A Fragment Calculus
– towards a model of Separate Compilation, Linking and Binary Compatibility

Sophia Drossopoulou, Susan Eisenbach and David Wragg
Department of Computing, Imperial College, 180 Queen's Gate, LONDON SW7 2BZ, England
email: sd,se,dpw@doc.ic.ac.uk

Abstract

We propose a calculus describing compilation and linking in terms of operations on fragments*, i.e. compilation units, without reference to their specific contents. We believe this calculus faithfully reflects the situation within modern programming systems.*

Binary compatibility *in Java prescribes conditions under which modification of fragments does not necessitate re-compilation of importing fragments. We apply our calculus to formalize binary compatibility, and demonstrate that several interpretations of the language specification are possible, each with different ramifications. We choose a particular interpretation, justify our choice, formulate and prove properties important for language designers and code library developers.*

1. Introduction

Separate compilation and linking, although supported by most language implementations, is under-specified in most language descriptions [3]. In the traditional arrangement in languages such as Ada [22, 4] or Modula-2 [23], the compiler checks for type consistency and the linker resolves references and checks the order of compilation. Any units importing modified units have to be re-compiled, and so separate compilation of several units corresponds to the compilation of all units together. Thus, the situation was sufficiently simple for this under-specification not to pose major problems.

However, there exist languages and systems where separate compilation and linking are complex, and justify a formal treatment. For instance in Java [12], because of its intended support for loading and executing remotely produced code whose source code is not necessarily accessible, solutions enforcing consistency through re-compilation are not suitable. Instead, the remit of the linker has been extended: not only does it have to resolve external references, it also has to ensure that binaries (the compiled units) are structurally correct and that they respect the types of entities they import from other binaries; the order of compilation need not correspond to the import relation.

Certain source code modifications, such as adding a method to a class, are *binary compatible* [8]. The Java language description does not require re-compilation of units importing units 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 units, 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. Separate compilation is *not* equivalent to compilation of all units together.

Binary compatibility has practical importance because of related security issues [5], and implications on library modification policies. It is quite complex – the language specification is sometimes inconsistent, as it considers some changes to be binary compatible, whose combination actually leads to programs which cannot link [7].

Formalizations of such issues tend to suggest calculi describing the underlying source and binary languages, and define modularization and linking in terms of these calculi. We now believe that such approaches have serious disadvantages:

- It is rather cumbersome to establish that a full-fledged calculus (*e.g.* [6, 1, 19, 17]) faithfully reflects the properties of a real language (*e.g.* Java) with respect to linking and separate compilation.
- Such calculi are at an inappropriate level of abstraction. Rather than think in terms of the particular language features, language designers think in terms of "programming in the large" and of properties satisfied by linking and compilation; library developers think in terms of linking capabilities of libraries.

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 — code

In this paper we explore a different avenue: We give an axiomatic definition compilation, linking, well-formedness for source and binary languages, and require some locality properties. We believe that our model distills the essential definitions and properties and reflects the situation in most real programming languages. Also we have taken into account feedback from Sun Java developers [2].

We use this model to formalize what it means for a source code modification to be a binary compatible change. We discovered that several interpretations of the definition in [12] are possible, and discuss their ramifications. We suggest the best interpretation in our view, and we prove properties which allow binary compatible modifications to be applied to interdependent libraries and preserve their linking capabilities. Thus, we clarify the issues around binary compatibility, and we offer a simple and abstract model.

The paper is organized as follows: In section 2 we introduce Java binary compatibility. In section 3 we describe a generic model of compilation and linking, in terms of a calculus of fragments. In section 4 we extend this model to describe updating and compiling into fragments. In sections 5 and 6 we define link compatibility, formulate and prove its properties. 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 [13], and in particular, to avoid the *fragile base class problem*, found in most C++ implementations, where a field (data member or instance variable) access is compiled into an offset from the beginning of the object, fixed at compile-time. If new fields are added and the class is re-compiled, then offsets may change, and object code that previously compiled using the original definition of the class may not execute safely together with the object code of the modified class. Similar problems may arise with virtual function calls.

Development environments usually attempt to compensate by automatically re-compiling all units importing the modified units; however, this strategy would be too restrictive in some cases. 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.

The example in 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. The class CStudent inherits the instance variable grade of type int. In class Lab the field guy, of class CStudent, is assigned grade 100. This program is well-formed and compiles producing 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 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; *cf.* figure 3, where guy[Student].grade represents the first and guy[CStudent].grade represents the second access.

Similar situations can arise for method calls.

3. Fragments

As in [3], we consider *fragments* as the basic units participating in compilation and linking. They represent parts of programs or libraries, and they need *not* be self-contained. The exact nature of fragments is language dependent: In Java fragments would be classes and interfaces, in Ada fragments would be packages, in Modula-2 fragments would be modules, *etc*.

For the current discussion we are not interested in the contents of the fragments. However, as we are interested in compilation and linking we distinguish $\mathcal{S}$ fragments from $\mathcal{B}$ fragments, where:

- $\mathcal{S}$ is the *source language*,
- $\mathcal{B}$ is the *binary language* containing all necessary information for execution and for compilation of importing fragments.

$\mathcal{S}$ may stand for Java, Pascal, Ada, *etc*. $\mathcal{B}$ may stand for the Java class files, the Modula-2 `.o` and `.sym` files, *etc*. In our previous work applied to Java[7, 6], $\mathcal{S}$ was represented by Java$_s$-program$\times$environment pairs, $\mathcal{B}$ was represented by Java$_{se}$-program$\times$environment pairs. Possible $\mathcal{S}$ fragments for the students example are shown in figure 2, and possible (high level) $\mathcal{B}$ fragments are shown in figure 3. The difference between the fragments in figure 2 and 3 is, that field accesses in the latter are enriched with information necessary for execution.

A *fragment system* distills the basic concepts necessary for the description of compilation and linking:

Definition 1 *A tuple* ($\mathcal{S}$, $\mathcal{B}$, ϵ, $\mathtt{Ids}$, $\mathcal{C}$, $\_+\_$, $\mathcal{D}$, $\_ \vdash \_ \diamond$, $\_ \vdash_{\mathcal{B}} \_ \diamond$) *is a* fragment system, *iff*

- $\mathcal{S}$, $\mathcal{B}$, $\mathtt{Ids}$ *are sets,* $\mathcal{S} \cap \mathcal{B} = \{\epsilon\}$
- $\mathcal{C}$ *is a mapping,* $\quad \mathcal{C} : \ \mathcal{S} \times \mathcal{B} \longrightarrow \mathcal{B}$
- $\_+\_$ *is a commutative, associative operator,* $\_+\_ : \ \mathcal{S} \times \mathcal{S} \longrightarrow \mathcal{S} \quad \cup \quad \mathcal{B} \times \mathcal{B} \longrightarrow \mathcal{B}$, *with* ϵ *as the identity element*
- $\mathcal{D}$ *a mapping,* $\quad \mathcal{D} : \ \mathcal{S} \cup \mathcal{B} \longrightarrow \mathtt{Ids}$
- $\_ \vdash \_ \diamond$ *is a relation in* $\mathcal{B} \times \mathcal{S}$, *and* $\_ \vdash_{\mathcal{B}} \_ \diamond$ *is a relation in* $\mathcal{B} \times \mathcal{B}$

and the axioms 1, 2, 3, 4 (given later) are satisfied. The elements of $\mathcal{S} \cup \mathcal{B}$ *are called* fragments.

The sets $\mathcal{S}$, $\mathcal{B}$ and $\mathtt{Ids}$ stand, respectively, for the source language, the binary language and a set of identifiers. The ϵ fragment denotes either an empty $\mathcal{S}$ fragment or an empty $\mathcal{B}$ fragment. $\mathcal{S}$ fragments are compiled into $\mathcal{B}$ fragments by the total function $\mathcal{C}$[1] using environment information from imported $\mathcal{B}$ fragments. Several fragments may be put together to form larger fragments, using the linking operator $+$. The total function $\mathcal{D}$ extracts the identifiers of all entities declared at the outer level of fragments. For $\mathcal{S}$ fragment $\mathtt{S}$, $\mathcal{B}$ fragment $\mathtt{B}$, the predicate $\mathtt{B} \vdash \mathtt{S} \diamond$ represents the checks performed on $\mathtt{S}$ by the compiler in the environment of type information from $\mathtt{B}$, whereas the predicate $\mathtt{B}_1 \vdash_{\mathcal{B}} \mathtt{B}_2 \diamond$ represents the checks performed by the linker.

$\mathcal{S}$	$\xrightarrow[\mathcal{C}]{}$	$\mathcal{B}$	$\xrightarrow[+]{}$	$\mathcal{B}$
compiler checks $\mathtt{B} \vdash \mathtt{S} \diamond$				linker checks $\mathtt{B}_1 \vdash_{\mathcal{B}} \mathtt{B}_2 \diamond$

$\mathcal{S}$ fragments will be named $\mathtt{S}$, $\mathtt{S}'$, $\mathtt{S}_1$, *etc*, $\mathcal{B}$ fragments will be named $\mathtt{B}$, $\mathtt{B}'$, $\mathtt{B}_1$, *etc*, and fragments which may belong to either $\mathcal{S}$ or $\mathcal{B}$ will be named $\mathtt{F}$, $\mathtt{F}'$ *etc*. In the remainder of this section we discuss the operations and predicates $\mathcal{C}$, $+$, $\mathcal{D}$, $\vdash \ \diamond$, $\vdash_{\mathcal{B}} \ \diamond$ in more detail, and formulate requirements on these in terms of axioms.

3.1. Compilation and well-formedness

Compilation ($\mathcal{C}$) of $\mathcal{S}$ code produces $\mathcal{B}$ code using environment information from $\mathcal{B}$ code. Thus, $\mathcal{B}$ is expected to contain two different kinds of information: the first is code for the execution of the particular fragment, the second is environment information for the compilation of importing fragments. In many language implementations this is stored in different formats and in different files – *e.g.* the `.o` and the `.sym` files of some Modula-2 implementations. However, since compilation produces both kinds of information, no generality is lost by not distinguishing them, and by expecting $\mathcal{B}$ to contain execution *and* environment information. We expect $\mathcal{C}\{\mathtt{S}^{\mathtt{st}}, \epsilon\} = \mathtt{B}^{\mathtt{st}}$, $\mathcal{C}\{\mathtt{S}^{\mathtt{lab}}, \mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}}\} = \mathtt{B}^{\mathtt{lab}}$, and $\mathcal{C}\{\mathtt{S}^{\mathtt{lab}}, \mathtt{B}^{\mathtt{st}}\} = \epsilon = \mathcal{C}\{\mathtt{S}^{\mathtt{lab}}, \mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}'}\}$.

We shall use the assertion $\vdash_{\mathcal{B}} \mathtt{B} \diamond$ as a shorthand for $\mathtt{B} \vdash_{\mathcal{B}} \mathtt{B} \diamond$. The first axiom expresses the requirement that empty fragments are compiled into empty fragments, and well-formedness of a non-empty $\mathcal{S}$ fragment in the environment of a $\mathcal{B}$ fragment is equivalent to well-formedness of the corresponding non-empty compilation.

Axiom 1 *For any* $\mathtt{B}$:

- $\mathcal{C}\{\epsilon, \mathtt{B}\} = \epsilon$

[1] In previous work $\mathcal{C}$ was a partial function; this distinction has important repercussions for the concept of binary compatibility.

1st phase

$\mathtt{S}^{\mathtt{st}}$ = `class Student { int grade; }`

$\mathtt{S}^{\mathtt{cs}}$ = `class CStudent extends Student { }`

$\mathtt{S}^{\mathtt{lab}}$ =
```
class Lab {
    CStudent guy;
    void f(){ guy.grade=100; }
  }
```

2nd phase

$\mathtt{S}^{\mathtt{st}}$ = *as in 1st phase*

$\mathtt{S}^{\mathtt{cs}'}$ =
```
class CStudent extends Student {
    char grade;
  }
```

$\mathtt{S}^{\mathtt{lab}}$ = *as in 1st phase*

3rd phase

$\mathtt{S}^{\mathtt{st}}$ = *as in 1st and 2nd phase*

$\mathtt{S}^{\mathtt{cs}'}$ = *as in 2nd phase*

$\mathtt{S}^{\mathtt{lab}}$ = *as in 1st and 2nd phase*

$\mathtt{S}^{\mathtt{m}}$ =
```
class Marker {
    CStudent guy;
    void g(){ guy.grade='A'; }
  }
```

Figure 2. Students — source fragments

1st phase

$\mathtt{B}^{\mathtt{st}}$ = `class Student { int grade; }`

$\mathtt{B}^{\mathtt{cs}}$ = `class CStudent extends Student { }`

$\mathtt{B}^{\mathtt{lab}}$ =
```
class Lab {
    CStudent guy;
    void f(){ guy[Student].grade=100; }
  }
```

2nd phase

$\mathtt{B}^{\mathtt{st}}$ = *as in 1st phase*

$\mathtt{B}^{\mathtt{cs}'}$ =
```
class CStudent extends Student {
    char grade;
  }
```

$\mathtt{B}^{\mathtt{lab}}$ = *as in 1st phase*

3rd phase

$\mathtt{B}^{\mathtt{st}}$ = *as in 1st and 2nd phase*

$\mathtt{B}^{\mathtt{cs}'}$ = *as in 2nd phase*

$\mathtt{B}^{\mathtt{lab}}$ = *as in 1st and 2nd phase*

$\mathtt{B}^{\mathtt{m}}$ =
```
class Marker {
    CStudent guy;
    void g(){ guy[CStudent].grade='A'; }
  }
```

Figure 3. Students — binary fragments

- $\forall \mathtt{S} \neq \epsilon: \ \mathcal{C}\{\!\{\mathtt{S},\mathtt{B}\}\!\} \neq \epsilon \iff \mathtt{B} \vdash \mathtt{S} \diamond \iff \mathtt{B} \vdash_{\mathcal{B}} \mathcal{C}\{\!\{\mathtt{S},\mathtt{B}\}\!\} \diamond$

Notice that the assertion $\mathtt{B}_1 \vdash_{\mathcal{B}} \mathtt{B}_2 \diamond$ does not imply the existence of an $\mathcal{S}$ fragment $\mathtt{S}$ such that $\mathtt{B}_2 = \mathcal{C}\{\!\{\mathtt{S},\mathtt{B}_1\}\!\}$. In Java, it is possible for definitions to be well-formed, when considered at binary level, and not to be well-formed, when considered at source level. For example, take $\mathtt{B}^{\mathtt{AB}}$:

$$\mathtt{B}^{\mathtt{AB}} = \texttt{class A \{ int f()\{return 5; \} \}} + \texttt{class B extends A \{ void f()\{return; \} \}}.$$

A Java linker would link byte-code corresponding to $\mathtt{B}^{\mathtt{AB}}$ without error. Therefore, $\vdash_{\mathcal{B}} \mathtt{B}^{\mathtt{AB}} \diamond$ should hold. Nevertheless, the above definitions are *not* well-formed if considered at source level, because the method `f()` in class `B` overrides that of `A`, but has a different result type [12]. Therefore, for

$$\mathtt{S}^{\mathtt{AB}} = \texttt{class A \{ int f()\{return 5; \} \}} + \texttt{class B extends A \{ void f()\{return; \} \}}$$

for all Bs: $\mathcal{C}\{\!\{\mathtt{S}^{\mathtt{AB}},\mathtt{B}\}\!\} = \epsilon$.

3.2. Linking

The operator $+$ combines fragments, and is used both at source and at binary level. At either level, we call the $+$ operator *linking*. The source code of the first phase of the students example consists of $\mathtt{S}^{\mathtt{st}} + \mathtt{S}^{\mathtt{cs}} + \mathtt{S}^{\mathtt{lab}}$; that of the second phase consists of $\mathtt{S}^{\mathtt{st}} + \mathtt{S}^{\mathtt{cs}'} + \mathtt{S}^{\mathtt{lab}}$. The binary code of the first phase consists of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}} + \mathtt{B}^{\mathtt{lab}}$; that of the second phase consists of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}'} + \mathtt{B}^{\mathtt{lab}}$.

The expression $\mathcal{D}(\mathtt{F})$ should denote the identifiers of all fragments introduced in $\mathtt{F}$. So, $\mathcal{D}(\mathtt{S}^{\mathtt{cs}} + \mathtt{S}^{\mathtt{m}})$ should be something like { `CStudent`, `Marker` }.

Linking binary code in actual systems may involve several steps, *e.g.* verification of format, resolution of references, and several checks, often applied in an interleaved manner. We are not interested in these steps themselves and we consider that all checks should take place when testing well-formedness of the fragment resulting from the linking process. Thus, the case where linking fragments $\mathtt{B}_1$ and $\mathtt{B}_2$ should flag an error can be modeled by $\vdash_{\mathcal{B}} \mathtt{B}_1 + \mathtt{B}_2 \diamond$ *not* holding.

Therefore, linking is based on concatenation – even though it may involve some more actions. This implies the following requirements: Linking introduces the identifiers that are separately introduced by the sub-fragments. A fragment consists of the linking of "simple" fragments each introducing one identifier ($\#(\mathcal{D}(\mathtt{F}_i)) = 1$). Compilation introduces the same identifiers as the original.

Axiom 2 *For fragments* $\mathtt{F}$, $\mathtt{F}'$, $\mathtt{S}$, $\mathtt{B}$:

- $\mathcal{D}(\mathtt{F} + \mathtt{F}') = \mathcal{D}(\mathtt{F}) \cup \mathcal{D}(\mathtt{F}')$
- $\mathtt{F} = \mathtt{F}' \implies$ *for* $n = \#(\mathcal{D}(\mathtt{F})), \exists \mathtt{F}_1, ...\mathtt{F}_n, \mathtt{F}'_1, ...\mathtt{F}'_n:$

$\mathtt{F} = \mathtt{F}_1 + ... + \mathtt{F}_n,\ \mathtt{F}' = \mathtt{F}'_1 + ... + \mathtt{F}'_n,$
and for $\mathtt{i} \in \{1...n\} : \mathtt{F}_i = \mathtt{F}'_i,\ \#(\mathcal{D}(\mathtt{F}_i)) = 1$

- $\mathtt{B} \vdash \mathtt{S} \diamond \implies \mathcal{D}(\mathcal{C}\{\!\{\mathtt{S},\mathtt{B}\}\!\}) = \mathcal{D}(\mathtt{S})$

Note that compilation after linking, *i.e.* $\mathcal{C}\{\!\{\mathtt{S}_1+\mathtt{S}_2,\mathtt{B}\}\!\}$, need *not* be equivalent to linking after compilation, *i.e.* to $\mathcal{C}\{\!\{\mathtt{S}_1,\mathtt{B}\}\!\}+\mathcal{C}\{\!\{\mathtt{S}_2,\mathtt{B}\}\!\}$. For example, $\mathcal{C}\{\!\{\mathtt{S}^{\mathtt{lab}},\mathtt{B}^{\mathtt{st}}\}\!\} = \epsilon$, whereas $\mathcal{C}\{\!\{\mathtt{S}^{\mathtt{lab}}+\mathtt{S}^{\mathtt{cs}},\mathtt{B}^{\mathtt{st}}\}\!\} = \mathtt{B}^{\mathtt{lab}}+\mathtt{B}^{\mathtt{cs}}$.

Fragments "containing" other fragments are said to subsume them:

Definition 2 $\mathtt{F}$ subsumes $\mathtt{F}_1$ *if* $\mathtt{F} = \mathtt{F}_1+\mathtt{F}_2$ *for some* $\mathtt{F}_2$.

For example, $\mathtt{S}^{\mathtt{lab}}+\mathtt{S}^{\mathtt{st}}+\mathtt{S}^{\mathtt{cs}'}+\mathtt{S}^{\mathtt{m}}$ subsumes $\mathtt{S}^{\mathtt{st}}+\mathtt{S}^{\mathtt{cs}'}$.

3.3. Disjoint fragments

In some cases we expect pairs of fragments to be *disjoint*, *i.e.* to introduce different entities:

Definition 3 *For fragments* $\mathtt{F}_1$, $\mathtt{F}_2$:

- $\mathtt{F}_1$, $\mathtt{F}_2$ *are* disjoint *iff* $\mathcal{D}(\mathtt{F}_1) \cap \mathcal{D}(\mathtt{F}_2) = \emptyset$.

Thus, $\mathtt{S}^{\mathtt{cs}}+\mathtt{S}^{\mathtt{st}}$ and $\mathtt{S}^{\mathtt{m}}$ are disjoint, whereas $\mathtt{S}^{\mathtt{st}}+\mathtt{S}^{\mathtt{cs}}$ and $\mathtt{S}^{\mathtt{m}}+\mathtt{S}^{\mathtt{cs}'}$ are not. The constituent sub-fragments of a well-formed fragment are disjoint.

Axiom 3 *For fragments* $\mathtt{S}_1$, $\mathtt{S}_2$, $\mathtt{B}$, $\mathtt{B}_1$, $\mathtt{B}_2$:

- $\mathtt{B} \vdash \mathtt{S}_1+\mathtt{S}_2 \diamond \implies \mathtt{S}_1$ *and* $\mathtt{S}_2$ *disjoint*
- $\mathtt{B} \vdash_{\mathcal{B}} \mathtt{B}_1+\mathtt{B}_2 \diamond \implies \mathtt{B}_1$ *and* $\mathtt{B}_2$ *disjoint*

3.4. Locality

In general, one expects properties that can be established in a certain environment, to hold for larger environments as well. For instance, one expects an expression which has a type in a certain environment to have the same type in any larger environment. Such properties were proven, in [6], and also used in [5].

In particular, for fragments we expect the following locality properties: Linking disjoint, well-formed $\mathcal{S}$ or $\mathcal{B}$ fragments produces well-formed fragments. If $\mathtt{B}$ is well-formed in environment $\mathtt{B}_1$ then it remains so in any well-formed larger environment $\mathtt{B}_1+\mathtt{B}_2$. Checking a binary in the empty environment is equivalent to checking it with itself as the environment. Finally, if $\mathtt{S}$ is well-formed in environment $\mathtt{B}_1$ and in the larger environment $\mathtt{B}_1+\mathtt{B}_2$, then compilation in the two environments produces identical results.

Axiom 4 *For fragments* $\mathtt{S}$, $\mathtt{S}_1$, $\mathtt{S}_2$, $\mathtt{B}$, $\mathtt{B}_1$, $\mathtt{B}_2$:

- $\mathtt{B} \vdash \mathtt{S}_1 \diamond,\quad \mathtt{B} \vdash \mathtt{S}_2 \diamond,\quad \mathtt{S}_1, \mathtt{S}_2$ *disjoint* $\implies$
  $\mathtt{B} \vdash \mathtt{S}_1+\mathtt{S}_2 \diamond$
- $\mathtt{B} \vdash_{\mathcal{B}} \mathtt{B}_1 \diamond,\quad \mathtt{B} \vdash_{\mathcal{B}} \mathtt{B}_2 \diamond,\quad \mathtt{B}_1, \mathtt{B}_2$ *disjoint* $\implies$
  $\mathtt{B} \vdash_{\mathcal{B}} \mathtt{B}_1+\mathtt{B}_2 \diamond$
- $\mathtt{B}_1 \vdash_{\mathcal{B}} \mathtt{B} \diamond,\quad \vdash_{\mathcal{B}} \mathtt{B}_1+\mathtt{B}_2 \diamond \implies \mathtt{B}_1+\mathtt{B}_2 \vdash_{\mathcal{B}} \mathtt{B} \diamond$
- $\epsilon \vdash_{\mathcal{B}} \mathtt{B} \diamond \iff \mathtt{B} \vdash_{\mathcal{B}} \mathtt{B} \diamond$
- $\mathtt{B}_1 \vdash \mathtt{S} \diamond$ *and* $\mathtt{B}_1+\mathtt{B}_2 \vdash \mathtt{S} \diamond \implies$
  $\mathcal{C}\{\!\{\mathtt{S},\mathtt{B}_1\}\!\} = \mathcal{C}\{\!\{\mathtt{S},\mathtt{B}_1+\mathtt{B}_2\}\!\}$

3.5 Strong locality

Strong locality requires the compilation of a well-formed $\mathcal{S}$ fragment to be identical to its compilation in a larger well-formed environment, ie:

$\mathtt{B}_1 \vdash \mathtt{S} \diamond$ and $\mathtt{B}_1+\mathtt{B}_2 \vdash_{\mathcal{B}} \mathtt{B} \diamond \implies$
$\mathcal{C}\{\!\{\mathtt{S},\mathtt{B}_1\}\!\} = \mathcal{C}\{\!\{\mathtt{S},\mathtt{B}_1+\mathtt{B}_2\}\!\}$.

This property actually corresponds to recasting the third point from axiom 4 for $\mathcal{S}$ fragments, and it is weaker than the fifth point from axiom 4.

Strong locality is satisfied by the Java subset we have formalized [6]. In the original version of this paper, we required this property as an axiom, and this axiom was central to the argumentation of that version.

However, because of its treatment of packages, strong locality does *not* hold in full Java.[2] This realization led to the adoption of a slightly different formalization of binary compatibility, *cf.* sections 5 and 5.1.

3.6 Faithfulness of the model

The above concludes the axiomatic description of the basic model for compilation and linking. We believe that it describes concisely most people's expectations of compilation and linking. The axioms are satisfied by the Java subset we have studied.

However, the question as to the faithfulness of the model to Java is open, as there does not exist a full formal specification of Java. Even if such a formal description existed, it would still be debatable how far this description corresponded to the developers' intentions, the language definition [12] and the language implementations.

On the other hand, Sun Java developers [2] have studied the previous version of this paper, have given us feedback, and pointed out a discrepancy with Java, which we have taken into account. Thus our confidence that this model is appropriate for Java has grown.

The fact that the Java developers responded to this calculus, and not to full blown formalizations of parts of the language, is, we believe, a strong indication that the fragment calculus represents the appropriate level of abstraction for the description of issues of separate compilation and linking.

[2] This was reported to us by Gilad Bracha [10].

4. Updating

Java binary compatibility is concerned with the effects of modifications to source and binary code. Therefore we need operators to describe such modifications: $\_\oplus\_$ describes the effect of updating some code, whereas $\_\oplus_{c}\_$ describes the effect of updating $\mathcal{B}$ code through compilation of $\mathcal{S}$ code.

Definition 4 *For fragments* $\mathtt{F}_1$, $\mathtt{F}_2$:

- $\mathtt{F}_1 \oplus \mathtt{F}_2 = \mathtt{F}_0 + \mathtt{F}_2$,

where $\mathtt{F}_0$ *such that* $\exists\ \mathtt{F}_3$ *with*
$\mathtt{F}_1 = \mathtt{F}_0 + \mathtt{F}_3$, $\mathcal{D}(\mathtt{F}_3) \subseteq \mathcal{D}(\mathtt{F}_2)$, $\mathtt{F}_0$, $\mathtt{F}_2$ *disjoint.*

For example $(\mathtt{S}^{\mathtt{st}} + \mathtt{S}^{\mathtt{cs}} + \mathtt{S}^{\mathtt{lab}}) \oplus \mathtt{S}^{\mathtt{cs}'} = \mathtt{S}^{\mathtt{st}} + \mathtt{S}^{\mathtt{cs}'} + \mathtt{S}^{\mathtt{lab}}$. Also, $(\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{lab}}) \oplus \mathtt{B}^{\mathtt{m}} = \mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{lab}} + \mathtt{B}^{\mathtt{m}}$.

We can show that $\oplus$ is well-defined using the first two points from axiom 2. Updating is associative but not commutative. For disjoint fragments updating is equivalent to linking. Furthermore,

Lemma 1 *For fragments* $\mathtt{F}$, $\mathtt{F}_1$, $\mathtt{F}_2$, $\mathtt{F}_3$, $\mathtt{F}_4$:

- $\mathtt{F}$ *disjoint from* $\mathtt{F}_2 \implies$
  $(\mathtt{F}_1 \oplus \mathtt{F}) + \mathtt{F}_2 = (\mathtt{F}_1 + \mathtt{F}_2) \oplus \mathtt{F}$
- $\mathtt{F}_1 = \mathtt{F}_3 + \mathtt{F}$, $\mathtt{F}_2 = \mathtt{F}_4 + \mathtt{F}$, *and* $\mathtt{F}_3$, $\mathtt{F}_4$ *disjoint* $\implies$
  $\mathtt{F}_1 \oplus \mathtt{F}_2 = \mathtt{F}_2 \oplus \mathtt{F}_1$
- $\mathtt{F}_1$, $\mathtt{F}_2$ *disjoint, and* $\mathtt{F}_1$, $\mathtt{F}_3$ *disjoint* $\implies$
  $\mathtt{F}_1 + (\mathtt{F}_2 \oplus \mathtt{F}_3) = (\mathtt{F}_1 + \mathtt{F}_2) \oplus \mathtt{F}_3$
- $\mathtt{F}_1, \mathtt{F}_2$ *disjoint* $\implies \mathtt{F} \oplus (\mathtt{F}_1 + \mathtt{F}_2) = \mathtt{F} \oplus \mathtt{F}_1 \oplus \mathtt{F}_2$.

The expression $\mathtt{B} \oplus_{c} \mathtt{S}$ denotes updating $\mathtt{B}$ by the compilation of $\mathtt{S}$ in environment $\mathtt{B}$.

Definition 5 *For fragments* $\mathtt{B}$ *and* $\mathtt{S}$, *we define*:

- $\mathtt{B} \oplus_{c} \mathtt{S} = \mathtt{B} \oplus \mathcal{C}\{\!\{\mathtt{S}, \mathtt{B}\}\!\}$

The second phase of our example updates $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}} + \mathtt{B}^{\mathtt{lab}}$ through compilation of $\mathtt{S}^{\mathtt{cs}'}$, giving:

$$(\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}} + \mathtt{B}^{\mathtt{lab}}) \oplus_{c} \mathtt{S}^{\mathtt{cs}'} = \mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}'} + \mathtt{B}^{\mathtt{lab}}$$

The third phase compiles the new fragment $\mathtt{S}^{\mathtt{m}}$ updating the result of the previous phase, giving:

$$(\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}'} + \mathtt{B}^{\mathtt{lab}}) \oplus_{c} \mathtt{S}^{\mathtt{m}} = \mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}'} + \mathtt{B}^{\mathtt{lab}} + \mathtt{B}^{\mathtt{m}}$$

Because the arguments of $\_\oplus_{c}\_$ come from different domains, the concepts of commutativity and associativity do not apply. We use $\_\oplus_{c}\_$ in a left-associative manner. In general, $\mathcal{C}\{\!\{\mathtt{S}, \mathtt{B}\}\!\} \oplus_{c} \mathtt{S}' \neq \mathcal{C}\{\!\{\mathtt{S} \oplus \mathtt{S}', \mathtt{B}\}\!\}$. The left hand side represents separate compilation of fragments whereas the right hand side represents compilation of the fragments together. As we mentioned earlier, in Java these can be different, and it is possible for one compilation to succeed, and the other not to. This happens in the second phase of the students example, where $\mathcal{C}\{\!\{\mathtt{S}^{\mathtt{lab}}, \mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}}\}\!\} \oplus_{c} \mathtt{S}^{\mathtt{cs}'} = \mathtt{B}^{\mathtt{lab}} + \mathtt{B}^{\mathtt{cs}'}$, whereas $\mathcal{C}\{\!\{\mathtt{S}^{\mathtt{lab}} \oplus \mathtt{S}^{\mathtt{cs}'}, \mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}}\}\!\} = \epsilon$.

The following lemma, used later to prove lemma 5, describes the result of interleaving compilation and linking. If $\mathtt{B}_1$ and $\mathtt{S}$ are disjoint, then $\mathtt{B}_1$ remains unaffected when updating $\mathtt{B}_1 + \mathtt{B}_2$ through compilation of $\mathtt{S}$, but may be taken into account when compiling $\mathtt{S}$.

Lemma 2 *For fragments* $\mathtt{S}$, $\mathtt{B}_1$, $\mathtt{B}_2$, *with* $\mathtt{B}_1$ *disjoint* $\mathtt{B}_2$, *and* $\mathtt{B}_1$ *disjoint* $\mathtt{S}$:

- $(\mathtt{B}_1 + \mathtt{B}_2) \oplus_{c} \mathtt{S} = \mathtt{B}_1 + (\mathtt{B}_2 \oplus \mathcal{C}\{\!\{\mathtt{S}, \mathtt{B}_1 + \mathtt{B}_2\}\!\})$

Proof by lemma 1, first and third point of axiom 2. □

5. Binary compatibility

The Java language specification [12] describes binary compatible changes as follows:

> "*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.*"

Our notion of *binary compatible change* aims to capture the above. It restricts source code modifications in terms of properties of the resulting compilation. Therefore, its formalization will have the general form:

$\mathtt{S}$ is a *binary compatible change* of $\mathtt{B}$ iff:
$\vdash_{\mathcal{B}} \ldots\mathtt{B}\ldots \diamond \quad \implies \quad \vdash_{\mathcal{B}} \ldots\mathtt{B} \oplus_{c} \mathtt{S}\ldots \diamond$

During the process of formalization we realized that the definition from [12] is *not* unambiguous, because it does not answer the following questions.

Q1 Should source code which cannot be compiled be considered a binary compatible change?

Q2 How many binaries are meant?

Q3 What should be the environment for the compilation of the modified code?

Regarding question Q1, in the previous version of this paper we considered changes which did not compile to be binary *incompatible*. This tied in with the fact that we described compilation through a *partial* function. Consideration of questions Q2 and Q3 led to four alternative interpretations, which are described in section 5.1. We had postulated strong locality, and thus could prove composition properties such as those in section 6.

Without strong locality, the composition properties do not hold for link compatibility as defined in section 5.1. However, such properties are part of the rationale for the concept of binary compatibility [2, 20]. Therefore, when the lack of strong locality was reported to us, we felt that our approach required revision.

Thus, we explore the approach whereby a change which does not compile *is* binary compatible. This allows us to re-establish the composition properties and prove them more simply.

This approach motivates our current description of compilation through a *total* function $\mathcal{C}$, where $\mathcal{C}\{\!\{\mathtt{S},\mathtt{B}\}\!\} = \epsilon$ if $\mathtt{B} \vdash \mathtt{S} \diamond$ does not hold. Moreover, source code which cannot be compiled does not produce binaries, and therefore does not modify the original binaries; so it can be understood as the empty change, and hence binary compatible.

We now consider the remaining questions Q2 and Q3 and discuss the three alternative interpretations I1, I2 and I3.

Definition 6 (I1-I3) *For $\mathcal{S}$ fragment $\mathtt{S}$ and $\mathcal{B}$ fragment $\mathtt{B}$:*

I1 $\mathtt{S}$ *is a* weak binary compatible change *of* $\mathtt{B}$ *if*:

$$\vdash_{\mathcal{B}} \mathtt{B} \diamond \quad \Longrightarrow \quad \vdash_{\mathcal{B}} \mathtt{B} \oplus_{\mathcal{C}} \mathtt{S} \diamond$$

I2 $\mathtt{S}$ *is a* local binary compatible change *of* $\mathtt{B}$*, iff for all* $\mathtt{B}_0$ *disjoint from* $\mathtt{S}$:

$$\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond \quad \Longrightarrow \quad \vdash_{\mathcal{B}} \mathtt{B}_0 + (\mathtt{B} \oplus_{\mathcal{C}} \mathtt{S}) \diamond$$

I3 $\mathtt{S}$ *is a* binary compatible change *of* $\mathtt{B}$*, iff for all* $\mathtt{B}_0$ *disjoint from* $\mathtt{S}$:

$$\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond \quad \Longrightarrow \quad \vdash_{\mathcal{B}} (\mathtt{B}_0 + \mathtt{B}) \oplus_{\mathcal{C}} \mathtt{S} \diamond$$

Thus, binary compatibility requires compilation to preserve linking capabilities, but only if it was successful. We now discuss the three interpretations:

Interpretation I1 is too weak. For instance, it allows the removal of a method definition `f()` from a class in $\mathtt{B}$, provided that `f()` were not called inside $\mathtt{B}$. However, further libraries which linked with $\mathtt{B}$ might rely on `f()`.

Interpretation I2 is too localized. Namely, $\mathtt{S}^{\mathtt{cs}'}$ is a local binary compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}} + \mathtt{B}^{\mathtt{lab}}$, and $\mathtt{S}^{\mathtt{cs}}$ is not a local binary compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{m}}$; these facts are as expected. However, $\mathtt{S}^{\mathtt{cs}}$ is trivially a local binary compatible change of $\mathtt{B}^{\mathtt{m}}$ (because $\mathcal{C}\{\!\{\mathtt{S}^{\mathtt{cs}},\mathtt{B}^{\mathtt{m}}\}\!\} = \epsilon$), even though compilation of $\mathtt{S}^{\mathtt{cs}}$ destroys the well-formedness of the environment $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}'} + \mathtt{B}^{\mathtt{m}}$.

Therefore, we adopt interpretation I3. With this, $\mathtt{S}^{\mathtt{cs}'}$ is a binary compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}} + \mathtt{B}^{\mathtt{lab}}$ and of $\mathtt{B}^{\mathtt{cs}} + \mathtt{B}^{\mathtt{lab}}$; $\mathtt{S}^{\mathtt{cs}}$ is not a local binary compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{m}}$, and not a binary compatible change of $\mathtt{B}^{\mathtt{m}}$.

5.1 Further interpretations

The interpretations I1, I2, and I3 were motivated by the absence of the strong locality property[10]. However, the absence of the strong locality property is due to an idiosyncratic treatment of packages, and may not be a desirable feature for Java anyway. It may not even be a feature of future "web-centered" languages. Furthermore, a treatment which considers an $\mathcal{S}$ fragment which does not compile, to be binary compatible, may be counter-intuitive.

Thus, it is worthwhile exploring a different approach to question Q1, whereby an $\mathcal{S}$ fragment which does not compile is considered binary incompatible. This approach can be based on a treatment of compilation as a *partial* function $\mathcal{C}_p$, where $\mathcal{C}_p\{\!\{\mathtt{S},\mathtt{B}\}\!\} = \mathcal{C}\{\!\{\mathtt{S},\mathtt{B}\}\!\}$ iff $\mathtt{B} \vdash \mathtt{S} \diamond$, and undefined otherwise. We also define $\oplus_{\mathcal{C}_p}$ as $\mathtt{B} \oplus_{\mathcal{C}_p} \mathtt{S} = \mathtt{B} \oplus \mathcal{C}_p\{\!\{\mathtt{S},\mathtt{B}\}\!\}$.

If we consider the remaining questions Q2 and Q3 we obtain four further interpretations:

Definition 7 (I4-I7) *For $\mathcal{S}$ fragment $\mathtt{S}$ and $\mathcal{B}$ fragment $\mathtt{B}$*

I4 $\mathtt{S}$ *is a* weak link compatible change *of* $\mathtt{B}$ *if*:

$$\vdash_{\mathcal{B}} \mathtt{B} \diamond \quad \Longrightarrow \quad \vdash_{\mathcal{B}} \mathtt{B} \oplus_{\mathcal{C}_p} \mathtt{S} \diamond$$

I5 $\mathtt{S}$ *is a* strong link compatible change *of* $\mathtt{B}$*, iff for all* $\mathtt{B}_0$ *disjoint from* $\mathtt{S}$:

$$\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond \quad \Longrightarrow \quad \vdash_{\mathcal{B}} \mathtt{B}_0 + (\mathtt{B} \oplus_{\mathcal{C}_p} \mathtt{S}) \diamond$$

I6 $\mathtt{S}$ *is a* link compatible change *of* $\mathtt{B}$*, iff for all* $\mathtt{B}_0$ *disjoint from* $\mathtt{S}$:

$$\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond \quad \Longrightarrow \quad \vdash_{\mathcal{B}} (\mathtt{B}_0 + \mathtt{B}) \oplus_{\mathcal{C}_p} \mathtt{S} \diamond$$

I7 $\mathtt{S}$ *is a* link compatible change *of* $\mathtt{B}_1$ in the context of $\mathtt{B}_2$ *iff* $\mathtt{B}_2$ *is disjoint from* $\mathtt{S}$ *and* $\mathtt{B}_1$*, and for all* $\mathtt{B}_0$ *disjoint from* $\mathtt{S}$:

$$\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B}_1 + \mathtt{B}_2 \diamond \quad \Longrightarrow \quad \vdash_{\mathcal{B}} (\mathtt{B}_0 + \mathtt{B}_1 + \mathtt{B}_2) \oplus_{\mathcal{C}_p} \mathtt{S} \diamond.$$

The difference between binary compatibility (I1-I3) and link compatibility(I4-I7) is the following: Link compatibility requires preservation of linking capabilities *and* successful compilation, whereas binary compatibility requires preservation of linking capabilities *only if* compilation is successful.

Interpretation I4 is too weak, for the same reasons which make interpretation I1 too weak.

Interpretation I5 is too strong, because it expects $\mathtt{B}$ to contain *all* information necessary for the compilation of $\mathtt{S}$. Thus, if $\mathtt{S}$ contained code using properties of the predefined class `String`, it would be a strongly link compatible change of `String` – even if $\mathtt{S}$ did not modify `String`, but only used it.

According to interpretation I6, $\mathtt{S}^{\mathtt{lab}}$ is a link compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}}$, and $\mathtt{S}^{\mathtt{cs}'}$ is a link compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}} + \mathtt{B}^{\mathtt{lab}}$. Interpretation I6 is weaker than interpretation I5, because it is possible for $(\mathtt{B}_0 + \mathtt{B}) \oplus_{\mathcal{C}_p} \mathtt{S}$ to be defined and for $\mathtt{B} \oplus_{\mathcal{C}_p} \mathtt{S}$ not to be. This subtlety allows $\mathtt{S}$ to be a link compatible change of a library $\mathtt{B}$, which imports other libraries, and which cannot be compiled in isolation, *i.e.* $\vdash_{\mathcal{B}}$ B $\diamond$ does not hold. Such a library can only be compiled in the presence of further libraries, represented by the fragment $\mathtt{B}_0$, with which $\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond$.

Also, B does not need to contain *all* the type information necessary to type check S; it only needs to contain *enough* information to ensure type correct compilation of S in the environment of all appropriate fragments $\mathtt{B}_0$.

Interpretation I6 is the one we had adopted in [7]. However, the reference binaries are still too extensive.[3] If for instance, S used features of the predefined class `String`, then in order for S to be a link compatible change of B, B would have *either* to use the same features of `String`, *or* to contain the class declaration of `String`. Since however, S only *uses* the class `String` and does not *modify* it, the distinction of the role of `String` should be reflected in the definition.

Thus, in interpretation I7 we distinguish $\mathtt{B}_2$, the *context* which may *not* be modified by the compilation of S, from $\mathtt{B}_1$, which *may*. Therefore, $\mathtt{S}^{\mathtt{m}}$ is a link compatible change of ϵ in the context of $\mathtt{B}^{\mathtt{cs}}$. Also, an S which uses `String` may be a link compatible change of B, in the context of class `String`.

In fragment systems which satisfy the strong locality property, link compatible changes, as in interpretations I5-I7, enjoy composition properties corresponding to these from the next section.

Link compatibility implies binary compatibility in the sense that any link compatible change of a B is also a binary compatible change of B. Also, strong link compatibility implies link compatibility. We expect that further entailment relationships between the interpretations hold; their proof may require a refinement of the fragment system definition.

6. Composition Properties

We now demonstrate the following five properties:

- **Preservation over larger fragments**: establishes binary compatibility for all fragments containing a fragment for which this property has already been established.
- **Preservation over sequences**: guarantees that combined binary compatible steps preserve their linking capabilities – provided that each step is a binary compatible change of the result of all previous modifications – *cf.* figure 4.
- **Preservation over libraries**: application of binary compatible changes to different fragments preserves well-formedness – *cf.* figure 5.
- **Lack of folding property**: in general, two binary compatible changes cannot be folded into one.
- **Lack of diamond property**: two binary compatible changes applied to the same fragment, cannot always be reconciled.

[3] Having small reference binaries is important, because this allows modifications to be applied to more fragments comprising a large program, as in lemma 5.

These properties are, we believe, crucial in delineating the exact nature of binary compatibility, and are central issues in the design of that feature [20].

Also, these properties affect the way library designers can evolve their libraries: The lack of a folding or a diamond property restricts the ways in which binary compatible changes may be combined. The lack of diamond property means that programmers may not apply *independent* binary compatible changes to the *same* fragment and expect the linking capabilities to be preserved. However, the preservation over libraries allows programmers to apply *independent* binary 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 binary 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 Any binary compatible change is also a binary compatible change of a larger fragment:

Lemma 3 *For* S, $\mathtt{B}_1$, $\mathtt{B}_2$, *where* S *and* $\mathtt{B}_2$ *are disjoint*:

$$\mathtt{S} \textit{ a binary compatible change of } \mathtt{B}_1 \implies$$
$$\mathtt{S} \textit{ a binary compatible change of } \mathtt{B}_1 + \mathtt{B}_2$$

Proof by commutativity and associativity of $+$. □

Preservation over sequences As outlined in figure 4, a sequence of binary compatible steps, $\mathtt{S}_1$, ... $\mathtt{S}_n$, applied to fragment B preserves its linking capabilities. In order to establish that a step is binary compatible, we need to know the effect of all prior steps, thus we require that $\mathtt{S}_{i+1}$ is binary compatible for $(\mathtt{B}_0 + \mathtt{B}) \oplus_c \mathtt{S}_1 ... \oplus_c \mathtt{S}_i$.

Lemma 4 *For $\mathcal{B}$ fragments* B, $\mathtt{B}_0$, *a sequence of $\mathcal{S}$ fragments* $\mathtt{S}_1$, ... $\mathtt{S}_n$, $\mathtt{B}_0$ *disjoint* $\mathtt{S}_i$, *if*

- *for* $1 \leq i \leq n$: $\mathtt{S}_{i+1}$ *binary compatible change of* $\mathtt{B}^i$ *where* $\mathtt{B}^i = (\mathtt{B}_0 + \mathtt{B}) \oplus_c \mathtt{S}_1 ... \oplus_c \mathtt{S}_i$
- $\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond$

then

- $\vdash_{\mathcal{B}} (\mathtt{B}_0 + \mathtt{B}) \oplus_c \mathtt{S}_1 ... \oplus_c \mathtt{S}_n \diamond$

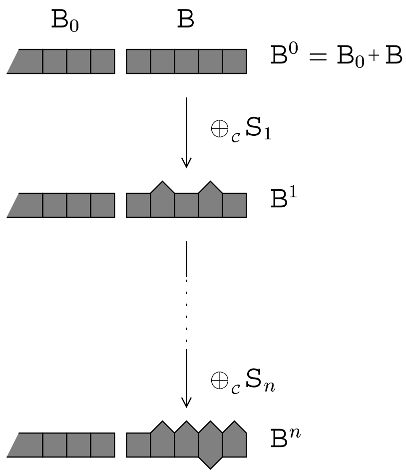

Figure 4. Preservation over sequences

Proof by induction on k; using that $\mathtt{B}^0$=$\mathtt{B}_0+\mathtt{B}$ and $\mathtt{B}^{k+1}$=$\mathtt{B}^k \oplus_c \mathtt{S}_{k+1}$, prove that $\vdash_{\mathcal{B}} \mathtt{B}^k \diamond$ for all k. Also, $\mathtt{B}^n = (\mathtt{B}_0+\mathtt{B}) \oplus_c \mathtt{S}_1 ... \oplus_c \mathtt{S}_n$. □

Preservation over libraries Binary compatible modifications $\mathtt{S}_i$ applied to fragments $\mathtt{B}_i$ which are parts of a program $\mathtt{B}_0 + \mathtt{B}_1 + ... + \mathtt{B}_n$, preserve the linking capabilities of that program, provided that the modifications are binary compatible for the particular fragments only – *i.e.* require $\mathtt{S}_i$ to be a binary compatible change of $\mathtt{B}_i$, which is stronger than requiring $\mathtt{S}_i$ to be a binary compatible change of $\mathtt{B}_0 + \mathtt{B}_1 + ... + \mathtt{B}_n$.

In contrast to preservation over sequences, we do not need to know the effect of another modification in order to establish that $\mathtt{S}_i$ is a binary compatible change of $\mathtt{B}_i$, but we take into account the effect of previous modifications. Thus, $\mathtt{B}_k$ is transformed to $\mathtt{B}'_k$, where $\mathtt{B}'_k = \mathtt{B}_k \oplus \mathcal{C}\{\!\{\mathtt{S}_k, \mathtt{B}_0+\mathtt{B}'_1+...\mathtt{B}'_{k-1}+\mathtt{B}_k...+\mathtt{B}_n\}\!\}$; as in figure 5.

This models the situation where programmers make changes to the particular fragments that belong to them, but *are aware* of each other's actions. When all modified fragments are put together, the resulting program $\mathtt{B}_0 + \mathtt{B}'_1 ...+ \mathtt{B}'_n$ preserves the linking capabilities of the original program. The order of the fragments is immaterial.

Lemma 5 *For* $\mathtt{B}_0, \mathtt{B}_1, ..., \mathtt{B}_n, \mathtt{S}_1,..., \mathtt{S}_n$, *where* $\mathtt{S}_i$ *disjoint from* $\mathtt{B}_k$, *from* $\mathtt{S}_k$ *and from* $\mathtt{B}_0$ *for all* $i \neq k$, $i, k \in \{1...n\}$, *if*

- $\mathtt{S}_i$ *is a binary compatible change of* $\mathtt{B}_i$ *for* $1 \leq i \leq n$
- $\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B}_1 + ... + \mathtt{B}_n \diamond$

then

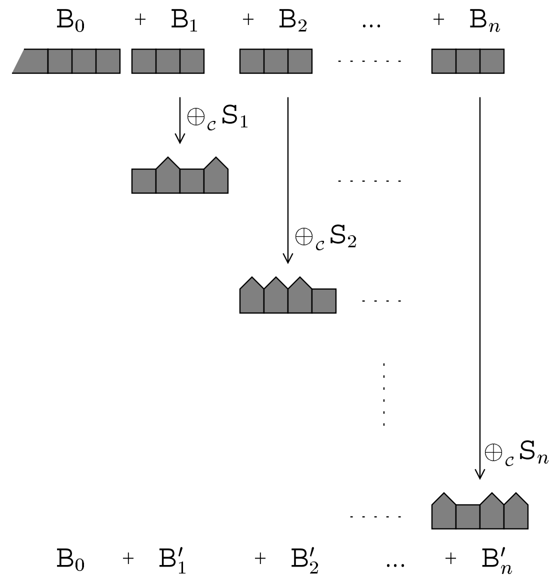

Figure 5. Preservation over libraries

- $\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B}'_1 ...+ \mathtt{B}'_n \diamond$
  where $\mathtt{B}'_k = \mathtt{B}_k \oplus \mathcal{C}\{\!\{\mathtt{S}_k, \mathtt{B}_0+\mathtt{B}'_1+...\mathtt{B}'_{k-1}+\mathtt{B}_k...+\mathtt{B}_n\}\!\}$

Proof Define $\mathtt{B}^k = \mathtt{B}_0+\mathtt{B}'_1+...\mathtt{B}'_k+\mathtt{B}_{k+1}...+\mathtt{B}_n$. By induction on $\mathtt{k}$, and using lemma 2, we show that $\mathtt{B}'_k$ and $\mathtt{B}^k$ are defined, and that $\vdash_{\mathcal{B}} \mathtt{B}^k \diamond$ for all $\mathtt{k}$. □

Interestingly, the other case of the lemma, whereby programmers apply binary compatible changes *unaware* of each other's actions, *cannot* be proven. That is, for $\mathtt{B}'_k = \mathtt{B}_k \oplus \mathcal{C}\{\!\{\mathtt{S}_k, \mathtt{B}_0+\mathtt{B}_1+...+\mathtt{B}_n\}\!\}$ we cannot prove that $\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B}'_1 ...+ \mathtt{B}'_n \diamond$. This is so, because binary compatibility guarantees preservation of well-formedness if the compilation took place in the *same* environment. In other words, from $\vdash_{\mathcal{B}} \mathtt{B}_0+...\mathtt{B}'_k+\mathtt{B}_{k+1}...+\mathtt{B}_n \diamond$ it is impossible to infer that $\vdash_{\mathcal{B}} \mathtt{B}_0+...\mathtt{B}'_k+(\mathtt{B}_{k+1} \oplus \mathcal{C}\{\!\{\mathtt{S}_{k+1}, \mathtt{B}_0+\mathtt{B}_1+...+\mathtt{B}_n\}\!\})...+\mathtt{B}_n \diamond$

The concepts of transitivity and reflexivity are not applicable to the binary compatibility relationship, because its domain and range do not match. Neither is there a folding or a diamond property:

Lack of folding property For $\mathtt{S}_1$ a binary compatible change of $\mathtt{B}$, and $\mathtt{S}_2$ a binary compatible change of $(\mathtt{B}_0+\mathtt{B}) \oplus_c \mathtt{S}_1$, it does not necessarily hold that $(\mathtt{B}_0+\mathtt{B}) \oplus_c \mathtt{S}_1 \oplus_c \mathtt{S}_2 = (\mathtt{B}_0+\mathtt{B}) \oplus_c (\mathtt{S}_1 \oplus \mathtt{S}_2)$. As a counter-example, consider $\mathtt{B} = \mathtt{B}^{\mathtt{st}}+\mathtt{B}^{\mathtt{cs}}$, $\mathtt{S}_1 = \mathtt{S}^{\mathtt{lab}}$ and $\mathtt{S}_2 = \mathtt{S}^{\mathtt{cs}'}$.

Lack of diamond property For $\mathtt{S}_1$ and $\mathtt{S}_2$ binary compatible changes of $\mathtt{B}$, there do not always exist fragments $\mathtt{S}_3$ and $\mathtt{S}_4$, such that $\mathtt{S}_3$, $\mathtt{S}_4$ disjoint with $\mathtt{S}_1$, $\mathtt{S}_2$, and $\mathtt{S}_3$ is a binary compatible change of $\mathrm{B}_0 \oplus_c \mathrm{S}_1$, and S_4 is a binary compatible change of $\mathrm{B} \oplus_c \mathrm{S}_2$, and $\mathrm{B} \oplus_c \mathrm{S}_1 \oplus_c \mathrm{S}_3$= $\mathrm{B} \oplus_c \mathrm{S}_2 \oplus_c \mathrm{S}_4$. For example, S_1 might be introducing a method `f` with signature `int` $\rightarrow$ `int` into a class `C`, and S_2 introducing another method `f` with signature `int`$\rightarrow$ `char` into the same class `C`.

7. Conclusions and further work

We gave an axiomatic definition of compilation and linking, and extended it to model binary compatibility. In our view, the contributions of this paper are:

- identification of the appropriate level of abstraction with respect to the description of separate compilation and linking,
- a distillation of the essential features with respect to separate compilation and linking,
- a clarification of the design space for binary compatibility,
- a formal framework for the description of binary compatibility and proof of its properties,
- a strengthening of the properties of binary compatibility proven earlier [7],
- demonstration that the properties of binary compatibility in Java stem from the few features described in the fragment calculus rather than from the rather large set of features of Java[16, 21] and its byte-code [18, 19, 9].

We believe that such a fragment calculus can serve as a basis for the description of the approaches to separate compilation and linking taken by other languages, and as a starting point for an abstract description of dynamic linking and loading in Java [15, 14, 11].

Acknowledgements

We acknowledge the financial support from the EPSRC Grant Ref:GR/L 76709.

We thank David Clarke for feedback, Phil Wadler for heated discussions and many useful suggestions and the anonymous LICS referees for insightful suggestions.

We are very grateful to Gilad Bracha, for detailed comments on the previous version of the paper, and for helping us keep on the right track.

References

[1] E. Boerger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523. Springer Verlag LNCS, 1999.

[2] G. Bracha. Private Communications, February 1999.

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

[4] 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.

[5] D. Dean. The Security of Static Typing with Dynamic Linking. In *Fourth ACM Conference on Computer and Communication Security*, 1997.

[6] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is Java Sound? *Theory and Practice of Object Systems*, 5(1), Jan. 1999. available at http://www-dse.doc.ic.ac.uk/projects/slurp/.

[7] S. Drossopoulou, D. Wragg, and S. Eisenbach. What *is* Java Binary Compatibility? In *OOPSLA Proceedings*, October 1998.

[8] I. Forman, M. Conner, S. Danforth, and L. Raper. Release-to-Release Binary Compatibility in SOM. In *OOPSLA'95 Proceedings*, 1995.

[9] S. N. Freund and J. C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. In *OOPSLA Proceeedings*, October 1998.

[10] from Gilad Bracha. Packages break Strong Locality. In *more at http://www-dse.doc.ic.ac.uk/ sue/packages.html*, February 1999.

[11] A. Goldberg. A Specification of Java Loading and Bytecode Verification. In *5th ACM Conference on Computer and Communications Security*, 1998.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, August 1996.

[13] J. Gosling and H. McGilton. The Java white paper, http://java.sun.com/java.sun.com/whitePaper/java-whitepaper 1.html 1995.

[14] T. Jensen, D. L. Metyayer, and T. Thorn. A Formalization of Visibility and Dynamic Loading in Java. In *IEEE ICCL*, 1998.

[15] S. Liang and G. Bracha. Dynamic Class Loading in the Java^{TM} Virtual Machine. In *OOPSLA Proceedings*, October 1998.

[16] T. Nipkow and D. von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *25th POPL Proceedings*, 1998.

[17] Z. Qian. A Formal Specification of the Java Virtual Machine Instructions. http://www.informatik.uni-bremen.de/ qian/abs-fsjvm.html, 1997.

[18] Z. Qian. Least Types for Memory Locations in Java Bytecode. In *FOOL 6*. http: //www.cs.williams.edu/ kim/FOOL/sched6.html, 1999.

[19] R. Stata and M. Abadi. A Type System For Java Bytecode Subroutines. In *POPL'98 Proceedings*, January 1998.

[20] G. Steele. Private Communication, January 1998.

[21] D. Syme. Proving Java Type Sound. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer, 1999.

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

[23] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.