A Formal Framework for Java Separate Compilation

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. We define a formal notion, called *compilation schema*, allowing to specify different possibilities for performing the overall process of Java compilation, which includes type-checking of source fragments with generation of corresponding binary code, type-checking of binary fragments, extraction of type information from fragments and definition of dependencies among them. We consider three compilation schemata of interest for Java, that is, *minimal*, *SDK* and *safe*, which correspond to a minimal set of checks, the checks performed by the SDK implementation, and all the checks needed to prevent run-time linkage errors, respectively. In order to demonstrate our approach, we define a kernel model for Java separate compilation and execution, consisting in a small Java subset, and a simple corresponding binary language for which we provide an operational semantics including run-time verification. We define a safe compilation schema for this language and formally prove type safety.

1 Introduction

In modern programming languages, the notion of "program" as a whole has become more and more obsolete. Now, the process of developing software typically consists in writing separate pieces of code, which we will call *fragments*, following [7], each one implementing some basic functionality and relying on other functionalities provided by other fragments. The language should provide facilities which allow the development of fragments to be as much modular as possible. In particular, a highly desirable property is *separate compilation*, which means, in its strongest formulation, that a single source fragment f can be type-checked in isolation, generating a corresponding binary fragment, in a context where only type information is available on the fragments it depends on, say $f_1, \ldots, f_n$, but no code. This phase is called *intra-checking* in [7]. Then, an executable application can be constructed linking together a collection of binary fragments s.t., for each fragment f, all the fragments $f_1, \ldots, f_n$ it depends on are available and satisfy the required type assumptions. This phase is called *inter-checking* in

* Partially supported by Murst - TOSCA Teoria della Concorrenza, Linguaggi di Ordine Superiore e Strutture di Tipi and APPlied SEMantics - Esprit Working Group 26142.

[7]. Inter-checking should guarantee that the resulting application does not raise linkage errors (for instance, a fragment needed during execution either does not exist or does not provide some expected functionality).

Though Java is a widely known paradigmatic example of language supporting separate compilation, it does not match the above schema in many respects.

1. A fragment f which depends on $f_1, \ldots, f_n$ can be type-checked in absence of their sources, but at least the corresponding binaries must be available, since in Java class files play the dual roles of interfaces and object files.
2. If (some of) $f_1, \ldots, f_n$ are only available in source form, then Java compilers enforce their type-checking, hence generation of the corresponding code[1], too.
3. Linking is *dynamic*, in the sense that starting an application corresponds to running just one fragment f and then, during the execution, other needed fragments are loaded and linked on demand.

The fact (3) that linking is dynamic has a very strong impact on changing the schema we outlined above based on two distinct phases of intra-checking (separate compilation) and inter-checking (linking).

First of all, run-time inter-checks become necessary in order to guarantee that execution does not crash. For instance, the Java Virtual Machine (JVM from now on) has a bytecode verifier which finds linkage errors and throws corresponding exceptions.

On the other hand, since run-time checks are performed, inter-checks at compile-time are in a sense redundant, since in any case the fact that the execution does not crash is guaranteed by the bytecode verifier. Nevertheless, Java compilers try to anticipate at compile-time the detection of some linkage errors, performing *some* (but not all) inter-checks at compile-time. In other words, the overall compilation process is not solely specified by the type-checking rules for the source code, but some additional aspects must be modeled in order to fully capture the behavior of a compiler.

For instance, if the fragment f to be type-checked depends on a source fragment f', Java compilers act like a typical static linker, by enforcing the type-checking of f' too, in order to ensure that, if f' either is missing or does not satisfy the type requirements of f, then the error is detected before execution.

However, Java compilers take this approach only partially, and do not perform *all* the checks which would be actually possible (for instance, no checks are performed if f' is in binary form). As a consequence, as well-known to experienced Java programmers, standard Java compilers, as SDK and Jikes, are *not type safe* in the sense that they can generate binary fragments whose execution throws linkage errors. This seems in contradiction with the fact that type safety results have been proved for the Java language [18, 9, 17]; the explanation is that these formal type systems, and the related type safety results, are only related

[1] In this paper, we will always use the terminology *type-checking* a fragment meaning the process that in the case of a source fragment also includes generation of binary code.

to the special case when a closed set of source fragments is type-checked, while (1) and (2) are not taken into account (apart from [11], see Sect.5). Even worse, there is *no* specification of separate compilation in [13], hence the outcome of compilations may strongly depend on the particular compiler implementation. Moreover, as will be shown in detail in the following, rules adopted by existing compilers are quite complex and cannot be easily explained informally.

For all these reasons, we believe necessary the definition of a formal framework for Java separate compilation, providing a rigorous basis for:

- defining and investigating different possibilities for the overall compilation process (for instance: a minimal set of checks, the checks performed by standard Java compilers, as many checks as possible)
- proving desirable properties, like type safety, for a compilation process.

This is what we achieve in this paper, by modeling the overall compilation process via the formal notion of *compilation schema*. A compilation schema defines two different judgments, a *source type judgment* $\Gamma \vdash \mathtt{S} \leadsto \mathtt{B}$ and a *binary type judgment* $\Gamma \vdash \mathtt{B} \diamond$ modeling type-checking, in a given type environment Γ, of source code $\mathtt{S}$ and binary code $\mathtt{B}$, respectively; in the former case the corresponding binary code $\mathtt{B}$ is also generated. These two components model the part of Java compilation which corresponds to truly separate compilation in the sense of [7]. The fact (2) that in Java type-checking of a fragment may enforce type-checking of other fragments is modeled by another component of a compilation schema, that is, a *dependency function*. Finally, the fact (1) that in Java type information for a fragment cannot be provided separately from code is modeled by a *type extraction function* which extracts from a compilation environment ce (collection of source and binary fragments) a type environment Γ providing the type information needed for type-checking fragments in ce.

We consider three different compilation schemata for Java. The first, which we call *minimal*, corresponds to true separate type-checking, in the sense that no other fragment is type-checked when compilation is invoked on a fragment f. In this case, all inter-checks are left to the run-time verifier. However, note that some of the fragments f depends on *must* be available, since some of the type information needed for type-checking f has to be extracted from them (see Sect.5 for some more comments).

The second compilation schema is that used by standard Java compilers (at least for what we have been able to understand by experiments, since no specification is available). In this case only some inter-checks are performed: invoking compilation on f enforces type-checking of other fragments, but not of all those which could be possibly loaded at run-time; moreover no checks are performed on binary fragments, that is, the type judgment $\Gamma \vdash \mathtt{B} \diamond$ is always trivially valid. As a consequence, binaries obtained as result of the compilation are not guaranteed to safely link at run-time, as we will show by the examples in Sect.2. Finally, we propose a compilation schema which is type safe, that is, guarantees safe linking at run-time. For this last schema we provide a full definition of the four components for a small Java subset and we prove type safety.

The paper is organized as follows. In Sect.2 we illustrate by means of some examples the three different schemata. In Sect.3 we introduce our framework, formally defining the notion of compilation schema and the related type safety property. In Sect.4 we define, for a small subset of Java, a type safe compilation schema. Note that this implies that we not only define the syntax for the source level, but also for the binary level, for which we also provide, in order to express and prove type safety, a simple execution model, partly inspired by those defined in [10, 8], but much simpler, since our aim here is not to provide a realistic model of Java dynamic loading and linking, but to focus on type safety. Hence a contribution of the paper is also to define a kernel calculus for the Java Virtual Machine, in the spirit of other calculi have been defined for modeling in isolation relevant aspects of the Java language [14, 3]. In Sect.5 we discuss related work. Finally, Sect.6 summarizes the contribution of this paper and outlines further work.

2 Some motivating examples

In this section we illustrate, by means of some examples, three different Java compilation schemata, called *minimal*, *SDK* and *safe*, respectively.

As already explained, the minimal compilation schema requires the minimal amount of checks over fragments: type-checking is performed only for those source fragments on which the compiler has been explicitly invoked and no checks (except those strictly necessary for compiling the sources) are performed on binary fragments. This schema fits well in open environments where source fragments to be compiled are expected to be later dynamically linked with fragments that are not available at compile-time. For instance, assume that the class `C1` we want to compile depends on a class `C2`. Even in the case the source of `C2` is available, it could be sensible to avoid to type-check `C2` if there is a high probability that it does not correspond to the actual code that will be linked with `C1` at runtime.

The SDK schema simply corresponds to the SDK implementation of Java[2]. As already said, this schema falls in between the minimal and the safe schemata: it enforces more checks than the former but less than the latter. For instance, the compilation of a class `C` requires that all source fragments[3] directly used by `C` must be type-checked, while for all binary fragments[4] directly used by `C`, only their existence and format is checked (but no real type-check is performed).

Finally, the safe schema can be sensibly applied when we expect that the fragments that will be linked at run-time are those available after the compilation; under this assumption, it makes sense to type-check all fragments (either source or binary) used (either directly or indirectly) by a class `C`, in order to ensure that no execution of `C` will throw a linkage error (like, for instance,

[2] The examples in this paper are based on version 1.4 beta 2 of SDK.
[3] Whose corresponding binary fragment is either unavailable or older.
[4] Which either are more recent than the corresponding source fragment or do not have a corresponding source fragment.

`NoClassDefFoundError` or `NoSuchMethodError`). To this aim, the compilation schema must include all those checks on binary fragments that a safe linker would perform if Java classes were statically linked[5].

While all these three schemata share the same source type judgment and type extraction function (corresponding to the Java type system defined in [13] and formalized in, e.g., [9]), they remarkably differ in the other two components (that is, dependency function and binary type judgment) as described in the following examples.

Consider the following class declarations, that we assume each one contained in a single `.java` file:

```
class Main {
  static void main(String[] args){new Used().m();}
  void g (UsedAsType x) {}
}
class Used extends UsedParent {
  int m(){return new TransUsed().m();}
}
class UsedParent{
  int m(){return 1;}
}
class TransUsed {
  int m(){return 1;}
}
class UsedAsType { ... }
```

As already stated, the same type extraction function is shared by the three schemata. The definition is straightforward: the type environment is a mapping associating to each available class `C` the type information which can be extracted from its code, that is, a pair consisting of the direct superclass of `C` and the list of method signatures[6] directly declared in `C`, which we will call the *direct type* of `C`.

Assume now that we want to compile `Main`. In the minimal schema, our aim is just to perform the separate type-checking (intra-checking), hence we only need the type information necessary to type-check the source code of `Main`. In particular, for each class used in `Main`, there are two possible situations: either the class is used only as a type, like in the method `g`, and in this case we only need the existence of the class in the type environment, or it is used as the receiver's type in a method call, like in `new Used().m()`. In this case we need to know which are *all* the method signatures of the class (either directly declared or inherited). This information, which we will call the *full type* of a class, can be safely constructed having all the direct types of the classes in its ancestor

[5] Of course, such checks are usually performed at run-time by the JVM.

[6] In these examples we will consider for simplicity only instance method declarations, as in the Java subset defined in Sect.4; in full Java the direct type of a class would also include other declared members. We assume that the method `main` is just used for starting execution.

hierarchy and provided that this hierarchy is acyclic[7] (this will be formalized in Fig.5 later on). In summary, `Main.java` can be successfully type-checked, producing a corresponding `Main.class`, in the type environment Γ, defined by

$\{\texttt{Main} \mapsto \langle\texttt{Object}, \texttt{void g()}\rangle, \texttt{Used} \mapsto \langle\texttt{UsedParent}, \texttt{int m}()\rangle,$
$\texttt{UsedParent} \mapsto \langle\texttt{Object}, \texttt{int m}()\rangle, \texttt{UsedAsType} \mapsto \ldots\},$

which can be extracted from a compilation environment ce_1 where only `Main.java`, `Used.java`, `UsedParent.java` and `UsedAsType.java` are available. This is formalized by the validity of the judgment $\Gamma \vdash \texttt{S} \leadsto \texttt{B}$ that will be defined in Fig.3. Note that, since, as already said, in Java type information on fragments cannot be provided separately from their code, either the `.java` or `.class` files for `Used`, `UsedParent` and `UsedAsType` must be available, even though no type-checking is performed on their code (for `UsedAsType` not even the type information is used). In the minimal schema, indeed, the set of dependencies of `Main.java` is $\{\texttt{Main}\}$, reflecting the fact that we are only interested in type-checking `Main.java`.

In both the SDK and safe schema, the set of dependencies of `Main` in ce_1 includes also `Used`, `UsedParent`, `UsedAsType` and `TransUsed`. However, the type-checking of `Used` clearly fails in absence of `TransUsed`. As a consequence, compilation of `Main.java` in ce_1 fails for both the SDK and safe schema.

Let us consider two compilation environments able to discriminate between the SDK and the safe schemata. First consider ce_2 which contains `Main.java`, `Used.class`, `UsedParent.class`, `UsedAsType.class`[8], and a changed version of `TransUsed.java` which does not satisfy intra-checks (for instance, the body of method `m` returns a boolean). The type environment extracted from ce_2 is still Γ. However, the set of dependencies of `Main` in ce_2 is $\{\texttt{Main}, \texttt{Used}, \texttt{UsedParent}, \texttt{UsedAsType}, \texttt{TransUsed}\}$ in the safe schema, while it is $\{\texttt{Main}, \texttt{Used}, \texttt{UsedParent}, \texttt{UsedAsType}\}$ in the SDK schema.

Hence, in SDK the source of class `TransUsed` is not type-checked; then, a new binary fragment `Main.class` is produced. However, in the new environment obtained by enriching ce_2 with the fragment `Main.class`, the execution of class `Main` throws the error `NoClassDefFoundError` (note that this error is raised instead of a type error since the class has not been compiled, hence there is no corresponding binary), whereas this error is detected at compile-time by the safe schema which performs the type-checking of the source code of `TransUsed`.

In this case the difference between the Java and the safe schema is given by the dependency function. However, even in the case dependencies are the same, the two schemata can still behave differently due to the fact that the safe schema also performs a significant binary type-checking (formalized by the validity of the judgment $\Gamma \vdash \texttt{B} \diamond$ that will be defined in Fig.6). For instance, invoking the compilation of both `Main` and `TransUsed` in the compilation environment ce_3 which contains `Main.java`, `Used.class`, `UsedParent.class`, `UsedAsType.class` and a changed version of `TransUsed.java` which does not satisfy type requirements in `Used` (for instance, declaring `boolean m() { return true;}`), in SDK no

[7] However, in JDK 1.4 beta 2 the compiler loops if a used binary class has a cyclic inheritance hierarchy!

[8] Obtained, e.g., by compiling the whole program in the example.

checks are performed on the the binary code of `Used`. Hence, again, in the binary environment obtained after the compilation, the execution of class `Main` throws the exception `NoSuchMethodError`, whereas this error is detected at compile-time by the safe schema.

3 Framework

We now formally define our framework for modeling the Java overall compilation process. Consistently with this aim, we will everywhere use a Java-related terminology. However, most of the notions presented here could be generalized to model the compilation process of other languages, as we will briefly discuss in Sect.5.

Let us denote by $\mathbb{C}$ the set of fragment names, that is, in Java, class/interface names[9], ranged over by `C`, and by $\mathbb{S}$ and $\mathbb{B}$ the set of source and binary fragments, respectively. We assume that source fragments are `.java` files containing (for simplicity) exactly one class/interface declaration, and binary fragments are `.class` files.

A *compilation environment* ce is a pair[10]

$$\langle ce_b, ce_s \rangle \in CE = [\mathbb{C} \rightharpoonup_{fin} \mathbb{B}] \times [\mathbb{C} \rightharpoonup_{fin} \mathbb{S}]$$

s.t. $Def(ce_b) \cap Def(ce_s) = \emptyset$. We will call ce_b and ce_s a *binary* and a *source environment*, respectively. Note that the assumption $Def(ce_b) \cap Def(ce_s) = \emptyset$ means that, even in the case a class has both a binary and a source definition, the compiler considers only one of them, according to some rule.

The results of (successful) compilations are binary environments. Hence, we can model the compilation process by a (partial) function, called *compilation function*:

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

where $\mathcal{C}(\langle ce_b, ce_s \rangle, \texttt{CS}) = ce'_b$ intuitively means that the compilation, invoked on fragments with names in `CS`, in the compilation environment consisting of binary fragments ce_b and source fragments ce_s, generates binary fragments ce'_b.

We introduce now the formal notion of *compilation schema*, meant to express different Java compilation processes.

A compilation schema consists of the following four components.

- A *dependency function* $\mathcal{D}$ which gives, for any compilation environment ce and fragment name `C`, the set `CS` of all the fragment names on which type-checking is enforced when the compilation is invoked on `C`. We assume that the definition of $\mathcal{D}$ is extended in the obvious way to set of fragment names.

[9] We will consider only classes in the Java subset in Sect.4.

[10] We denote by $[A \rightharpoonup_{fin} B]$ the set of *finite partial functions* from A into B, that is, functions f from A into B which are defined on a finite subset of A, denoted $Def(f)$.

– A *type extraction function* $\mathcal{T}$ which extracts from a compilation environment ce a *type environment* Γ providing the type information necessary to type-check fragments in ce.
– A *source type judgment* $\Gamma \vdash \mathtt{S} \leadsto \mathtt{B}$ expressing that in the type environment Γ the source fragment S is successfully type-checked generating the binary fragment B.
– A *binary type judgment* $\Gamma \vdash \mathtt{B} \diamond$ expressing that in the type environment Γ the binary fragment B is successfully type-checked.

To compile a set of fragments CS in a compilation environment ce, first the needed type environment Γ is extracted applying $\mathcal{T}$ to ce. Then, all the fragments in the set $\mathtt{CS}_d$ computed using $\mathcal{D}$ are type-checked generating corresponding binaries for those which were in source form. This can be formalized by the inference rule in Fig.3 which defines a compilation function $\mathcal{C}$ in terms of the four components of a compilation schema. The second side condition, $\mathtt{CS} \subseteq \mathit{Def}(ce_s)$, states that a compilation can be only invoked on a set of sources.

$$\frac{\begin{array}{c}\forall \mathtt{C} \in \mathtt{CS}_b \ \Gamma \vdash ce_b(\mathtt{C}) \diamond \\ \forall \mathtt{C} \in \mathtt{CS}_s \ \Gamma \vdash ce_s(\mathtt{C}) \leadsto \mathtt{B}_\mathtt{C}\end{array}}{\mathcal{C}(\langle ce_b, ce_s\rangle, \mathtt{CS}) = \{\mathtt{C} \mapsto \mathtt{B}_\mathtt{C} \mid \mathtt{C} \in \mathtt{CS}_s\}} \quad \begin{array}{l}\Gamma = \mathcal{T}(\langle ce_b, ce_s\rangle) \\ \mathtt{CS} \subseteq \mathit{Def}(ce_s) \\ \mathtt{CS}_d = \mathcal{D}(\langle ce_b, ce_s\rangle, \mathtt{CS}) \\ \mathtt{CS}_b = \mathtt{CS}_d \cap \mathit{Def}(ce_b) \\ \mathtt{CS}_s = \mathtt{CS}_d \cap \mathit{Def}(ce_s)\end{array}$$

Fig. 1. Definition of compilation function

Let us now apply the above definitions for specifying the three different compilation schemata informally introduced in Sect.2.

The type extraction function is the same for the three schemata: the type environment extracted from a compilation environment $ce = \langle ce_b, ce_s\rangle$ is a finite partial function which associates to each $\mathtt{C} \in \mathit{Def}(ce_b) \cup \mathit{Def}(ce_s)$ its *direct type*, that is, a pair consisting of the superclass of C and the list of the method signatures declared in C (see the formal definition in Fig.15 later on).

Again, the source type judgment is the same for the three schemata, and corresponds to the Java type system defined in [13] and formalized, e.g., in [9]. The formalization for the small Java subset for which we define a type safe compilation schema is given in Fig.3 later on.

On the contrary, the three schemata remarkably differ in the other two components.

For what concerns the dependency function, $\mathcal{D}(ce, \mathtt{C})$ contains only C in the minimal schema; in the safe schema $\mathcal{D}(ce, \mathtt{C})$ contains C and all the classes transitively used by C, regardless that C is in source or binary form (see the formal definition in Fig.14 later on). In the SDK schema the definition is much more involved. First of all, $\mathcal{D}(ce, \mathtt{C})$ contains C and all the classes directly used by C. For each of these classes, say $\mathtt{C}'$, $\mathcal{D}(ce, \mathtt{C})$ also (recursively) contains $\mathcal{D}(ce, \mathtt{C}')$

if $\mathtt{C}'$ is in source form. If $\mathtt{C}'$ is in binary form, then the behavior is different depending whether $\mathtt{C}'$ is only used in $\mathtt{C}$ as "abstract" type (for instance, field type, parameter type, method return type) or information on the components provided by $\mathtt{C}$ is also needed (for instance, there is a method call with receiver's type $\mathtt{C}$). In the former case $\mathcal{D}(ce, \mathtt{C})$ contains only $\mathtt{C}'$, in the latter it contains $\mathtt{C}'$, all the ancestor classes of $\mathtt{C}'$ and (recursively) $\mathcal{D}(ce, \mathtt{C}'')$ for each ancestor $\mathtt{C}''$ which is in source form.

This rule is quite complex, and has been extrapolated only by performing a number of compilation tests, since it is hard to deduce it from the compiler source and no other form of documentation seems to be available.

The binary type judgment also differs from schema to schema. In the minimal schema no type-checks are performed (that is, the judgment $\Gamma \vdash \mathtt{B} \diamond$ trivially holds). In the safe schema the checks performed on a binary fragment are similar to those performed on a source fragment. A difference is, for instance, the way a method invocation is checked. In the source case the method must be found searching in all the ancestor classes, and overloading must be resolved, while in the binary case a method call is already annotated with the class where the method should be found together with its signature. The formalization for the small Java subset for which we define a type safe compilation schema is given in Fig.3 (last rule) and Fig.4 for the source case and in Fig.6 (last rule) for the binary case.

In the SDK schema no type-checks are performed on binary code. The only checks which are performed, together with the existence check on $\mathtt{C}'$, when type-checking a class $\mathtt{C}$ which uses $\mathtt{C}'$, are on the format of the binary (analogous to the fact that Java grammar is respected in the source case) and on the correspondence between the fragment name and the name of the class defined inside. In the formal model in Sect.4 we assume for simplicity that fragments are well-formed in this sense.

We introduce now the formal property of type safety for separate compilation. We assume a judgment of the form $\mathtt{C} \leadsto_{ce_b} V$ which is valid if and only if execution of class $\mathtt{C}$ in the binary environment ce_b terminates producing a value V which can be either a normal value or an exception. Intuitively, this judgment corresponds to start the execution from class $\mathtt{C}$ in an environment where the binaries available to the JVM are ce_b, hence some of them could be dynamically linked during the execution.

The formal definition of this judgment for the small Java subset for which we define a type safe compilation schema is given in Fig.13.

Definition 1. *A compilation function $\mathcal{C}$ is* type safe *iff for any compilation environment $\langle ce_b, ce_s\rangle$ and set of class names CS, if $\mathcal{C}(CS, \langle ce_b, ce_s\rangle) = ce'_b$, then, for all class names $C \in Def(ce'_b)$ and values V, if $C \leadsto_{ce_b[ce'_b]} V$, then V is not an exception.*

Note that type safety requires that execution does not throw linkage errors only when started from classes that are the product of the compilation. Indeed, an error raised by an execution started from a class $\mathtt{C}$ present in the original binary environment ce_b can be either an error which was already present, hence not due to the compilation, or an error due to the fact that some binary used by `C` has been modified. In this case we will say that the compilation function does not satisfy a different property, which we call *contextual binary compatibility* (see the Conclusion).

4 A safe compilation schema for Java

$$
\begin{array}{rcl}
\mathtt{S} & ::= & \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \}\ \mathtt{main\ E}^s \\
\mathtt{MDS}^s & ::= & \mathtt{MD}^s_1 \ldots\ \mathtt{MD}^s_n \\
\mathtt{MD}^s & ::= & \mathtt{MH}\ \{\ \mathtt{return\ E}^s;\ \} \\
\mathtt{MH} & ::= & \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n) \\
\mathtt{E}^s & ::= & \mathtt{new\ C} \mid \mathtt{x} \mid \mathtt{N} \\
 & & \mathtt{E}^s_0.\mathtt{m}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n) \\
 & & \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{int} \\
 & & \\
\mathtt{B} & ::= & \langle \mathtt{C}, \mathtt{C'}, \mathtt{MDS}^b, \mathtt{E}^b \rangle \\
\mathtt{MDS}^b & ::= & \mathtt{MD}^b_1 \ldots\ \mathtt{MD}^b_n \\
\mathtt{MD}^b & ::= & \mathtt{MH}\ \{\ \mathtt{return\ E}^b;\ \} \\
V & ::= & \mathtt{new\ C} \mid \mathtt{N} \mid \varepsilon \\
\mathtt{E}^b & ::= & V \mid \mathtt{x} \mid \mathtt{E}^b_0.\mathtt{m} \prec \boxed{\mathtt{C}}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T} \succ (\mathtt{E}^b_1, \ldots, \mathtt{E}^b_n) \\
 & & \mathtt{E}^b_0.\mathtt{m} \prec \mathtt{C}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T} \succ (\mathtt{E}^b_1, \ldots, \mathtt{E}^b_n) \mid \mathtt{new}\ \boxed{\mathtt{C}} \\
\varepsilon & ::= & \mathit{ClassNotFound} \mid \mathit{ClassCircularityError} \mid \mathit{VerifyError} \mid \mathit{NoSuchMethod} \\
 & & \\
\mathtt{MS} & ::= & \mathtt{T\ m}(\mathtt{T}_1 \ldots \mathtt{T}_n) \\
\mathtt{AMS} & ::= & \mathtt{C\ MS} \\
\mathtt{AMSS} & ::= & \mathtt{AMS}_1 \ldots\ \mathtt{AMS}_n
\end{array}
$$

Fig. 2. Syntax and types

The language we consider is a small subset of Java, defined in Fig.2; 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, a sequence of method declarations $\mathtt{MDS}^s$, and an expression $\mathtt{E}^s$ playing the role of the (static) `main` method, that for simplicity we assume present in all classes. A method declaration $\mathtt{MD}^s$ consists of a method header and a method body (an expression). A method header `MH` consists of a (return) type, a method name and a sequence of parameter types and names. There are four kinds of expression: instance creation, parameter name, integer literal and method invocation. A type is either a class name or `int`. We will use the abbreviation $\bar{\mathtt{T}}$ for $\mathtt{T}_1 \ldots \mathtt{T}_n$ in the following.

Our description of bytecode[11]. is rather abstract: we basically enrich the source language with two kinds of annotation. Each method invocation is annotated with a method descriptor, which describes the method which has been statically selected for the call. Moreover, each reference of a class name is (initially) boxed. The idea is that a reference to a class is "sealed" in a box until it has been verified (at runtime). Such a reference cannot be used until it has been unboxed.

A binary fragment $\mathtt{B}$ consists of the name of the class, the name of the superclass, a set of binary method declarations $\mathtt{MDS}^b$ and the binary expression corresponding to the method `main`. This expression is used as the starting point of the execution when the class is run. A binary method declaration $\mathtt{MD}^b$ is structurally equivalent to a source method declaration except that the body is a binary expression.

Binary expressions can be either *values*, or parameters, or (either boxed or unboxed) method calls, or an unboxed creation expression. Values correspond to the normal forms of the reduction semantics of binary fragments (defined in Fig.13), and can be either unboxed creation expressions, or integer literals, or exceptions (in case of abnormal termination).

The expressions which can only be produced by a reduction step are ε, $\mathtt{new}\ \mathtt{C}$, and $\mathtt{E}^b_0.\mathtt{m} \prec \mathtt{C}, \bar{\mathtt{T}}, \mathtt{T} \succ (\mathtt{E}^b_1, \ldots, \mathtt{E}^b_n)$.

In the last part of Fig.2 we define *method signatures* and *annotated method signatures*, which are not part of the syntax but will be used in the type judgments. A method signature $\mathtt{MS}$ is a method header without the argument names; an annotated method signature $\mathtt{AMS}$ is a method signature prefixed by an annotation indicating the class which contains the method declaration.

We start now the formal definition of the four components of our safe compilation schema, which will be used (as shown in Fig.3 in Sect.3 to define the corresponding compilation function.

The dependency function $\mathcal{D}$ is defined as follows:

$$\mathcal{D}(ce, \mathtt{CS}) = \{\mathtt{C}' \mid \exists \mathtt{C} \in \mathtt{CS}\ s.t.\ \mathtt{C} \xrightarrow{*}_{ce} \mathtt{C}'\}$$

where $\xrightarrow{*}_{ce}$ is the reflexive and transitive closure of the relation $\rightarrow_{ce}$ defined by $\mathtt{C} \rightarrow_{ce} \mathtt{C}'$ iff $\mathtt{C}' \in \mathit{refClasses}(\mathtt{C}, ce)$. This latter function, defined in Fig.14 in the Appendix, gives the set of all classes explicitly mentioned explicitly in the code of $\mathtt{C}$.

A type environment Γ is a finite (partial) function from class names into *direct types* of classes, that is, pairs $\langle \mathtt{C}', \mathtt{MS}_1 \ldots \mathtt{MS}_n \rangle$ where $\mathtt{C}'$ is the (direct) superclass of $\mathtt{C}$ and $\mathtt{MS}_1 \ldots \mathtt{MS}_n$ is the list of the signatures of methods declared in $\mathtt{C}$.

The type extraction function $\mathcal{T}$, defined in Fig.15 in the Appendix, simply throws away method bodies and parameter names, retaining type information from all classes in the compilation environment.

[11] For simplicity, we do not distinguish, at the syntax level, between the language originally contained in binary fragments and the run-time language used in execution

$$\frac{\Gamma \vdash \mathtt{C} ::_\diamond \_ \quad \Gamma \vdash \mathtt{MDS}^s \leadsto \mathtt{MDS}^b \quad \Gamma;\emptyset \vdash \mathtt{E}^s : \_ \leadsto \mathtt{E}^b}{\Gamma \vdash \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \}\ \mathtt{main}\ \mathtt{E}^s \leadsto \langle \mathtt{C}, \mathtt{C'}, \mathtt{MDS}^b, \mathtt{E}^b \rangle}$$

$$\frac{\forall i \in 1..n\ \Gamma \vdash \mathtt{MD}_i^s \leadsto \mathtt{MD}_i^b}{\Gamma \vdash \mathtt{MD}_1^s\ \ldots\ \mathtt{MD}_n^s \leadsto \mathtt{MD}_1^b \ldots \mathtt{MD}_n^b}$$

$$\frac{\begin{array}{l}\Gamma; \{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash \mathtt{E}^s : \mathtt{T} \leadsto \mathtt{E}^b \\ \Gamma \vdash \mathtt{T} \leq \mathtt{T}_0\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{E}^s;\ \} \leadsto \\ \qquad \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{E}^b;\ \}\end{array}} \quad \forall i \in 1..n\ \mathtt{T}_i \in \mathit{Def}(\Gamma) \cup \{\mathtt{int}\}$$

$$\frac{}{\Gamma; \Pi \vdash \mathtt{new\ C} : \mathtt{C} \leadsto \mathtt{new}\ \boxed{\mathtt{C}}} \quad \mathtt{C} \in \mathit{Def}(\Gamma)$$

$$\frac{}{\Gamma; \Pi \vdash \mathtt{N} : \mathtt{int} \leadsto \mathtt{N}}$$

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

$$\frac{\begin{array}{l}\Gamma; \Pi \vdash \mathtt{E}_0^s : \mathtt{C} \leadsto \mathtt{E}_i^b \\ \forall i \in 1..n\ \Gamma; \Pi \vdash \mathtt{E}_i^s : \mathtt{T}_i \leadsto \mathtt{E}_i^b\end{array}}{\begin{array}{l}\Gamma; \Pi \vdash \mathtt{E}_0^s.m(\mathtt{E}_1^s, \ldots, \mathtt{E}_n^s) : \mathtt{T'} \leadsto \\ \qquad \mathtt{E}_0^b.m \prec \mathtt{C'}, \bar{\mathtt{T}}', \mathtt{T'} \succ (\mathtt{E}_1^b, \ldots, \mathtt{E}_n^b)\end{array}} \quad \prec \mathtt{C'}, \bar{\mathtt{T}}', \mathtt{T'} \succ = \mathit{methRes}(\Gamma, \mathtt{C}, m, \mathtt{T}_1 \ldots \mathtt{T}_n)$$

Fig. 3. Source type-judgment

In Fig.3 are defined the rules for type-checking, in a given type environment, source fragments (with generation of the corresponding binary code).

The first rule defines the type-checking of a class declaration.

First, it is checked that C has a well-formed *full type* in Γ. The full type of a class is the list of all the annotated method signatures of the class, either directly declared or inherited, and it can be safely constructed if there are no cycles in the inheritance hierarchy of C and the Java rules on Java method overriding are respected. This is formalized by the judgment $\Gamma \vdash \mathtt{C} ::_{\diamond} \_$ defined in Fig.7.

Then, the method bodies and the main expression are checked and compiled.

The second rule defines the type-checking of a sequence of method declarations. Each method declaration is correct (third rule) if the type of the expression body is a subtype of the declared return type (premises) and all argument types exist in Γ (note that *no* check is performed on the fact that they have well-formed full types). ir wewhether they are well-formed in Γ or not. The judgment $\Gamma \vdash \mathtt{T} \leq \mathtt{T}'$ is valid whenever T is a subtype of $\mathtt{T}'$ in the type environment Γ, and is defined in Fig.7.

Other rules define the type-cheking of expressions, which also needs a local type environment Π which is a (partial) function from parameters into types.

An instance creation expression, `new C`, is well-typed, and has type C, in Γ and Π if C exists in Γ. An integer literal is trivially well-typed, and has type `int`, in every Γ and Π).

A parameter is well-typed in Γ and Π if it belongs to the domain of the local type environment, and it has the corresponding type.

A method call expression is type-checked in two steps: first, the receiver expression and all the argument expressions are type-checked finding their types $\mathtt{T}_i$ (and generating the corresponding binary expressions $\mathtt{E}^b_i$). Then, using this information, the most specific among the applicable methods is selected, as formally defined by the function *methRes*, defined in Fig.4, which returns the corresponding method descriptor δ. This descriptor is used to annotate the binary method call produced as the result of the compilationas annotation of the method invocation expression; the annotation is used at runtime by the JVM (see Fig.13). Moreover, the third component of the descriptor (return type of the selected method) gives the return type of method call.

In Fig.5 we define the judgment $\Gamma \vdash \mathtt{C} :: \mathtt{AMSS}$, associating to a class its full type AMSS, and the judgment $\Gamma \vdash \mathtt{C} ::_{\diamond} \mathtt{AMSS}$, which is valid only if AMSS is well-formed.

As already said, the full type of a class consists of the sequence of the annotated signatures of the methods either directly declared in C or inherited.

A full class type AMSS is well-formed if it does not contain duplicate method signatures (predicate *noRep*) and if the Java rules on overriding are satisfied (predicate *okOverride*).

In Fig.5 we have used the notation $\mathtt{AMS} \in \mathtt{AMSS}$ as a shortcut for $\exists \mathtt{AMSS}_0, \mathtt{AMSS}_1 : \mathtt{AMSS} = \mathtt{AMSS}_0\ \mathtt{AMS}\ \mathtt{AMSS}_1$.

Note that neither $\Gamma \vdash \mathtt{C} :: \mathtt{AMSS}$ nor $\Gamma \vdash \mathtt{C} ::_{\diamond} \mathtt{AMSS}$ can be deduced for C if it has a cyclic inheritance hierarchy in Γ.

$$methRes(\Gamma, \mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \begin{cases} \prec \boxed{\mathtt{C}'}, \bar{\mathtt{T}}', \mathtt{T}' \succ \text{ if } \Gamma \vdash \mathtt{C} :: \mathtt{AMSS} \wedge \\ \qquad\qquad\qquad mostSpec(\Gamma, appMeth(\Gamma, \mathtt{AMSS}, \mathtt{m}, \bar{\mathtt{T}})) = \mathtt{C}'\ \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}') \\ \bot \qquad\qquad\quad \text{otherwise} \end{cases}$$

$$appMeth(\Gamma, \Lambda, \mathtt{m}, \bar{\mathtt{T}}) = \Lambda$$

$$appMeth(\Gamma, \mathtt{C}\ \mathtt{T}\ \mathtt{m}'(\bar{\mathtt{T}}')\ \mathtt{AMSS}, \mathtt{m}, \bar{\mathtt{T}}) = \begin{cases} \mathtt{C}\ \mathtt{T}\ \mathtt{m}'(\bar{\mathtt{T}}')\ appMeth(\Gamma, \mathtt{AMSS}, \mathtt{m}, \bar{\mathtt{T}}) \text{ if } \mathtt{m} = \mathtt{m}' \wedge \Gamma \vdash \bar{\mathtt{T}} \leq \bar{\mathtt{T}}' \\ appMeth(\Gamma, \mathtt{AMSS}, \mathtt{m}, \bar{\mathtt{T}}) \qquad\qquad \text{otherwise} \end{cases}$$

$$mostSpec(\Gamma, \Lambda) = \bot$$
$$mostSpec(\Gamma, \mathtt{C}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})) = \mathtt{C}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})$$
$$mostSpec(\Gamma, \mathtt{C}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})\ \mathtt{AMSS}) = min(\Gamma, \mathtt{C}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}), mostSpec(\Gamma, \mathtt{AMSS})) \text{ with } \mathtt{AMSS} \neq \Lambda$$

$$min(\Gamma, \mathtt{C}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}), \mathtt{C}'\ \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}')) = \begin{cases} \mathtt{C}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) & \text{if } \Gamma \vdash \mathtt{C}\ \bar{\mathtt{T}} \leq \mathtt{C}'\ \bar{\mathtt{T}}' \\ \mathtt{C}'\ \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}') & \text{if } \Gamma \vdash \mathtt{C}'\ \bar{\mathtt{T}}' \leq \mathtt{C}\ \bar{\mathtt{T}} \\ \bot & \text{otherwise} \end{cases}$$

Fig. 4. Definition of *methRes*

$$\frac{}{\Gamma \vdash \mathtt{Object} :: \Lambda}$$

$$\frac{\Gamma \vdash \mathtt{C}' :: \mathtt{AMSS}'}{\Gamma \vdash \mathtt{C} :: \mathtt{AMSS}'\ \mathtt{C}\ \mathtt{MS}_1\ \ldots\ \mathtt{C}\ \mathtt{MS}_n} \qquad \Gamma(\mathtt{C}) = \langle \mathtt{C}', \mathtt{MS}_1 \ldots \mathtt{MS}_n \rangle$$

$$\frac{\Gamma \vdash \mathtt{C} :: \mathtt{AMSS}}{\Gamma \vdash \mathtt{C} ::_\diamond \mathtt{AMSS}} \qquad noRep(\mathtt{AMSS}) \wedge okOverride(\mathtt{AMSS})$$

$$noRep(\mathtt{AMS}_1 \ldots \mathtt{AMS}_n) \iff \forall i, j \in 1..n\ \mathtt{AMS}_i = \mathtt{AMS}_j \implies i = j$$

$$okOverride(\mathtt{AMSS}) \iff$$
$$\forall \mathtt{AMS}, \mathtt{AMS}' \in \mathtt{AMSS}\ \mathtt{AMS} = \mathtt{C}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \wedge \mathtt{AMS}' = \mathtt{C}'\ \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}) \implies \mathtt{T} = \mathtt{T}'$$

Fig. 5. Full type of a class

$$\frac{\mathit{\Gamma} \vdash \mathtt{C} ::_{\diamond} \_ \quad \mathit{\Gamma} \vdash \mathtt{MDS}^b \ \diamond \quad \mathit{\Gamma}; \emptyset \vdash \mathtt{E}^b : \_ \diamond}{\mathit{\Gamma} \vdash \langle \mathtt{C}, \mathtt{C}', \mathtt{MDS}^b, \mathtt{E}^b \rangle \diamond}$$

$$\frac{\forall i \in 1..n \ \mathit{\Gamma} \vdash \mathtt{MD}_i^b \diamond}{\mathit{\Gamma} \vdash \mathtt{MD}_1^b \ \ldots \ \mathtt{MD}_n^b \diamond}$$

$$\frac{\begin{array}{l} \mathit{\Gamma}; \{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash \mathtt{E}^b : \mathtt{T} \diamond \\ \mathit{\Gamma} \vdash \mathtt{T} \leq \mathtt{T}_0 \end{array}}{\mathit{\Gamma} \vdash \mathtt{T}_0 \ \mathtt{m}(\mathtt{T}_1 \ \mathtt{x}_1, \ldots, \mathtt{T}_n \ \mathtt{x}_n) \ \{ \ \mathtt{return} \ \mathtt{E}^b; \ \} \diamond} \qquad \forall i \in 1..n \ \mathtt{T}_i \in \mathit{Def}(\mathit{\Gamma}) \cup \{\mathtt{int}\}$$

$$\frac{}{\mathit{\Gamma}; \mathit{\Pi} \vdash \mathtt{new} \ \boxed{\mathtt{C}} : \mathtt{C} \diamond} \qquad \mathtt{C} \in \mathit{Def}(\mathit{\Gamma})$$

$$\frac{}{\mathit{\Gamma}; \mathit{\Pi} \vdash \mathtt{N} : \mathtt{int} \diamond}$$

$$\frac{}{\mathit{\Gamma}; \mathit{\Pi} \vdash \mathtt{x} : \mathtt{T} \diamond} \qquad \mathit{\Pi}(\mathtt{x}) = \mathtt{T}$$

$$\frac{\begin{array}{l} \mathit{\Gamma}; \mathit{\Pi} \vdash \mathtt{E}_0^b : \mathtt{C} \diamond \\ \forall i \in 1..n \ \mathit{\Gamma}; \mathit{\Pi} \vdash \mathtt{E}_i^b : \mathtt{T}_i \diamond \\ \forall i \in 1..n \ \mathit{\Gamma} \vdash \mathtt{T}_i \leq \mathtt{T}'_i \\ \mathit{\Gamma} \vdash \mathtt{C} \lhd \mathtt{C}' \ \mathtt{T}' \ m(\mathtt{T}'_1 \ldots \mathtt{T}'_n) \end{array}}{\mathit{\Gamma}; \mathit{\Pi} \vdash \mathtt{E}_0^b.m \prec \mathtt{C}', \mathtt{T}'_1 \ldots \mathtt{T}'_n, \mathtt{T}' \succ (\mathtt{E}_1^b, \ldots, \mathtt{E}_n^b) : \mathtt{T}' \diamond}$$

Fig. 6. Binary type-judgment

Fig.6 shows the rules for type-checking binary fragments, which are analogous to those for source fragments shown in Fig.3, except for the last rule, concerning method calls.

Indeed, as already mentioned, in a binary method call the descriptor annotation indicates exactly which method to look for, and we only have to check that the types of the receiver expression and of the parameters are subtypes of those specified in the descriptor, and the class of the actual receiver $\mathtt{T}_0$ still implements such a method, premise $\Gamma \vdash \mathtt{T}_0 \lhd \mathtt{T}'_0\ \mathtt{T}'\ m(\mathtt{T}'_1 \ldots \mathtt{T}'_n)$. This last judgment is defined in Fig.7.

$$\frac{}{\Gamma \vdash \mathtt{int} \leq \mathtt{int}}$$

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

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

$$\frac{\Gamma \vdash \mathtt{C} \leq \mathtt{C}' \quad \Gamma \vdash \mathtt{C}' \leq \mathtt{C}''}{\Gamma \vdash \mathtt{C} \leq \mathtt{C}'}$$

$$\frac{\forall i \in 1..n\ \Gamma \vdash \mathtt{T}_i \leq \mathtt{T}'_i}{\Gamma \vdash \mathtt{T}_1 \ldots \mathtt{T}_n \leq \mathtt{T}'_1 \ldots \mathtt{T}'_n}$$

$$\frac{\Gamma \vdash \mathtt{C} \leq \mathtt{C}_k}{\Gamma \vdash \{\mathtt{C}_1\ \mathtt{MS}_1, \ldots, \mathtt{C}_n\ \mathtt{MS}_n\} \lhd \mathtt{C}\ \mathtt{MS}} \qquad k \in 1..n \wedge \mathtt{MS} = \mathtt{MS}_k$$

$$\frac{\Gamma \vdash \mathtt{C} :: \mathtt{AMSS} \quad \Gamma \vdash \mathtt{AMSS} \lhd \mathtt{AMS}}{\Gamma \vdash \mathtt{C} \lhd \mathtt{AMS}}$$

Fig. 7. Implementation and widening

The judgment $\Gamma \vdash \mathtt{int} \leq \mathtt{int}$ is trivially valid in every Γ.

Every class $\mathtt{C}$ defined in Γ is a subtype of itself (second rule) and of its (direct) superclass (third rule). Note that every class in Γ is considered subclass of itself, even if its inheritance hierachy is cyclical, because this does not lead to wrong type assumption. Vice versa, a class $\mathtt{C}$ is considered subclass of $\mathtt{C}'$ only if its inheritance relation is acyclical.

Subtyping relation is transitive, fourth rule. The fifth rule extends the subtype relation to tuples of types.

The judgment $\Gamma \vdash \{\mathtt{C}_1\ \mathtt{MS}_1, \ldots, \mathtt{C}_n\ \mathtt{MS}_n\} \lhd \mathtt{C}\ \mathtt{MS}$ is valid whenever one of the annotated method signatures $\mathtt{C}_i\ \mathtt{MS}_i$ "implements" the annotated method signature $\mathtt{C}\ \mathtt{MS}$, that is, there is a method with the same signature in $\mathtt{C}$ or any of its superclasses. If a class has full type $\mathtt{AMSS}$ and the latter implements the method signature $\mathtt{AMS}'$, then the class is said to implement $\mathtt{AMS}'$.

The last rule defines the judgment $\mathit{\Gamma} \vdash \mathtt{C} \lhd \mathtt{AMS}'$, that is valid whenever class $\mathtt{C}$ implements $\mathtt{AMS}'$.

We define now execution and verification of binary fragments. We premit some auxiliary definitions.

$$S_1 \cup^E S_2 =_{\text{def}} \begin{cases} S_1 & \text{if } S_1 \in \varepsilon \\ S_2 & \text{if } S_2 \in \varepsilon, S_1 \notin \varepsilon \\ S_1 \cup S_2 & \text{otherwise} \end{cases}$$

$$b_1 \wedge^E b_2 =_{\text{def}} \begin{cases} b_1 & \text{if } b_1 \in \varepsilon \\ b_2 & \text{if } b_2 \in \varepsilon, b_1 \notin \varepsilon \\ \mathit{Ok} & \text{otherwise} \end{cases}$$

Fig. 8. Exception-aware union and disjunction

Fig.8 shows the definition of the two operations $\cup^E$ and $\wedge^E$, whose arguments are either set of class names or exceptions.

The former is just set union when both arguments are not exceptions, whereas it returns one of its arguments when it is an exception (giving priority to the left argument).

The latter is similar to a boolean conjunction; it returns *Ok* when both arguments are *Ok* and one of its argument when it is an exception (again, giving priority to the left argument).

$$\mathit{MBody}(ce_b, \mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T}) = \begin{cases} \mathit{NoSuchMethod} & \text{if } \mathtt{C} = \mathtt{Object} \\ \mathit{ClassNotFound} & \text{if } \mathtt{C} \neq \mathtt{Object}, \mathtt{C} \notin \mathit{Def}(ce_b) \\ \langle \mathtt{E}^b, \mathtt{x}_1 \ldots \mathtt{x}_n \rangle & \text{if } \mathtt{T}\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\{\mathtt{E}^b\} \in \mathit{code}(ce_b(\mathtt{C})) \\ \mathit{MBody}(ce_b, \mathit{superclass}(ce_b(\mathtt{C})), \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T}) & \text{otherwise} \end{cases}$$

$$\mathit{code}(\langle \_, \_, \mathtt{MDS}^b, \_ \rangle) = \mathtt{MDS}^b$$

$$\mathit{superclass}(\langle \_, \mathtt{C}', \_, \_ \rangle) = \mathtt{C}'$$

Fig. 9. Definition of *MBody*

The function $\mathit{MBody}(ce_b, \mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T})$, defined in Fig.9, models method look-up at run-time; it searches in the binary environment ce_b for the body of a method named $\mathtt{m}$ whose argument types are $\mathtt{T}_1 \ldots \mathtt{T}_n$ and return type is $\mathtt{T}$, starting from class $\mathtt{C}$. If such a method is not found in $\mathtt{C}$, then it is searched, recursively, in its superclass.

The result of *MBody* can be either the body of the method (if found) or an exception if it is not found. There are two kinds of error that can happen during the method look-up: if $\mathtt{C} = \mathtt{Object}$ then this means that the method cannot be found, so the exception *NoSuchMethod* is returned. Otherwise, if C, cannot be found in the binary environment, then the exception *ClassNotFound* is returned.

$$
\begin{array}{lll}
[\cdot]^{Exp} & ::= & [\cdot].m \prec \boxed{\mathtt{C}}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T} \succ (\mathtt{E}^b{}_1, \ldots, \mathtt{E}^b{}_n) \mid \\
 & & \mathtt{new}\ \mathtt{C.m} \prec \boxed{\mathtt{C}'}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T} \succ (v_1, \ldots, v_{i-1}, [\cdot], \mathtt{E}^b_{i+1}, \ldots, \mathtt{E}^b_n) \\
[\cdot]^{Type} & ::= & \mathtt{new}\ \mathtt{C.m} \prec [\cdot], \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T} \succ (v_1, \ldots, v_n) \mid \\
 & & \mathtt{new}\ [\cdot]
\end{array}
$$

Fig. 10. Contexts

In Fig.10 we introduce two kinds of contexts: expression contexts $[\cdot]^{Exp}$ and type contexts $[\cdot]^{Type}$. In rewrite semantics, given in fig. Fig.13, the former are used to propagate execution to sub-expressions and the latter to verify class references in order to unbox them (making it possible to "normal" execution to proceed). The function *Weak_subtype* is used, at verification time, to check

$$
\mathit{Weak\_subtype}(\mathtt{T}, \mathtt{T}', ce_b) = \begin{cases}
\mathit{Ok} & \text{if } \mathtt{T} = \mathtt{T}' \\
\varepsilon & \text{if } \mathtt{T} \neq \mathtt{T}',\ \mathit{Supertypes}(\mathtt{T}, ce_b) = \varepsilon \\
\mathit{Ok} & \text{if } \mathtt{T} \neq \mathtt{T}', \mathtt{T}' \in \mathit{Def}(ce_b), \\
 & \quad \mathtt{T}' \in \mathit{Supertypes}(\mathtt{T}, ce_b) \\
\mathit{VerifyError} & \text{if } \mathtt{T} \neq \mathtt{T}', \mathtt{T}' \in \mathit{Def}(ce_b), \\
 & \quad \mathtt{T}' \notin \mathit{Supertypes}(\mathtt{T}, ce_b) \\
\mathit{ClassNotFound} & \text{otherwise}
\end{cases}
$$

$$
\begin{array}{l}
\mathit{Supertypes}(\mathtt{T}, ce_b) = \mathit{Supertypes}_{aux}(\mathtt{T}, \emptyset, ce_b) \\
\mathit{Supertypes}_{aux}(\mathtt{T}, LC, ce_b) = \\
\qquad \begin{cases}
\mathit{ClassCircularityError} & \text{if } \mathtt{T} \in LC \\
\mathit{ClassNotFound} & \text{if } \mathtt{T} \notin \mathit{Def}(ce_b) \cup \{\mathtt{int}, \mathtt{Object}\} \\
\{\mathtt{Object}\} \cup LC & \text{if } \mathtt{T} = \mathtt{Object} \\
\{\mathtt{int}\} & \text{if } \mathtt{T} = \mathtt{int} \\
\{\mathtt{T}\} \cup^E \mathit{Supertypes}_{aux}(\mathit{superclass}(ce_b(\mathtt{T})), \{\mathtt{T}\} \cup LC, ce_b) & \text{if } \mathtt{T} \notin LC, \mathtt{T} \in \mathit{Def}(ce_b)
\end{cases}
\end{array}
$$

Fig. 11. *Weak_subtype* and *Supertypes* definitions

whether a type is subtype of another; $\mathit{Weak\_subtype}(\mathtt{T}, \mathtt{T}', ce_b)$, defined in Fig.11, returns *Ok* when the type T is a subtype of T′ in the binary environment ce_b or an appropriate exception when it is not. The subtype is "weak" because of the special case $\mathtt{T} = \mathtt{T}'$: any type T is *always* considered subtype of itself; it does not even matter whether T exists or not in the binary environment ce_b. In all the other cases both T and T′ must exist, otherwise an exception *ClassNotFound* is returned. When both exist in ce_b, the auxiliary function *Supertypes* is used to check the relationship between T and T′; indeed, T is subtype of T′ iff T′ is a supertype of T. The function $\mathit{Supertypes}(\mathtt{T}, ce_b)$ can either return the set of supertypes of T in ce_b, when they can be computed, or an exception in case of error. There are two possible error situations: when a class (directly or indirectly) extends itself and when a parent class is not found in ce_b; in these cases the exceptions *ClassCircularityError* and *ClassNotFound* are, respectively, returned. Fig.12 shows the verification judgments. The top-level judgment $\vdash_{ce_b} \mathtt{C} : \mathit{Ok}$

$$\frac{\forall i \in 1..n \;\; \vdash_{ce_b} \mathtt{MD}^b_i : b_i \quad \emptyset \vdash_{ce_b} \mathtt{E}^b : \langle \_, b\rangle}{\vdash_{ce_b} \mathtt{C} : \mathit{noDup}(\mathtt{MD}^b{}_1 \ldots \mathtt{MD}^b{}_n) \bigwedge^E_{i \in 1..n} b_i \wedge^E b} \qquad ce_b(\mathtt{C}) = \langle \_, \_, \mathtt{MD}^b{}_1 \ldots \mathtt{MD}^b{}_n, \mathtt{E}^b\rangle$$

$$\frac{}{\vdash_{ce_b} \mathtt{C} : \mathit{ClassNotFound}} \qquad \mathtt{C} \notin \mathit{Def}(ce_b)$$

$$\frac{\{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash_{ce_b} \mathtt{E}^b : \langle \mathtt{T}, b\rangle}{\vdash_{ce_b} \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\mathtt{return}\ \mathtt{E}^b;\} : b \wedge^E \mathit{Weak\_subtype}(\mathtt{T}, \mathtt{T}_0, ce_b)}$$

$$\frac{\forall i \in 0..n \;\; \sigma \vdash_{ce_b} \mathtt{E}^b_i : \langle \mathtt{T}_i, b_i\rangle}{\sigma \vdash_{ce_b} \mathtt{E}^b_0.\mathtt{m} \prec \boxed{\mathtt{T}'_0}, \mathtt{T'}_1 \ldots \mathtt{T'}_n, \mathtt{T} \succ (\mathtt{E}^b{}_1, \ldots, \mathtt{E}^b{}_n) : \langle \mathtt{T}, \bigwedge^E_{i \in \{0,\ldots,n\}} s_i\rangle} \qquad \begin{array}{l} s_i = b_i \wedge^E \\ \quad \mathit{Weak\_subtype}(\mathtt{T}_i, \mathtt{T}'_i, ce_b) \end{array}$$

$$\frac{}{\sigma \vdash_{ce_b} \mathtt{E}^b_0.\mathtt{m} \prec \boxed{\mathtt{T}'_0}, \mathtt{T'}_1 \ldots \mathtt{T'}_n, \mathtt{T} \succ (\mathtt{E}^b{}_1, \ldots, \mathtt{E}^b{}_k) : \langle \mathtt{T}, \mathit{VerifyError}\rangle} \qquad k \neq n$$

$$\frac{}{\sigma \vdash_{ce_b} n : \langle \mathtt{int}, \mathit{Ok}\rangle} \qquad n = 0, 1, -1, 2, -2, \ldots$$

$$\frac{}{\sigma \vdash_{ce_b} \mathtt{new}\ \boxed{\mathtt{C}} : \langle \mathtt{C}, \mathit{Ok}\rangle}$$

$$\frac{}{\sigma \vdash_{ce_b} \mathtt{x} : \langle \sigma(\mathtt{x}), \mathit{Ok}\rangle} \qquad \mathtt{x} \in \mathit{Def}(\sigma)$$

$$\mathit{noDup}(\mathtt{MD}^b{}_1 \ldots \mathtt{MD}^b{}_n) = \begin{cases} \mathit{Ok} & \text{if } \forall i, j \in 1..n \;\; \mathit{methSig}(\mathtt{MD}^b_i) = \mathit{methSig}(\mathtt{MD}^b_j) \implies i = j \\ \mathit{VerifyError} & \text{otherwise} \end{cases}$$

$$\mathit{methSig}(\mathtt{T}\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\{\mathtt{E}^b\}) = \mathtt{T}\ \mathtt{m}(\mathtt{T}_1 \ldots \mathtt{T}_n)$$

Fig. 12. Verification

is valid whenever the class C can be verified in the binary environment ce_b, otherwise it is valid a judgment $\vdash_{ce_b} \mathtt{C} : \varepsilon$, where ε indicates the error occurred in the verification steps. That is, verification process is deterministic and always terminates (either with *Ok* or with an exception). When a class exists in the current binary environment ce_b its verification consists of: checking that there are no different method declarations with the same signature in the code of the class (predicate *noDup*) and verifying that all method declaration ($\vdash_{ce_b} \mathtt{MD}^b_i : b_i$) and the class' main expression ($\emptyset \vdash_{ce_b} \mathtt{E}^b : \langle \_, b\rangle$) are *Ok*. When a class does not exists in a binary environment ce_b its verification simply gives *ClassNotFound* (second metarule). The judgment $\sigma \vdash_{ce_b} \mathtt{E}^s : \langle \mathtt{T}, b\rangle$ is valid whenever the expression $\mathtt{E}^s$ in a binary context ce_b and local type environment σ has type T and the result of its verification is b. The value b can be either Ok, when the verification succeeds, or an exception, indicating the problem, when the verification fails. In this latter case the value of T is immaterial. A local type environment σ is a (partial) function from argument names to types. The verification of a method declaration (third rule) succeeds when the verification of its body succeeds and the type of the body is a subtype of the declared return type. The verification of a method invocation succeeds when the number of actual arguments is the same of the number of arguments in the method descriptor, the verification of the receiver and of each argument succeeds and the type of the receiver and of each actual argument is a subtype of the corresponding type contained in the method descriptor. The third rule covers the case in which the number of actual arguments is the same of the number of arguments in the descriptor, while the fourth one covers the "bad" cases when the numbers of arguments differs (side condition $k \neq n$). Note that if a binary fragment is the result of a compilation of a source fragment, the number of actual arguments is indeed equal to the number of arguments in the descriptor; only in "malicious" binary fragments it may be found such a mismatch. Fig.13 show the rewriting rules for the program

$$\frac{\vdash_{ce_b} \mathtt{C} : Ok \quad \mathtt{E}^b \overset{*}{\leadsto}_{ce_b} V}{\mathtt{C} \leadsto_{ce_b} V} \qquad ce_b(\mathtt{C}) = \langle \_, \_, \_, \mathtt{E}^b\rangle$$

$$\frac{\vdash_{ce_b} \mathtt{C} : \varepsilon}{\mathtt{C} \leadsto_{ce_b} \varepsilon}$$

$$\frac{\vdash_{ce_b} \mathtt{C} : \varepsilon}{[\boxed{\mathtt{C}}]^{Type} \leadsto_{ce_b} \varepsilon}$$

$$\frac{\vdash_{ce_b} \mathtt{C} : Ok}{[\boxed{\mathtt{C}}]^{Type} \leadsto_{ce_b} [\mathtt{C}]^{Type}}$$

$$\frac{\mathtt{E}^b \leadsto_{ce_b} \mathtt{E}_1^b}{[\mathtt{E}^b]^{Exp} \leadsto_{ce_b} [\mathtt{E}_1^b]^{Exp}} \qquad \mathtt{E}_1^b \neq \varepsilon$$

$$\frac{\mathtt{E}^b \leadsto_{ce_b} \varepsilon}{[\mathtt{E}^b]^{Exp} \leadsto_{ce_b} \varepsilon}$$

$$\frac{}{\mathtt{new}\ \mathtt{C}.m \prec \mathtt{C}', \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T} \succ (v_1, \ldots, v_n) \leadsto_{ce_b} \varepsilon} \qquad MBody(ce_b, \mathtt{C}', \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T}) = \varepsilon$$

$$\frac{\mathtt{E}^b[v_1/\mathtt{x}_1, \ldots, v_n/\mathtt{x}_n] \leadsto_{ce_b} \mathtt{E}_1^b}{\mathtt{new}\ \mathtt{C}.m \prec \mathtt{C}', \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T} \succ (v_1, \ldots, v_n) \leadsto_{ce_b} \mathtt{E}_1^b} \qquad \begin{array}{l} MBody(ce_b, \mathtt{C}', \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T}) \neq \varepsilon \\ MBody(ce_b, \mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T}) = \langle \mathtt{E}^b, \mathtt{x}_1 \ldots \mathtt{x}_n\rangle \end{array}$$

Fig. 13. Rewriting

execution. The first two rules cover the loading/verification process. The former is used in case of error: the whole term is rewritten in the exception thrown by the verifier. The latter is used when the verification is carried out successfully; in this case the term is rewritten in itself except for the reference to the class C that is unboxed. The third rule is just the standard closure. The fourth rule propagate an exception rewriting an entire term containing an exception ε in the exception itself. The fifth and the sixth rules deal with method invocation. When the method cannot be found starting the search from the class contained in the method descriptor the entire expression is rewritten in the exception; otherwise a second call to *MBody*, passing as starting class the dynamic type of the receiver, returns the method body and the name of the arguments. These information are used to expand the method call.

4.1 Main Results

We prove three main theorems claiming the safety of source and binary type-checking and of the safe compilation schema, respectively; the former two theorems are necessary for proving the latter.

Theorem 1 (Safe Source Type-Checking). *For all type environment Γ, sources S and binaries B, if $\Gamma \vdash S \leadsto B$, then $\Gamma \vdash B \diamond$.*

Theorem 2 (Safe Binary Type-Checking). *Let $\langle ce_b, ce_s \rangle$ and C be a compilation environment and a class name, respectively. For all values V, if $\forall C' \in \mathcal{D}(\langle ce_b, ce_s \rangle, C) \quad \mathcal{T}(\langle ce_b, ce_s \rangle) \vdash ce_b(C') \diamond$ and $C \leadsto_{ce_b} V$, then V is not an exception.*

We can state know the main property of the safe compilation schema: if a set of classes is successfully compiled w.r.t. the safe schema, then the execution of any binary produced by such compilation in the updated binary environment never throws a linkage exception.

Theorem 3 (Safety). *Let $\langle ce_b, ce_s \rangle$, CS and ce'_b be a compilation environment, a set of class names and a binary environment, respectively. For all class names $C \in Def(ce'_b)$ and values V, if $\mathcal{C}(\langle ce_b, ce_s \rangle, CS) = ce'_b$ and $C \leadsto_{ce_b[ce'_b]} V$, then V is not an exception.*

4.2 Formal Proofs (Sketched)

Safe Source Type-Checking: For reasons of space we omit the proof, which is by induction over the rules defining the source type checking judgment $\Gamma \vdash \mathtt{S} \leadsto \mathtt{B}$. The proof uses a number of lemmas claiming the same property for all kinds of subcomponents of a source; the most interesting one concerns expressions.

Lemma 1. *For all type and variable environments Γ and Π, source and binary expressions $\mathtt{E}^s$ and $\mathtt{E}^b$ and types T, if $\Gamma; \Pi \vdash \mathtt{E}^s : T \leadsto \mathtt{E}^b$ then $\Gamma; \Pi \vdash \mathtt{E}^b : T \diamond$.*

Safe Binary Type-Checking: Safety of binary type-checking comes from the following two theorems, the former connecting static with dynamic binary type-checking (that is, the binary type checking judgment $\Gamma \vdash \mathtt{B} \diamond$ with the verification judgment $\vdash_{ce_b} \mathtt{C} : b$), the latter expressing subject reduction for binary expressions. For reasons of space we omit the proofs of these two theorems; the former can be proved by induction over the rules for binary type-checking, while the latter can be proved in the standard way, that is, by induction over the rewriting rules for binary expressions.

Theorem 4 (Binary Type-Checking Implies Verification). *Let* $\langle ce_b, ce_s \rangle$ *and* C *be a compilation environment and a class name, respectively, s.t. the following condition holds:* $\mathcal{D}(\langle ce_b, ce_s \rangle, C) \subseteq \mathit{Def}(ce_b)$*. If* $\mathcal{T}(\langle ce_b, ce_s \rangle) \vdash ce_b(C) \diamond$*, then* $\vdash_{ce_b} C : \mathit{Ok}$.

Note that the converse implication does not hold, since type-checking at run time is more accurate than that at compile time.

Theorem 5 (Binary Subject Reduction). *Let* $\langle ce_b, ce_s \rangle$ *and* $\mathtt{E}^b$ *be a compilation environment and a binary expression, respectively. If* $\mathcal{T}(\langle ce_b, ce_s \rangle); \emptyset \vdash \mathtt{E}^b : T \diamond$ *and* $\mathtt{E}^b \rightsquigarrow_{ce_b} \mathtt{E}^{b'}$*, then there exists a type* T' *s.t.* $\mathcal{T}(\langle ce_b, ce_s \rangle); \emptyset \vdash \mathtt{E}^{b'} : T' \diamond$ *and* $\mathcal{T}(\langle ce_b, ce_s \rangle) \vdash T' \leq T$.

We are now able to prove safety of binary type-checking. Let us assume that $\forall \mathtt{C}' \in \mathcal{D}(\langle ce_b, ce_s \rangle, \mathtt{C}) \;\; \mathcal{T}(\langle ce_b, ce_s \rangle) \vdash ce_b(\mathtt{C}') \diamond$ and $\mathtt{C} \rightsquigarrow_{ce_b} V$. From the first assumption we easily deduce $\mathcal{D}(\langle ce_b, ce_s \rangle, \mathtt{C}) \subseteq \mathit{Def}(ce_b)$ and $\mathcal{T}(\langle ce_b, ce_s \rangle) \vdash ce_b(\mathtt{C}) \diamond$ (since, trivially, $\mathtt{C} \in \mathcal{D}(\langle ce_b, ce_s \rangle, \mathtt{C})$).
As a consequence, theorem 4 can be applied, therefore $\vdash_{ce_b} \mathtt{C} : \mathit{Ok}$ holds. This means that $\mathtt{C} \rightsquigarrow_{ce_b} V$ has been deduced by instantiating the first (and not the second) meta-rule in Fig.13, so $\mathtt{E}^b \overset{*}{\rightsquigarrow}_{ce_b} V$ must hold. Furthermore, the validity of $\mathcal{T}(\langle ce_b, ce_s \rangle) \vdash ce_b(\mathtt{C}) \diamond$ implies the validity of $\mathcal{T}(\langle ce_b, ce_s \rangle); \emptyset \vdash \mathtt{E}^b : \mathtt{T} \diamond$, since there is only one meta-rule that can be instantiated in Fig.6. Therefore we can apply theorem 5 and deduce the validity of $\mathcal{T}(\langle ce_b, ce_s \rangle); \emptyset \vdash V : \mathtt{T}' \diamond$, with $\mathtt{T}'$ subtype of $\mathtt{T}$. Since exceptions do not type-check (see rules in Fig.6), we can conclude that V is not an exception.

Safety: To prove safety we need two lemmas claiming that both the dependency and type extraction functions are invariant w.r.t. source type-checking. These lemmas can be proved by induction over the definition of $\mathcal{D}$ and $\mathcal{T}$, respectively; for reasons of space we omit their proofs.

Lemma 2. *Let* $\langle ce_b, ce_s \rangle$*,* C *and* B *be a compilation environment, a class name, and a binary fragment, respectively. If* $\mathcal{T}(\langle ce_b, ce_s \rangle) \vdash ce_b(C) \rightsquigarrow B$*, then for all class name* C' *the following equality holds:*

$$\mathcal{D}(\langle ce_b, ce_s \rangle, C') = \mathcal{D}(\langle ce_b[C \mapsto B], ce_{s \setminus C} \rangle, C').$$

Lemma 3. *Let $\langle ce_b, ce_s\rangle$, C and B be a compilation environment, a class name, and a binary fragment, respectively. If $\mathcal{T}(\langle ce_b, ce_s\rangle) \vdash ce_b(C) \rightsquigarrow B$, then the following equality holds:*

$$\mathcal{T}(\langle ce_b, ce_s\rangle) = \mathcal{T}(\langle ce_b[C \mapsto B], ce_{s\setminus C}\rangle).$$

Now assume that $\mathcal{C}(\langle ce_b, ce_s\rangle, \mathtt{CS}) = ce'_b$. By virtue of the top-level rule in Fig.3, the following judgments are valid:

$$\begin{array}{l}\forall \mathtt{C} \in \mathtt{CS}_b\ \Gamma \vdash ce_b(\mathtt{C}) \diamond \\ \forall \mathtt{C} \in \mathtt{CS}_s\ \Gamma \vdash ce_s(\mathtt{C}) \rightsquigarrow \mathtt{B}_\mathtt{C}\end{array}$$

where $\mathtt{CS}_b = \mathtt{CS}_d \cap \mathit{Def}(ce_b)$, $\mathtt{CS}_s = \mathtt{CS}_d \cap \mathit{Def}(ce_s)$, $\mathtt{CS}_d = \mathcal{D}(\langle ce_b, ce_s\rangle, \mathtt{CS})$ and $\Gamma = \mathcal{T}(\langle ce_b, ce_s\rangle)$. Furthermore, $ce'_b(\mathtt{C}) = \{\mathtt{C} \mapsto \mathtt{B}_\mathtt{C} \mid \mathtt{C} \in \mathtt{CS}_s\}$.

By Theorem 1, $\Gamma \vdash ce'_b(\mathtt{C}) \diamond$ for all $\mathtt{C} \in \mathtt{CS}_s$, therefore we can easily deduce $\Gamma \vdash ce_b[ce'_b](\mathtt{C}) \diamond$ for all $\mathtt{C} \in \mathtt{CS}_d$.

Let us now prove the main theorem by assuming that $\mathtt{C}$ is a class name in $\mathtt{CS}_b$ (recall that $\mathit{Def}(ce'_b) = \mathtt{CS}_b$) and that $\mathtt{C} \rightsquigarrow_{ce_b[ce'_b]} V$ for a certain value V. By lemmas 2 and 3 and by induction on the cardinality of $\mathtt{CS}$, $\mathcal{D}(\langle ce_b, ce_s\rangle, \mathtt{C}) = \mathcal{D}(\langle ce_b[ce'_b], ce_{s\setminus\mathtt{CS}}\rangle, \mathtt{C})$ and $\mathcal{T}(\langle ce_b, ce_s\rangle) = \mathcal{T}(\langle ce_b[ce'_b], ce_{s\setminus\mathtt{CS}}\rangle)$. Therefore we can apply Theorem 2 and conclude that V cannot be an exception.

5 Related Work

We already mentioned in the Introduction that the seminal paper on separate type-checking of fragments is [7]. There, the basic idea is to distinguish a phase of *intra-checking*, which models separate compilation, in which a single fragment is type-checked w.r.t. a typing environment (which expresses the interface of the fragment in terms of both imported and exported services), and a phase of *inter-checking* which models (static) linking, in which it is checked that all the fragments we want to link have been type-checked w.r.t. compatible type environments.

Formally[12], intra-checking is modeled by a judgment $\Gamma \vdash f : T$ (in [7] issues of code generation are avoided by always working at the source level, even when discussing linking), expressing that the fragment f has type T in the type environment Γ. Inter-checking takes place on *linksets* which are, roughly, collections of named fragment $x_i \mapsto \Gamma_i \vdash f_i : T_i^{i\in 1..n}$, and succeeds if and only if intra-checking succeeds (that is, each $\Gamma_i \vdash f_i : T_i$ holds) and, moreover, for each $j, k \in 1..n$, x_j has type T_j in Γ_k. This corresponds to require *exact* agreement among the actual interface of a fragment and that assumed in another: in realistic systems, this condition should be weakened, for instance requiring some subtyping relation (see below for the Java case).

As already discussed in the Introduction, Java has many features which make this view not immediately applicable: class files play the dual roles of interfaces

[12] We use slightly different notations from [7] in order to conform to those used in this paper.

(type environments) and object files; there is no separate linking phase, since linking takes place at run-time; compilers usually incorporate *some* inter-checks, but not enough to guarantee safe run-time linking.

In this paper, we present a framework which models separate compilation in the Java sense. Though a detailed formal comparison of our framework with that in [7] is matter of further work, we can list the main contact points and differences.

- The intra-checking phase is modeled in the same way, apart from the fact that here we are interested in distinguishing source and code fragments, hence in modeling code generation.
- Instead of starting from a fixed linkset, here we assume that the set of fragments to be linked is determined by the dependency function.
- Most importantly, instead of having that each fragment f_i is equipped with its own type environment Γ_i (expressing its interface), here we assume a *global* type environment. This reflects the fact that in Java there is no separate notion of interface[13] of a fragment (class $\mathtt{C}$) describing both imported and exported services, but this interface must in some way be extracted from the code. Here, since our aim was to model separate compilation as it happens in Java rather than to compare with truly separate compilation in the sense of [7], we have taken the simpler approach to type-check all the fragments in the same type environment, which trivially consists in the compilation environment where we drop method bodies. Hence the phase which consists in checking compatibility of different interfaces is trivial. An approach more in the spirit of [7], which we leave to further investigation, would be to extract the interface Γ_i for each class $\mathtt{C}_i$ containing the minimal assumptions needed for successfully type-checking $\mathtt{C}_i$, and then to check that different interfaces are compatible. This approach poses non trivial problems both on how to perform the type extraction and how to define the "right" subtyping relation among interfaces. However, the investigation is very interesting because it could lead to innovative techniques for Java compilation supporting truly separate compilation and then the possibility of performing static checks on binaries.

Another important stream of research related to this paper is that devoted to the formal definition of Java (see [1] for a survey). As already mentioned, the type judgment which we consider at the source level is based (though much simpler, since we consider a small Java subset) on the many existing formal Java type systems, in particular those in [9, 2]. For what concerns an integrated formal model covering all Java aspects, the most remarkable amount of work in this direction is that of Sophia Drossopoulou and her group. The already cited [9] provides a formal type system at the source level for a substantial subset $\mathrm{Java}^{\mathsf{s}}$ of Java and a translation of this language into a binary language $\mathrm{Java}^{\mathsf{b}}$, which is in turn a subset of a language $\mathrm{Java}^{\mathsf{r}}$ of run-time terms for which an operational semantics is given. This allows to prove type safety of the Java subset. In [12,

[13] Not to be confused with a Java interface.

11] the focus is on binary compatibility. In [8] a model is defined for dynamic loading and linking, distinguishing five components in a Java implementation: evaluation, resolution, loading, verification, and preparation, with their associated checks. These five together are proved to guarantee type soundness. This paper is the most important reference for the execution model of our small binary language; however, in our case the main aim is not to define a realistic model of the JVM, taking into account all features, but to show how absence of linkage errors can be guaranteed by a compilation schema, so we take a much more abstract view. Finally, [11] enhances the previous formal description of Java in [9], introducing, among other improvements, an account of separate compilation. Indeed, type information used in type checking Java$^{\mathsf{s}}$ can also be extracted from the binary language Java$^{\mathsf{r}}$, analogously to what we do in this paper by means of the $\mathcal{T}$ function. However, the judgment for type-checking source classes defined in [11] do not correspond to separate compilation as happens in our framework, simply because its validity requires the type environment extracted from the compilation environment to be well-formed.

Finally, several interesting papers can be found in literature on separate compilation for ML (see among many others [16, 15, 6]). All these papers clearly show that separate compilation in ML is not a simple issue, and for this reason, needs to be properly formalized. However, ML separate compilation is based on traditional static linking, therefore many problems arising in Java disappear in ML; for instance, the static type-checks proposed in [16] are sensible for a static linker, but cannot be performed at run-time by a virtual machine without seriously compromising efficiency. Furthermore, it seems that no unifying frameworks have been defined for investigating ML separate compilation, and in fact, this would be useful to compare all the technical results and to understand how they can contribute all together to the design of a better compiler/linker for ML. For instance, using the terminology used in our paper to model the overall compilation process, [16] is mainly concerned with the definition of the type extraction function, while [15] with the type-checking of sources and [6] with the definition of the dependency function.

6 Conclusion

We have introduced a formal framework modeling Java separate compilation. The overall compilation process is modeled by the formal notion of compilation schema, in which the aspects which concern truly separate type-checking of fragments (source type judgment and binary type judgment) are isolated from the definition of dependencies and extraction from the type information from the fragments. We have considered three compilation schemata of interest for Java, that is, minimal, SDK, and safe, correspondingly to perform, when a single fragment is type-checked, no type-checks on other fragments, only the type-checks performed by SDK, and enough type-checks in order to ensure absence of linkage errors at run-time.

In order to demonstrate our approach, we have defined a kernel model for Java separate compilation and execution, consisting in a small Java subset, and a simple corresponding binary language for which we provide an operational semantics including run-time verification. We have defined a safe compilation schema for this language and formally proved type safety.

In this paper we have focused on the safety property (a preliminary work pointing out that Java compilers are not safe in the context of separate compilation was [4]); however, there are other interesting properties of compilation schemata we want to investigate, like *monotonicity* and *contextual binary compatibility*. By monotonicity we mean 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 [7], and a preliminary formalization is given in [5]). By *contextual binary compatibility* we mean a property analogous to Java binary compatibility, but related to compilation: we say that a compilation schema respects contextual binary compatibility if all the binary fragments which could be safely linked before compilation still safely link after. Note that, even though a formal analysis of their relation is still matter of further work (see below), these three properties seem at a first sight to be independent.

The work presented in this paper is a first step, and many interesting research directions are open. On the more theoretical side, we plan to formally analyze the difference between separate compilation in the Java sense, modeled in this paper, and truly separate compilation in the sense of, e.g., [7]. This should lead to the definition of a very abstract framework, like that in [7], but including dynamic linking and verification, in which we will express formal properties like type safety, monotonicity and contextual binary compatibility and analyze their relations.

On the side of application to Java, we plan to extend the safe type system defined here to more substantial Java subsets and to develop extended compilers which satisfy good properties like type safety.

References

1. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. Number 1523 in Lecture Notes in Computer Science. Springer, 1999.
2. D. Ancona, G. Lagorio, and E. Zucca. Jam: A smooth extension of Java with mixins. In E. Bertino, editor, *ECOOP'00 - European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 154–178. Springer, 2000.
3. D. Ancona, G. Lagorio, and E. Zucca. A core calculus for Java exceptions. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 2001*. ACM Press, October 2001. SIGPLAN Notices.
4. D. Ancona, G. Lagorio, and E. Zucca. Java separate type checking is not safe. In *3th Intl. Workshop on Formal Techniques for Java Programs 2001*, June 2001.
5. D. Ancona, G. Lagorio, and E. Zucca. Separate compilation in Java: Avoiding ambiguity via monotonicity. Technical Report, DISI, July 2001.

6. M. Blume. Dependency analysis for standard ML. *ACM Transactions on Programming Languages and Systems*, 21(4):790–812, 1999.
7. L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, 1997.
8. S. Drossopoulou. Towards an abstract model of Java dynamic linking and verfication. In R. Harper, editor, *TIC'00 - Third Workshop on Types in Compilation (Selected Papers)*, volume 2071 of *Lecture Notes in Computer Science*, pages 53–84. Springer, 2001.
9. 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, 1999.
10. S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus - towards a model of separate compilation, linking and binary compatibility. In *Proc. 14th Ann. IEEE Symp. on Logic in Computer Science*, July 1999.
11. S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Dept. of Computing - Imperial College of Science, Technology and Medicine, September 2000.
12. S. Drossopoulou, D. Wragg, and S. Eisenbach. What *is* Java Binary Compatibility? In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*, volume 33(10) of *ACM SIGPLAN Notices*, pages 341–358, October 1998.
13. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification, Second Edition.* Addison-Wesley, 2000.
14. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.
15. X. Leroy. Manifest types, modules and separate compilation. In *ACM Symp. on Principles of Programming Languages 1994*, pages 109–122. ACM Press, 1994.
16. Z. Shao and A.W. Appel. Smartest recompilation. In *ACM Symp. on Principles of Programming Languages 1993*, pages 439–450. ACM Press, 1993.
17. 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, 1999.
18. 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, 1999.

A Appendix

$$refClasses(\mathtt{C}, \langle ce_b, ce_s\rangle) = \begin{cases} refClasses(ce_b(\mathtt{C})) & \text{if } \mathtt{C} \in Def(ce_b) \\ refClasses(ce_s(\mathtt{C})) & \text{if } \mathtt{C} \in Def(ce_s) \\ \emptyset & \text{otherwise} \end{cases}$$

$$refClasses(\texttt{class C extends C}' \; \{ \; \mathtt{MDS}^s \; \} \; \texttt{main } \mathtt{E}^s) = \{\mathtt{C}, \mathtt{C}'\} \cup refClasses(\mathtt{MDS}^s) \cup refClasses(\mathtt{E}^s)$$
$$refClasses(\mathtt{MD}_1^s \ldots \; \mathtt{MD}_n^s) = \bigcup_{i \in \{1,\ldots,n\}} refClasses(\mathtt{MD}_i^s)$$
$$refClasses(\mathtt{MH} \; \{ \; \texttt{return } \mathtt{E}^s; \; \}) = refClasses(\mathtt{MH}) \cup refClasses(\mathtt{E}^s)$$
$$refClasses(\mathtt{T}_0 \; \mathtt{m}(\mathtt{T}_1 \; \mathtt{x}_1, \ldots, \mathtt{T}_n \; \mathtt{x}_n)) = \{\mathtt{T}_0, \ldots, \mathtt{T}_n\}$$
$$refClasses(\texttt{new C}) = \{\mathtt{C}\}$$
$$refClasses(\mathtt{x}) = refClasses(\mathtt{N}) = \emptyset$$
$$refClasses(\mathtt{E}_0^s.\mathtt{m}(\mathtt{E}_1^s, \ldots, \mathtt{E}_n^s)) = \bigcup_{i \in \{0,\ldots,n\}} refClasses(\mathtt{E}_i^s)$$

$$refClasses(\langle \mathtt{C}, \mathtt{C}', \mathtt{MDS}^b, \mathtt{E}^b\rangle) = \{\mathtt{C}, \mathtt{C}'\} \cup refClasses(\mathtt{MDS}^b) \cup refClasses(\mathtt{E}^b)$$
$$refClasses(\mathtt{MD}_1^b \ldots \; \mathtt{MD}_n^b) = \bigcup_{i \in \{1,\ldots,n\}} refClasses(\mathtt{MD}_i^b)$$
$$refClasses(\mathtt{MH} \; \{ \; \texttt{return } \mathtt{E}^b; \; \}) = refClasses(\mathtt{MH}) \cup refClasses(\mathtt{E}^b)$$
$$refClasses(\mathtt{E}_0^b.\mathtt{m} \prec \boxed{\mathtt{C}}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T}_0 \succ (\mathtt{E}_1^b, \ldots, \mathtt{E}_n^b)) = \{\mathtt{C}\} \cup \bigcup_{i \in \{0,\ldots,n\}} (refClasses(\mathtt{E}_i^s) \cup \{\mathtt{T}_i\})$$
$$refClasses(\texttt{new } \boxed{\mathtt{C}}) = \{\mathtt{C}\}$$

Fig. 14. Definition of the dependency function

$$\forall \mathtt{C} \; \mathcal{T}(\langle ce_b, ce_s\rangle)(\mathtt{C}) = \begin{cases} \langle \mathtt{C}', \mathcal{T}(\mathtt{MDS}^b)\rangle & \text{if } ce_b(\mathtt{C}) = \langle \mathtt{C}, \mathtt{C}', \mathtt{MDS}^b, \mathtt{E}^b\rangle \\ \langle \mathtt{C}', \mathcal{T}(\mathtt{MDS}^s)\rangle & \text{if } ce_s(\mathtt{C}) = \texttt{class C extends C}' \; \{ \; \mathtt{MDS}^s \; \} \; \texttt{main } \mathtt{E}^s \\ \bot & \text{otherwise} \end{cases}$$
$$\mathcal{T}(\mathtt{MD}_1^s \ldots \; \mathtt{MD}_n^s) = \mathcal{T}(\mathtt{MD}_1^s) \ldots \; \mathcal{T}(\mathtt{MD}_n^s)$$
$$\mathcal{T}(\mathtt{MD}_1^b \ldots \; \mathtt{MD}_n^b) = \mathcal{T}(\mathtt{MD}_1^b) \ldots \; \mathcal{T}(\mathtt{MD}_n^b)$$
$$\mathcal{T}(\mathtt{MH} \; \{ \; \texttt{return } \mathtt{E}^s; \; \}) = \mathcal{T}(\mathtt{MH})$$
$$\mathcal{T}(\mathtt{MH} \; \{ \; \texttt{return } \mathtt{E}^b; \; \}) = \mathcal{T}(\mathtt{MH})$$
$$\mathcal{T}(\mathtt{T}_0 \; \mathtt{m}(\mathtt{T}_1 \; \mathtt{x}_1, \ldots, \mathtt{T}_n \; \mathtt{x}_n)) = \mathtt{T}_0 \; \mathtt{m}(\mathtt{T}_1 \ldots \mathtt{T}_n)$$

Fig. 15. Definition of the type extraction function