Java Separate Type Checking is not Safe
(Extended Abstract)

Davide Ancona, Giovanni Lagorio, and Elena Zucca*

DISI - Università di Genova
Via Dodecaneso, 35, 16146 Genova (Italy)
email: {davide,lagorio,zucca}@disi.unige.it

Abstract. Java supports *separate type-checking* in the sense that compilation can be invoked on a single source fragment, and this may enforce type-checking of other either source or binary fragments existing in the environment. However, the Java specification does not define precise rules on how this process should be performed, therefore the outcome of compilation may strongly depend on the particular compiler implementation. Furthermore, rules adopted by standard Java compilers, as SDK and Jikes, can produce binary fragments whose execution throws linking related errors. We introduce a simple framework which allows to formally express the process of separate compilation and the related formal notion of type safety. Moreover, we define, for a small subset of Java, a type system for separate compilation which we conjecture to be safe.

1 Introduction

Traditional type systems for programming languages define the well-formedness of self-contained programs, and are said to be *safe* if the (result of the compilation of) a well-typed program is guaranteed to well-behave at run time (see [6, 4, 5] for the Java case).

However, in languages supporting *separate compilation* and *dynamic linking*, like Java, this simple framework is no longer adequate. Indeed, it is possible to type-check a single source fragment in a context where other fragments are present either in source or in binary form. Hence, there are two main new ingredients to be considered in typing rules: checks can be performed not only on source, but also on binary fragments, and, for type-checking a fragment, it can be necessary to type-check other (source or binary) fragments, following some strategy.

Moreover, the output of the compilation phase is not a self-contained executable program, but a collection of binary fragments which can be linked and executed in many different ways. Hence, the type safety notion must be expressed in a more flexible form.

* Partially supported by Murst - TOSCA Teoria della Concorrenza, Linguaggi di Ordine Superiore e Strutture di Tipi.

In this paper, we introduce a simple framework for separate compilation, modeled as a function which, given a set of fragment names and a *compilation context* consisting of both source and binary fragments, produces a collection of binary fragments, and we define a related notion of type safety.

Our aim is to face the following problems related to Java separate compilation.

- There is *no* specification of separate compilation in [2], hence the outcome of compilations may strongly depend on the particular compiler implementation.
- Rules adopted by existing compilers can be quite complex and cannot be easily explained informally.
- As known by Java programmers, rules adopted by standard Java compilers, as SDK and Jikes, can produce binary fragments whose execution throws linking related errors. This seems in contradiction with the fact that type safety results have been proved for the Java language [6, 4, 5]; the explanation, as we will illustrate in more detail in the following, is that these type systems, and the related type safety results, are only related to a special case, which is the compilation of a self-contained set of source fragments.

Our framework is a formal basis for defining type systems for languages supporting separate compilation, notably Java, and formally reasoning about them by defining and proving good properties. For reasons of space, here we focus on type safety, however there exist other kinds of good properties one could expect from separate compilation (see end of Sect.3 and the Conclusion).

In order to illustrate our approach, we define, for a small Java subset, a type system for separate compilation which we conjecture to be safe (a formal proof would require the definition of a simple execution model, not considered here for lack of space).

The work presented in this paper is a first step towards the formal definition and comparison of different type systems for Java separate compilation, corresponding, e.g., either to standard Java compilers, , or to extended compilers which perform additional checks. The overall motivation of this research is the following.

As illustrated in detail in the following, standard compilers perform very few checks on binary fragments, relying on the fact that these checks can be in practice delegated to the JVM[1], which finds linking related errors and throws corresponding exceptions (see examples in Sect.2), thus guaranteeing that execution does not crash. However, we argue that this is not a good enough motivation. Indeed, the fact that the JVM has a run-time verifier (hence intercepts error situations) cannot be used as a justification for not trying to anticipate at compile-time checks which actually *can* be performed earlier; otherwise, following the same principle, one could also throw away checks on source fragments since in any case the fact that the execution does not crash is guaranteed by the bytecode verifier, hence these checks are in a sense redundant. In our opinion,

[1] Java Virtual Machine.

even though the run-time verification cannot, of course, be eliminated in Java[2], it is worthwhile to investigate the possibility of anticipate at compile-time as many checks as possible, as it is in the long tradition of type systems. The obvious advantage is earlier error detection; then, in principle, the possibility that execution in a context of "certified" bytecode fragments obtained by a "smart" compiler could be performed without some run-time checks (as it is already the case for a context of binary fragments resulting from the compilation of all source fragments).

The paper is organized as follows. In Sect.2 we present simple examples to illustrate type-checking rules adopted by the SDK and Jikes compilers and to show that these rules are not safe. In Sect.3 we introduce our framework and formally express type safety. In Sect.4 we show, for a small subset of Java, a type system for separate compilation which we conjecture to be safe. Finally, Sect.5 summarizes the contribution of the paper and outlines further work.

2 Some motivating examples

In this section we illustrate by means of some examples the type-checking rules adopted by the two Java compilers SDK 1.3 and Jikes 1.11 (which apparently seem to coincide[3]), and we show that these rules are not safe.

In the following, we will call *compilation context* all the source[4] and binary fragments which are available to the compiler (the notion will be formalized in the next section). If both the source and the corresponding binary fragment are present for a class, then standard compilers inspect the binary and ignore the source, while the source is inspected if the binary is obsolete, that is, source has been changed after last compilation.

The first example illustrates non-safe behavior due to the fact that, when checking a binary fragment, standard compilers do not enforce checking of all used fragments.

```
class A{ static void main(String[] args){new B().m();} }
class B{ int m(){return new C().m();} }
class C{ int m(){return 1;} }
```

If, in a compilation context cc_0 consisting of the three source fragments, we invoke the compiler on `A.java`, then compilation of `B.java` and `C.java` is enforced, so that, after compilation, we obtain a new context cc_1 where the binary fragments of the three classes are available. However, if we re-compile `A.java` in the context cc_2 obtained by removing from cc_1 the binary fragment of `C`, then re-compilation of `C.java` is not enforced[5], therefore we obtain again the context cc_2 (hence, no static error has been detected); however, if we try to execute class `A` in this context, then error `NoClassDefFoundError` is thrown.

[2] To deal with fragments which are not known to be the result of some compilation.
[3] Except that Jikes supports compilation options that enforce more checks.
[4] We assume for simplicity a unique file for each class.
[5] In Jikes re-compilation of `C.java` can be enforced with the option `+F` or `+U`.

Indeed, in standard compilers, when a fragment named N is checked, this always enforces (transitively) checking the parent of N, regardless N is in source or binary form[6], whereas used fragments are (transitively) checked only when N is in source form. This rule is not safe since it can lead to linking related errors, as shown above. In the type system in Sect.4, instead, parent and used fragments are always (transitively) checked.

Next examples illustrates cases in which the non-safe behavior is not related to dependencies among checking fragments, but rather to the fact that some checks which could be in principle performed on binary fragments are not actually performed.

In the context cc_1, as previously defined, assume to modify `C.java` in the following way:

```
class C{ C m(){return new C();} }
```

Let cc'_2 denote the context obtained from cc_1 by modifying the source fragment of C as shown above. If we re-compile both `A.java` and `C.java`, then we obtain a new context cc'_3 (hence, no static error has been detected). However, in this new context, the execution of class A throws `NoSuchMethodError`. The problem is that, when checking `B.class`, compilers do not check that class C should have a method `int m()`, as would be checked if only the source of B were available.

A similar situation arises in the following example:

```
class A{ static void main(String[] args){new B().m()} }
class B{ D m(){return new C();} }
class C extends D {}
class D {}
```

Assume, analogously to the example above, to first compile all fragments, then modify `C.java` as follows:

```
class C {}
```

If we re-compile `A.java` and `C.java` in this context, then we get no static error, but the execution of class A throws `VerifyError`. The problem, again, is that, when checking `B.class`, compilers do not check that class C should be a subtype of D, as would be checked if only the source of B were available.

Finally, consider the following source fragments:

```
class A{ static void main(String[] args){new B().m()} }
class B{ int m(){return new C().m();} }
class C extends D {}
class D { int m(){return 1;} }
```

[6] Hence in an analogous example where A extends B which extends C re-compilation of `C.java` would be enforced even by checking `B.class`, thus causing no run-time error.

and the situation in which we start from the context containing the source fragments above, we compile all of them, and then we remove `B.java`[7] and modify `C.java` and `D.java` as follows:

```
class D {}
class C extends D{ int m() { return 1;} }
```

Again, re-compiling `A.java`, `C.java` and `D.java` we get no static error and obtain a context in which the execution of class `A` throws `NoSuchMethodError`. Here the problem is that the call `new C().m()` in `B.class` is annotated with the class `D` where method `m` was previously declared and the JVM verifies that `m` is actually declared either in `D` or in some superclass of `D`. Note that, as in the preceding example, in presence of `B.java` the problem can be fixed by re-compiling it; in this case, however, no static error is detected, but a new binary fragment for `B` where the call is annotated with `C` is produced.

In summary, these three examples show that standard compilers do not perform on binary fragments some checks which could be possibly performed at compile-time. These are either checks which are performed on source fragments, or checks related to additional informations stored in the bytecode which make it less "abstract" w.r.t. to source. In the type system we define in the following, on the contrary, these checks on binaries are performed, hence in the three examples a static error would be raised.

As final remark, the examples above also show that rules for Java separate compilation are not trivial to understand and express and that, therefore, the behavior of the existing compilers cannot be always easily predicted; other examples, not related to violating type safety, where the compilers exhibit unexpected behavior can be found in [1].

3 Framework

We introduce now a simple framework allowing to model separate compilation and to express the property of type safety in a formal way.

*Notations.*We denote by $[A \rightharpoonup_{fin} B]$ the set of the *finite partial functions* from A into B, that is, functions from A into B which are defined on a finite subset of A. For each $f \in [A \rightharpoonup_{fin} B]$, we set $Def(f) = \{a \in A | f(a) \in B\}$. $\square$

Let us denote by $\mathbb{C}$ the set of fragment names, ranged over by c, and by $\mathbb{S}$ and $\mathbb{B}$ the set of source and binary fragments, respectively. We assume that $\mathbb{S} \cap \mathbb{B} = \emptyset$. In the Java case, fragment names will be class/interface names, source fragments will be `.java` files containing (for simplicity) exactly one class/interface declaration, and binary fragments will be `.class` files. However, the model we present is general and can be applied to fragments of different nature.

[7] In presence of `B.java` the counter-example works as well, but the error can be detected by forcing its re-compilation.

A *compilation context* cc is a pair $\langle cc_b, cc_s\rangle \in CC = [\mathbb{C} \rightharpoonup_{fin} \mathbb{B}] \times [\mathbb{C} \rightharpoonup_{fin} \mathbb{S}]$. In general $Def(cc_b) \cap Def(cc_s) \neq \emptyset$, since for some fragment both the source and the binary can be available (intuitively, this means that the binary is obsolete).

The results of (successful) compilations are finite partial functions from class names into binary fragments. Hence, we can model the compilation process by a (partial) function:

$$\mathcal{C} : \wp(\mathbb{C}) \times CC \rightharpoonup [\mathbb{C} \rightharpoonup_{fin} \mathbb{B}]$$

where $\mathcal{C}(C, \langle cc_b, cc_s\rangle) = cc'_b$ intuitively means that the compilation, invoked on fragments with names in C, in the compilation context consisting of binary fragments cc_b and source fragments cc_s, produces binary fragments cc'_b.

We introduce now the formal property of type safety for separate compilation. For our purposes, we can abstract from all details of the linking and execution model and just assume a very general judgment of the form $cc_b \vdash c \rightsquigarrow \mathrm{OK}$ which is valid if and only if execution of c in the context of binary fragments cc_b does not throw any linking related error. In the Java case, for instance, this judgment corresponds to start execution from class[8] c in a context where all binaries in cc_b are available to the JVM, hence some of them could be dynamically linked during execution.

Definition 1. *A compilation function $\mathcal{C}$ is* type safe *iff for any compilation context $\langle cc_b, cc_s\rangle$ and set of fragment names C, if $\mathcal{C}(C, \langle cc_b, cc_s\rangle) = cc'_b$, then, for any $c \in Def(cc'_b)$, $cc_b[cc'_b] \vdash c \rightsquigarrow \mathrm{OK}$.*

Note that type safety requires that execution does not raise linking related errors only when started from classes that were the product of the compilation. An error raised by an execution started from a class c present in the original binary context cc_b can be either an error which was already present (that is, $cc_b \vdash c \rightsquigarrow \mathrm{OK}$ does not hold), hence not due to compilation, or is due to the fact that some binary used by c has been modified. In this case we say that the compilation function does not satisfy *contextual binary compatibility* [1].

4 A safe type system for separate compilation

In this section, we define a type system (that we conjecture to be safe) which models separate compilation for a small Java subset.

The language we consider is shown in Fig. 1; metavariables `C`, `m`,`x` and `N` range over sets of class, method and parameter names, and integer literals, respectively. Both source and binary fragments are specified.

A source fragment `S` is a class declaration consisting of the class name, the name of the superclass and a set of method declarations. A method declaration consists of a method header and a method body (an expression). A method header consists of a (return) type, a method name and a sequence of parameter

[8] We also ignore for simplicity the fact that c should have a `main` method.

types and names. There are four kinds of expressions: instance creation, parameter name, integer literal and method invocation. A type is either a class name or `int`.

A binary fragment B consists of the name of the superclass, a set of annotated method headers and a set of type constraints KS. An annotated method header is a method header prefixed by an annotation indicating the class which contains the method declaration. A type constraint K is either a subtype constraint $\mathtt{C}_1 \leq \mathtt{C}_2$, or an implementation constraint $\mathtt{C} \lhd \mathtt{AMHS}$, stating that class C must provide annotated methods AMHS.

Note that here, for simplicity, binary fragments contain no code, but only some type information which can, however, easily retrieved from a regular Java `.class` file.

$$
\begin{array}{rcll}
\mathtt{S} & ::= & \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}\ \} & \\
\mathtt{MDS} & ::= & \mathtt{MD}_1 \ldots\ \mathtt{MD}_n & (n \geq 0) \\
\mathtt{MD} & ::= & \mathtt{MH}\ \{\ \mathtt{return\ E;}\ \} & \\
\mathtt{MH} & ::= & \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n) & (n \geq 0) \\
\mathtt{E} & ::= & \mathtt{new\ C} \mid \mathtt{x} \mid \mathtt{N} & \\
& & \mathtt{E}_0.\mathtt{m}(\mathtt{E}_1, \ldots, \mathtt{E}_n) & (n \geq 0) \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{int} & \\
& & & \\
\mathtt{B} & ::= & \langle \mathtt{C}, \mathtt{AMHS}, \mathtt{KS} \rangle & \\
\mathtt{KS} & ::= & \mathtt{K}_1 \ldots\ \mathtt{K}_n & (n \geq 0) \\
\mathtt{K} & ::= & \mathtt{C}_1 \leq \mathtt{C}_2 \mid \mathtt{C} \lhd \mathtt{AMHS} & \\
\mathtt{AMHS} & ::= & \mathtt{C}_1\ \mathtt{T}_1\ \mathtt{m}(\bar{\mathtt{T}}_1) \ldots \mathtt{C}_n\ \mathtt{T}_n\ \mathtt{m}(\bar{\mathtt{T}}_n) & (n \geq 0) \\
\bar{\mathtt{T}} & ::= & \mathtt{T}_1 \ldots \mathtt{T}_n & (n \geq 0)
\end{array}
$$

Fig. 1. Syntax and types

The top-level rules of the type system are defined in Fig.2.
The main judgment $cc \vdash \mathtt{CS} \rightsquigarrow cc_b$ is valid whenever the compilation invoked on the class names in CS in compilation context cc successfully produces the binary context cc_b.

The compilation can be split in two distinct phases; first, all classes in CS (and, implicitly, all classes which classes in CS depends on) are type-checked (hypotheses), then binary fragments are produced for all the type-checked classes which were not yet in binary form (conclusion).

The side condition $\mathtt{CS} \subseteq \mathit{Def}(cc_s)$ ensures that all classes in CS have a source fragment in cc; if not so, compilation fails, otherwise classes in CS are sequentially type-checked (hypotheses).

Judgment $cc; \Gamma \vdash \mathtt{C} \rightsquigarrow \Gamma'$ is valid whenever class C is well-typed w.r.t. compilation context cc and class environment Γ; Γ' is the new class environment produced during the type-checking of C. A class environment is a finite map associating with each class name C a pair $\langle \mathtt{C}', \mathtt{AMHS} \rangle$, where $\mathtt{C}'$ denotes the superclass of C, while AMHS is the set of all annotated method headers (either inherited

$$\frac{\langle cc_b, cc_s\rangle; \Gamma_0 \vdash \mathtt{C}_1 \leadsto \Gamma_1 \quad \ldots \quad \langle cc_b, cc_s\rangle; \Gamma_{n-1} \vdash \mathtt{C}_n \leadsto \Gamma_n}{\langle cc_b, cc_s\rangle \vdash \mathtt{CS} \leadsto cc'_b} \quad \begin{array}{l} \mathtt{CS} = \{\mathtt{C}_1, \ldots, \mathtt{C}_n\} \subseteq \mathit{Def}(cc_s) \\ \mathit{Def}(\Gamma_0) = \{\mathtt{Object}\}, \Gamma_0(\mathtt{Object}) = \langle \bot, \emptyset\rangle \\ \mathit{Def}(cc'_b) = \mathit{Def}(\Gamma_n) \setminus \mathit{Def}(cc_b) \\ \forall\, \mathtt{C} \in \mathit{Def}(cc'_b)\ cc'_b(\mathtt{C}) = \mathit{bin}(cc_s, \Gamma_n, \mathtt{C}) \end{array}$$

$$\frac{}{cc; \Gamma \vdash \mathtt{int} \leadsto \Gamma} \qquad \frac{}{cc; \Gamma \vdash \mathtt{C} \leadsto \Gamma}\ \mathtt{C} \in \mathit{Def}(\Gamma)$$

$$\frac{\langle cc_b, cc_s\rangle; \Gamma \vdash \mathtt{C}_1 \leadsto \Gamma_1 \quad \langle cc_b, cc_s\rangle; \Gamma_1[\mathtt{C} \mapsto \langle \mathtt{C}_1, \mathtt{AMHS}\rangle] \vdash \mathtt{KS} \leadsto \Gamma_2}{\langle cc_b, cc_s\rangle; \Gamma \vdash \mathtt{C} \leadsto \Gamma_2} \quad \begin{array}{l} \mathtt{C} \notin \mathit{Def}(\Gamma) \\ cc_b(\mathtt{C}) = \langle \mathtt{C}_1, \mathtt{AMHS}', \mathtt{KS}\rangle \\ \Gamma_1(\mathtt{C}_1) = \langle \_, \mathtt{AMHS}_1\rangle \\ \mathtt{AMHS}_1[\mathtt{AMHS}'] = \mathtt{AMHS} \end{array}$$

$$\frac{\langle cc_b, cc_s\rangle; \Gamma \vdash \mathtt{C}_1 \leadsto \Gamma_1 \quad \langle cc_b, cc_s\rangle; \Gamma_1[\mathtt{C} \mapsto \langle \mathtt{C}_1, \mathtt{AMHS}\rangle] \vdash \mathtt{MDS} \leadsto \Gamma_2}{\langle cc_b, cc_s\rangle; \Gamma \vdash \mathtt{C} \leadsto \Gamma_2} \quad \begin{array}{l} \mathtt{C} \notin \mathit{Def}(\Gamma) \cup \mathit{Def}(cc_b) \\ cc_s(\mathtt{C}) = \mathtt{class\ C\ extends\ C_1}\ \{\ \mathtt{MDS}\ \} \\ \mathtt{AMHS}' = \mathit{Amhs}(\mathtt{C}, \mathtt{MDS}) \\ \Gamma_1(\mathtt{C}_1) = \langle \_, \mathtt{AMHS}_1\rangle \\ \mathtt{AMHS}_1[\mathtt{AMHS}'] = \mathtt{AMHS} \end{array}$$

Fig. 2. Top-level rules

or declared) of C. Class environments model the needed type information about classes collected by the compiler while inspecting source and binary fragments in the compilation context.

The initial class environment Γ_0 (see the corresponding side condition) contains only the predefined empty class (with no superclass) Object[9]. Class environment Γ_1, produced while type-checking $\mathtt{C}_1$, contains (besides Γ_0) type information about all classes needed for type-checking $\mathtt{C}_1$: all superclasses of $\mathtt{C}_1$ and all classes used (both directly and indirectly) by $\mathtt{C}_1$.

The new class environment Γ_1 is used for checking next class $\mathtt{C}_2$ and so on, until producing an environment Γ_n containing all classes which have been type-checked; from this set we can easily retrieve the set of all classes which need to be compiled (see the side condition defining cc'_b).

The remaining rules specify type-checking of primitive types and classes. Type-checking of primitive types and or classes already collected in the class environment is trivial.

The other two rules concern classes which have not been inspected yet (the former deals with binary fragments, whereas the latter with source fragments). They are almost symmetric, except that when both binary and source fragment are present, priority is given to the former[10]. First, the direct parent class $\mathtt{C}_1$ is type-checked; then, from the annotated method headers of $\mathtt{C}_1$ and those declared in C, the annotated method headers of C are derived (and rules on overriding are

[9] For simplicity, we ignore all the predefined methods of Object.

[10] For simplicity, we assume that a binary fragment is always more recent than its corresponding source.

checked). Finally, either the set of type constraints (in the binary case) or the set of method declarations (in the source case) of C is type-checked.

For lack of space, all other rules and auxiliary functions are defined in the Appendix.

Finally we show how the second example discussed in Sect.2 can be modeled in the framework defined above.

The compilation context $cc_0 = \langle cc_b^0, cc_s^0 \rangle$ is defined by $cc_b^0 = \emptyset$, $cc_s^0 = \{\mathtt{A} \mapsto \mathtt{S_A}, \mathtt{B} \mapsto \mathtt{S_B}, \mathtt{C} \mapsto \mathtt{S_C}\}$, where $\mathtt{S_A}$, $\mathtt{S_B}$, and $\mathtt{S_C}$ are the source code of A, B, and C as defined in the example[11].

The compilation context $cc_1 = \langle cc_b^1, cc_s^0 \rangle$ (corresponding to the context after invoking the compiler on A) is obtained by updating the previous binary context cc_b^0 with the binary context cc_b derived from the judgment $cc_0 \vdash \{\mathtt{A}\} \rightsquigarrow cc_b$. Since in this case cc_b^0 is empty, we have $cc_b^1 = cc_b = \{\mathtt{A} \mapsto \mathtt{B_A}, \mathtt{B} \mapsto \mathtt{B_B}, \mathtt{C} \mapsto \mathtt{B_C}\}$, where

$$\begin{aligned}
\mathtt{B_A} &= \langle \mathtt{Object}, \{\mathtt{A\ int\ main()}\}, \{\mathtt{B} \leq \mathtt{B}, \mathtt{B} \lhd \{\mathtt{B\ int\ m()}\}\}\rangle \\
\mathtt{B_B} &= \langle \mathtt{Object}, \{\mathtt{B\ int\ m()}\}, \{\mathtt{C} \leq \mathtt{C}, \mathtt{C} \lhd \{\mathtt{C\ int\ m()}\}\}\rangle \\
\mathtt{B_C} &= \langle \mathtt{Object}, \{\mathtt{C\ int\ m()}\}, \emptyset\rangle
\end{aligned}$$

The subtype constraints $\mathtt{B} \leq \mathtt{B}$ and $\mathtt{C} \leq \mathtt{C}$ simply require the existence of class B and C, respectively, otherwise no constructor could be correctly invoked on them.

The context cc_2' is obtained from cc_1 by changing the source code of C (according to the example), therefore $cc_2' = \langle cc_b^1, cc_s^2 \rangle$, where $cc_s^2 = cc_s^0[\mathtt{S'_C}/\mathtt{C}]$ is obtained from cc_s^0 by updating C with the new source $\mathtt{S'_C}$.

Finally, A cannot be successfully compiled in context cc_2, since there is no cc_b s.t. the judgment $cc_2 \vdash \{\mathtt{A}\} \rightsquigarrow cc_b$ is valid.

5 Conclusion

We have shown that typing rules for Java separate compilation can be quite complex and cannot easily explained informally. Moreover, they can be be unsafe, as happens for SDK and Jikes compilers since they perform very few checks on binary fragments delegating them to the JVM. We argue that a more robust compiler implementation should perform as much checks as possible at compile time, delegating to the JVM only those checks that can only be performed at run time.

We have introduced a simple framework which allows to formally model separate compilation and the related properties. Within this framework, we have defined, for a small subset of Java, a type system for separate compilation which we conjecture to be type safe.

In this paper, for lack of space, we have focused on the safety property; however, there are other interesting properties one can express for separate compilation, like *contextual binary compatibility* (mentioned at the end of Sect.3) and

[11] Where, however, `static` has been removed and `void` replaced with `int`.

monotonicity, that is, the fact that when a subset of the source fragments composing a program is changed, re-compiling only this set gives the same result as re-compiling the whole program (this property is mentioned as desirable in [3] and formalized in [1]).

The work presented in this paper is a first step towards the formal definition and comparison of different type systems for Java separate compilation, corresponding, e.g., either to standard Java compilers, or to extended compilers which perform additional checks. A lot of work still has to be done. On the theoretical side, we plan to define a complete execution and linking model for the toy language defined in this paper, including a toy bytecode, thus allowing to formally prove type safety. We also want to study the formal relations between the type safety property analyzed in this paper and other properties like monotonicity and contextual binary compatibility [1]. On the practical side, we plan to extend the safe type system defined here to more relevant Java subsets and to develop extended compilers which satisfy good properties like type safety.

Acknowledgments: We warmly thank Sophia Drossopoulou for her precious contribution to stimulate and enhance this work.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Monotone separate compilation in Java. Technical Report, DISI. Submitted for publication, April 2001.
2. G. Bracha, J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification, Second Edition.* Addison-Wesley, 2000.
3. L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, January 1997.
4. S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer Verlag, Berlin, 1999.
5. D. Syme. Proving Java type sound. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 83–118. Springer Verlag, 1999.
6. D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 119–156. Springer Verlag, 1999.

A Appendix

Class environments:

$$\begin{aligned} \Gamma &::= \mathtt{C}_1{:}\langle \mathtt{C}_1^{\perp}, \mathtt{AMHS}_1\rangle, \ldots, \mathtt{C}_n{:}\langle \mathtt{C}_n^{\perp}, \mathtt{AMHS}_n\rangle \ (n \geq 0) \\ \mathtt{C}^{\perp} &::= \perp \mid \mathtt{C} \end{aligned}$$

Binary class generation:

$$bin(cc_s, \Gamma, \mathtt{C}) = \langle \mathtt{C}_1, \mathtt{AMHS}, \mathtt{KS} \rangle \qquad \begin{array}{l} \text{if } cc_s(\mathtt{C}) = \mathtt{class\ C\ extends\ C_1\ \{\ MDS\ \}} \\ \mathtt{AMHS} = Amhs(\mathtt{MDS}) \\ \Gamma \vdash \mathtt{MDS} \rightsquigarrow \mathtt{KS} \end{array}$$

Annotated methods update: A set of method headers AMHS is well-formed if it does not contain overloaded methods.

$$\mathtt{AMHS}'[\mathtt{AMHS}] = \begin{cases} \mathtt{AMHS} \to \mathtt{AMHS}' \text{ if } \mathtt{AMHS} \to \mathtt{AMHS}' \text{ is well-formed} \\ \text{undefined} \qquad \text{otherwise} \end{cases}$$

where $\mathtt{AMHS} \to \mathtt{AMHS}' = \mathtt{AMHS} \cup \{\mathtt{C\ T\ m(\bar{T})} \mid \nexists\, \mathtt{C}_1\ s.t.\ \mathtt{C}_1\ \mathtt{T\ m(\bar{T})} \in \mathtt{AMHS}\}$

Annotation and extraction of method headers:

$$Amhs(\mathtt{C}, \mathtt{MDS}) = annotate(\mathtt{C}, Mhs(\mathtt{MDS}))$$

$$annotate(\mathtt{C}, \mathtt{MH}_1\ \ldots\ \mathtt{MH}_n) = \mathtt{C}\ \mathtt{MH}_1\ \ldots\ \mathtt{C}\ \mathtt{MH}_n$$

$$Mhs(\mathtt{MH}_1\{\ \mathtt{return}\ \mathtt{E}_1;\ \}\ldots\ \mathtt{MH}_n\{\ \mathtt{return}\ \mathtt{E}_n;\ \}) = \mathtt{MH}_1 \ldots \mathtt{MH}_n$$

Method resolution

$$RetType(\Gamma, \mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n) = \mathtt{T}' \text{ if } \begin{cases} \Gamma(\mathtt{C}) = \mathtt{AMHS} \\ \mathtt{C}_1\ \mathtt{T}'\ \mathtt{m}(\mathtt{T}'_1\ \mathtt{x}_1, \ldots, \mathtt{T}'_n\ \mathtt{x}'_n) \in \mathtt{AMHS} \\ \Gamma \vdash \mathtt{T}_i \leq \mathtt{T}'_i \text{ for } i = 1..n \end{cases}$$

$$\frac{cc; \Gamma_0 \vdash \mathtt{K}_1 \rightsquigarrow \Gamma_1 \ldots cc; \Gamma_{n-1} \vdash \mathtt{K}_n \rightsquigarrow \Gamma_n}{cc; \Gamma_0 \vdash \mathtt{K}_1\ \ldots\ \mathtt{K}_n \rightsquigarrow \Gamma_n}$$

$$\frac{cc; \Gamma \vdash \mathtt{C}_1 \rightsquigarrow \Gamma_1 \quad cc; \Gamma_1 \vdash \mathtt{C}_2 \rightsquigarrow \Gamma_2 \quad \Gamma_2 \vdash \mathtt{C}_1 \leq \mathtt{C}_2}{cc; \Gamma \vdash \mathtt{C}_1 \leq \mathtt{C}_2 \rightsquigarrow \Gamma_2}$$

$$\frac{cc; \Gamma \vdash \mathtt{C} \rightsquigarrow \Gamma_1 \quad \Gamma_1 \vdash \mathtt{AMHS}_1 \lhd \mathtt{AMHS}}{cc; \Gamma \vdash \mathtt{C} \lhd \mathtt{AMHS} \rightsquigarrow \Gamma_1} \quad \Gamma_1(\mathtt{C}) = \mathtt{AMHS}_1$$

Fig. 3. Type-checking sets of constraints

$$\frac{\Gamma \vdash \mathtt{MD}_1 \leadsto \mathtt{KS}_1 \ldots \Gamma \vdash \mathtt{MD}_n \leadsto \mathtt{KS}_n}{\Gamma \vdash \mathtt{MD}_1\ \ldots\ \mathtt{MD}_n \leadsto \mathtt{KS}_1\ \ldots\ \mathtt{KS}_n}$$

$$\frac{\Gamma; \{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash \mathtt{E} : \mathtt{T} \leadsto \mathtt{KS}}{\Gamma \vdash \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{E};\ \} \leadsto \mathtt{KS}\ \mathtt{T}_1 \leq \mathtt{T}_1 \ldots \mathtt{T}_n \leq \mathtt{T}_n\ \mathtt{T} \leq \mathtt{T}_0}$$

$$\frac{}{\Gamma; \Pi \vdash \mathtt{new}\ \mathtt{C} : \mathtt{C} \leadsto \mathtt{C} \leq \mathtt{C}} \qquad \frac{}{\Gamma; \Pi \vdash \mathtt{N} : \mathtt{int} \leadsto \Lambda} \qquad \frac{}{\Gamma; \Pi \vdash \mathtt{x} : \mathtt{T} \leadsto \Lambda}\ \Pi(\mathtt{x}) = \mathtt{T}$$

$$\frac{\begin{array}{l}\Gamma; \Pi \vdash \mathtt{E}_0 : \mathtt{C} \leadsto \mathtt{KS}_0 \\ \Gamma; \Pi \vdash \mathtt{E}_1 : \mathtt{T}'_1 \leadsto \mathtt{KS}_1\ \ldots\ \Gamma; \Pi \vdash \mathtt{E}_n : \mathtt{T}'_n \leadsto \mathtt{KS}_n \\ \Gamma \vdash \mathtt{T}'_1 \leq \mathtt{T}_1\ \ldots\ \Gamma \vdash \mathtt{T}'_n \leq \mathtt{T}_n\end{array}}{\Gamma; \Pi \vdash \mathtt{E}_0.m(\mathtt{E}_1, \ldots, \mathtt{E}_n) : \mathtt{T} \leadsto \mathtt{KS}_0 \ldots \mathtt{KS}_n\ \mathtt{C} \lhd \{\mathtt{C}_1\ \mathtt{T}\ \mathtt{m}(\mathtt{T}_1 \ldots \mathtt{T}_n)\}}\ \Gamma(\mathtt{C}) = \mathtt{AMHS}_1\ \mathtt{C}_1\ \mathtt{T}\ \mathtt{m}(\mathtt{T}_1 \ldots \mathtt{T}_n)\ \mathtt{AMHS}_2$$

Fig. 4. Code generation

$$\frac{cc; \Gamma_0 \vdash \mathtt{MD}_1 \leadsto \Gamma_1 \ldots cc; \Gamma_{n-1} \vdash \mathtt{MD}_n \leadsto \Gamma_n}{cc; \Gamma_0 \vdash \mathtt{MD}_1\ \ldots\ \mathtt{MD}_n \leadsto \Gamma_n}$$

$$\frac{\begin{array}{l}cc; \Gamma \vdash \mathtt{T}_0 \leadsto \Gamma_0 \ldots cc; \Gamma \vdash \mathtt{T}_n \leadsto \Gamma_n \\ cc; \Gamma_n; \{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash \mathtt{E} : \mathtt{T} \leadsto \Gamma' \\ \Gamma' \vdash \mathtt{T} \leq \mathtt{T}_0\end{array}}{cc; \Gamma \vdash \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{E};\ \} \leadsto \Gamma'}$$

$$\frac{cc; \Gamma \vdash \mathtt{C} \leadsto \Gamma'}{cc; \Gamma; \Pi \vdash \mathtt{new}\ \mathtt{C} : \mathtt{C} \leadsto \Gamma'} \qquad \frac{}{cc; \Gamma; \Pi \vdash \mathtt{N} : \mathtt{int} \leadsto \Gamma}$$

$$\frac{}{cc; \Gamma; \Pi \vdash \mathtt{x} : \mathtt{T} \leadsto \Gamma}\ \Pi(\mathtt{x}) = \mathtt{T}$$

$$\frac{\begin{array}{l}cc; \Gamma; \Pi \vdash \mathtt{E}_0 : \mathtt{C} \leadsto \Gamma_0 \\ cc; \Gamma_0; \Pi \vdash \mathtt{E}_1 : \mathtt{T}_1 \leadsto \Gamma_1 \\ \ldots \\ cc; \Gamma_{n-1}; \Pi \vdash \mathtt{E}_n : \mathtt{T}_n \leadsto \Gamma_n\end{array}}{cc; \Gamma; \Pi \vdash \mathtt{E}_0.m(\mathtt{E}_1, \ldots, \mathtt{E}_n) : \mathtt{T} \leadsto \Gamma_n}\ \mathit{RetType}(\Gamma, \mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n) = \mathtt{T}$$

Fig. 5. Type-checking of source class bodies

$$\frac{\Gamma \vdash \mathtt{C}'_1 \leq \mathtt{C}_1 \ldots \Gamma \vdash \mathtt{C}'_n \leq \mathtt{C}_n}{\Gamma \vdash \{\mathtt{C}_1\ \mathtt{T}_1\ \mathtt{m}_1(\bar{\mathtt{T}}_1), \ldots, \mathtt{C}_k\ \mathtt{T}_k\ \mathtt{m}_k(\bar{\mathtt{T}}_k)\} \lhd \{\mathtt{C}'_1\ \mathtt{T}_1\ \mathtt{m}_1(\bar{\mathtt{T}}_1), \ldots, \mathtt{C}'_n\ \mathtt{T}_n\ \mathtt{m}_n(\bar{\mathtt{T}}_n)\}}\ n \leq k$$

$$\frac{}{\Gamma \vdash \mathtt{int} \leq \mathtt{int}} \qquad \frac{}{\Gamma \vdash \mathtt{C} \leq \mathtt{C}}\ \mathtt{C} \in \mathit{Def}(\Gamma)$$

$$\frac{}{\Gamma \vdash \mathtt{C} \leq \mathtt{C}'}\ \Gamma(\mathtt{C}) = \langle \mathtt{C}', \_ \rangle \qquad \frac{\Gamma \vdash \mathtt{C} \leq \mathtt{C}' \quad \Gamma \vdash \mathtt{C}' \leq \mathtt{C}''}{\Gamma \vdash \mathtt{C} \leq \mathtt{C}'}$$

Fig. 6. Implementation and widening