True Separate Compilation of Java Classes*

D. Ancona
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
davide@disi.unige.it

G. Lagorio
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
lagorio@disi.unige.it

E. Zucca
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
zucca@disi.unige.it

ABSTRACT

We define a type system modeling true separate compilation for a small but significant Java subset, in the sense that a single class declaration can be intra-checked (following the Cardelli's terminology) and compiled providing a minimal set of type requirements on missing classes. These requirements are specified by a local type environment associated with each single class, while in the existing formal definitions of the Java type system classes are typed in a global type environment containing all the type information on a closed program. We also provide formal rules for static inter-checking and relate our approach with compilation of closed programs, by proving that we get the same results.

Categories and Subject Descriptors

F.3 [**Logics and meanings of programs**]: Studies of Program Constructs—*object-oriented constructs, type structure*

General Terms

Languages, Theory

Keywords

Types, separate compilation, object-oriented programming

1. INTRODUCTION

1.1 What separate compilation means

In the seminal paper [3], Cardelli has discussed and made precise the notion of *separate compilation*, which is one of the most desired properties of modern programming environments, especially in contexts where dynamic reconfiguration and mobility is allowed; we adopt his approach and terminology in this paper, with some slight adjustment.

By separate compilation we mean the separate typechecking of source fragments including generation of binary code (e.g., bytecode in the Java case). A source fragment cannot be compiled in isolation, but it can be compiled in an environment where adequate type information about missing fragments is available. This can be modeled by a judgment of the form $\Gamma \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$, where $\mathtt{S}$ is the source code, τ is the fragment's inferred type, $\mathtt{B}$ is the generated binary code[1] and Γ is a type environment playing the role of interface, providing information on the fragments whose names appear as free variables in $\mathtt{S}$. Note that this view abstracts from the concrete way in which this type environment is obtained, either from an ad-hoc interface file written by the programmer (as, for instance, in Modula-2 or Ada), or by extracting or inferring this information from the source code of the fragment which is typechecked and/or from others (see the discussion on the Java case in the following subsection).

The process of separate compilation described above is also called in [3] *intra-checking*, to stress that only the checks related to the internal coherence of a single fragment are performed, under some assumptions on the others.

Of course, a single binary code fragment $\mathtt{B}$ obtained from separate compilation is (in general) incomplete, and cannot be executed in isolation. In order to be sure that we have a collection of fragments which can be successfully linked together, *inter-checking* must be performed. Assume to have a collection of fragments with associated names, say, $f_1, \ldots, f_n$ (a *linkset* following the terminology in [3]), which have been successfully intra-checked, formally deriving named judgments $f_1 \mapsto \Gamma_1 \vdash \mathtt{S}_1 : \tau_1 \leadsto \mathtt{B}_1, \ldots, f_n \mapsto \Gamma_n \vdash \mathtt{S}_n : \tau_n \leadsto \mathtt{B}_n$. Then, inter-checking consists in checking that, for each $i = 1..n$, the inferred type τ_i of the $i-$th fragment conforms to the assumptions made on f_i in all $\Gamma_1, \ldots, \Gamma_n$. In the simple example language adopted in [3], where a type environment consists of a sequence of judgments of the form $f : \tau$, and there is no subtyping, this mutual consistency check amounts to require that in each Γ_j the required type for f_i, if any, must be τ_i; of course this definition needs to be refined in more complex type systems.

In strongly typed languages supporting static linking, inter-checking is actually performed by the linker, and this also includes generation of an executable program from the single binary fragments $\mathtt{B}_1, \ldots, \mathtt{B}_n$. Inter-checking should guarantee that the resulting program never raises linkage errors

*Partially supported by DART - Dynamic Assembly, Reconfiguration and Type-checking - EC project IST-2001-33477, Murst NAPOLI - Network Aware Programming: Oggetti, Linguaggi, Implementazioni, and APPlied SEMantics - Esprit Working Group 26142.

[1] In the original formulation in [3] compilation is simplified to typechecking.

PPDP'02, October 6-8, 2002, Pittsburgh, Pennsylvania, USA.

at run-time. In languages supporting dynamic linking, as in Java, there is no assembly of binary code before execution. However, inter-checking, if performed, still guarantees safe execution, in the following sense. In an execution environment where the available code fragments are some $f_1 \mapsto \mathtt{B}_1, \ldots, f_n \mapsto \mathtt{B}_n$ which have been inter-checked as described above, then starting the execution from any[2] $\mathtt{B}_i$ never raises linkage errors. We will discuss in the following whether and how inter-checking is performed in standard Java programming environments.

1.2 Does Java support true separate compilation?

Though Java is widely known as a paradigmatic example of language supporting separate compilation, neither standard Java compilers nor existing formal definitions of the Java type system (e.g., [4]; see [7] for more references) match the above schema. First, let us briefly recall how a standard Java compiler works. Assume for simplicity that source fragments coincide with `.java` files containing exactly one class or interface declaration[3], and that we invoke the compiler on only one class, say `C`. First of all, the classes $\mathtt{C}_1, \ldots, \mathtt{C}_n$ which `C` depends on must be present at least in binary form. This is due to two reasons: in Java there are no separate interface files, hence type information must be extracted from code, and moreover assumptions on $\mathtt{C}_1, \ldots, \mathtt{C}_n$ cannot be extracted by type inference from the code of `C`, for reasons that we will discuss in detail later on. Second, if some of $\mathtt{C}_1, ..., \mathtt{C}_n$ are only available in source form, then Java compilers enforce their compilation too.

Hence, the behavior of a standard Java compiler when invoked on `C` does not simply consist in intra-checking `C`. We could still argue that Java supports the intra-checking and inter-checking schema in the sense that, even though these two phases are interleaved in a standard Java compiler, it is possible to formalize its overall behavior following this schema. In other words, a standard Java compiler could be seen as playing the double role of a separate compilation mechanism and an inter-checker (that is, a tool which performs the checks a static linker would do, but with no code assembly). However, this is only partially true, in the sense that a standard Java compiler *does not* perform all the inter-checks a safe mechanism of separate compilation plus inter-checking would perform; this is only true in the particular case in which all $\mathtt{C}_1, \ldots, \mathtt{C}_n$ are in source form. We will come back to this point in Sect.3.

We consider now existing formal definitions of the Java type system. These definitions (we refer for instance to [4]) do not model separate compilation of classes, but "global" compilation of a "closed program" (self-contained collection of classes in source form). More in detail, this means that a "global" type environment Γ^G containing the type information part of the program (roughly, the program deprived of method bodies) is considered; the internal coherence of this type environment is checked (in this phase, for instance, cycles in the inheritance hierarchy are detected) and then the well-formedness of each class body is checked against the type environment Γ^G.

These definitions formalize the behavior of Java compilers only in the particular case in which they are invoked on a closed collection of classe declarations in source form. Some generalization is achieved in [5], where it is assumed that the information in the global type environment Γ^G can be extracted from a binary as well as from a source fragment, as Java compilers actually do. In [2] we have proposed a formalization of the Java compilation process where the judgment corresponding to intra-checking is clearly isolated from other components (extraction of the type environment from the program and determination of the fragments on which compilation is propagated). However, still each class is intra-checked against a unique global type environment extracted from the compilation environment. Note that with this approach inter-checking trivially succeeds, since the type environment is directly extracted from the code and is the same for all fragments, but in intra-checking a single fragment we use much more type information than needed; in other words the type system is not as abstract as it could be (see next subsection).

1.3 True separate compilation of Java classes

In this paper, we start from a different point of view, that is, from considering a Java class in isolation and wondering which is the type information on other classes (modeled by a *local type environment* Γ^L) needed for intra-checking the class and generating the corresponding bytecode.

Let us consider a simple example in the toy Java subset on which we will formally define intra-checking and inter-checking in Sect.2.

```
class C extends Parent {
    Pure h (Pure x){ return x;}
    int h1 (Cons x) { return 0;}
    int m1 (Type1 x) { return new C.h1(new Cons);}
    int m2 (Type2 x) {  return new C.m1(x);}
    Type1 m (Type2 x) { return new Used.g(x);}
}
```

First consider class `Pure`; it is used as a "pure abstract type" with no subtyping constraints, therefore there are no requirements on `Pure` in order for `C` to be compilable and, in fact, class `C` can be correctly executed even in environments where no bytecode for `Pure` is available (see the specification of the Java Virtual Machine in [8]). On the contrary, classes `Cons`, `Type1` and `Type2` must satisfy some constraints. Class `Cons` must at least exist since otherwise creation of `Cons` instances would not be possible (for simplicity we assume that each class only has the default constructor, and we omit parentheses in the invocation), while class `Type2` must be a subtype of `Type1`. We will formalize these constraints by the judgments $\Gamma^L \vdash_{\mathtt{L}} \exists\mathtt{Cons}$ and $\Gamma^L \vdash_{\mathtt{L}} \mathtt{Type2} \leq \mathtt{Type1}$, respectively.

Let us look now at the method invocation[4] in the body of `m`. The class `C` can be intra-checked in any type environment where a class `Used` is available which provides a method (either directly declared or inherited) with name `g` and one

[2]For simplicity we assume that execution can be started from any fragment; in Java this holds only for fragments corresponding to classes with a `main` method.

[3]Unfortunately there is a name clash between interfaces in the sense of [3] and Java interfaces, hence in the sequel we will always say "class" meaning either class or interface. In the small Java subset in Sect.2 we will only consider classes.

[4]In the toy Java subset class members are only methods; however, method invocations are the most challenging issue, and the generalization to field accesses and constructor invocations is trivial.

parameter of a supertype of Type2; moreover we have the constraint that its return type must be a subtype of Type1. For instance, class C can be typechecked in the following environment (1):

```
class Parent{}
class Cons{}
class Type1{}
class Type2 extends Type1{}
class Type3 extends Type2{}
class UsedParent { Type3 g(Type1 x) { ...}}
class Used extends UsedParent {}
```

and also in this environment (2):

```
class Parent{}
class Cons{}
class Type1{}
class Type2 extends Type1{}
class Used {
    Type2 g(Type2 x) {...}
    int f() {...}
}
```

Hence, we would be tempted to store in the local type environment Γ^L just the information that class Used must have a method g with one parameter of a supertype of Type2 and return type subtype of Type1. However, in order to produce the corresponding bytecode, a Java compiler must know *exactly* which are the parameter and return type of the method, and even in which superclass of User it has been declared. Indeed, in bytecode method invocations are annotated with a *method descriptor* indicating the method which has been selected for the invocation at compile-time. A method descriptor is a triple consisting of the class which contains the method declaration, the parameter types and the return type[5]. We get new Used[UsedParent, Type1, Type3].g(x) in the first example and new Used[Used, Type2, Type2].g(x) in the second.

Formally, this means that local type environments can require that, for a method invocation where the receiver type is C, method name m and argument types $\bar{\mathtt{T}}$, the method descriptor for the invocation must be selected among those in a given set $\{\mu_1, \ldots, \mu_n\}$. We will formalize this constraint by the judgment $\Gamma^L \vdash \mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu_1, \ldots, \mu_n$.

In the example, C can be intra-checked, e.g., in a type environment Γ^L_1 where the method g selected for method invocations with receiver type Used and argument type Type2 is declared in class UsedParent, has return type Type3, and parameter type Type1, as in environment (1); this constraint is formalized by the judgment $\Gamma^L_1 \vdash \mathit{Sel}(\mathtt{Used}, \mathtt{g}, \mathtt{Type2}) = [\mathtt{UsedParent}, \mathtt{Type1}, \mathtt{Type3}]$.

As well, C can be intra-checked in a type environment Γ^L_2 where the selected method is declared in class Used, and has return and parameter type Type2, as in environment (2); this constraint is formalized by the judgment $\Gamma^L_1 \vdash \mathit{Sel}(\mathtt{Used}, \mathtt{g}, \mathtt{Type2}) = [\mathtt{Used}, \mathtt{Type2}, \mathtt{Type2}]$.

Note that in the example there is exactly one selectable method; however, in the general case, more than one method can be selectable, and, when typing the invocation, the most specific among them must be chosen according to the Java rules for overloading resolution.

The example clearly shows that there is no "least" local type environment in this case; this is due to the fact that Java bytecode is much "less abstract" than Java source code, as we will discuss in more detail in Sect.3.

Finally, let us consider the parent class. Class Parent is never used inside the body of C; however, we cannot conclude that there are no requirements on Parent, since some choice could violate Java rules on overriding, for instance the following: class Parent { Parent m (Type2 x) { ...}}. In other words, we must require in Γ^L that, if the class Parent has a (directly declared or inherited) method with name m and one argument of type Type2, then this method has return type Type1. We will write this constraint as Parent#Type1 m (Type2).

In summary, the local type environment needed for intra-checking a class (generating the corresponding bytecode) will express the following kinds of constraints on other classes: a class C must exist, written $\exists\,\mathtt{C}$, a class $\mathtt{C}_1$ must be a subtype of another class $\mathtt{C}_2$, written $\mathtt{C}_1 \leq \mathtt{C}_2$, the method selected for invocations with receiver type C, method name m and argument types $\bar{\mathtt{T}}$ must be chosen in a certain set of methods, written $\mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu_1, \ldots, \mu_n$, and a class C cannot have a method with name m and parameter types $\bar{\mathtt{T}}$ unless this method has return type T, written C#T m ($\bar{\mathtt{T}}$).

Let us briefly illustrate the advantages of this approach in comparison with that of traditional type systems for object-oriented languages.In these type systems, e.g., those used in existing Java formal definitions, typically a type environment associates with each class a *class type* consisting of its parent class and (considering only methods as we do here), the sequence of all the method headers.

The problem with this approach is that it hardly supports separate compilation, in the sense that, if a class C1 can be compiled in an environment where another class C2 has some class type CT, then it is not guaranteed that compilation succeeds and gives the same result in an environment with a different C2 which has as class type (a supertype of) CT. To see this, consider the environment (1) of the previous example where class C can be successfully typechecked. Suppose we model the corresponding type information as follows.

$$
\begin{array}{l}
\mathtt{Parent} \mapsto <\mathtt{Object}, \Lambda> \\
\mathtt{Cons} \mapsto <\mathtt{Object}, \Lambda> \\
\mathtt{Type1} \mapsto <\mathtt{Object}, \Lambda> \\
\mathtt{Type2} \mapsto <\mathtt{Type1}, \Lambda> \\
\mathtt{Type3} \mapsto <\mathtt{Type2}, \Lambda> \\
\mathtt{UsedParent} \mapsto <\mathtt{Object}, \mathtt{Type3\ g\ (Type1)}> \\
\mathtt{Used} \mapsto <\mathtt{UsedParent}, \Lambda>
\end{array}
$$

We would expect that class C can be successfully typechecked, generating the same bytecode as in environment (1), in any other environment where classes Parent, Cons, Type1, Type2, Type3, UsedParent and Used have all the methods indicated above, and possibly others. This is true in most cases, but there are two situations in which the existence of other methods in these classes can affect the typechecking of C. First, overloading resolution for some invocation can change, as in the following case

```
class UsedParent {
    Type3 g(Type1 x) { ...}
```

[5] We refer in this paper to SDK compilers until version 1.3; in version 1.4, the first component of the annotation has become the receiver's type instead of the class which contains the method declaration; however, the class where the method is declared is still needed for overloading resolution which has remained the same.

```
    Type2 g(Type2 x) {...}
}
```

where method `g(Type2)` instead of `g(Type1)` is selected for the invocation `new Used.g(x)`, thus generating a different bytecode, hence different executions (it is easy to construct an analogous example where typechecking not even succeeds since the invocation becomes ambiguous).

Second, constraints on overriding can be violated, as in the case we already mentioned.

In summary, local type environments not only express "positive" requirements (like "this class must provide this method"), but two kinds of judgments embody a "negative" requirement:

- $\mathit{Sel}(\mathtt{C},\mathtt{m},\bar{\mathtt{T}}) = \mu_1,\ldots,\mu_n$, which states that class `C` *cannot* have a method `m` with parameter types more general than $\bar{\mathtt{T}}$ and more specific than all those in $\{\mu_1,\ldots,\mu_n\}$ (for instance, class `UsedParent` as above would violate the constraint
$\mathit{Sel}(\mathtt{Used},\mathtt{g},\mathtt{Type2}) = [\mathtt{UsedParent},\mathtt{Type1},\mathtt{Type3}]$),
- $\mathtt{C}\#\mathtt{T}\ \mathtt{m}\ (\bar{\mathtt{T}})$ which states that class `C` *cannot* have a method `m` with parameter types $\bar{\mathtt{T}}$ and return type different from `T` (for instance, class `Parent` above would violate the constraint $\mathtt{Parent}\#\mathtt{Type1\ m\ (Type\ 2)}$).

1.4 Inter-checking

True separate compilation of a Java class `C` in a type environment Γ as described in the preceding section is formalized by a judgment $\Gamma \vdash_{\mathtt{L}} \mathtt{S} : \mathtt{CT} \leadsto \mathtt{B}$ where `S` is the source of `C`, `CT` is the inferred class type and `B` is the generated binary fragment (`.class` file). Assume that classes $\mathtt{C}_1,\ldots,\mathtt{C}_n$ are separately compiled into binary fragments $\mathtt{B}_1,\ldots,\mathtt{B}_n$, respectively. In other words, there exist valid judgments $\Gamma_i \vdash_{\mathtt{L}} \mathtt{S}_i : \mathtt{CT}_i \leadsto \mathtt{B}_i$, where $\mathtt{S}_i$ is the source of $\mathtt{C}_i$, for $i \in 1..n$. Then, inter-checking the set of binary fragments $\mathtt{B}_1,\ldots,\mathtt{B}_n$ amounts to check that, for each class $\mathtt{C}_i$, the other classes satisfy the type assumptions Γ_i required by $\mathtt{C}_i$; the formal definition will be given in Sect.2.3.

Inter-checking guarantees safe execution in the sense that starting execution from any $\mathtt{B}_i$ in the binary context $\mathtt{B}_1,\ldots,\mathtt{B}_n$ never raises linkage errors. Note that, since Java supports dynamic class loading, inter-checking at run time performed by the JVM cannot be avoided (to deal with fragments which are not known to be the result of some compilation); nevertheless, whenever it is possible, static inter-checking should be performed (and Java standard compilers perform static inter-checking indeed in some cases, see Sect.3).

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). Moreover, due to the lazy nature of the Java Verifier, some linkage errors might not be detected during the testing phase, but only later on when the application has already been delivered. Finally, the JVM is not able to detect some kinds of unwanted behavior; for instance, since method overloading is resolved statically rather then dynamically, it may happen that a method different from that intended is executed. Consider again the example of the preceding section. By compiling the invocation `new Used.g(x)` in environment (1) we get

```
new Used[UsedParent,Type1,Type3].g(x)
```

Now, if we change `UsedParent` by adding method `Type2 g(Type2)`, then all classes can be safely linked by the JVM; the problem is that the behavior of the code is not that expected, since the method invocation in `C` still invokes the method `Type3 g(Type1)` rather than `Type2 g(Type2)` which is more specific.

An unpleasant consequence of this problem is illustrated by the following scenario. Class `Used` and `UsedParent` are part of a library, class `C` is a client which has only access to the bytecode of the library, and initially class `UsedParent` declares only the `Type3 g(Type1)` method. After some time a new version of the library is released in which method `Type2 g(Type2)` is added in `UsedParent`. In the Java standard environment, the client can remain unaware of the change, since its code still safely links with the new version. However, we have the unpleasant effect that a method invocation `new Used.g(t2)` with `t2` of type `Type2` has two different behaviors when it appears in the library's code, which has been recompiled (the method `Type2 g(Type2)` is executed) and in the client's code (the method `Type3 g(Type1)` is executed).

In our approach, on the contrary, assuming that the client class `C` is equipped with its interface, corresponding to a local environment Γ_1^L s.t. the judgments

$$\Gamma_1^L \vdash_{\mathtt{L}} \mathtt{Type3} \leq \mathtt{Type1}$$
$$\Gamma_1^L \vdash \mathit{Sel}(\mathtt{Used},\mathtt{g},\mathtt{Type2}) = [\mathtt{UsedParent},\mathtt{Type1},\mathtt{Type3}]$$

are valid, the problem could be detected by the client before execution, since class `C` does not inter-check with classes `Used` and `UsedParent` (see the formal rule given in Sect.2.3).

1.5 Summary

We have so far recalled the notions of true separate compilation (intra-checking) and inter-checking as introduced in [3], pointed out that standard Java compilers and existing formal definitions of the Java type system do not obey this schema, and illustrated the kinds of constraints on other classes needed for intra-checking a single Java class.

The rest of the paper is structured as follows. In Sect.2 we formally define a small Java subset which embodies the relevant cases of type constraints previously illustrated. We also define a corresponding bytecode language which is a very abstract version of Java bytecode. We formally define a type system corresponding to a true separate compilation schema, that is a judgment $\Gamma \vdash_{\mathtt{L}} \mathtt{S} : \mathtt{CT} \leadsto \mathtt{B}$ where `S` is a source fragment (class declaration), `B` is a binary fragment, `CT` is the inferred class type and Γ is a type environment expressing constraints on other classes. Moreover, we formalize the inter-checking phase. At the end of the section, we state theorems relating true separate compilation, as defined here, to standard Java compilation (formalized by a type system which is an adaptation to our Java subset of existing formal definitions of Java).

In Sect.3, we compare more in detail the behavior or standard Java compilers with true separate compilation and outline how the given formal definition of true separate compilation could lead to the development of a tool to be used as an alternative to a standard Java compiler.

Finally in the Conclusion we summarize the results we have achieved and describe further work.

$$
\begin{array}{rcll}
\mathtt{S} & ::= & \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \} & \\
\mathtt{MDS}^s & ::= & \mathtt{MD}^s_1 \ldots\ \mathtt{MD}^s_n & (n \geq 0) \\
\mathtt{MD}^s & ::= & \mathtt{MH}\ \{\ \mathtt{return\ E}^s;\ \} & \\
\mathtt{MH} & ::= & \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n) & (n \geq 0) \\
\mathtt{E}^s & ::= & \mathtt{new\ C} \mid \mathtt{x} \mid \mathtt{N} \mid \mathtt{E}^s_0.\mathtt{m}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n) & \\
 & & & \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{int} & \\
 & & & \\
\mathtt{B} & ::= & \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^b\ \} & \\
\mathtt{MDS}^b & ::= & \mathtt{MD}^b_1 \ldots\ \mathtt{MD}^b_n & (n \geq 0) \\
\mathtt{MD}^b & ::= & \mathtt{MH}\ \{\ \mathtt{return\ E}^b;\ \} & \\
\mathtt{E}^b & ::= & \mathtt{new\ C} \mid \mathtt{x} \mid \mathtt{N} \mid \mathtt{E}^b_0.\mathtt{m}\,\mu\,(\mathtt{E}^b_1, \ldots, \mathtt{E}^b_n) & (n \geq 0) \\
\mu & ::= & [\mathtt{C}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T}] &
\end{array}
$$

Figure 1: Syntax and types

2. FORMALIZATION

2.1 A Type System for Separate Compilation

The toy language we consider is a small subset of Java, defined in Fig.1; metavariables C, m, x and N range over sets of class, method and parameter names, and integer literals, respectively.

A source fragment S is a class declaration consisting of the name of the class, the name of the superclass and a sequence of method declarations $\mathtt{MDS}^s$. 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. In the following we will use the metavariable $\bar{\mathtt{T}}$ for type tuples.

Our toy bytecode is rather abstract: we only annotate method invocations with *method descriptors* μ; as already explained in the Introduction, a method descriptor is a triple consisting of the class which contains the method declaration, and the parameter and return types, and specifies the method which has been selected for the invocation at compile time. A binary fragment B is structurally equivalent to a source class declaration except that method bodies are binary expressions.

$$
\begin{array}{rcl}
\Gamma^G & ::= & {\gamma^G}_1 \ldots {\gamma^G}_n \\
\gamma^G & ::= & \mathtt{C} \mapsto \mathtt{<C', MSS>} \\
\mathtt{MSS} & ::= & \mathtt{MS}_1 \ldots \mathtt{MS}_n \\
\mathtt{MS} & ::= & \mathtt{T\ m}(\bar{\mathtt{T}}) \\
 & & \\
\Gamma^L & ::= & {\gamma^L}_1 \ldots {\gamma^L}_n \\
\gamma^L & ::= & \exists \mathtt{C} \mid \mathtt{C} \leq \mathtt{C'} \mid \mathit{Sel}(\mathtt{C}, m, \bar{\mathtt{T}}) = \mu\mathtt{s} \mid \mathtt{C\#MS} \\
\mu\mathtt{s} & ::= & \mu_1 \ldots \mu_n \\
 & & \\
\Gamma & ::= & \gamma_1 \ldots \gamma_n \\
\gamma & ::= & \gamma^G \mid \gamma^L
\end{array}
$$

Figure 2: Environments

In Fig.2 we define local type environments for compilation of open programs as opposite to *global* type environments for compilation of closed programs (self-contained collection of classes); the former are sequences of type assumptions γ^L that we call *local*, while the latter are sequences of type assumptions γ^G that we call *global*. However, for achieving more uniformity and expressive power, we have also introduced the more generic notion of *type environment* Γ where local and global type assumptions can be mixed up.

Global type environments are used in existing formalizations of the Java type system [4, 5], and associate with each class C its superclass C′ and the sequence MSS of the signatures of methods declared in the class. A method signature MS consists of a method header deprived of the parameter names. This type environments can be trivially extracted from the programs.

Local type environments contain four kinds of local type assumptions:

- $\exists\mathtt{C}$ requires class C to be defined and is needed for compiling object creation;
- $\mathtt{C} \leq \mathtt{C'}$ requires class C to be a subclass of class C′;
- $\mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}$ requires the following properties:
  - the method descriptors in $\mu\mathtt{s}$ determine a set of *selectable* methods that are effectively applicable (in the sense of Java [6] 15.12.2.1) to an invocation of method m on a receiver of static type C and with arguments of static type $\bar{\mathtt{T}}$; more formally, for all $[\mathtt{C'}, \bar{\mathtt{T}}', \mathtt{T}]$ in $\mu\mathtt{s}$, method $\mathtt{T\ m}(\bar{\mathtt{T}}')$ must be declared in class C′ and $\mathtt{C} \leq \mathtt{C'}$ and $\bar{\mathtt{T}} \leq \bar{\mathtt{T}}'$ must hold; clearly, both class C and C′ must exist;
  - this set of selectable methods must contain all most specific methods (see [6] 15.12.2.2); more formally, for all class C″ s.t. $\mathtt{C} \leq \mathtt{C''}$ and for all methods $\mathtt{T'\ m}(\bar{\mathtt{T}}'')$ declared in C″, if $\bar{\mathtt{T}} \leq \bar{\mathtt{T}}''$ (that is, m is applicable), then there exists a descriptor $[\mathtt{C'}, \bar{\mathtt{T}}', \mathtt{T}]$ in $\mu\mathtt{s}$ s.t. $\mathtt{C'} \leq \mathtt{C''}$ and $\bar{\mathtt{T}}' \leq \bar{\mathtt{T}}''$ (that is, among the set of selectable methods, there exists a method which is more specific than method $\mathtt{T'\ m}(\bar{\mathtt{T}}'')$ declared in C″).

  In other words, given an invocation of method m on a receiver of static type C and with arguments of static type $\bar{\mathtt{T}}$, the set of selectable methods determined by $\mu\mathtt{s}$ must be contained in the set of all applicable methods (for a given program) and must contain, in turn, all most specific methods, so that method resolution can be correctly performed by picking the most specific among selectable methods. Note that we do not impose selected methods to coincide with the set of all applicable methods, since this would make the requirement stronger without any change in the expressive power of the type system. The same consideration holds if we would impose selected methods to coincide with the set of all most specific applicable methods.

  Finally, note that restricting local type environments to type assumptions of the form $\mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu$ where the right hand side is a single method descriptor rather than a sequence (that is, there must be exactly one most specific method) would decrease the expressive power of the type system; indeed, in the case no assumptions of the form $\mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu$ can be derived for a given method invocation, we are not able to discriminate the following three different cases:

1. there are no enough type information for resolving and, therefore, compiling the method invocation;
2. there is no applicable method;
3. there is more than one most specific method (that is, the invocation is ambiguous).

On the contrary, these cases can be discriminated allowing as right hand side a sequence of method descriptors: in case 1 no judgment of the form $Sel(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}$ holds, in case 2 the judgment $Sel(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \Lambda$ holds, while in case 3 the judgment $Sel(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}$ holds with $\mu\mathtt{s}$ a sequence of more than one (not comparable) method descriptors.

- $\mathtt{C}\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})$ requires that parent class C exists and can be correctly extended with method $\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})$, that is, if C has (that is, either directly declares or inherits) a method named m with argument types $\bar{\mathtt{T}}$, then its return type is T. As already explained in the Introduction, this requirement ensures the correctness of method overriding (see [6]).

The rules for typechecking and compiling a source fragment in a type environment Γ are defined in Fig.3. Note that, even though the rules can be instantiated with generic type environments, the most interesting case occurs when Γ is a local type environment, since this ensures that separate compilation can be performed under a set of local type assumptions.

The main rule defines the typechecking of a class declaration in a type environment Γ. Since we expect Γ to contain type information only on the classes used by C, but not on C itself, the compilation of all method declarations of C must be performed on the type environment $\Gamma, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}>$, where Γ has been enriched by the global type assumption $\mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}>$. The judgment $\Gamma, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}> \vdash_{\mathtt{L}} \mathtt{MDS}^s : \mathtt{MSS}' \leadsto \mathtt{MDS}^b$ holds whenever the method declarations $\mathtt{MDS}^s$ have type $\mathtt{MSS}'$ and are compiled into the binary method declarations $\mathtt{MDS}^b$ in the type environment $\Gamma, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}>$. Clearly, the type MSS associated with class C in the environment must coincide with the type returned by the judgment for compilation of method declarations (note that MSS can be easily extracted from the declaration of C).

The judgment $\vdash_{\mathtt{L}} \Gamma, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}>\diamond$ ensures that the environment where the compilation is performed is well-formed (see the rule below); note that compilation can be successful even when Γ contains assumptions on C itself, providing that such assumptions are consistent with the global assumption $\mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}>$.

The class declaration is correct if the methods of the superclass $\mathtt{C}'$ of C are correctly overridden by the methods declared in C; therefore, for all method signatures $\mathtt{MS}_i = \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})$ in MSS declared in C, if $\mathtt{C}'$ has the method $\mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}})$, then $\mathtt{T} = \mathtt{T}'$ must hold (judgment $\Gamma \vdash_{\mathtt{L}} \mathtt{C}'\#\mathtt{MS}_i$).

Finally, we must check the existence of the superclass $\mathtt{C}'$; this hypothesis is really necessary only when class C declares no methods (otherwise, indeed, from the validity of $\vdash_{\mathtt{L}} \mathtt{C}'\#\mathtt{MS}$ we can derive $\vdash_{\mathtt{L}} \exists\mathtt{C}'$; see comments on Fig.4).

The second rule defines the typechecking and compilation of a sequence of method declarations by simply typechecking and compiling each single method declaration. The side condition ensures that the Java rule for overloading is verified (see [6]) by checking that there are no different declarations for a method with the same name and argument types. The auxiliary function *name&Par* (defined in Fig.5) returns the name and the parameter types of a method declaration.

Each method declaration is correct (third rule) in the type environment Γ if the body is compilable in Γ and in the environment Π assigning their types to the parameter names of the method; furthermore, the type T inferred for the body $\mathtt{E}^s$ must be a subtype of the return type $\mathtt{T}_0$. Note that, according to the rules of the Java verifier (see [8]), existence of the types of the arguments is not required.

An instance creation expression, **new** C, is well-typed in Γ, and has type C, only when we can infer from Γ the existence of C. This is sufficient in our language where every class has always exactly one constructor (with no parameters); however in Java, where classes may have more than one constructor (or even no constructors at all can be invoked for a given class) we should introduce a type assumption similar to that defined for selectable methods.

An integer literal is always trivially well-typed and has type **int**.

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

Method invocation is the only construct of our toy source language whose translation in the toy bytecode is not trivial, since overloading must be resolved and an appropriate method descriptor must be computed to annotate the method invocation. To this aim, first, the receiver and all the argument expressions are typechecked and compiled obtaining their type and bytecode. Then, from the method name and the receiver and argument types, the sequence of all selectable methods is inferred in the type environment Γ. If no sequence can be inferred, then compilation fails because the type assumption in Γ are not sufficient for resolving the invocation; otherwise, if we can only infer sequences which are either empty or contain more than one method descriptor, then compilation fails because either no applicable method can be found or the invocation is ambiguous, respectively. Finally, if we can infer a single method descriptor, then it corresponds to the most specific applicable methods (see rules for selectable methods in Fig.4).

The first rule of Fig.4 defines well-formed type environments; this is done in a rather indirect way on top of the definition for well-formed global type environments as given in [4] (the corresponding rules can be found in Fig.5): a type environment Γ is well-formed (in the sense of separate compilation) if there exists a global type environment which entails Γ and is well-formed (in the sense of closed programs compilation). Since a global environment Γ^G must be closed in order for the judgment $\vdash_{\mathtt{G}} \Gamma^G\diamond$ to be valid, this means that a type environment Γ is well-formed if there exists a closed and statically correct program P satisfying Γ (that is, the global environment extracted from P entails Γ). In other words, well-formedness of type environments coincides with their consistency.

This rule has the drawback that it cannot be directly converted into an algorithm for checking consistency of type environments. However, the problem is decidable and an intuitive idea of a possible algorithm is given in Sect.3 and developed in [1].

is valid whenever $\gamma \in \Gamma^L$.

The remaining rules in Fig.4 define type environment entailment: if Γ_1 entails Γ_2 (that is, $\Gamma_1 \vdash \Gamma_2$ is valid), then we expect that every closed and statically correct program sat-

$$\frac{\begin{array}{l}\Gamma, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}> \vdash_{\mathtt{L}} \mathtt{MDS}^s : \mathtt{MSS} \leadsto \mathtt{MDS}^b \quad \vdash_{\mathtt{L}} \Gamma, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}> \diamond \\ \forall\ i \in 1..n \quad \Gamma \vdash_{\mathtt{L}} \mathtt{C}' \# \mathtt{MS}_i \quad \Gamma \vdash_{\mathtt{L}} \exists\, \mathtt{C}'\end{array}}{\Gamma \vdash_{\mathtt{L}} \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \} : <\mathtt{C}', \mathtt{MSS}> \leadsto \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^b\ \}}\ \mathtt{MSS} = \mathtt{MS}_1 \ldots \mathtt{MS}_n$$

$$\frac{\forall i \in 1..n\ \Gamma \vdash_{\mathtt{L}} \mathtt{MD}_i^s : \mathtt{MS}_i \leadsto \mathtt{MD}_i^b}{\Gamma \vdash_{\mathtt{L}} \mathtt{MD}_1^s\ \ldots\ \mathtt{MD}_n^s : \mathtt{MS}_1 \ldots \mathtt{MS}_n \leadsto \mathtt{MD}_1^b \ldots \mathtt{MD}_n^b}\ \mathit{name}\&\mathit{Par}(\mathtt{MD}_i^s) = \mathit{name}\&\mathit{Par}(\mathtt{MD}_j^s) \implies i = j$$

$$\frac{\Gamma; \{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash_{\mathtt{L}} \mathtt{E}^s : \mathtt{T} \leadsto \mathtt{E}^b \quad \Gamma \vdash_{\mathtt{L}} \mathtt{T} \leq \mathtt{T}_0}{\Gamma \vdash_{\mathtt{L}} \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return\ E}^s;\ \} : \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \ldots\ \mathtt{T}_n) \leadsto \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return\ E}^b;\ \}}$$

$$\frac{\Gamma \vdash_{\mathtt{L}} \exists\, \mathtt{C}}{\Gamma; \Pi \vdash_{\mathtt{L}} \mathtt{new\ C} : \mathtt{C} \leadsto \mathtt{new\ C}} \qquad \frac{}{\Gamma; \Pi \vdash_{\mathtt{L}} \mathtt{N} : \mathtt{int} \leadsto \mathtt{N}} \qquad \frac{}{\Gamma; \Pi \vdash_{\mathtt{L}} \mathtt{x} : \mathtt{T} \leadsto \mathtt{x}}\ \Pi(\mathtt{x}) = \mathtt{T}$$

$$\frac{\begin{array}{l}\Gamma; \Pi \vdash_{\mathtt{L}} \mathtt{E}_0^s : \mathtt{C} \leadsto \mathtt{E}_0^b \\ \forall i \in 1..n\ \Gamma; \Pi \vdash_{\mathtt{L}} \mathtt{E}_i^s : \mathtt{T}_i \leadsto \mathtt{E}_i^b \\ \Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}, \mathtt{m}, \mathtt{T}_1\ \ldots\ \mathtt{T}_n) = [\mathtt{C}', \bar{\mathtt{T}}', \mathtt{T}']\end{array}}{\Gamma; \Pi \vdash_{\mathtt{L}} \mathtt{E}_0^s.m(\mathtt{E}_1^s, \ldots, \mathtt{E}_n^s) : \mathtt{T}' \leadsto \mathtt{E}_0^b.m\,[\mathtt{C}', \bar{\mathtt{T}}', \mathtt{T}']\,(\mathtt{E}_1^b, \ldots, \mathtt{E}_n^b)}$$

$j \in \{1, \ldots, n\}$

Figure 3: Separate compilation

isfying Γ_1 must also satisfy Γ_2. Note that entailment does not implies well-formedness: if Γ_1 is not well-formed then any judgment of the form $\Gamma_1 \vdash \Gamma_2$ that can be inferred is sound since there are no programs satisfying Γ_1; on the contrary, if Γ_2 is not well-formed, then Γ_1 must necessarily be not well-formed.

The first two rules for environment entailment simply say that $\Gamma_1 \vdash \Gamma_2$ is valid if each type assumption γ contained in Γ_2 is entailed by Γ_1; the remaining rules cover the cases when Γ_2 is a single atomic type assumption γ.

There are three rules for class existence; the side condition $\mathtt{C} \neq \mathtt{C}', \mathtt{C}' \neq \mathtt{Object}$ corresponds to the fact that a subtyping check of the form either $\mathtt{C} \leq \mathtt{C}$ or $\mathtt{C} \leq \mathtt{Object}$ is always passed by the Java Verifier (see [8]), even when `C` is not available. A type assumption of the form $\mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}_1\,[\mathtt{C}', \bar{\mathtt{T}}', \mathtt{T}']\,\mu\mathtt{s}_2$ clearly implies the existence of the static type of the receiver `C` and of the classes $\mathtt{C}'$ where selectable methods are found, but not of the arguments and return types [8]. A type assumption of the form $\Gamma \vdash_{\mathtt{L}} \mathtt{C}\#\_$ implies the existence of the (parent) class `C`.

Rules for subtyping assumptions are straightforward.

There are seven rules for assumptions on selectable methods.

The first four rules allow to construct a sequence of selectable methods valid for a given invocation, that is, a sequence of methods which are all applicable and which contains all those which are most specific.

The set of selectable methods for any invocation on a receiver having static type `Object` is always empty, since in our language we assume that the root class declares no methods.

If the sequence `MSS` of the signatures of all methods declared in class `C` contains the method $\mathtt{T\ m}(\bar{\mathtt{T}})$, then a sequence of selectable methods for the invocation of `m` on a receiver of type `C` with arguments of type $\bar{\mathtt{T}}$ is just the descriptor $[\mathtt{C}, \bar{\mathtt{T}}, \mathtt{T}]$. This means that in this case the compilation of the method invocation can be successful even when no information on the methods of the superclass of `C` is available.

If a sequence of selectable methods for a given invocation on a receiver of the superclass $\mathtt{C}'$ of `C` is available, and we know which are all the methods declared in `C`, then we can compute all applicable methods of `C` for the given invocation and add them to the selectable methods of the superclass in order to obtain the selectable methods of class `C`.

Finally, a sequence of selectable methods for a given invocation is still a sequence of selectable methods for an invocation with less specific receiver and argument types provided that all the methods in the sequence are still applicable.

The following two rules are used for restricting a set of selectable methods: if method descriptors μ_1 and μ_2 belong to the same sequence $\mu\mathtt{s}$ of selectable methods, and μ_1 is less specific than μ_2, then μ_1 can be safely removed from $\mu\mathtt{s}$; if in the sequence of selectable methods there is a method descriptor which exactly corresponds to the invocation, then all other method descriptors can be safely removed.

The last rule has been introduced only for safe of completeness (actually, it could be omitted without changing the validity of the separate compilation judgment) and states that a sequence of selectable methods can always be enriched by other methods which are applicable to the invocation.

The auxiliary function $\mathit{Appl}(\Gamma, \mathtt{C}, \mathtt{MSS}, \mathtt{m}, \bar{\mathtt{T}})$ computes in Γ (needed for the subtyping relation) the method descriptors obtained by prefixing by `C` all the method signatures in `MSS` which are applicable to an invocation of method `m` with arguments of type $\bar{\mathtt{T}}$:

$$\mathit{Appl}(\Gamma, \mathtt{C}, \mathtt{MSS}, \mathtt{m}, \bar{\mathtt{T}}) = \{[\mathtt{C}, \bar{\mathtt{T}}', \mathtt{T}] \mid \mathtt{T\ m}(\bar{\mathtt{T}}') \in \mathtt{MSS}, \Gamma \vdash \bar{\mathtt{T}} \leq \bar{\mathtt{T}}'\}.$$

Finally, there are four rules for assumptions on parent classes. Class `Object` can be safely extended with any method, since it is empty. The second rule states that if a class `C` can be correctly extended with a certain method, then its superclasses can be extended as well with the same method. The next rule expresses the trivial fact that any class can be correctly extended with its methods. The last rule says that if a class can be correctly extended with a certain method, then each subclass that does not declare such method (that is, there are no methods with the same name and arguments type) can be safely extended with it as well.

$$\frac{\Gamma^G \vdash_{\mathtt{L}} \Gamma \quad \vdash_{\mathtt{G}} \Gamma^G\diamond}{\vdash_{\mathtt{L}} \Gamma\diamond} \qquad \frac{}{\Gamma \vdash_{\mathtt{L}} \Lambda} \qquad \frac{\Gamma \vdash_{\mathtt{L}} \Gamma_1 \quad \Gamma \vdash_{\mathtt{L}} \gamma}{\Gamma \vdash_{\mathtt{L}} \Gamma_1, \gamma} \qquad \frac{}{\Gamma_1, \gamma, \Gamma_2 \vdash_{\mathtt{L}} \gamma}$$

$$\frac{\Gamma \vdash_{\mathtt{L}} \mathtt{C} \leq \mathtt{C}'}{\begin{array}{l}\Gamma \vdash_{\mathtt{L}} \exists \mathtt{C}\\ \Gamma \vdash_{\mathtt{L}} \exists \mathtt{C}'\end{array}} \begin{array}{l}\mathtt{C} \neq \mathtt{C}',\\ \mathtt{C}' \neq \mathtt{Object}\end{array} \qquad \frac{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}_0, \_, \_) = [\mathtt{C}_1, \_, \_] \ldots [\mathtt{C}_n, \_, \_]}{\forall\ i \in 0..n\ \Gamma \vdash_{\mathtt{L}} \exists \mathtt{C}_i} \qquad \frac{\Gamma \vdash_{\mathtt{L}} \mathtt{C}\#\_}{\Gamma \vdash_{\mathtt{L}} \exists \mathtt{C}}$$

$$\frac{}{\Gamma \vdash_{\mathtt{L}} \mathtt{T} \leq \mathtt{Object}} \qquad \frac{}{\Gamma \vdash_{\mathtt{L}} \mathtt{T} \leq \mathtt{T}} \qquad \frac{\Gamma \vdash_{\mathtt{L}} \mathtt{C}_1 \leq \mathtt{C}_2 \quad \Gamma \vdash_{\mathtt{L}} \mathtt{C}_2 \leq \mathtt{C}_3}{\Gamma \vdash_{\mathtt{L}} \mathtt{C}_1 \leq \mathtt{C}_3} \qquad \frac{}{\Gamma_1, \mathtt{C} \mapsto <\mathtt{C}', \_>, \Gamma_2 \vdash_{\mathtt{L}} \mathtt{C} \leq \mathtt{C}'}$$

$$\frac{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n) = \mu\mathtt{s}_1\, [\mathtt{C}', \mathtt{T'}_1 \ldots \mathtt{T'}_n, \mathtt{T}']\, \mu\mathtt{s}_2}{\begin{array}{r}\forall i \in 1..n \quad \Gamma \vdash_{\mathtt{L}} \mathtt{T}_i \leq \mathtt{T}'_i\\ \Gamma \vdash_{\mathtt{L}} \mathtt{C} \leq \mathtt{C}'\end{array}} \qquad \frac{\forall i \in 1..n \quad \Gamma \vdash_{\mathtt{L}} \mathtt{T}_i \leq \mathtt{T}'_i}{\Gamma \vdash_{\mathtt{L}} \mathtt{T}_1 \ldots \mathtt{T}_n \leq \mathtt{T'}_1 \ldots \mathtt{T'}_n}$$

$$\frac{}{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{Object}, \mathtt{m}, \bar{\mathtt{T}}) = \Lambda} \qquad \frac{}{\Gamma_1, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}_1\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})\ \mathtt{MSS}_2>, \Gamma_2 \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = [\mathtt{C}, \bar{\mathtt{T}}, \mathtt{T}]}$$

$$\frac{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}', \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}}{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}\, \mu_1\, \ldots\, \mu_n} \begin{array}{l}\Gamma = \Gamma_1, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}>, \Gamma_2\\ \mathit{Appl}(\Gamma, \mathtt{C}, \mathtt{MSS}, \mathtt{m}, \bar{\mathtt{T}}) = \{\mu_1, \ldots, \mu_n\}\end{array}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = [\mathtt{C}_1, \bar{\mathtt{T}}_1, \mathtt{T}_1] \ldots [\mathtt{C}_n, \bar{\mathtt{T}}_n, \mathtt{T}_n]\\ \forall\ i \in 1..n\ \Gamma \vdash_{\mathtt{L}} \mathtt{C}' \leq \mathtt{C}_i\ \Gamma \vdash_{\mathtt{L}} \bar{\mathtt{T}}' \leq \bar{\mathtt{T}}_i \quad \Gamma \vdash_{\mathtt{L}} \mathtt{C} \leq \mathtt{C}' \quad \Gamma \vdash_{\mathtt{L}} \bar{\mathtt{T}} \leq \bar{\mathtt{T}}'\end{array}}{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}', \mathtt{m}, \bar{\mathtt{T}}') = [\mathtt{C}_1, \bar{\mathtt{T}}_1, \mathtt{T}_1] \ldots [\mathtt{C}_n, \bar{\mathtt{T}}_n, \mathtt{T}_n]}$$

$$\frac{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}_1\, [\mathtt{C}_1, \bar{\mathtt{T}}_1, \mathtt{T}_1]\, \mu\mathtt{s}_2 \quad \Gamma \vdash_{\mathtt{L}} \mathtt{C}_2 \leq \mathtt{C}_1 \quad \Gamma \vdash_{\mathtt{L}} \bar{\mathtt{T}}_2 \leq \bar{\mathtt{T}}_1}{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}_1\, \mu\mathtt{s}_2}\ \mu\mathtt{s}_1\, \mu\mathtt{s}_2 = \mu\mathtt{s}_3\, [\mathtt{C}_2, \bar{\mathtt{T}}_2, \mathtt{T}_2]\, \mu\mathtt{s}_4$$

$$\frac{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}_1\, [\mathtt{C}', \bar{\mathtt{T}}', \mathtt{T}']\, \mu\mathtt{s}_2}{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}', \mathtt{m}, \bar{\mathtt{T}}') = [\mathtt{C}', \bar{\mathtt{T}}', \mathtt{T}']}$$

$$\frac{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}_1 \quad \Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}', \mathtt{m}, \bar{\mathtt{T}}') = \mu\mathtt{s}_2 \quad \Gamma \vdash_{\mathtt{L}} \mathtt{C} \leq \mathtt{C}' \quad \Gamma \vdash_{\mathtt{L}} \bar{\mathtt{T}} \leq \bar{\mathtt{T}}'}{\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}_1\, \mu\mathtt{s}_2}$$

$$\frac{}{\Gamma \vdash_{\mathtt{L}} \mathtt{Object}\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})} \qquad \frac{\Gamma \vdash_{\mathtt{L}} \mathtt{C}\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \quad \Gamma \vdash_{\mathtt{L}} \mathtt{C} \leq \mathtt{C}'}{\Gamma \vdash_{\mathtt{L}} \mathtt{C}'\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})} \qquad \frac{}{\Gamma_1, \mathtt{C} \mapsto <\_, \mathtt{MSS}_1\ \mathtt{MS}\ \mathtt{MSS}_2>, \Gamma_2 \vdash_{\mathtt{L}} \mathtt{C}\#\mathtt{MS}}$$

$$\frac{\Gamma_1, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MS}_1 \ldots \mathtt{MS}_n>, \Gamma_2 \vdash_{\mathtt{L}} \mathtt{C}'\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})}{\Gamma_1, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MS}_1 \ldots \mathtt{MS}_n>, \Gamma_2 \vdash_{\mathtt{L}} \mathtt{C}\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})} \forall\ i \in 1..n\ \mathtt{MS}_i \neq\ \_\ \mathtt{m}(\bar{\mathtt{T}})$$

Figure 4: Type environments well-formedness and entailment

2.2 Results

The following three theorems relate separate compilation as formally defined here with the standard compilation of closed programs as defined for instance in [4, 5] (see Fig.5 for the formal rules).

The first theorem claims that separate compilation "gives the same result" of standard compilation. More precisely, assume that a closed source program P is compiled (that is, by using $\vdash_{\mathtt{G}}$) yielding a collection of binary fragments ce_b (formally, a finite partial function from class names to binaries). Assume also that a source fragment $\mathtt{S}$ declaring class $\mathtt{C} = \mathit{className}(\mathtt{S})$ in P is separately compiled (that is, by using $\vdash_{\mathtt{L}}$) yielding a binary fragment $\mathtt{B}$ in a type environment Γ entailed by the global type environment $\Gamma^G = \tau(P)$ extracted from the program P. Then, the the binary $ce_b(\mathtt{C})$ of $\mathtt{C}$ obtained by compiling the whole program must coincide with the binary $\mathtt{B}$ obtained by separately compiling $\mathtt{C}$.

The auxiliary functions *className* and τ (defined in Fig.5) returns the class name in a class declaration and the global type environment extracted from a program, respectively.

We only sketch the proof of this theorem and show the main lemmas required to prove it.

Lemma 2.1. *If* $\Gamma^G \vdash_{\mathtt{L}} \Gamma$, $\Gamma \vdash C_1 \leq C_2$, $\{C_1, C_2\} \subseteq \mathit{Def}(\Gamma^G)$ *then* $\Gamma^G \vdash_{\mathtt{G}} C_1 \leq C_2$.

Lemma 2.2. *If* $\vdash_{\mathtt{G}} \Gamma^G\diamond$, $\Gamma^G \vdash_{\mathtt{L}} \Gamma$, $\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(C, m, \bar{T}) = \mu\mathtt{s}$ *and* $\mathit{MostSpecs}_{\Gamma^G}(C, m, \bar{T}) = \{\mu\}$, *then* $\Gamma \vdash_{\mathtt{L}} \mathit{Sel}(C, m, \bar{T}) = \mu$.
the two following cases:
the most specific one;
$\mu\mathtt{s} \subseteq \mathit{Appl}_{\Gamma^G}(C, m, \bar{T})$. *So, if a most Lemma 2.1.*

Lemmas 1 and 2 are necessary to prove Lemma 3 below.

Lemma 2.3. *If* $\vdash_{\mathtt{G}} \Gamma^G\diamond$, $\Gamma^G \vdash_{\mathtt{L}} \Gamma$ *then:*

1. *if* $\Gamma, \Pi \vdash_{\mathtt{L}} \mathtt{E}^s : T_1 \rightsquigarrow \mathtt{E}^b_1$ *and* $\Gamma^G, \Pi \vdash_{\mathtt{G}} \mathtt{E}^s : T_2 \rightsquigarrow \mathtt{E}^b_2$, *then* $T_1 = T_2$ *and* $\mathtt{E}^b_1 = \mathtt{E}^b_2$

2. $\Gamma \vdash_{\mathtt{L}} \mathtt{MD}^s : \mathit{MS} \rightsquigarrow \mathtt{MD}^b_1$ *and* $\Gamma^G \vdash_{\mathtt{G}} \mathtt{MD}^s \rightsquigarrow \mathtt{MD}^b_2$, *then* $\mathtt{MD}^b_1 = \mathtt{MD}^b_2$;

$$\frac{\forall i\in 1..n\ \Gamma^G \vdash_{\mathtt{G}} \mathtt{S}_i \rightsquigarrow \mathtt{B}_i \quad \vdash_{\mathtt{G}} \Gamma^G\diamond}{\vdash_{\mathtt{G}} \mathtt{S}_1\ldots\mathtt{S}_n \rightsquigarrow \bigcup_{i\in 1..n} \mathit{className}(\mathtt{S}_i)\mapsto \mathtt{B}_i}\ \begin{array}{l} \mathit{className}(\mathtt{S}_i)=\mathit{className}(\mathtt{S}_j) \implies i=j \\ \Gamma^G=\tau(\mathtt{S}_1\ldots\mathtt{S}_n)\end{array}$$

$$\frac{\forall i\in 1..n\ \Gamma^G \vdash_{\mathtt{G}} \mathtt{MD}^s_i \rightsquigarrow \mathtt{MD}^b_i}{\Gamma^G \vdash_{\mathtt{G}} \mathtt{class\ C\ extends\ C'}\ \mathtt{MD}^s_1\ldots\mathtt{MD}^s_n \rightsquigarrow \mathtt{class\ C\ extends\ C'}\ \mathtt{MD}^b_1\ldots\mathtt{MD}^b_n}\ \mathit{name\&Par}(\mathtt{MD}^s_i)=\mathit{name\&Par}(\mathtt{MD}^s_j) \implies i=j$$

$$\frac{\forall i\in 0..n\ \Gamma^G \vdash_{\mathtt{G}} \mathtt{T}_i \diamond_{\mathtt{type}} \quad \Gamma^G,\{\mathtt{x}_1\mapsto\mathtt{T}_1,\ldots,\mathtt{x}_n\mapsto\mathtt{T}_n\}\vdash_{\mathtt{G}} \mathtt{E}^s:\mathtt{T}\rightsquigarrow\mathtt{E}^b \quad \Gamma^G\vdash_{\mathtt{G}}\mathtt{T}\leq\mathtt{T}_0}{\Gamma^G\vdash_{\mathtt{G}}\mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n)\ \{\mathtt{return}\ \mathtt{E}^s;\}\rightsquigarrow\mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n)\ \{\mathtt{return}\ \mathtt{E}^b;\}}\ \mathtt{x}_i=\mathtt{x}_j \implies i=j$$

$$\frac{}{\Gamma^G,\Pi\vdash_{\mathtt{G}}\mathtt{new\ C}:\mathtt{C}\rightsquigarrow\mathtt{new\ C}}\ \mathtt{C}\in\mathit{Def}(\Gamma^G) \qquad \frac{}{\Gamma^G,\Pi\vdash_{\mathtt{G}}\mathtt{x}:\Pi(\mathtt{x})\rightsquigarrow\mathtt{x}}\ \mathtt{x}\in\mathit{Def}(\Pi) \qquad \frac{}{\Gamma^G,\Pi\vdash_{\mathtt{G}}\mathtt{N}:\mathtt{int}\rightsquigarrow\mathtt{N}}$$

$$\frac{\begin{array}{c}\Gamma^G,\Pi\vdash_{\mathtt{G}}\mathtt{E}^s_0:\mathtt{C}\rightsquigarrow\mathtt{E}^b_0\\ \forall i\in 1..n\ \Gamma^G,\Pi\vdash_{\mathtt{G}}\mathtt{E}^s_i:\mathtt{T}_i\rightsquigarrow\mathtt{E}^b_i\end{array}}{\Gamma^G,\Pi\vdash_{\mathtt{G}}\mathtt{E}^s_0.\mathtt{m}(\mathtt{E}^s_1,\ldots,\mathtt{E}^s_n):\mathtt{T}'\rightsquigarrow\mathtt{E}^b_0.\mathtt{m}[\mathtt{C}',\bar{\mathtt{T}}',\mathtt{T}'](\mathtt{E}^b_1,\ldots,\mathtt{E}^b_n)}\ \mathit{MostSpecs}_{\Gamma^G}(\mathit{Appl}_{\Gamma^G}(\mathtt{C},\mathtt{m},\mathtt{T}_1\ldots\mathtt{T}_n))=\{[\mathtt{C}',\bar{\mathtt{T}}',\mathtt{T}']\}$$

$\mathit{Appl}_{\Gamma^G}(\mathtt{Object},\mathtt{m},\bar{\mathtt{T}})=\emptyset$
$\mathit{Appl}_{\Gamma^G}(\mathtt{C},\mathtt{m},\bar{\mathtt{T}})=\{[\mathtt{C},\bar{\mathtt{T}}_i,\mathtt{T}_i] | i\in 1..n, \Gamma^G\vdash_{\mathtt{G}}\bar{\mathtt{T}}\leq\bar{\mathtt{T}}_i, \mathtt{m}=\mathtt{m}_i\}\cup\mathit{Appl}_{\Gamma^G}(\mathtt{C}',\mathtt{m},\bar{\mathtt{T}})$ if $\Gamma^G(\mathtt{C})=<\mathtt{C}',\mathtt{T}_1\ \mathtt{m}_1(\bar{\mathtt{T}}_1)\ldots\mathtt{T}_n\ \mathtt{m}_n(\bar{\mathtt{T}}_n)>$
else $\perp$
$\mathit{MostSpecs}_{\Gamma^G}(\{[\mathtt{C}_1,\bar{\mathtt{T}}_1,\mathtt{T}_1]\ldots[\mathtt{C}_n,\bar{\mathtt{T}}_n,\mathtt{T}_n]\})=\{[\mathtt{C}_i,\bar{\mathtt{T}}_i,\mathtt{T}_i] | \forall i\in 1..n, j\in 1..n\ \Gamma^G\vdash_{\mathtt{G}}\mathtt{C}_i\leq\mathtt{C}_j,\ \Gamma^G\vdash_{\mathtt{G}}\bar{\mathtt{T}}_i\leq\bar{\mathtt{T}}_j\}$

$$\frac{}{\Gamma^G\vdash_{\mathtt{G}}\emptyset\diamond} \qquad \frac{}{\Gamma^G\vdash_{\mathtt{G}}\mathtt{C}\diamond_{\mathtt{type}}}\ \mathtt{C}\in\mathit{Def}(\Gamma^G) \qquad \frac{}{\Gamma^G\vdash_{\mathtt{G}}\mathtt{int}\diamond_{\mathtt{type}}} \qquad \frac{}{\Gamma^G\vdash_{\mathtt{G}}\mathtt{Object}:\Lambda}$$

$$\frac{\Gamma^G\vdash_{\mathtt{G}}\mathtt{C}':\mathtt{MSS}'}{\Gamma^G\vdash_{\mathtt{G}}\mathtt{C}:\mathtt{MSS}'[\mathtt{MSS}]}\ \begin{array}{l}\mathtt{MSS}'[\mathtt{MSS}]\neq\perp\\ \Gamma^G(\mathtt{C})=<\mathtt{C}',\mathtt{MSS}>\end{array} \qquad \frac{\Gamma^G\vdash_{\mathtt{G}}\Gamma^G_1\diamond \quad \Gamma^G\vdash_{\mathtt{G}}\mathtt{C}:<\mathtt{C}',\mathtt{MSS}>}{\Gamma^G\vdash_{\mathtt{G}}\Gamma^G_1\cup\{\mathtt{C}\mapsto<\mathtt{C}',\mathtt{MSS}>\}\diamond}\ \mathtt{C}\notin\mathit{Def}(\Gamma^G_1) \qquad \frac{\Gamma^G\vdash_{\mathtt{G}}\Gamma^G\diamond}{\vdash_{\mathtt{G}}\Gamma^G\diamond}$$

$$\frac{}{\Gamma^G\vdash_{\mathtt{G}}\mathtt{int}\leq\mathtt{int}} \qquad \frac{}{\Gamma^G\vdash_{\mathtt{G}}\mathtt{C}\leq\mathtt{C}}\ \mathtt{C}\in\mathit{Def}(\Gamma^G) \qquad \frac{}{\Gamma^G\vdash_{\mathtt{G}}\mathtt{C}\leq\mathtt{C}'}\ \Gamma^G(\mathtt{C})=<\mathtt{C}',\_> \qquad \frac{\Gamma^G\vdash_{\mathtt{G}}\mathtt{C}\leq\mathtt{C}' \quad \Gamma^G\vdash_{\mathtt{G}}\mathtt{C}'\leq\mathtt{C}''}{\Gamma^G\vdash_{\mathtt{G}}\mathtt{C}\leq\mathtt{C}''}$$

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

$\mathit{className}(\mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \})=\mathtt{C}$

If $\mathtt{MD}^s=\mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n)\{\ \mathtt{return}\ \mathtt{E}^s;\ \}$, then
$\mathit{signature}(\mathtt{MD}^s)=\mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ldots\mathtt{T}_n)$,
$\mathit{name\&Par}(\mathtt{MD}^s)=<\mathtt{m},\mathtt{T}_1\ldots\mathtt{T}_n>$,
$\mathit{returnType}(\mathtt{MD}^s)=\mathtt{T}_0$

$\tau(\mathtt{S}_1\ldots\mathtt{S}_n)=\bigcup_{i\in\{1,\ldots,n\}}\tau(\mathtt{S}_i)$
$\tau(\mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MD}^s_1\ldots\mathtt{MD}^s_n\ \})=\mathtt{C}\mapsto<\mathtt{C}',\mathit{signature}(\mathtt{MD}^s_1)\ldots\mathit{signature}(\mathtt{MD}^s_n)>$

$$\mathtt{MS'}_1\ldots\mathtt{MS'}_n[\mathtt{MS}_1\ldots\mathtt{MS}_k]=\begin{cases}\perp & \text{if } \exists\ i\in 1..k, j\in 1..n:\ \mathit{name\&Par}(\mathtt{MS}_i)=\mathit{name\&Par}(\mathtt{MS}'_j)\wedge \\ & \qquad \mathit{returnType}(\mathtt{MS}_i)\neq\mathit{returnType}(\mathtt{MS}'_j)\\ \mathtt{MS}'_{i_1}\ldots\mathtt{MS}'_{i_p}\ \mathtt{MS}_1\ldots\mathtt{MS}_k & \text{otherwise, where: } \{i_1,\ldots,i_p\}= \\ & \qquad \{i\in 1..n |\ \nexists j\in 1..k:\mathit{name\&Par}(\mathtt{MS}'_i)=\mathit{name\&Par}(\mathtt{MS}_j)\}\end{cases}$$

Figure 5: Compilation of a closed program, well-formedness of global environments, and auxiliary functions

3. *$\Gamma\vdash_{\mathtt{L}}\mathtt{MDS}^s:\mathit{MSS}\rightsquigarrow\mathtt{MDS}^b_1$ and $\Gamma^G\vdash_{\mathtt{G}}\mathtt{MDS}^s\rightsquigarrow\mathtt{MDS}^b_2$, then $\mathtt{MDS}^b_1=\mathtt{MDS}^b_2$.*

Lemma 2.2;

Theorem 2.4. *If $\Gamma\vdash_{\mathtt{L}} S:CT\rightsquigarrow B$, $\mathit{className}(S)=C$, $P=P_1\ S\ P_2$, $\vdash_{\mathtt{G}} P\rightsquigarrow ce_b$ and $\tau(P)\vdash_{\mathtt{L}}\Gamma$ then $ce_b(C)=B$.*
Proof
Using Lemma 2.3.

The remaining two theorems state that there exists a local type environment Γ^L where class declaration S is separately compilable if and only if there exists a closed program P satisfying Γ^L and containing S which is compilable.

Theorem 2.5. *If $P=P_1\ S\ P_2$, $\mathit{className}(S)=C$ and $\vdash_{\mathtt{G}} P\rightsquigarrow ce_b$ then $\exists\ \Gamma^L:\ \tau(P)\vdash_{\mathtt{L}}\Gamma^L$ and $\Gamma^L\vdash_{\mathtt{L}} S:CT\rightsquigarrow B$.*

Theorem 2.6. *If* $\Gamma \vdash_{\mathtt{L}} S : CT \rightsquigarrow B$ *then* $\exists\, P$ *s.t.* $\tau(P) \vdash_{\mathtt{L}} \Gamma$ *and* $\vdash_{\mathtt{G}} P \rightsquigarrow ce_b$.

2.3 Inter-checking

In this section we formally define *inter-checking* for Java classes; assume that classes $\mathtt{C}_1, \ldots, \mathtt{C}_n$ are separately compiled into binary fragments $\mathtt{B}_1, \ldots, \mathtt{B}_n$, respectively. In other words, there exist valid judgments $\Gamma_i \vdash_{\mathtt{L}} \mathtt{S}_i : \mathtt{CT}_i \rightsquigarrow \mathtt{B}_i$, where $\mathtt{S}_i$ is the source of $\mathtt{C}_i$, for $i \in 1..n$. If we assume that the program consisting of classes $\mathtt{C}_1, \ldots, \mathtt{C}_n$ is closed, then binary fragments successfully inter-check if for each class $\mathtt{C}_i$ the other classes $\mathtt{C}_1, \ldots, \mathtt{C}_{i-1}, \mathtt{C}_{i+1}, \ldots, \mathtt{C}_n$ satisfy the assumptions required in type environment Γ_i; this amounts requiring that each type environment Γ_i must be entailed by the global type environment $\Gamma^G = \mathtt{C}_1 \mapsto \mathtt{CT}_1, \ldots, \mathtt{C}_n \mapsto \mathtt{CT}_n$.

However, if we want to be able to inter-check open programs as well, then we need to introduce a type environment Γ_0 containing all assumptions on classes declared outside the program. As a consequence we obtain the following typing rule for inter-checking:

$$\frac{\begin{array}{l}\vdash_{\mathtt{L}} \Gamma_0 \diamond \\ \forall i \in 1..n\ \Gamma_i \vdash_{\mathtt{L}} \mathtt{S}_i : \mathtt{CT}_i \rightsquigarrow \mathtt{B}_i \\ \forall i \in 1..n\ \Gamma_0, \Gamma^G \vdash_{\mathtt{L}} \Gamma_i\end{array}}{\Gamma_0 \vdash_{\mathtt{L}} \mathtt{B}_1 \ldots \mathtt{B}_n \diamond \textit{inter-checked}} \quad \begin{array}{l}\forall i \in 1..n\ \mathtt{C}_i = \textit{className}(\mathtt{S}_i) \\ \mathtt{C}_1, \ldots, \mathtt{C}_n \text{ distinct} \\ \Gamma^G = \mathtt{C}_1 \mapsto \mathtt{CT}_1, \ldots, \mathtt{C}_n \mapsto \mathtt{CT}_n\end{array}$$

3. STANDARD COMPILATION VERSUS TRUE SEPARATE COMPILATION

In this section we compare more in detail the behavior of existing Java compilers with true separate compilation and show how a Java compiler fully supporting true separate compilation could be developed on the basis of the type theory presented in the previous section.

We only touch some design and implementation issues, leaving for future work a more complete treatment.

3.1 When Java compilers perform true separate compilation

As already mentioned, all formal specifications of Java typechecking/compilation defined so far (e.g., [4, 5], but see [7] for a more complete list of references) do not consider the issue of true separate compilation: programs (that is, collections of class definitions either in source or in bytecode) are assumed to be closed, that is, they cannot refer to classes whose definition is not available. However in practice this constraint is too strict; for instance, in some cases Java compilers allow the user to successfully compile an open program. Consider for example the program P consisting of the following class declaration[6]:

```
class C1 extends Object{int f(){return  new C2.g();}}
```

together with the following available in binary form:

```
class C2 extends Object{
   int g(){return 1;}
   C3 h(C3 c){return c;}
}
```

Even though P is not closed (no definition for `C3` is available), SDK compilers successfully compiles class `C1`. This happens because in this case the compiler uses the bytecode of `C2` only for extracting the type information needed to compile class `C1` but no typechecking is performed on the code of `C2`. In other words, the SDK compiler simply intra-checks class `C1` (generating a corresponding bytecode) by using the type information extracted from `C2`. In general, SDK compiler exhibits a behavior corresponding to true separate compilation when all the classes used in the source code which are accessed by the compiler are in binary form.

This is modeled in our type system by the fact that class `C1` is compilable in the local type environment

$\Gamma^L \equiv \textit{Sel}(\mathtt{C2}, \mathtt{g}, \Lambda) = [\mathtt{C2}, \Lambda, \mathtt{int}]$,

while in the other type systems proposed in literature (like that defined in Fig.5) `C1` is not compilable since the global type environment

$\Gamma^G \equiv \mathtt{C1} \mapsto <\mathtt{Object}, \mathtt{int\ f}()>$,
$\mathtt{C2} \mapsto <\mathtt{Object}, \mathtt{int}\ g()\ \mathtt{C3}\ h(\mathtt{C3})>$,

extracted from P is not well formed.

However the ability of modeling real compilers is not the main motivation of the type system defined in Sect.2; rather, the principal aim is to define a framework for true separate Java compilation able to express the set of constraints required for successfully compiling a class in isolation.

3.2 When Java compilers *do not* perform true separate compilation

Currently available Java compilers support true separate compilation only partly for the following main reasons:

1. Separate compilation and inter-checking are blurred together;

2. Fragment interfaces are not separated from class definitions.

Point 1 can be illustrated by the following example; consider the definition of `C1` given above together with the corresponding source code for `C2` (as defined above). Furthermore assume that no bytecode is available for `C2`. Now, even though the two class declarations are stored in different files, it is not possible to compile class `C1` separately from `C2`, as happened in the previous example; indeed, in this case the SDK compiler, even though invoked only on `C1`, tries to compile `C2` as well, with a subsequent compilation error in case no definition for class `C3` is available.

In other words, the SDK compiler assumes that class `C1` must be linked with the class `C2` available at compile time, therefore it performs a complete inter-checking of the classes by compiling `C2` as well. In general, SDK exhibits a behavior corresponding to complete inter-checking when all the classes used in the source code which are accessed by the compiler are in source form. However, this assumption is arbitrary for a language, like Java, supporting dynamic linking: we can envisage situations where the user does not want `C2` to be compiled, because `C1` will be dynamically linked with a class `C2` whose definition is not currently available to the compiler.

3.3 Are local environments necessary?

A compiler supporting true separate compilation could be designed without introducing an explicit notion of fragment interface, but rather by simply extracting from `C2` the type information needed for correctly compiling `C1`.

However, this solution can still be considered unsatisfactory for the following reasons:

[6]For uniformity we use the syntax defined in Sect.2.

- The user would probably prefer to directly define the type assumptions on C2, rather than providing a complete definition containing useless and error prone implementation details (point 2);
- The user could be interested in the type assumptions on C2 used by the compiler for generating the bytecode of C1 in order to know which kind of classes C2 can be safely linked with C1; such information can also be exploited by a static inter-checker (see Sect.1.4).

Type assumptions could be represented by class declarations deprived of method bodies, but, as already pointed out in the Introduction, this naive solution prevents to express minimal requirements on used classes. We show another example illustrating the problem. Consider the following two class declarations:

```
class C1 extends C2{int f(C1 c){return c.g(c);}}
class C2 extends Object{int g(C2 c){return 1;}}
```

We can successfully compile class C1 under the assumption extracted from C2 stating that C2 extends Object and declares method int g(C2); more formally, we can separately compile class C1 w.r.t. the global type environment

$\Gamma^G \equiv \mathtt{C2} \mapsto <\mathtt{Object}, \mathtt{int\ g(C2)}>$

However, this assumption is too strict since requires C2 not to have extra methods; for instance, the following definition for C2, which is compatible with the declaration of C1 above, does not match Γ^G:

```
class C2 extends Object{
  int g(C2 c){return 1;}
  int h(C2 c){return 1;}
  C2 m(){return new C2;}}
```

On the other side, the intuitive subtyping rules stating that C2 can be any class (extending Object) and having at least method int g(C2) cannot be applied since it is unsound. To see this, we can replace C1 with the following new declaration:

```
class C1 extends C2{
  int h(Object o){return 2;}
  int m(){return new C1.h(new C2);}}
```

Class C1 can still be separately compiled w.r.t. Γ^G, but now it is no longer compatible with the second declaration for C2 since method m is not correctly overridden; furthermore note also that method resolution for invocation new C1.h(new C2) becomes ambiguous.

Our type system offers a more flexible mechanism for separate compilation since type assumptions on used classes are expressed by means of local rather than global type environments, while global type environments are mainly used for specifying the classes declared in the fragment. Following this approach, the required type assumptions for compiling the first declaration C1 could correspond to the local type environment

$\Gamma^L_1 \equiv \mathtt{C2}\#\mathtt{int\ f(C1)},\ \mathit{Sel}(\mathtt{C2}, \mathtt{g}, \mathtt{C1}) = [\mathtt{C2}, \mathtt{C2}, \mathtt{int}]$

while for the second declaration of C1 we could have

$\Gamma^L_2 \equiv \mathtt{C2}\#\mathtt{int\ h(Object)},\ \mathtt{C2}\#\mathtt{int\ m()},\ \mathit{Sel}(\mathtt{C2}, \mathtt{h}, \mathtt{C2}) = \Lambda$

The reader may verify that the first declaration of C2 matches both Γ^L_1 and Γ^L_2, whereas the second only matches Γ^L_1.

3.4 Implementation Issues

In this section we discuss some implementation issues related to the design of a prototype Java compiler supporting true separate compilation and driven by our framework.

The first problem we face is how an interface can be extracted from a given fragment f.

An appealing solution consists in inferring the interface from the code of f; as already shown in the Introduction, unfortunately, this is not possible for Java without avoiding radical changes to the overall architecture. For instance, consider again the example in the Introduction:

```
class C extends Parent {
...
Type1 m(Type2 x){ return new Used.g(x);}}
```

Class C can be correctly linked with any class Used having a method α g(β), for any types α, β s.t. $\alpha \leq \mathtt{Type1}$ and $\mathtt{Type2} \leq \beta$; such method can either be directly declared in Used or inherited from some ancestor γ of Used. Clearly, all these classes cannot be captured by a unique local type environment Γ^L in our type system. In order to do that, we should introduce type variables in the type environments, analogously to the approach followed in [10]; so we could infer the following class interface for C:

$\Gamma^{\mathrm{L}} = \ldots, \alpha \leq \mathtt{Type1}, \mathtt{Type2} \leq \beta, \mathit{Sel}(\mathtt{Used}, \mathtt{g}, \mathtt{Type2}) = [\gamma, \alpha, \beta]$
$\mathtt{CT} = <\mathtt{Parent}, \ldots \mathtt{Type1\ m(Type2)}>$

However, in this way the compiler cannot generate Java bytecode for C, since method descriptors cannot contain type variables; as a consequence, either JVM should be radically modified, or we should introduce a sort of pre-bytecode that may contain type variables that must be instantiated during static inter-checking in order to produce valid Java bytecode (indeed, the solution proposed in [10] relies on static linking; see also [9]). Furthermore, it seems hard to define a system ensuring the existence of principal types.

For these reasons, our prototype compiler will require users to explicitly annotate fragments with their interfaces; for instance, following a style common to many module languages, we may assume that the interface for the fragment contained in C.java can be found in C.def.

At this point we can follow two different approaches for implementing our compiler: either radically modify a currently available compiler (like SDK, for instance), or implement a forward engineering procedure able to convert .def (interfaces) into .java files (class declarations) (the analogous tool in [9] is called a *stub generator*).

The first solution clearly produces a more efficient implementation and allows to include additional features, as automatic generation of interfaces on demand, when only source and binary files are available (as commonly happens).

However, the second solution is more appealing for a prototype version and has also the advantage of being more modular, since the forward engineering procedure can be implemented by a pre-processor allowing the use of any Java compiler. We outline how this last approach should work by means of a simple example (a detailed formal description is presented in [1]).

Consider the following class declaration:

```
class H extends P{
  A1 f(H x){return x.g(x,x.m(x));}
  H g(A2 a,int i){return new H;}
}
```

Assume that the local type environment corresponding to the interface of H is defined as follows:

$$\Gamma^L \equiv \texttt{P} \leq \texttt{A1}, \texttt{P} \leq \texttt{A2}, \texttt{P}\#\texttt{A1 f(H)}, \texttt{P}\#\texttt{H g(A2,int)}, \\ Sel(\texttt{P},\texttt{g},\texttt{H int}) = \Lambda, Sel(\texttt{P},\texttt{m},\texttt{H}) = [\texttt{P},\texttt{P},\texttt{int}]\ [\texttt{A1},\texttt{A1},\texttt{int}]$$

The first two constraints require the classes A1 and A2 to be superclasses of the parent class P; the third and fourth constraints express the requirements on the parent class necessary to guarantee that rules on overriding are respected. Finally, the last two constraints are necessary for typechecking the two method invocations and generating corresponding bytecode. The former allows to resolve overloading for invocations `h.g(h,i)` with `h` of type H and `i` of type `int`, selecting the method `g` declared in H; indeed the constraint ensures that the superclass cannot have a method `g(H int)` which would make the invocation ambiguous. The latter allows to resolve overloading for invocations `h.m(h)` with `h` of type H, selecting a method `int m (P)` which must be declared in P; moreover class A1 must declare a method `int m (A1)`.

Starting from Γ^L the pre-processor can generate the following dummy classes:

```
class P extends A1{int m(P p){return 0;}}
class A1 extends A2{int m(A1 a){return 0;}}
class A2 extends Object{}
```

The program P consisting of the declaration of H plus P, A1 and A2 is closed; furthermore the global type environment Γ^G extracted from P verifies $\Gamma^G \vdash \Gamma^L$ (the reader may easily check that this property holds). Then, we expect that class H is separately compilable and generates the binary B if and only if the program P is compilable in SDK and the binary generated for H is B.

Note that there exists an infinite number of programs satisfying the property above, but among them P can be considered minimal, in a sense that needs to be formalized but corresponds to the intuition that it contains the minimal amount of declarations needed for satisfying Γ^L. However in this case there exists another minimal fragment (therefore, there is no least fragment) defined by:

```
class P extends A2{ int h(P p){return 0;}}
class A1 extends Object{ int h(A1 a){return 0;}}
class A2 extends A1{}
```

This happens because for successfully compiling H, both classes A1 and A2 must be supertypes of H (or, equivalently, of P), as correctly specified in Γ^L; however, since Java does not supports multiple inheritance among classes, it must be the case that either A1 is a subclass of A2 or the converse. Both choices are legal and correspond to the two different fragments defined above, but none of them is more specific than the other, so either can be arbitrary chosen by the pre-processor.

4. CONCLUSION

We have defined a type system modeling true separate compilation for a small but significant Java subset, in the sense that a single class declaration can be intra-checked (following the terminology in [3]) and compiled providing a set of type requirements on missing classes. These type requirements are specified by a local type environment associated with each single class, while in the existing formal definitions of the Java type system, classes are typed in a global type environment containing all the type information on classes composing a closed program.

We have also provided formal rules for static inter-checking of a collection of classes and related our approach with existing formal definitions of Java typechecking and compilation of closed programs, by proving that we get the same results.

We plan to extend our formalization to a larger Java subset and to develop tools for compilation and static analysis of Java code on the basis of the type theory presented in this paper. In particular, in [1] we describe in detail the forward engineering procedure informally illustrated in Sect.3.4.

5. ACKNOWLEDGEMENTS

We are extremely grateful to Sophia Drossopoulou for the stimulating discussions and precious suggestions.

6. REFERENCES

[1] D. Ancona and G. Lagorio. Supporting true separate compilation in Java: A modular approach. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2002. Submitted for publication.

[2] D. Ancona, G. Lagorio, and E. Zucca. A formal framework for Java separate compilation. In B. Magnusson, editor, *ECOOP'02 - European Conference on Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 609–635. Springer, 2002.

[3] L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, 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, 1999.

[5] 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.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification, Second Edition*. Addison-Wesley, 2000.

[7] P. H. Hartel and L. Moreau. Formalizing the safety of Java, the Java Virtual Machine and Java card. *ACM Computing Surveys*, 33(4):517–558, 2001.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Second edition, 1999.

[9] S. McDirmid, M.Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned java. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 2001*. ACM Press, October 2001. SIGPLAN Notices.

[10] Z. Shao and A. Appel. Smartest recompilation. In *ACM Symp. on Principles of Programming Languages 1993*, pages 439–450. ACM Press, 1993.