost correct. Technical Report 3/98, Imperial College Department of Computing, February 1998. available at http://www-dse.doc.ic.ac.uk/projects/slurp/.

Appendix

A Modelling link compatibility

In this section we discuss the concept of link compatibility, analyze and justify our approach, and give alternative definitions. As we said earlier, link compatibility was introduced to capture the guarantee of binary compatibility. Consider again the description from the Java language specification:

> "*A change to a type is binary compatible with (equivalently, does not break compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error.*"

A.1 The issues

Five issues arose when considering the formalization of the above description:

- representation of "binaries";
- representation of "change";
- the extent of the role of the pre-existing binaries;
- the number of "pre-existing binaries" involved;
- representation of "linking" and "linking without error";

which we shall discuss in some detail.

The representation of "binaries" In most current Java implementations binaries are Java byte-code programs (*i.e.* `.class` files) However, this does not have to be so; indeed, any code satisfying the requirements outlined in ch 13.1 of the Java specification may be used. Furthermore, the byte-code is at a different level of abstraction from most programmers' view of Java. Therefore, we represent "binaries" as Java_{se} bodies. Java_{se} has the advantage of having a type system, and of containing all necessary information for execution.

The representation of "change" Since Java programs are represented by environment and body pairs, change consists of a new environment and body. Should the body of the change be a Java_{se} or a Java_{s} body? We chose to have Java_{s} bodies, because this models more accurately source code modifications as introduced by a programmer, and also expresses the fact that binary compatible changes allow parts of a program to have been compiled with different versions of the environment.

The extent of the role of the pre-existing binaries In how far is the context $\mathtt{F}_0$ crucial for the compilation of the modification $\mathtt{F}'$? Do we allow the modifications to depend on contexts? Our answer is yes, because we want to model modifications to libraries that are not stand-alone. This is why in definition 6 we require

$$\vdash_{se}\mathtt{F}_0 \circ \mathtt{F} \ \diamond \quad \Longrightarrow \quad \vdash_{se}(\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}' \ \diamond$$

as opposed to the stronger requirement

$$\vdash_{se}\mathtt{F}_0 \circ \mathtt{F} \ \diamond \quad \Longrightarrow \quad \vdash_{se}\mathtt{F}_0 \circ (\mathtt{F} \oplus_c \mathtt{F}') \ \diamond.$$

The number of "pre-existing binaries" involved The term "pre-existing binaries" is used twice in the quote from before, but it is not necessarily clear, how many different pre-existing binaries are involved. Either *one* set is involved, meaning:

> A change is binary compatible with pre-existing binaries if these pre-existing binaries link without error and continue to do so after the change.

or, *two* sets are involved, meaning:

> A change is binary compatible with pre-existing binaries if any further pre-existing binaries that link without error with the former pre-existing binaries continue to do so after the change to the former pre-existing binaries.

We have chosen the second interpretation, and distinguish $\mathtt{F}$, the binaries being modified, from $\mathtt{F}_0$, the "context" binaries that linked without error with $\mathtt{F}$.

In definition 6 the modifications $\mathtt{F}'$ are considered link compatible for $\mathtt{F}$, iff for all contexts $\mathtt{F}_0$, such that $\mathtt{F}$ and $\mathtt{F}_0$ linked without error, the effect of $\mathtt{F}'$ onto $\mathtt{F}$ will link with $\mathtt{F}_0$ without error. However, in section A.2 we shall discuss the repercussions of considering *one set* of pre-existing binaries.

The representation of "linking", and of "linking without error" Linking is described in some detail in 12.3 of [10], as a process taking place after loading, and consisting of *verification*, *preparation* and *resolution* of symbolic references. Verification ensures that a binary is structurally correct; for the byte-code it is described in some detail in [12] and also in [15]. Preparation involves creation of static fields and their initialization to default values. Resolution involves checking symbolic references (containing type information) to methods and fields of other classes and replacing them by more direct references [10].

A formal description of the linker requires the development of more formal apparatus, *e.g.* [9]. However, for the purposes of the current investigation, we do not need a complete description of the linking process, because we clearly are not interested in the *outcome* of the linker, we are only interested in the possible *errors reported* by it. All checks performed during verification and resolution correspond to checking type correctness of Java$_{se}$ terms.

Thus, we claim for Java$_{se}$ fragments $\mathtt{F}_1$, $\mathtt{F}_2$, that if $\vdash_{se}\mathtt{F}_1 \ \diamond$, then the code corresponding to $\mathtt{F}_1$ would pass the verifier checks, and if $\vdash_{se}\mathtt{F}_1 \circ \mathtt{F}_2 \ \diamond\!\!\diamond$, then all symbolic references in the code corresponding to $\mathtt{F}_1$ and $\mathtt{F}_2$ would be successfully resolved. Therefore, the requirement $\vdash_{se}\mathtt{F}_1 \circ \mathtt{F}_2 \ \diamond$ together with the requirements that all declared classes have a class body, adequately represents "linking without error". In section A.2 we shall discuss the repercussions of an alternative representation of "linking without errors' through *run-time safety*, a property whereby program execution will never raise linker-related exceptions *c.f.* definition 10.

A.2 Alternative definitions

The approach described in the main body of this paper represents a certain stance on the issues identified above, one which we have found to be the most reasonable and fruitful. Naturally we have given some consideration to other possibilities, and in this section we compare three alternatives to definition 6, which correspond to different answers to the last two of the five issues.

We consider the representation of "linking without error" either through type-safety of the program, or though the run-time safety, For the number of pre-existing binaries, we consider the cases where either *one* or *two* sets are taken into account. This produces the following four alternatives:

pre-existing binaries / *linking without error*	*two*	*one*
type safe	link compatible	weak link compatible
run-time safe	global link compatible	local link compatible

Definition 9 describes a variation of link compatibility where we consider a modification $\mathtt{F}'$ with respect to some *specific* pre-exiting binaries $\mathtt{F}$ only, and require the result to link without error:

Definition 9 *A Java$_s$ fragment* $\mathtt{F}'$ *is a* weak link compatible change *of a Java$_{se}$ fragment* $\mathtt{F}$, *iff*

$$\vdash \mathtt{F} \oplus_c \mathtt{F}' \ \diamond$$

This definition would allow the removal of a method from a class, provided that that method were not called inside any of the method bodies in $\mathtt{F}$. Therefore, this definition is appropriate only in cases where we have an exact knowledge of the classes which we want to link with the modified classes. For well-formed fragments link compatibility implies weak link compatibility.

Lemma 8 *If a Java$_s$ fragment* $\mathtt{F}'$ *is a link compatible change of a Java$_{se}$ fragment* $\mathtt{F}$*, and* $\vdash \mathtt{F} \diamond$*, then* $\mathtt{F}'$ *is a weak link compatible change of* $\mathtt{F}$*.*

We shall now consider an alternative representation of "links without error", in terms of the *run-time behaviour* of the resulting program, whereby we call a Java$_{se}$ program *run-time safe* if its execution does not cause the exceptions that would be detected by a linker (*i.e.* absence of a method body, or absence of a field).

We call *linker exceptions* those exceptions that could be raised by resolution; these are `AbstractMethodError`, `IllegalAccessError`, `InstantiationError`, *etc.* In other words, execution of a run-time safe program may terminate, or may halt or because of a predefined or user defined exception, but *not* because an appropriate body or field was absent.

Definition 10 *A Java$_{se}$ fragment* $\mathtt{F} = (\Gamma, \mathtt{p})$ *is* run-time safe *iff, for all terms* $\mathtt{t}$*, states* σ*, with execution of* $\mathtt{p}$ *leads to configuration* $\langle \mathtt{t}, \sigma \rangle$:

- $\mathtt{t} = \mathtt{throw}\ \iota_{\mathtt{i}},\ \sigma(\iota_{\mathtt{i}}) = \ll \ldots \gg^{E} \implies$
  $\mathtt{E}$ *is* not *a linker exception.*

The subject reduction theorem implies that type safety and completeness guarantee run-time safety.

Conjecture 1 *If* $\vdash_{\bar{s}e} \mathtt{F} \Diamond\!\!\!\!\times$*, then* $\mathtt{F}$ *is run-time safe.*

Our next attempt at a formal definition of the guarantee of binary compatibility will be in terms of run-time safety. In definition 11 we only consider one set of pre-existing binaries, whereas in definition 12 we consider two.

Definition 11 *A Java$_s$ fragment* $\mathtt{F}'$ *is a* local link compatible change *of a Java$_{se}$ fragment* $\mathtt{F}$*, iff*

$$\mathtt{F} \oplus_c \mathtt{F}' \text{ is run-time safe.}$$

Therefore, provided that $\mathtt{F} \oplus_c \mathtt{F}'$ is run-time safe, $\mathtt{F}'$ is a local link compatible change, even if $\vdash_{\bar{s}e} \mathtt{F} \oplus_c \mathtt{F}' \diamond$ did not hold! Thus local link compatibility seems to guarantee no more than what is required. The above definition would allow the addition of a method to an interface, provided that this method was never called from $\mathtt{F}$; this corresponds to the second phase from our example in section 2.2. However, we see no practical way of ensuring that a change satisfies the local link compatible change property. More importantly, after a local link compatible change and a locally type correct compilation run-time safety is not guaranteed any more, as demonstrated by the third phase of the example from section 2.2.

Therefore, a type correct compilation cannot be considered a local link compatible step, and a type-correct compilation of a new fragment $\mathtt{F}'$ does not guarantee run-time safety, unless the original fragment $\mathtt{F}$ was type correct:

Conjecture 2 *If a Java$_s$ fragment* $\mathtt{F}'$ *is weak link compatible change of a Java$_{se}$ fragment* $\mathtt{F}$*, then* $\mathtt{F}'$ *is a local link compatible change of* $\mathtt{F}$*.*

The opposite direction of the implication does not hold. For example, the addition of a method to an interface, although a local link compatible change, does not always create a type correct fragment and therefore is not *not* weak link compatible.

The requirement of local link compatibility is weak, because it cannot guarantee much after subsequent locally type correct compilations. In the next definition we require the property of run-time safety to be preserved in all appropriate contexts, and by subsequent locally type-correct compilations of class bodies.

Definition 12 *A Java$_s$ fragment* $\mathtt{F}'$ *is a* global link compatible change *of a Java$_{se}$ fragment* $\mathtt{F}$*, iff for all Java$_s$ fragments* $\mathtt{F}''$*, Java$_{se}$ bodies* $\mathtt{p}''$*, Java$_{se}$ fragment* $\mathtt{F}'' = (\epsilon, \mathtt{p}'')$*, where* $\mathtt{F}_0$ *disjoint from* $\mathtt{F}'$, $\mathtt{F}''$:

$$\begin{gathered} \mathtt{F}_0 \circ \mathtt{F} \text{ is run-time safe} \\ \implies \\ (\mathtt{F}_0 \circ \mathtt{F}) \oplus_c \mathtt{F}' \oplus_c \mathtt{F}'' \text{ is run-time safe} \\ (\text{or is undefined}). \end{gathered}$$

Thus, the addition of a method to an interface is not a global link compatible change even if this method were not called in $\mathtt{F}$, $\mathtt{F}_0$ or $\mathtt{F}'$, as it may be called in a subsequent modification $\mathtt{F}''$. Global link compatible changes are local link compatible changes.

Lemma 9 *If a Java$_s$ fragment* $\mathtt{F}'$ *is global link compatible change of a Java$_{se}$ fragment* $\mathtt{F}$*, then* $\mathtt{F}'$ *is a local link compatible change of* $\mathtt{F}$*.*

It seems to us that global link compatibility is the weakest possible description of the guarantee of binary compatibility. It remains open, in how far global link compatibility is equivalent to link compatibility, and if it is not, whether there are useful cases covered by one but not the other. The following diagram summarizes the relationship between the four definitions given in this section:

$$\begin{tikzcd}
\text{link compatible} \arrow[r, Rightarrow, "\text{if } \vdash_{\bar{s}e} \mathtt{F} \diamond"'] \arrow[d, Rightarrow, shift left, "?"] & \text{weak link compatible} \arrow[d, Rightarrow] \\
\text{global link compatible} \arrow[u, Rightarrow, shift left, "?"] \arrow[r, Rightarrow, "\text{if } \mathtt{F} \text{ run-time safe}"'] & \text{local link compatible}
\end{tikzcd}$$

B The syntax of environments

$$
\begin{array}{lll}
\mathit{Env} & ::= & [\ \mathit{StandardEnv}\ ;\ ]\ \mathit{Decls} \\
\mathit{StandardEnv} & ::= & \texttt{Exception ext Object}\ldots\texttt{NullPE ext Exception}\ldots\texttt{;}\ \ldots \\
\mathit{Decls} & ::= & \mathit{Decl}\ ;\ \mathit{Decls}\ \mid\ \epsilon \\
\mathit{Decl} & ::= & \texttt{ClassId ext ClassName impl }(\texttt{InterfName})^* \\
\mathit{VarType} & ::= & \mathit{SimpleType}\ \mid \mathit{ArrayType} \\
\mathit{SimpleType} & ::= & \mathit{PrimType}\ \mid \texttt{ClassName}\ \mid \texttt{InterfaceName} \\
\mathit{ArrayType} & ::= & \mathit{SimpleType}\texttt{[ ]}\ \mid \mathit{ArrayType}\texttt{[ ]} \\
 & \mid & \texttt{InterfaceName} \\
\mathit{PrimType} & ::= & \texttt{bool}\ \mid \texttt{char}\ \mid \texttt{int}\ \mid \ldots \\
\mathit{Type} & ::= & \mathit{VarType}\ \mid\ \texttt{void}\ \mid\ \texttt{nil} \\
 & & \texttt{\{}(\texttt{VarId} : \mathit{VarType})^*\ (\texttt{MethId} : \mathit{MethType})^*\texttt{\}} \\
 & \mid & \texttt{InterfId ext InterfName}^*\texttt{\{}(\texttt{MethId} : \mathit{MethType})^*\texttt{\}} \\
 & \mid & \texttt{VarId} : \mathit{VarType} \\
\mathit{MethType} & ::= & \mathit{ArgType} \rightarrow (\mathit{VarType} \mid \texttt{void}) \\
\mathit{ArgType} & ::= & [\mathit{VarType}\ (\times VarType)^*]
\end{array}
$$

C The syntax of Java$_s$

$$
\begin{array}{lll}
\mathit{ProgramBody} & ::= & (\ \mathit{ClassBody}\ )^* \\
\mathit{ClassBody} & ::= & \texttt{ClassId ext ClassName \{}(\ \mathit{MethBody}\ )^*\texttt{\}} \\
\mathit{MethBody} & ::= & \texttt{MethId is } (\lambda\ \texttt{ParId} : \mathit{VarType.})^* \\
 & & \texttt{\{}\mathit{Stmts}\ ;\ \texttt{return}\ [\mathit{Expr}]\ \texttt{\}} \\
\mathit{Stmts} & ::= & \mathit{Stmt}\ \mid \mathit{Stmts}\ ;\ \mathit{Stmt} \\
\mathit{Stmt} & ::= & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stmts}\ \texttt{else}\ \mathit{Stmts} \\
 & \mid & \mathit{Var} = \mathit{Expr}\ \mid\ \mathit{Expr}\ \mid\ \texttt{throw}\ \mathit{Expr} \\
 & \mid & \texttt{try}\ \mathit{Stmts}\ (\texttt{catch ClassName Id}\ \mathit{Stmts})^*\ \texttt{finally}\ \mathit{Stmts} \\
 & \mid & \texttt{try}\ \mathit{Stmts}\ (\texttt{catch ClassName Id}\ \mathit{Stmts})^+ \\
\mathit{Expr} & ::= & \mathit{Value}\ \mid\ \mathit{Var}\ \mid \\
 & & \mathit{Expr}.\texttt{MethName}\ (\ \mathit{Expr}^*)\ (\texttt{[}\ \mathit{Expr}\ \texttt{]})^+(\texttt{[ ]})^* \\
\mathit{Var} & ::= & \texttt{Name}\ \mid\ \mathit{Var}.\texttt{VarName}\ \mid\ \mathit{Var}\texttt{[}\mathit{Expr}\texttt{]}\ \mid\ \texttt{this} \\
\mathit{Value} & ::= & \mathit{PrimValue}\ \mid\ \texttt{null} \\
\mathit{PrimValue} & ::= & \mathit{intValue}\ \mid \mathit{charValue}\ \mid \mathit{byteValue}\ \mid \ldots
\end{array}
$$

D Some of the Java$_s$ type checking rules

$$
\frac{\Gamma \vdash \diamond \qquad \texttt{i} \text{ is integer}, \quad \texttt{c} \text{ is character}, \quad \texttt{x} \text{ is identifier}}
{\begin{array}{l}\Gamma \vdash \texttt{null} : \texttt{nil}, \quad \Gamma \vdash \texttt{true} : \texttt{bool}, \quad \Gamma \vdash \texttt{false} : \texttt{bool}, \quad \Gamma \vdash \texttt{i} : \texttt{int}, \quad \Gamma \vdash \texttt{c} : \texttt{char}, \quad \Gamma \vdash \texttt{x} : \Gamma(\texttt{x}) \\ \mathcal{C}\{\!\!\{\Gamma, \texttt{z}\}\!\!\} = \texttt{z} \quad \text{if } \texttt{z} \text{ is integer, character, identifier, } \texttt{null}, \ \texttt{true}, \text{ or } \texttt{false}\end{array}}
$$

$$
\frac{\begin{array}{l}\Gamma \vdash \texttt{v} : \texttt{T} \\ \Gamma \vdash \texttt{e} : \texttt{T}' \\ \Gamma \vdash \texttt{T}' \leq_{wdn} \texttt{T}\end{array}}
{\begin{array}{l}\Gamma \vdash \texttt{v} := \texttt{e} : \texttt{void} \\ \mathcal{C}\{\!\!\{\Gamma, \texttt{v} := \texttt{e}\}\!\!\} = \mathcal{C}\{\!\!\{\Gamma, \texttt{v}\}\!\!\} := \mathcal{C}\{\!\!\{\Gamma, \texttt{e}\}\!\!\}\end{array}}
\qquad
\frac{}{\begin{array}{l}\Gamma \vdash \texttt{return} : \texttt{void} \\ \mathcal{C}\{\!\!\{\Gamma, \texttt{return}\}\!\!\} = \texttt{return}\end{array}}
$$

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

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

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

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

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

$$\frac{\begin{array}{l}\mathtt{n} \geq 0, \mathtt{k} \geq 0, \mathtt{m} \geq 0, \Gamma \vdash \Gamma \diamond \\ \Gamma(\mathtt{C}) = \mathtt{C\ ext\ C'\ \ impl\ I_1...I_n}\{\mathtt{v_1} : \mathtt{T_1}...\mathtt{v_k} : \mathtt{T_k}, \mathtt{m_1} : \mathtt{MT_1}...\mathtt{m_l} : \mathtt{MT_l}\} \\ \mathtt{cBody} = \mathtt{C\ ext\ C'}\ \{\mathtt{mBody_1}, ...\mathtt{mBody_l}\}, \qquad \mathtt{stmts}' = \mathtt{stmts}[\mathtt{z_1}/\mathtt{x_1}, ..., \mathtt{z_n}/\mathtt{x_n}] \\ \Gamma(\mathtt{this}) = \mathtt{Undef} \\ \mathtt{mBody_i} = \mathtt{m_i\ is\ mPrsSts_i} \quad \mathtt{i} \in \{1...\mathtt{l}\} \\ \Gamma, \mathtt{this} : \mathtt{C} \vdash \mathtt{mBody_i} : \mathtt{MT_i} \quad \mathtt{i} \in \{1...\mathtt{l}\}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{cBody} \diamond \\ \mathcal{C}\{\!\{\Gamma, \mathtt{cBody}\}\!\} = \mathtt{C\ \ ext\ C'}\ \{\mathcal{C}\{\!\{\Gamma, \mathtt{mBody_1}\}\!\}...\mathcal{C}\{\!\{\Gamma, \mathtt{mBody_l}\}\!\}\}\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{p} = \mathtt{p_1 p_2} \implies \mathcal{C}l(\mathtt{p_1}) \cap \mathcal{C}l(\mathtt{p_2}) = \emptyset \\ \mathtt{n} \geq 0, \quad \mathtt{p} = \mathtt{cBody_1}, ...\mathtt{cBody_n} \\ \mathtt{cBody_i} = \mathtt{C_i\ ext\ }...\{...\} \quad \text{for } \mathtt{i} \in \{1...\mathtt{n}\} \\ \Gamma \vdash \mathtt{cBody_i} \diamond \quad \mathtt{i} \in \{1...\mathtt{n}\}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{p} \diamond \\ \mathcal{C}\{\!\{\Gamma, \mathtt{p}\}\!\} = \mathcal{C}\{\!\{\Gamma, \mathtt{this} : \mathtt{C}, \mathtt{cBody_1}\}\!\}...\mathcal{C}\{\!\{\Gamma, \mathtt{this} : \mathtt{C}, \mathtt{cBody_n}\}\!\}\end{array}}$$

$$\frac{\Gamma \vdash \mathtt{p} \diamond}{\vdash (\Gamma, \mathtt{p}) \diamond}$$

$$\frac{\mathcal{C}l(\Gamma) = \mathcal{C}l(\mathtt{p}) \quad \Gamma \vdash \mathtt{p} \diamond}{\Gamma \vdash \mathtt{p} \otimes}$$

E Altering the syntax of Java$_s$ to obtain Java$_{se}$ syntax

Expr	::=	...	
	|	*Expr*.[*ArgType*]`MethName`(*Expr**)	*replaces Expr*.`MethName`(*Expr**)
	|	*Stmts*	
Var	::=	...	
	|	*Var*.[`ClassName`]`VarName`	*replaces Var*.`VarName`

F Some of the Java$_{se}$ type checking rules

$$\frac{\Gamma \vdash_{se} \mathtt{v} : \mathtt{T} \quad \Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{C} \quad \mathit{FDec}(\Gamma, \mathtt{C}, \mathtt{f}) = (\mathtt{C}, \mathtt{T}')}{\Gamma \vdash_{se} \mathtt{v}.[\mathtt{C}]\mathtt{f} : \mathtt{T}'}$$

$$\frac{\Gamma \vdash_{se} \mathtt{e_i} : \mathtt{T_i'} \quad \mathtt{i} \in \{1...\mathtt{n}\}, \mathtt{n} \geq 0 \quad \Gamma \vdash \mathtt{T_i'} \leq_{wdn} \mathtt{T_i} \quad \mathtt{i} \in \{2...\mathtt{n}\} \quad \mathit{FirstFit}(\Gamma, \mathtt{m}, \mathtt{T_1'}, \mathtt{T_2} \times ... \times \mathtt{T_n}) = \{(\mathtt{T}, \mathtt{MT})\}}{\Gamma \vdash_{se} \mathtt{e_1}.[\mathtt{T_2} \times ... \times \mathtt{T_n}]\mathtt{m}(\mathtt{e_2}...\mathtt{e_n}) : \mathit{Res}(\mathtt{MT})}$$