Stronger Typings for Separate Compilation of Java-like Languages*

Davide Ancona and Giovanni Lagorio

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

Abstract. We formally define and implement a formal system supporting separate compilation for a small but significant Java-like language. This system is formally proved to be stronger than the standard compilation of both Java and C#, in the sense that it better supports software reuse by avoiding unnecessary recompilation steps after code modification which are usually performed by using the standard compilers.
This is achieved by introducing the notion of local type assumption allowing the user to specify weaker requirements on the source fragments which need to be compiled in isolation.
Another important property satisfied by our system is compositionality, which corresponds to the intuition that if a set of fragments can be separately compiled and such fragments are compatible, then it is possible to compile all the fragments together as a unique program and obtain the same result.
The system has been fully implemented by an available prototype, which takes a type environment and a set of class declarations as input and produces standard Java bytecode for those classes. The main algorithm is described, and its correctness is informally motivated.

1 Introduction

Separate compilation of statically typed languages is an important feature of modern software systems, since it promotes software reuse while retaining type safety without the need for unnecessary recompilation steps.

As pointed out by Cardelli in [6], the essence of separate compilation corresponds to a typing judgment which, at least in the simplest cases, has the general form $\Gamma \vdash \texttt{S}{:}\tau$, where Γ is a type environment containing all necessary information needed for compiling in isolation the source fragment `S`, and τ is the inferred result type of `S`. Using the terminology introduced in [6], we say that fragment `S` *intrachecks*.

For instance, in Java separate compilation as supported by the standard SDK implementation, `S` corresponds to the declaration of a class `H`, Γ contains

* Partially supported by Dynamic Assembly, Reconfiguration Type-checking - EC project IST-2001-33477, and Murst NAPOLI - Network Aware Programming: Oggetti, Linguaggi, Implementazioni.

all the information on the classes needed by $\mathtt{H}$ and has the general form $\mathtt{C}_1 \mapsto \tau_1, \ldots, \mathtt{C}_n \mapsto \tau_n$, and τ and τ_i, for $i = 1..n$, are class types obtained by extracting all type annotations in the code, including the direct superclass, the headers of all methods and constructors and the declarations of all fields declared in the class (we refer to [3] for more details).

Another crucial notion for separate compilation is *interchecking* [6]; in general, the fact that two fragments $\mathtt{S}_1$ and $\mathtt{S}_2$, named N_1 and N_2, respectively, intracheck does not guarantee that the program obtained by merging N_1 with N_2 is statically correct. To ensure this we can either just recompile $\mathtt{S}_1$ and $\mathtt{S}_2$ as a whole (but this solution obviously contradicts separate compilation), or we can first verify that N_1 and N_2 intercheck by proving that N_1 and N_2 meet the requirements in Γ_2 and Γ_1, respectively; if this is the case, then recompilation of $\mathtt{S}_1$ and $\mathtt{S}_2$ can be avoided.

Formally, we can verify intercheck by proving that the two assumptions $N_1{:}\tau_1$ and $N_2{:}\tau_2$ together entail both the type assumptions in Γ_1 and those in Γ_2, which intuitively means that the type environments Γ_1 and Γ_2 are not stronger than the type environment $N_1{:}\tau_1, N_2{:}\tau_2$; in its simplest case, entailment corresponds to type environment inclusion, but in systems like that presented here and in [4] the entailment relation between type environments is not so trivial.

Since interchecking allows programmers to avoid useless recompilation steps without compromising static correctness, weaker type environments for compiling S imply less chances to need to recompile S in case some fragments which S depends on have been changed; therefore, weaker type environments are better for separate compilation and software reuse.

Ideally, the best situation consists in a type system with principal typings, as well pointed out in [15], where for each fragment it is possible to prove a typing which is at least as strong as all other provable typings for S.

According to the formal definitions given in [15], a *typing* t is a pair $(\Gamma \vdash \tau)$ and if $\mathsf{Term}(t)$ denotes the set of all source fragments S s.t. $\Gamma \vdash \mathtt{S}{:}\tau$ is provable in the system, then a typing t_1 is defined to be *stronger* than a typing t_2 if and only if $\mathsf{Term}(t_1) \subset \mathsf{Term}(t_2)$. Intuitively, if t_1 is stronger than t_2 and $\mathtt{S} \in \mathsf{Term}(t_1)$ (and, therefore, $\mathtt{S} \in \mathsf{Term}(t_2)$ as well), then typing t_1 approximates S better than t_2 and, therefore, t_1 is preferable to t_2 since it enlarges the set of contexts where S can be used in a type safe way without recompiling it.

Unfortunately, as stressed in [4], the typing $(\Gamma \vdash \tau)$ as implemented by SDK is not so strong, in the sense that the assumptions in Γ are far from being minimal and, as a consequence, S can be reused in much less contexts than one would expect. For instance, let us consider the following class declaration:

```
class H extends P{...}
```

In order to successfully compile class H, the SDK compiler needs to retrieve all the type information on class P, including its direct superclass, its method headers, and so on; assume, for instance, that the compilation has been successful by using the information that class P extends `Object` and its body is empty.

Now, if we need to use class H in a context where, for instance, class P extends another class rather than `Object` or declares some methods, then class P does not satisfy any longer the assumptions under which class H has been compiled, therefore, in order to statically ensure type safety we are forced to recompile class H.

In this paper we do not propose to attack the problem of principality for Java-like languages, but rather to investigate the properties and the implementation issues related to a system where it is possible to derive typings which are stronger (in the sense of [15]) than those derivable in the SDK Java implementation.

Another crucial point is that separate compilation in both Java and C# cannot be reduced to separate typechecking as happens for the simple functional language considered in [6]. To see this, let us consider again our example class H:

```
class H extends P {
int g(P p) { return new P.f(p) ; } ...}
```

If we are only concerned with typechecking, then we can safely state that the body of method `int g(P)` in H is statically correct whenever class P has a method[1] `int f`(α) with α supertype of P; whenever class P satisfies this type assumption we can avoid to typecheck again H.

However, if we turn to consider code generation, then we cannot state that class H is compiled into the binary fragment B under the assumption that class P has a method `int f`(α) with α supertype of P, since binary fragments cannot be annotated by type variables. For instance, if the argument type of f in P is `Object`, then the invocation of method f in the binary code B of H is annotated with `Object`, but if we change later the argument type of f, then class H needs to be recompiled in order to properly update the type annotation in its binary code.

Type annotations in the bytecode are needed for performing dynamic typechecking during the verification phase of class H [10, 7]. In fact, the latter example applies to any Java-like language, that is, any statically typed object-oriented language supporting safe dynamic typechecking by means of type annotations in the generated code, like Java and C#.

As a consequence, separate compilation of Java-like languages is captured more correctly by a typing judgment of the form $\Gamma \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$ saying that in the type environment Γ the source fragment S has type τ and compiles to the binary fragment B.

In [4] a first step towards stronger typings supporting separate compilation of Java-like languages has been taken, by defining a type system for a small but significant subset of Java where, for each source fragment S, it is possible to derive typings of the form $(\Gamma \vdash \tau \leadsto \mathtt{B})$ which are stronger than those derivable in the system corresponding to the SDK Java implementation.

This is made possible by introducing the notion of *local* type assumption as opposite to the notion of *global* type assumption used by the SDK compiler; as already shown, a global assumption on a class C has the form $\mathtt{C} \mapsto \tau$, where the

[1] In this preliminary part of the paper, for sake of simplicity we do not consider overloaded methods, which, however, are covered in the technical part.

class type τ contains all the possible type information on `C`. A quite standard example of local assumption is given by `C1` $\leq$ `C2`, which only requires class `C1` to be a subtype of `C2`, but says nothing, for instance, on the methods of `C1`; however, in [4] other new kinds of local type assumption are introduced.

This paper proposes a solution for two main issues left open in [4].

First, the system presented in this paper is a substantial modification of that presented in [4] and it is proved to be *compositional*, an important property for a system supporting separate compilation which, unfortunately, is not satisfied by the system in [4].

The second problem concerns the implementation; the system presented here is fully implemented by a prototype[2] written in Java, and some related issues are discussed in the paper.

Compositionality is an expected property of separate compilation informally defined in [6] as follows:

> The linked program should have the same effect as a program obtained by merging all the source together and compiling the result in a single step.

Formally, in our system this amounts to require that if $\Gamma_1 \vdash \mathtt{S}_1 : \tau_1 \leadsto \mathtt{B}_1$, $\Gamma_2 \vdash \mathtt{S}_2 : \tau_2 \leadsto \mathtt{B}_2$ and $\mathtt{S}_1$ and $\mathtt{S}_2$ intercheck, then the program $<\mathtt{S}_1\ \mathtt{S}_2>$ obtained by putting together the two fragments compiles successfully in the empty environment and produces the pair of binaries $<\mathtt{B}_1\ \mathtt{B}_2>$.

Another interesting property we prove ensures that our system is effectively "stronger" than the SDK compilation: if Γ contains only global assumptions and the typing judgment $\Gamma \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$ can be proved, then there exists a stronger typing (in the sense of [15]) ($\Gamma' \vdash\ : \tau \leadsto \mathtt{B}$) which can be proved for `S`, and, more interestingly, the environment Γ' can be effectively constructed by collecting all local assumptions needed to prove $\Gamma \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$.

This last result shows that, during the compilation of a closed program `P` (that is, a set of self-contained code fragments compilable in the empty environment) it is possible to infer for each fragment of `P` the set of local type assumptions (that is, a type environment) which are effectively used by the compiler for that particular fragment; the compiler can take advantage of these automatically generated type environments for avoiding the propagation of useless recompilation steps due to some later changes in `P`.

Finally, the system presented here is fully implemented by a prototype written in Java; while the typechecking phase is directly driven by the type rules of the system, for simplicity code generation relies on the SDK implementation; by using a technique similar to that adopted in [11], after checking the validity of $\Gamma \vdash \mathtt{S}{:}\tau$, the prototype generates from Γ a bunch of class declarations which can be merged with `S`; finally, the resulting closed program can be safely compiled by the SDK compiler.

[2] The code and some running examples are available on the web at the URL `http://www.disi.unige.it/person/LagorioG/separate-compilation/`.

One of the most challenging problems related to the implementation is the consistency check of the type environment Γ, which must ensure that Γ does not contain type assumptions which are in contradiction in order to guarantee the existence of some statically correct Java program satisfying all assumptions in Γ. Indeed, this problem is proved to be NP-complete, even though in many cases the algorithm drops to polynomial time complexity; however, an average case complexity analysis would be needed for understanding how this negative result has a practical impact on efficiency.

The rest of the paper is structured as follows. Section 2 contains an informal presentation of the system, whereas Section 3 contains all the technical details and the proofs concerning the type system. Section 4 describes the prototype implementation, and Section 5 contains some discussions on related and further work.

2 An Informal Presentation of the System

This section is a gentle introduction to the type system formally defined in Section 3. More precisely, the two basic notions of *local* type assumption and *entailment* relation between type environments are informally presented and motivated.

All the examples in this Section have been tested with our prototype implementation and can be found at the address specified in the Introduction.

2.1 Local Type Assumptions

Let us consider the declaration written in our Java-like language of class `H` mentioned in the Introduction:

Example 1.
```
class H extends P {
 int g(P p) {return new P.f(p);}
 int m() {return new H.g(new P);}
 U id(U u){return u;}
 X em(Y y){return y;}
}
```

and let us analyze under which assumptions class `H` can be successfully compiled. If we take the approach of the SDK compiler, then we may want to impose rather strong requirements on the classes used by `H`, by asking for the most detailed type information about such classes.

In our system this corresponds to compile `H` in a type environment Γ_g which at least must contain global assumptions on the classes `P`, `U`, `X` and `Y`. For instance, if Γ_g is the collection of the following assumptions:

$$\texttt{P} \mapsto <\texttt{Object}, \texttt{int f(Object)}>, \texttt{U} \mapsto <\texttt{Object}, >, \texttt{Y} \mapsto <\texttt{X}, >, \texttt{X} \mapsto <\texttt{Object}, >$$

then we are assuming that class P extends `Object` and declares only `int f(Object)`, classes U and X both extend `Object` and are empty, and class Y extends X and is empty.

In our system a global type assumption has the form $\mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}>$ with the meaning "C extends C′ and declares exactly all methods specified by MSS" (for simplicity, in our language instance methods are the only members a class can contain).

Under the assumptions contained in Γ_g class H can be successfully compiled to the following binary fragment $\mathtt{B_h}$:

```
class H extends P {
 int g(P p) {return new P<<P.f(Object)int>>(p);}
 int m() {return new H.<<H.g(P)int>>(new P);}
 U id(U u){return u;}
 X em(Y y){return y;}
}
```

Note that in our system a binary fragment is just like a source fragment except that invocations contain a *symbolic reference* $\ll \mathtt{C.m(T_1 \ldots T_n)T} \gg$ to a method, giving the name m, the parameter types $\mathtt{T}_1 \ldots \mathtt{T}_n$ and the return type T of the method, as well as the class C in which the method is to be found (see [10] 5.1). Indeed, from our perspective the most critical difference between source and binary fragments is type annotations in the method invocations, since it makes the problem of separate compilation (that is, separate typechecking plus code generation) substantially different from that of separate typechecking, as already pointed out in the Introduction.

Example 2. Let us now try to relax the strong assumptions in Γ_g by seeking an environment Γ_l containing other kinds of type assumptions which still guarantee that H compiles to the same binary fragment $\mathtt{B_h}$, but impose fairly weaker requirements on classes P, U, X and Y.

A first basic request is that the compilation environment where H is contained must provide a definition for the four classes on which H depends on. In our system this can be expressed by a local assumption of the form $\exists\,\mathtt{C}$, therefore Γ_l will contain at least the four assumptions $\exists\,\mathtt{P}, \exists\,\mathtt{U}, \exists\,\mathtt{X}, \exists\,\mathtt{Y}$.

Let us now focus on each single class used by H.

Class P: in order to correctly compile class H the following additional assumptions on class P must be added to Γ_l:

- $\mathtt{P} \not\leq \mathtt{H}$: P must not be a subtype of H since inheritance cannot be cyclic;
- $\mathtt{P}\#\mathtt{int\ g(P)}, \mathtt{P}\#\mathtt{int\ m()}, \mathtt{P}\#\mathtt{U\ id(U)}, \mathtt{P}\#\mathtt{X\ em(Y)}$: P cannot contain method declarations which would violate Java rules on overriding;
- $\mathit{Sel}(\mathtt{P}, \mathtt{f}, \mathtt{P}) = [\mathtt{P}, \mathtt{Object}, \mathtt{int}]$: P has a most specific method f for the invocation in the body of method g in H and such method f is declared in P with signature `int f(Object)`. This assumption ensures that the body of g in H is successfully compiled to the same bytecode of method g in $\mathtt{B_h}$ (in other words, the same symbolic reference to the method is generated);

– $\mathit{Sel}(\mathtt{P},\mathtt{g},\mathtt{P}) = \Lambda$: P has no methods applicable for the invocation in the body of method m in H; this fact implies that method g in H is actually the most specific method for that invocation, and, therefore, the body of m in H is successfully compiled to the same bytecode of method m in $\mathtt{B_h}$ (in other words, the same symbolic reference to the method is generated).

Class U: no additional requirements on U are needed, since the static correctness of method id in H only requires the existence of U.

Classes X and Y: in order to correctly compile class H, class Y must be a subtype of class X, otherwise method em in H would not be statically correct. Therefore we need to add the assumption $\mathtt{Y} \leq \mathtt{X}$ to Γ_l.

In conclusion, Γ_l is defined by the following sequence of local assumptions

$$\exists\,\mathtt{P}, \exists\,\mathtt{U}, \exists\,\mathtt{X}, \exists\,\mathtt{Y}, \mathtt{P} \not\leq \mathtt{H}, \mathtt{P}\#\mathtt{int\ g}(\mathtt{P}), \mathtt{P}\#\mathtt{int\ m}(), \mathtt{P}\#\mathtt{U\ id}(\mathtt{U}), \mathtt{P}\#\mathtt{X\ em}(\mathtt{Y}),$$
$$\mathit{Sel}(\mathtt{P},\mathtt{f},\mathtt{P}) = [\mathtt{P},\mathtt{Object},\mathtt{int}], \mathit{Sel}(\mathtt{P},\mathtt{g},\mathtt{P}) = \Lambda, \mathtt{Y} \leq \mathtt{X}$$

and our system supplies the following five kinds[3] of local assumptions:

– $\exists\,\mathtt{C}$ with the meaning "C must be declared";
– $\mathtt{C} \not\leq \mathtt{C}'$ with the meaning "C must not be a subtype of C′";
– $\mathtt{C}\#\mathtt{T\ m}(\mathtt{T}_1 \ldots \mathtt{T}_n)$ with the meaning "C can be correctly extended with method $\mathtt{T\ m}(\mathtt{T}_1 \ldots \mathtt{T}_n)$", that is, if C has a method (either declared or inherited) named m with parameter types $\mathtt{T}_1 \ldots \mathtt{T}_n$, then its return type must be T;
– $\mathit{Sel}(\mathtt{C},\mathtt{m},\mathtt{T}_1 \ldots \mathtt{T}_n) = \mu\mathtt{s}$ with the meaning "all methods named m determined by the specifiers in the sequence $\mu\mathtt{s}$ are *selectable* for the method invocation $\mathtt{C.m}(\mathtt{T_1} \ldots \mathtt{T_n})$".
  A *specifier* $[\mathtt{C}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T}]$ of a method gives the class C where the method is defined, the parameter types $\mathtt{T}_1 \ldots \mathtt{T}_n$ and the return type T of the method. A set S of methods is said to be selectable for the method invocation $\mathtt{C.m}(\mathtt{T_1} \ldots \mathtt{T_n})$ iff (1) all methods in S are applicable (according to the Java terminology used in [8] 15.12.2.1) to an invocation of a method named m on a receiver of static type C and with arguments of static type $\mathtt{T}_1 \ldots \mathtt{T}_n$; (2) S contains at least all maximally specific methods (see [8] 15.12.2.2), so that method resolution for that particular method invocation can be performed correctly by only looking at the methods in S.
– $\mathtt{C} \leq \mathtt{C}'$ with the meaning "C must be a subtype of C′".

Finally, class H can be successfully compiled to the same binary $\mathtt{B}_h$ both in Γ_l and Γ_g, but intuitively Γ_l is weaker than Γ_g (and, therefore, better for software reuse): for instance, class U must extend Object and be empty in Γ_g, while in Γ_l it can extend any class and declare any method. The notion of stronger (or weaker) type environment is captured in our system by an entailment relation on type environments.

[3] Actually, the global assumption $\mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}>$ is a convenient syntactic abbreviation for the conjunction of other two kinds of local assumptions: $\mathtt{C} <_1 \mathtt{C}'$ and $\mathtt{C} = \mathtt{MSS}$.

2.2 Entailment of Type Environments

Referring to the previous example, in our system the intuition that Γ_l is not stronger than Γ_g corresponds to the fact that Γ_g *entails* Γ_l, that is, the judgment $\Gamma_g \vdash \Gamma_l$ is provable. Indeed, an expected property of the entailment relation (which, in fact, holds in our system) is that if Γ is a consistent type environment (in the sense that it does not contain contradictory assumptions) and $\Gamma \vdash \Gamma'$ is provable, then for any provable judgment $\Gamma' \vdash \mathtt{S} : \tau \rightsquigarrow \mathtt{B}$ the judgment $\Gamma \vdash \mathtt{S} : \tau \rightsquigarrow \mathtt{B}$ is provable as well.

This means that, as expected, weaker type environments make typings stronger in the sense of [15]; indeed, recalling the definitions given in the Introduction, directly from the previous property of the entailment relation we can deduce that if Γ is a consistent type environment and $\Gamma \vdash \Gamma'$ is provable, then $\mathsf{Term}(\Gamma' \vdash \tau \rightsquigarrow \mathtt{B}) \subseteq \mathsf{Term}(\Gamma \vdash \tau \rightsquigarrow \mathtt{B})$, which means that the typing $(\Gamma \vdash \tau \rightsquigarrow \mathtt{B})$ is not stronger than the typing $(\Gamma' \vdash \tau \rightsquigarrow \mathtt{B})$.

On the other hand our intuition suggests that not only Γ_l is not stronger than Γ_g, but is really weaker than Γ_g.

This can be proved by showing, for instance, that in our system $\Gamma_g \vdash \mathit{Sel}(P, g, H) = \Lambda$ is provable, but not $\Gamma_l \vdash \mathit{Sel}(P, g, H) = \Lambda$; as a consequence, $\Gamma_l \vdash \Gamma_g$ is not provable.

Example 3. To see this, let us modify class `H` by changing the body of `m` with the expression `return new H.g(new H)`, so that now `g` is invoked on an argument of static type `H` rather than `P`. The modified class `H` still compiles successfully in the environment Γ_g to a binary $\mathtt{B}'_\mathtt{h}$ (which is obviously different from $\mathtt{B}_\mathtt{h}$ because of the change in `H`), while it does not compile in the environment Γ_l which does not guarantee the fact that class `P` does not have methods `g` selectable for the invocation `H.g(H)`.

Note that, since `H` is a subtype of `P`, this last fact is not a consequence of the assumption that `P` does not have methods `g` selectable for the invocation `H.g(P)` (for instance, `P` could declare method `int g(H)` which is not applicable to the invocation `H.g(P)`); in other words, is not possible to entail $\mathit{Sel}(\mathtt{P}, \mathtt{g}, \mathtt{H}) = \Lambda$ from $\mathit{Sel}(\mathtt{P}, \mathtt{g}, \mathtt{P}) = \Lambda$.

Example 4. Finally, we end this section with some more examples of entailments provable in our system, by showing that Γ_l contains redundant assumptions which, in fact, can be removed without affecting the outcome of the compilation of class `H`.

A first assumption we can easily remove is $\exists\,\mathtt{P}$, since the intuition suggests that all other assumptions on `P` in Γ_l need the declaration of class `P` in order to be verified, hence they all entail $\exists\,\mathtt{P}$.

The assumption $\mathit{Sel}(\mathtt{P}, \mathtt{g}, \mathtt{P}) = \Lambda$ ensures that class `P` cannot have a method named `g` with one parameter of type `P`; as a consequence, the method `int g(P)` in `H` cannot violate any overriding rule and, in fact, $\mathtt{P}\#\mathtt{int}\ \mathtt{m}()$ turns out to be entailed by $\mathit{Sel}(\mathtt{P}, \mathtt{g}, \mathtt{P}) = \Lambda$, and, hence, redundant.

A probably less intuitive fact is that $\mathtt{Y} \leq \mathtt{X}$ entails $\exists\,\mathtt{Y}$, but not $\exists\,\mathtt{X}$, therefore the former assumption can be removed from Γ_l, but not the latter. To see this, let us consider, for instance, the program fragment `class Y extends X {}`: it verifies $\mathtt{Y} \leq \mathtt{X}$ and $\exists\,\mathtt{Y}$, but not $\exists\,\mathtt{X}$, therefore it seems sensible to state that $\mathtt{Y} \leq \mathtt{X}$ does not entail $\exists\,\mathtt{X}$; on the other hand, it is not difficult to prove that any program fragment verifying $\mathtt{Y} \leq \mathtt{X}$ must necessarily contain the declaration of class Y.

3 Formalization

In this section we present the formalizations of the system presented in the previous section, as well as the main technical results.

3.1 Definition of the System

The Java-like language we consider is rather small but significant, since it includes overloading of instance methods which is one of the most critical features for separate compilation (see [6], end of pag. 1).

$$
\begin{array}{rcl}
\mathtt{P} & ::= & \langle \mathtt{S}_1..\mathtt{S}_n \rangle \\
\mathtt{S} & ::= & \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \} \\
\mathtt{MDS}^s & ::= & \mathtt{MD}^s_1 \ldots\ \mathtt{MD}^s_n \\
\mathtt{MD}^s & ::= & \mathtt{MH}\ \{\ \mathtt{return\ E}^s;\ \} \\
\mathtt{MH} & ::= & \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n) \\
\mathtt{E}^s & ::= & \mathtt{new\ C} \mid \mathtt{x} \mid \mathtt{N} \mid \mathtt{E}^s_0.\mathtt{m}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n) \\
 & & \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{int} \\
 & & \\
\mathtt{B} & ::= & \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^b\ \} \\
\mathtt{MDS}^b & ::= & \mathtt{MD}^b_1 \ldots\ \mathtt{MD}^b_n \\
\mathtt{MD}^b & ::= & \mathtt{MH}\ \{\ \mathtt{return\ E}^b;\ \} \\
\mathtt{E}^b & ::= & \mathtt{new\ C} \mid \mathtt{x} \mid \mathtt{N} \mid \mathtt{E}^b_0 \ll \mathtt{C.m}(\bar{\mathtt{T}})\mathtt{T} \gg (\mathtt{E}^b_1, \ldots, \mathtt{E}^b_n)
\end{array}
$$

implicit assumptions:

- class names in P are distinct;
- method signatures in $\mathtt{MDS}^s$ and $\mathtt{MDS}^b$ are distinct;
- parameter names in MH are distinct.

Fig. 1. Syntax

The syntax of the language is defined in Figure 1; metavariables C, m, x and N range over sets of class, method and parameter names, and integer literals, respectively.

A program P is a sequence of source fragments; 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 can be either a class name or `int`.

In the following we will use the abbreviation $\bar{\mathtt{T}}$ for type tuples.

As already mentioned, the bytecode of our language differs from the source code only for method invocations which contain a symbolic reference $\ll \mathtt{C.m(\bar{T})T} \gg$ to the method to be invoked (see Section 2).

Type environments are defined in Figure 2. A global type environment Γ^G is a sequence of global type assumptions. A global type assumption γ^G maps a class name to the corresponding class type τ. A class type τ is a pair consisting of the superclass name and the set of method signatures. A type assumption $\mathtt{C} \mapsto \langle \mathtt{C'}, \mathtt{MSS} \rangle$ is just a convenient syntactic abbreviation for the conjunction of the two local type assumptions $\mathtt{C} <_1 \mathtt{C'}$ and $\mathtt{C} = \mathtt{MSS}$.

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

Fig. 2. Type environments

A local type environment Γ^L is a sequence of local type assumptions of several kinds: $\exists\,\mathtt{C}$ (class existence), $\mathtt{C} \leq \mathtt{C'}$ (subtyping), $\mathit{Sel}(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}$ (method selection), $\mathtt{C\#T\ m(\bar{T})}$ (method overriding), $\mathtt{C} \not\leq \mathtt{C'}$ (negative subtyping), $\mathtt{C} <_1 \mathtt{C'}$ (class extension), and $\mathtt{C} = \mathtt{MSS}$ (declared methods).

The assumption $\mathtt{C} <_1 \mathtt{C'}$ means "C extends C′", while $\mathtt{C} = \mathtt{MSS}$ means "C declares all and only all the methods specified in MSS". The intuitive meaning of the other assumptions has already been explained in Section 2.

Finally, a type environment Γ can contain both global and local type assumptions.

Typing rules for separate compilation are defined in Figure 3.

The top-level rule defines the compilation of a program P, whose type is $\langle \tau_1..\tau_n \rangle$, into a set of binary fragments $\langle \mathtt{B}_1..\mathtt{B}_n \rangle$. The provided environment, Γ, is enriched with the type of the fragments to compile because of the possibility of recursion among them. The resulting environment Γ' must be well-formed;

$$\frac{\begin{array}{l}\vdash \Gamma'\diamond \\ \forall i \in 1,..,n\ \Gamma' \vdash \mathtt{S}_i : \tau_i \leadsto \mathtt{B}_i\end{array}}{\Gamma \vdash \mathtt{P} : \langle \tau_1..\tau_n \rangle \leadsto \langle \mathtt{B}_1..\mathtt{B}_n \rangle} \quad \begin{array}{l}\mathtt{P} = \langle \mathtt{S}_1..\mathtt{S}_n \rangle \\ \forall i \in 1,..,n\ \ \mathtt{C}_i = \mathit{className}(\mathtt{S}_i) \\ \qquad\qquad \tau_i = \mathit{extractType}(\mathtt{S}_i) \\ \Gamma' = \Gamma, \mathtt{C}_1 \mapsto \tau_1, .., \mathtt{C}_n \mapsto \tau_n\end{array}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{MDS}^s \leadsto \mathtt{MDS}^b \\ \forall\ i \in 1..n\ \ \Gamma \vdash \mathtt{C}'\#\mathtt{MS}_i \\ \Gamma \vdash \mathtt{C}' \not\leq \mathtt{C}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{class\ C\ extends\ C'}\ \{\mathtt{MDS}^s\} : \langle \mathtt{C}', \mathtt{MS}_1..\mathtt{MS}_n \rangle \leadsto \\ \qquad \mathtt{class\ C\ extends\ C'}\ \{\mathtt{MDS}^b\}\end{array}} \quad \mathit{extractType}(\mathtt{MDS}^s) = \mathtt{MS}_1..\mathtt{MS}_n$$

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

$$\frac{\begin{array}{l}\Gamma; \{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash \mathtt{E}^s : \mathtt{T} \leadsto \mathtt{E}^b \\ \Gamma \vdash \mathtt{T} \leq \mathtt{T}_0 \\ \forall\ i \in 0..n\ \ \Gamma \vdash \exists\, \mathtt{T}_i\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{E}^s;\ \} \leadsto \\ \qquad \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{E}^b;\ \}\end{array}}$$

$$\frac{\Gamma \vdash \exists\, \mathtt{C}}{\Gamma; \Pi \vdash \mathtt{new\ C} : \mathtt{C} \leadsto \mathtt{new\ C}}$$

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

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

$\mathit{extractType}(\mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \}) = \langle \mathtt{C}', \mathit{extractType}(\mathtt{MDS}^s) \rangle$
$\mathit{extractType}(\mathtt{MDS}^s) = \mathit{extractType}(\mathtt{MD}_1^s) \ldots \mathit{extractType}(\mathtt{MD}_n^s)$
$\mathit{extractType}(\mathtt{MH}\ \{\ \mathtt{return}\ \mathtt{E}^s;\ \}) = \mathit{extractType}(\mathtt{MH})$
$\mathit{extractType}(\mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)) = \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1..\mathtt{T}_n)$

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

Fig. 3. True separate compilation

the well-formedness of type environments (judgment $\vdash \Gamma\diamond$ defined in Figure 4) relies on that for well-formed global type environments. In other words, Γ is well-formed (that is, consistent) iff there exists a well-formed global type environment which entails Γ.

$$\frac{\mathit{\Gamma}^G \vdash \mathit{\Gamma} \quad \vdash \mathit{\Gamma}^G\diamond}{\vdash \mathit{\Gamma}\diamond} \qquad \frac{\mathit{\Gamma}^G \vdash \mathit{\Gamma}^G\diamond}{\vdash \mathit{\Gamma}^G\diamond} \qquad \frac{}{\mathit{\Gamma}^G \vdash \emptyset\diamond}$$

$$\frac{}{\mathit{\Gamma}^G \vdash \mathtt{T}\diamond_{\mathtt{type}}} \quad \begin{array}{l} \mathtt{T} = \mathtt{int}\vee \\ \mathtt{T} = \mathtt{Object}\vee \\ \mathtt{T} \in \mathit{Def}(\mathit{\Gamma}^G) \end{array} \qquad \frac{\forall i \in 0..n \ \mathit{\Gamma}^G \vdash \mathtt{T}_i\diamond_{\mathtt{type}}}{\mathit{\Gamma}^G \vdash \mathtt{T}_0 \ \mathtt{m}(\mathtt{T}_1..\mathtt{T}_n)\diamond_{\mathtt{MS}}} \qquad \frac{\forall i \in 1..n \ \mathit{\Gamma}^G \vdash \mathtt{MS}_i\diamond_{\mathtt{MS}}}{\mathit{\Gamma}^G \vdash \mathtt{MS}_1 \ldots \mathtt{MS}_n\diamond_{\mathtt{MSS}}}$$

$$\frac{}{\mathit{\Gamma}^G \vdash \mathtt{Object} : \mathit{\Lambda}} \qquad \frac{\begin{array}{c}\mathit{\Gamma}^G \vdash \mathtt{C}' : \mathtt{MSS}' \\ \mathit{\Gamma}^G \vdash \mathtt{MSS}\diamond_{\mathtt{MSS}}\end{array}}{\mathit{\Gamma}^G \vdash \mathtt{C} : \mathtt{MSS} \cup \mathtt{MSS}'} \quad \begin{array}{l} \mathit{\Gamma}^G(\mathtt{C}) = <\mathtt{C}', \mathtt{MSS}> \\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{MSS}, \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{MSS}' \implies \mathtt{T} = \mathtt{T}' \\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}), \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{MSS} \implies \mathtt{T} = \mathtt{T}' \end{array}$$

$$\frac{\mathit{\Gamma}^G \vdash \mathit{\Gamma}_1^G \diamond \quad \mathit{\Gamma}^G \vdash \mathtt{C} : <\mathtt{C}', \mathtt{MSS}>}{\mathit{\Gamma}^G \vdash \mathit{\Gamma}_1^G, \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}>\diamond} \quad \mathtt{C} \notin \mathit{Def}(\mathit{\Gamma}_1^G)$$

Fig. 4. Well-formed type environments

The rule which defines the compilation of a single fragment for class C requires that the superclass C′ can be safely extended by the signatures of methods C declares and that $\mathtt{C}' \not\leq \mathtt{C}$ in order not to create a cycle in the subclassing hierarchy. There is not an explicit assumption on the existence of the superclass because it is entailed by the latter judgment.

The rule which compiles a single body requires the existences of return type and each parameter type.

The entailment of type environments is given in Figure 5.

The first two rules for environment entailment simply say that $\mathit{\Gamma}_1 \vdash \mathit{\Gamma}_2$ is valid if each type assumption γ contained in $\mathit{\Gamma}_2$ is entailed by $\mathit{\Gamma}_1$; the remaining rules cover the cases when $\mathit{\Gamma}_2$ is a single atomic type assumption γ.

Rules for class existence and subtyping are quite straightforward except, maybe, rule (*single inh*) which imposes that the subclassing hierarchy must be a tree. If $\mathtt{C}_1$ is a subclass of both $\mathtt{C}_2$ and $\mathtt{C}_3$ then either $\mathtt{C}_2$ is a subclass of $\mathtt{C}_3$ or vice versa. If $\mathtt{C}_1$ directly extends $\mathtt{C}_2$, then the only possibility is $\mathtt{C}_2 \leq \mathtt{C}_3$.

The other rules deal with the selection and method overriding.

The axiom (*sel obj*) states that 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.

The rule (*sel define*) covers the case in which the sequence MSS of the signatures of all methods declared in class C contains the method $\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})$. In this case 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 specifier $[\mathtt{C}, \bar{\mathtt{T}}, \mathtt{T}]$. This means that in this case the method invocation can be successfully compiled even when no explicit information on the methods of the superclass of C is available.

The rule (*sel down*) permits to derive a selection assumption for a class C given the selection assumption for its superclass C′ and the knowledge of the method signatures declared by C. The side-condition of the rule uses two

$$(empty)\ \frac{}{\Gamma \vdash \Lambda} \qquad (conc)\ \frac{\Gamma \vdash \Gamma_1 \quad \Gamma \vdash \gamma}{\Gamma \vdash \Gamma_1, \gamma} \qquad (singleton)\ \frac{}{\Gamma_1, \gamma, \Gamma_2 \vdash \gamma}$$

$$(ex\text{-}sub)\ \frac{\Gamma \vdash \mathtt{C} \leq \mathtt{C}'}{\Gamma \vdash \exists \mathtt{C}}\ \mathtt{C}' \notin \{\mathtt{C}, \mathtt{Object}\} \qquad (ex\text{-}sel)\ \frac{\Gamma \vdash Sel(\mathtt{C}_0, \_, \_) = [\mathtt{C}_1, \_, \_]..[\mathtt{C}_n, \_, \_]}{i \in 0..n\ \Gamma \vdash \exists \mathtt{C}_i}$$

$$(ex\text{-}\#)\ \frac{\Gamma \vdash \mathtt{C}\#\_\ \_(\_)}{\Gamma \vdash \exists \mathtt{C}} \qquad (ex\text{-}\not\leq)\ \frac{\Gamma \vdash \mathtt{C} \not\leq \mathtt{C}'}{\Gamma \vdash \exists \mathtt{C}}$$

$$(ax \not\leq)\ \frac{}{\Gamma \vdash \mathtt{C} \not\leq \mathtt{C}'}\ \mathtt{isNotSubtype}(\Gamma, \mathtt{C}, \mathtt{C}') \qquad (\leq obj)\ \frac{}{\Gamma \vdash \mathtt{C} \leq \mathtt{Object}}$$

$$(refl)\ \frac{}{\Gamma \vdash \mathtt{T} \leq \mathtt{T}} \qquad (trans)\ \frac{\Gamma \vdash \mathtt{C}_1 \leq \mathtt{C}_2 \quad \Gamma \vdash \mathtt{C}_2 \leq \mathtt{C}_3}{\Gamma \vdash \mathtt{C}_1 \leq \mathtt{C}_3} \qquad (imm)\ \frac{\Gamma \vdash \mathtt{C} <_1 \mathtt{C}'}{\Gamma \vdash \mathtt{C} \leq \mathtt{C}'}$$

$$(single\ inh)\ \frac{\Gamma \vdash \mathtt{C}_1 <_1 \mathtt{C}_2 \quad \Gamma \vdash \mathtt{C}_1 \leq \mathtt{C}_3}{\Gamma \vdash \mathtt{C}_2 \leq \mathtt{C}_3}\ \mathtt{C}_3 \notin \{\mathtt{C}_1, \mathtt{C}_2\}$$

$$(sub\ sel)\ \frac{\begin{array}{c}\Gamma \vdash Sel(\mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n) = \mu\mathtt{s}_1\, [\mathtt{C}', \mathtt{T'}_1 \ldots \mathtt{T'}_n, \mathtt{T}']\, \mu\mathtt{s}_2 \\ \forall i \in 1..n\ \ \mathtt{T}_i \neq T'_i \implies \Gamma \vdash \mathtt{T}_i \leq \mathtt{T}'_i\end{array}}{\Gamma \vdash \mathtt{C} \leq \mathtt{C}'}$$

$$(sub\ vector)\ \frac{\forall\, i \in 1..n\ \ \Gamma \vdash \mathtt{T}_i \leq \mathtt{T}'_i}{\Gamma \vdash \mathtt{T}_1 \ldots \mathtt{T}_n \leq \mathtt{T'}_1 \ldots \mathtt{T'}_n} \qquad (sel\ obj)\ \frac{}{\Gamma \vdash Sel(\mathtt{Object}, \mathtt{m}, \bar{\mathtt{T}}) = \Lambda}$$

$$(sel\ define)\ \frac{\Gamma \vdash \mathtt{C} = \mathtt{MSS}_1\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})\, \mathtt{MSS}_2}{\Gamma \vdash Sel(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = [\mathtt{C}, \bar{\mathtt{T}}, \mathtt{T}]}$$

$$(sel\ down)\ \frac{\begin{array}{l}\Gamma \vdash \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MSS}> \\ \Gamma \vdash Sel(\mathtt{C}', \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}\end{array}}{\Gamma \vdash Sel(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}\, \mu\mathtt{s}'}\ \begin{array}{l}Applicable(\Gamma, \mathtt{C}, \mathtt{MSS}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}' \\ (Matchable(\mathtt{C}, \mathtt{MSS}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}' \vee \mathtt{isWellFounded}(\Gamma, \bar{\mathtt{T}}))\end{array}$$

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

$$(simplify)\ \frac{\begin{array}{l}\Gamma \vdash Sel(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}_1\, [\mathtt{C}_1, \bar{\mathtt{T}}_1, \mathtt{T}_1]\, \mu\mathtt{s}_2 \\ \Gamma \vdash \mathtt{C}_2 \leq \mathtt{C}_1 \\ \Gamma \vdash \bar{\mathtt{T}}_2 \leq \bar{\mathtt{T}}_1\end{array}}{\Gamma \vdash Sel(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}_1\, \mu\mathtt{s}_2}\ \mu\mathtt{s}_1\, \mu\mathtt{s}_2 = \mu\mathtt{s}_3\, [\mathtt{C}_2, \bar{\mathtt{T}}_2, \mathtt{T}_2]\, \mu\mathtt{s}_4$$

$$(sel\ exact)\ \frac{\Gamma \vdash Sel(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}_1\, [\mathtt{C}', \bar{\mathtt{T}}', \mathtt{T}']\, \mu\mathtt{s}_2}{\Gamma \vdash Sel(\mathtt{C}', \mathtt{m}, \bar{\mathtt{T}}') = [\mathtt{C}', \bar{\mathtt{T}}', \mathtt{T}']} \qquad (\#\ obj)\ \frac{}{\Gamma \vdash \mathtt{Object}\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})}$$

$$(\#\ up)\ \frac{\Gamma \vdash \mathtt{C}\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \quad \Gamma \vdash \mathtt{C} \leq \mathtt{C}'}{\Gamma \vdash \mathtt{C}'\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})} \qquad (\#\ sel)\ \frac{\Gamma \vdash Sel(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \Lambda}{\Gamma \vdash \mathtt{C}\#\mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}})}$$

$$(\#\ down\ 1)\ \frac{\begin{array}{l}\Gamma \vdash \mathtt{C} \mapsto <\mathtt{C}', \mathtt{MS}_1 \ldots \mathtt{MS}_n> \\ \Gamma \vdash \mathtt{C}'\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})\end{array}}{\Gamma \vdash \mathtt{C}\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})}\ \forall\ i \in 1..n\ \mathtt{MS}_i =\ \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}) \implies \mathtt{T} = \mathtt{T}'$$

$$(\#\ down\ 2)\ \frac{\begin{array}{l}\Gamma \vdash \mathtt{C} <_1 \mathtt{C}' \\ \Gamma \vdash \mathtt{C}'\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \\ \Gamma \vdash Sel(\mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s}\end{array}}{\Gamma \vdash \mathtt{C}\#\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})}\ [\mathtt{C}, \bar{\mathtt{T}}, \mathtt{T}'] \in \mu\mathtt{s} \implies \mathtt{T} = \mathtt{T}'$$

Fig. 5. Type environments entailment

$$\texttt{isWellFounded}(\Gamma, \texttt{T}_1 \ldots \texttt{T}_n) = \texttt{isWellFounded}(\Gamma, T_1) \wedge \ldots \wedge \texttt{isWellFounded}(\Gamma, \texttt{T}_n)$$

$$\texttt{isWellFounded}(\Gamma, \texttt{T}) = \begin{cases} \texttt{true} & \text{if } \texttt{T} \in \{\texttt{Object}, \texttt{int}\} \\ \texttt{isWellFounded}(\Gamma, \texttt{C}') & \text{if } \Gamma \vdash \texttt{C} <_1 \texttt{C}' \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$\texttt{isNotSubtype}(\Gamma, \texttt{C}, \texttt{C}') = \texttt{isWellFounded}(\Gamma, \texttt{C}) \wedge \texttt{C}' \notin \texttt{parents}(\Gamma, \texttt{C})$$

$$\texttt{parents}(\Gamma, \texttt{C}) = \begin{cases} \{\texttt{Object}\} & \text{if } \texttt{C} = \texttt{Object} \\ \{\texttt{C}\} \cup \texttt{parents}(\Gamma, \texttt{C}') & \text{if } \Gamma \vdash \texttt{C} <_1 \texttt{C}' \\ \bot & \text{otherwise} \end{cases}$$

$$\begin{aligned} &\mathit{Applicable}(\Gamma, \texttt{C}, \texttt{MSS}, \texttt{m}, \bar{\texttt{T}}) = \{[\texttt{C}, \bar{\texttt{T}}', \texttt{T}] \mid \texttt{T m}(\bar{\texttt{T}}') \in \texttt{MSS}, \Gamma \vdash \bar{\texttt{T}} \leq \bar{\texttt{T}}' \ \} \\ &\mathit{Matchable}(\texttt{C}, \texttt{MSS}, \texttt{m}, \bar{\texttt{T}}) = \{[\texttt{C}, \texttt{T}'_1 \ldots \texttt{T}'_n, \texttt{T}] \mid \ \texttt{T m}(\texttt{T}'_1 \ldots \texttt{T}'_n) \in \texttt{MSS}, \\ &\qquad\qquad\qquad\qquad\qquad\qquad \forall i \in 1..n \ (\texttt{T}_i = \texttt{int}) \iff (\texttt{T}'_i = \texttt{int})\} \end{aligned}$$

Fig. 6. Auxiliary functions

auxiliary functions defined in Figure 6: *Applicable* and *Matchable*. The former, $\mathit{Applicable}(\Gamma, \texttt{C}, \texttt{MSS}, \texttt{m}, \bar{\texttt{T}})$, gives the set of specifiers for all applicable methods for an invocation of a method named `m` with argument type $\bar{\texttt{T}}$ in `MSS`. A method is applicable for such an invocation if: (1) it is named `m`, (2) its parameter type is a supertype of $\bar{\texttt{T}}$ in Γ. The latter, $\mathit{Matchable}(\texttt{C}, \texttt{MSS}, \texttt{m}, \bar{\texttt{T}})$, gives the set of methods which *could* be applicable in some environment; that is, the one whose name is `m` and whose parameter type is a supertype of $\bar{\texttt{T}}$ in some environment. Of course, the set of *Applicable* methods is always contained in the *Matchable* one. Sometimes, it is strictly contained: a method `m`, in `MSS`, with parameter type $\bar{\texttt{T}}'$ is *not* applicable in Γ, but *Matchable*, when we cannot derive $\Gamma \vdash \bar{\texttt{T}} \leq \bar{\texttt{T}}'$. The key point is: there can be only two reasons why we cannot derive such a judgment. If all the supertypes of $\bar{\texttt{T}}$ are known, in which case $\texttt{isWellFounded}(\Gamma, \bar{\texttt{T}})$ holds, we are sure that the set of *Applicable* methods remains the same in every well-formed larger environment that entails Γ. Otherwise, there is not enough information in Γ to decide whether a method would be applicable in a larger environment or not. The side-condition ensure that a selection assumption is propagated to a subclass only when there are no methods which are not *Applicable* in Γ but *may* become applicable in a larger environment.

Propagating specifiers to a superclass is easier, see rule (*sel up*): 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.

Finally, rules (*simplify*) and (*sel exact*) simplify the set of specifiers removing some less specific specifier.

The remaining rules defines the overriding assumptions. The axiom (*# obj*) states that `Object` can be extended by any method.

The rule ($\#$ *up*) states that if we know that a certain class $\mathtt{C}$ can be extended by a method whose signature is $\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})$, then every superclass of it, $\mathtt{C}'$, can be extended with the same signature too.

When we know that no methods named $\mathtt{m}$ are applicable for an invocation on class $\mathtt{C}$ with argument type $\bar{\mathtt{T}}$ we know for sure that class $\mathtt{C}$ does not declare or inherit such a method so we can derive, see rule ($\#$ *sel*), that class $\mathtt{C}$ can be extended by a method named $\mathtt{m}$ with parameter type $\bar{\mathtt{T}}$ and any return type.

The last two rules, ($\#$ *down 1*) and ($\#$ *down 2*), say that if we know a class $\mathtt{C}'$ can be extended by a method $\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})$ then, a subclass $\mathtt{C}$ can be extended by such a method too unless it declares a method with the same name, same parameter type but *different* return type. Note that these two rules are independent: one cannot be derived from the other.

3.2 Formal Results

The main result presented in [4] states that separate compilation coincides with standard SDK Java compilation when restricted to closed programs.

Even though the same result could be proved here in a similar way, in this paper we are mainly concerned with properties one would expect in principle for any system supporting separate compilation and based on the notion of entailment of type environments, independently of the particular programming language.

A first basic property is *monotonicity* of the entailment relation.

Theorem 1 (Entailment Monotonicity). *For any type environment Γ_1, Γ_2 and Γ_3, if $\Gamma_1 \vdash \Gamma_2$ and $\Gamma_2 \vdash \Gamma_3$ are provable, then $\Gamma_1 \vdash \Gamma_3$ is provable as well.*

Proof. (Sketch) The proof is by induction on the proof structure for $\Gamma_2 \vdash \Gamma_3$ and by case analysis on the last applied meta-rules for that proof.

All cases are trivial except for meta-rules (*singleton*) and (*sel down*).

- (*singleton*) the claim comes directly from the fact that if $\Gamma_1 \vdash \Gamma_2$ is provable, then for any $\gamma \in \Gamma_2$, $\Gamma_1 \vdash \gamma$ is provable as well; this property can be proved by re-iterating application of meta-rule (*conc*) (that is, by induction on the number of assumptions in Γ_2);
- (*sel down*) the claim comes from the following two facts:
  1. for any Γ, $\mathtt{C}$ and $\mathtt{C}'$, $\mathtt{C} <_1 \mathtt{C}' \in \Gamma$ iff $\Gamma \vdash \mathtt{C} <_1 \mathtt{C}'$;
  2. for any Γ, γ, $\mathtt{C}$ and $\mathtt{C}'$, if $\Gamma \vdash \gamma$ and $\gamma \vdash \mathtt{C} \leq \mathtt{C}'$, then $\Gamma \vdash \mathtt{C} \leq \mathtt{C}'$.

  Fact 1 trivially holds because the only applicable meta-rule is (*singleton*), while fact 2 is a specific instance of this theorem which, however, can be independently proved again by induction on the structure of the proof of $\gamma \vdash \mathtt{C} \leq \mathtt{C}'$.

Monotonicity of entailment ensures that weaker environments make typings stronger. The definitions below are inspired by [15].

Definition 1 (Stronger Environment). *We say that type environment Γ_1 is* at least as strong as Γ_2 *(or, equivalently, Γ_2 is* at least as weak as Γ_1*) iff $\Gamma_1 \vdash \Gamma_2$ is provable.*

Definition 2 (Typing). *A typing t for programs is a triple of the form $(\Gamma \vdash \langle\tau_1..\tau_n\rangle \rightsquigarrow \langle B_1..B_n\rangle)$.*

Definition 3 (Stronger Typing). *Let $\mathsf{Term}(\Gamma \vdash \langle\tau_1..\tau_n\rangle \rightsquigarrow \langle B_1..B_n\rangle)$ denotes the set $\{P \mid \Gamma \vdash P : \langle\tau_1..\tau_n\rangle \rightsquigarrow \langle B_1..B_n\rangle$ is provable $\}$.*

Then typing t_1 is at least as strong as typing t_2 iff $\mathsf{Term}(t_1) \subseteq \mathsf{Term}(t_2)$ and stronger than t_2 iff $\mathsf{Term}(t_1) \subset \mathsf{Term}(t_2)$.

Theorem 2. *Let Γ_1 and Γ_2 be two type environments s.t. Γ_1 is consistent (that is, $\vdash \Gamma_1\diamond$ is provable) and at least as strong as Γ_2. Then, for each $\tau_1,\ldots,\tau_n$, $B_1,\ldots,B_n$, the typing $\Gamma_2 \vdash \langle\tau_1..\tau_n\rangle \rightsquigarrow \langle B_1..B_n\rangle$ is at least as strong as the typing $\Gamma_1 \vdash \langle\tau_1..\tau_n\rangle \rightsquigarrow \langle B_1..B_n\rangle$.*

Proof. (Sketch) The proof requires to extend the claim to all other kinds of typings, but in practice the most interesting case is the typing for expressions; in that case the proof proceeds by induction on the structures of proofs and uses the monotonicity of the entailment.

Compositionality (see also the Introduction) corresponds to the intuition that if we can separately compile a bunch of fragments and such fragments are compatible (that is, they intercheck) and form a closed program (that is, do not depend on externally defined classes), then we can compile the whole program in the empty environment and obtain the same result.

Theorem 3 (Compositionality). *Let $S_1,\ldots,S_n$ be the declarations of n (with $n \geq 0$) distinct classes $C_1,\ldots,C_n$, respectively; if there exist $\Gamma_1,\ldots,\Gamma_n$, $\tau_1,\ldots,\tau_n$, $B_1,\ldots,B_n$ s.t. the following conditions hold:*

- *(intrachecking) for $i \in 1..n$ $\Gamma_i \vdash \langle S_i\rangle : \langle\tau_i\rangle \rightsquigarrow B_i$ is provable;*
- *(interchecking) for $i \in 1..n$ $C_1 \mapsto \tau_1,\ldots,C_n \mapsto \tau_n \vdash \Gamma_i$ is provable.*

Then $\emptyset \vdash \langle S_1..S_n\rangle : \langle\tau_1..\tau_n\rangle \rightsquigarrow \langle B_1..B_n\rangle$ is provable as well.

Proof. (Sketch) Let $\Gamma = \mathtt{C}_1 \mapsto \tau_1,\ldots,\mathtt{C}_n \mapsto \tau_n$. Then, by the intrachecking hypothesis, the monotonicity of the entailment and the meta-rules for programs, classes and a single method declarations, we have that the global environment Γ does not contain inheritance cycles, respects the overriding and overloading rules and is closed, therefore $\vdash \Gamma\diamond$ is provable. By both the intrachecking and the interchecking hypotheses and by Theorem 2, $\Gamma \vdash \langle \mathtt{S}_i\rangle : \langle\tau_i\rangle \rightsquigarrow \mathtt{B}_i$ is provable for each $i \in 1..n$; finally, we can prove the claim by applying the meta-rule for compiling programs.

The last claim in this section proves the opposite directions of Theorem 3 and shows that our system is effectively stronger than SDK (see also the Introduction).

Theorem 4. *For each non-empty* $P = \langle S_1..S_n \rangle$, $\langle \tau_1..\tau_n \rangle$ *and* $\langle B_1..B_n \rangle$ *if* $\emptyset \vdash P : \langle \tau_1..\tau_n \rangle \rightsquigarrow \langle B_1..B_n \rangle$ *is provable, then for all* $i \in 1..n$ *there exists a local type environment* Γ_i^L *s.t.*

1. $\Gamma_i^L \vdash \langle S_i \rangle : \langle \tau_i \rangle \rightsquigarrow \langle B_i \rangle$ *is provable;*
2. *the typing* $(\Gamma_i^L \vdash \langle \tau_i \rangle \rightsquigarrow \langle B_i \rangle)$ *is stronger than* $(\Gamma^G \vdash \langle \tau_i \rangle \rightsquigarrow \langle B_i \rangle)$.

Proof. (Sketch) Let $\mathtt{C}_1, \ldots, \mathtt{C}_n$ be the names of $\mathtt{S}_1, \ldots, \mathtt{S}_n$, respectively and let us construct Γ_i^L by collecting all the local assumptions needed for applying the meta-rules in Figure 3 in the proof tree for $\Gamma^G \vdash \mathtt{S}_i : \tau_i \rightsquigarrow \mathtt{B}_i$, with $\Gamma^G = \mathtt{C}_1 \mapsto \tau_1, \ldots, \mathtt{C}_n \mapsto \tau_n$ (for instance, any application of the meta-rule for object creation requires a local assumption of the form $\exists\, \mathtt{C}$).

Then, by construction $\Gamma^G \vdash \Gamma_i^L$ is provable and since $\vdash \Gamma^G \diamond$ is provable by hypothesis, then $\vdash \Gamma_i^L, \mathtt{C}_i \mapsto \tau_i$ is provable as well by the meta-rule for well-formed type environment (first one in Figure 4) and the meta-rule (*conc*) in Figure 5 and by entailment monotonicity. Furthermore, by construction again, $\Gamma_i^L, \mathtt{C}_i \mapsto \tau_i \vdash \mathtt{S}_i : \tau_i \rightsquigarrow \mathtt{B}_i$ is provable, therefore $\Gamma_i^L \vdash \langle \mathtt{S}_i \rangle : \langle \tau_i \rangle \rightsquigarrow \langle \mathtt{B}_i \rangle$ is provable as well.

The second claim comes from Theorem 2 and from the fact that Γ^G is global while Γ_i^L is local.

4 Implementation Issues

This section describes the algorithm used in our prototype for implementing the system defined in Section 3, and gives some informal argumentations on its correctness. Our prototype takes a local environment Γ and a set of class declarations $\mathtt{C}_1 \ldots \mathtt{C}_n$ as input and produces standard bytecode for them as output.

There are three reasons why our prototype does not infer Γ from the code of $\mathtt{C}_1 \ldots \mathtt{C}_n$.

- Java-like languages are not based on type inference.
- Type inference cannot be used since binary fragments cannot be annotated with type variables (see Section 5).
- As pointed out in [1], inferring type information tends to slow down error removal because interface mismatches between fragments are detected at a later stage (at intercheck time instead of intracheck time). So this idea could be even counter productive from a software engineering perspective.

The prototype directly implements the type system defined in Figure 3, except that it does not generate code. For simplicity, code generation relies on the SDK compiler: if typechecking succeeds, then the prototype produces a bunch of skeleton classes, along $\mathtt{C}_1, \ldots, \mathtt{C}_n$, and invokes the standard compiler on them to produce the binaries[4].

The behavior of our prototype can be split into the following tasks:

[4] It could easily produce C# binaries as well, emitting C# sources instead of Java ones.

1. enrich the input Γ into Γ^+, by adding a type assumption of the form $\mathtt{C}_i \mapsto$ *extractType*$(\mathtt{S}_i)$ for each given declaration $\mathtt{S}_i$ for class $\mathtt{C}_i$, as in the main compilation rule;
2. process the subtyping/existence judgments contained in Γ^+;
3. verify the well-formedness of Γ^+, by looking for a well-formed global environment $\Gamma^G_{\mathit{final}}$ that entails Γ^+;
4. typecheck each class declaration;
5. produce a Java source based on $\Gamma^G_{\mathit{final}}$ and the given sources[5] and invoke the standard Java compiler on them producing the final binary fragments.

The first and the last point are trivial, so we focus only on the other ones. Because the global environment Γ^G found at step three is not discarded but used in the last step (when the compilation succeeds), we can roughly say that such a step "converts" a mixed environment Γ^+ into a global environment $\Gamma^G_{\mathit{final}}$. The key point, of course, is that such a conversion succeeds if and only if Γ^+ is consistent.

All the subtyping and existence judgments derivable in Γ^+ can be found in polynomial time from the judgments directly contained in Γ^+, so we calculate them first. The other kind of judgments are calculated on demand (when the typechecker needs them), except for the axioms and some special cases.

For subtyping in a first pass we scan all judgments in Γ^+ and collect the subtyping we can directly deduce by the rules given in Figure 5. Then, we close the relation just built by applying rules (*trans*) and (*single inh*). Note that this approach works because every constraint not already contained in Γ^+, but derivable from it, does *not* change the set of derivable subtypes.

After having calculated the subtyping relation we scan another time the judgments contained in Γ^+ to find the derivable existence judgments. Note that because of the rule ($\#$ *up*), we need to know the subclassing relation to discover all the existence judgments. In fact, rule ($\#$ *up*) may, indirectly, introduce the existence of a class (because of rule (*ex-$\#$*)).

Once the subtyping and existence judgments are known, our engine can find out whether a parent or selection judgment is derivable or not in Γ^+. Even if the involved rules are obviously different, the idea behind the way our engine can decide whether a parent or selection judgment is derivable is the same, so we describe the general approach.

$$\boxed{\dfrac{\cdots \quad \dfrac{\cdots \quad \dfrac{\text{A tree}}{\Gamma \vdash \gamma^L}}{\cdots}}{\Gamma \vdash \gamma^L} \rightsquigarrow \dfrac{\text{A tree}}{\Gamma \vdash \gamma^L}}$$

Fig. 7. Derivation tree pruning

[5] The translation of a global environment to a set of classes just consists of creating skeleton classes with dummy bodies.

A crucial observation is depicted in Figure 7: if a derivation tree for $\Gamma \vdash \gamma^L$ exists and it uses a sub-tree for the same $\Gamma \vdash \gamma^L$, then there exists a shorter derivation, which does not derive such a judgment twice. This observation means that we can search for a derivation building recursively a derivation tree for the target judgment and backtrack each time we need a proof for a judgment that needs already to be proved in the derivation tree we are building. Another observation is that the types we use in searching for the derivations are finite: they are just the primitive type `int` and the types explicitly mentioned in Γ^+. These two observations, together, ensure that we can decide, in finite time, whether a judgment can be derived in Γ^+.

The search for a well-formed $\Gamma^G_{\mathit{final}}$ entailing Γ^+ can be described by the following steps:

1. **classes discovery**; collecting all the class names used in Γ^+ we build Γ^G_ϵ, a global environment where all classes are temporarily empty (that is, without methods) and without a parent class. Note that Γ^G_ϵ may contain classes for which the existence cannot be derived in Γ^+. Consider, for instance, $\Gamma_2 = \mathit{Sel}(\mathtt{C}, \mathtt{m}, \_) = [\mathtt{C}, \_, \mathtt{C}']$. From this constraint alone we cannot derive the existence of $\mathtt{C}'$ in our system, that is, $\Gamma_2 \not\vdash \exists\, \mathtt{C}'$, but every well-formed global environment Γ^G_2 such that $\Gamma^G_2 \vdash \Gamma_2$ must contain a definition of $\mathtt{C}'$ because it must contain a method named `m` with return type $\mathtt{C}'$.
2. **first step of method addition**; we process each global constraint $\mathtt{C} = \mathtt{MSS}$ contained[6] in Γ^+, assigning to class `C` the set of method signatures `MSS`. During this step we have to ensure that for each class there is no conflicting global class definitions; there *may* be more than one global class definition for the same class, but the set of methods they define must coincide. Furthermore, no method can be defined twice in each definition. The classes so built are "frozen"; that is, no more methods can be added to them. The other, still empty, classes can instead be changed (enriched) by the following step.
3. **second step of method addition**; for each selection constraint $\mathit{Sel}(\_, \mathtt{m}, \_) = [\mathtt{C}_1, \bar{\mathtt{T}}_1, \mathtt{T}_1] \ldots [\mathtt{C}_n, \bar{\mathtt{T}}_n, \mathtt{T}_n]$ contained in Γ we add to every class $\mathtt{C}_i$ for $i \in 1..n$ a method named `m` with parameter type $\bar{\mathtt{T}}_i$ and return type $\mathtt{T}_i$; if $\mathtt{C}_i$ is frozen the "addition" succeeds only if the method is already present in the class.
4. **subclassing construction**; from the subtype information contained in Γ we find, using an adaptation of the topological sort algorithm, for each class in $\mathit{Def}(\Gamma^G_\epsilon)$ a unique parent class, obtaining $\Gamma^G_{\mathit{final}}$. Note that sometimes there is not a *unique* parent class, for instance: suppose both $\mathtt{C} \leq \mathtt{C}_1$ and $\mathtt{C} \leq \mathtt{C}_2$ hold without any relationship between $\mathtt{C}_1$ and $\mathtt{C}_2$. Since Java and C# do not support multiple inheritance, we have to make a choice and "flatten" the graph to $\mathtt{C} \leq \mathtt{C}_1 \leq \mathtt{C}_2$ or $\mathtt{C} \leq \mathtt{C}_2 \leq \mathtt{C}_1$. For now let us cheat and assume the algorithm can choose the "right" branch, we describe the real behavior

[6] Here "contained" is used with a loose meaning: remember that $\mathtt{C} \mapsto \langle \mathtt{C}', \mathtt{MSS} \rangle$ is a shortcut for $\mathtt{C} <_1 \mathtt{C}'$ and $\mathtt{C} = \mathtt{MSS}$.

later on. If we find a cyclical subtyping dependency then no well-formed global environment can entail Γ^+ and we just give up here.

Such a construction of global environment $\Gamma^G_{\mathit{final}}$ automatically guarantees:

- $\Gamma^+ \vdash \exists\mathtt{C} \implies \Gamma^G_{\mathit{final}} \vdash \exists\mathtt{C}$ because any *mentioned* class in Γ^+ is defined in $\Gamma^G_{\mathit{final}}$;
- $\Gamma^+ \vdash \mathtt{C} \leq \mathtt{C}' \implies \Gamma^G_{\mathit{final}} \vdash \mathtt{C} \leq \mathtt{C}'$

It remains to check whether $\Gamma^G_{\mathit{final}}$ is well-formed or not and if it respects the other constraints. In order to check the well-formedness of $\Gamma^G_{\mathit{final}}$ we have to check if the rules for method overriding (we adopt the Java rules, see 8.4.6.3 of [8]) are satisfied (we do not need to check the acyclicity of class hierarchy because we already checked it and built a proper hierarchy). So, we have to check that if a method m_1 overrides another method m_2 then their return types must be the same. If $\Gamma^G_{\mathit{final}}$ is well-formed, we must finally check:

1. The rules on selection constraints; that is, for each selection constraint γ^L contained in Γ we must check that for every applicable method in $\Gamma^G_{\mathit{final}}$ there is a method specifier in γ^L corresponding to a more specific method (of course, they can coincide).
2. That for every $\gamma^L = \mathtt{C} \not\leq \mathtt{C}'$ in Γ^+, class $\mathtt{C}$ does not extend $\mathtt{C}'$ in $\Gamma^G_{\mathit{final}}$.

These last two steps ensure that $\Gamma^G_{\mathit{final}} \vdash \Gamma^+$ is provable. If these checks fail there are two subcases: if there are other choices that can be made in building the hierarchy tree we just go back and try another tree (that is, we go on until we have tried *all* the possible trees) otherwise we just give up because a global environment Γ^G, satisfying our requirements, can not exist. The idea behind this claim is that we try all the possible single inheritance hierarchies that respect the ordering constraints in a global environment which contains the "minimum" number of classes and methods in order to satisfy the other constraints. Intuitively, if it did exist a well-formed Γ^G_2 which entails Γ^+ containing some extra-classes (or methods), then these extraneous classes/methods can be removed and the resulting global environment is still well-formed and entails Γ^+ but, if this were the case, our search must have found it in the first place.

If all the previous checks succeed, we finally know Γ^+ is consistent and we can go on typechecking each class declaration.

Note that we can *not* typecheck the source fragments, using the standard Java typechecking, with the just found $\Gamma^G_{\mathit{final}}$ because there may exist two different well-formed global environments Γ_a and Γ_b, such that both entail Γ^+, on which the standard typechecking algorithm gives two different results. For instance, consider: $\Gamma_a = \mathtt{T} \mapsto \langle \mathtt{T}_1, \rangle, \mathtt{T}_1 \mapsto \langle \mathtt{T}_2, \rangle, \mathtt{T}_2 \mapsto \langle \mathtt{Object}, \rangle$ and $\Gamma_b = \mathtt{T} \mapsto \langle \mathtt{T}_2, \rangle, \mathtt{T}_2 \mapsto \langle \mathtt{T}_1, \rangle, \mathtt{T}_1 \mapsto \langle \mathtt{Object}, \rangle$. It is easy to see that the following class can be compiled only in Γ_a, whereas the typechecking in Γ_b fails because `int` is not a subtype of any class.

```
class C extends Object {
```

```
    int m() { return new C.foo(new T) ; }
    int foo(T1 t) { return 42 ; }
    C foo(T2 t) { return new C ; }
}
```

Analogously, there exist examples in which both compilations are defined but their resulting bytecodes differ.

So, we need to typecheck the class declaration in Γ^+ with rules given in Figure 3. The typechecking described in Figure 3 is "standard", given the ability to decide whether the judgment listed in Figure 5 are derivable or not. The implemented algorithm (for the described subset of Java) is available at `http://www.disi.unige.it/person/LagorioG/separate-compilation/`.

Unfortunately, in the worst case the complexity of our algorithm is exponential in the length of the input environment. Indeed, as it has been already proved in [2], checking the consistency of a local environment is an NP-complete problem. Anyway, in many cases the algorithm drops to polynomial time complexity (for instance, when Γ is a global environment); however, an average case complexity analysis is needed for understanding how this negative result has a practical impact on efficiency.

5 Related and Further Work

Separate compilation is an issue that has been deeply studied for progamming languages, especially in the context of *selective recompilation* [1] which seeks to reduce rebuilding time due to source modifications.

According to the classification given in [1], [9] adopts for ML an approach which involves both cut-off elimination and smart recompilation, while [13] investigates smartest recompilation, by employing type inference to derive the type assumptions needed for compiling an ML code fragment in isolation. Smart and smarter recompilations have been considered as well for C-like languages in [14, 12].

Unfortunately, the same kind of problems seem not to be seriously considered for Java-like languages. Even though selective recompilation might be considered not strictly essential in the presence of dynamic linking and verification, we claim that software development for both Java and C# would benefit from a better support for separate compilation and selective recompilation, since *earlier error detection* is an important principle of software engineering.

This paper, together with [5, 3, 4], can be considered as a first step towards a better support for separate compilation of Java-like languages, intended as object-oriented languages supporting dynamic linking and verification and relying on type annotations in the bytecode.

We have already discussed the problems of inferring the needed type assumptions directly from the code, like happens in the smartest recompilation [13].

The solution presented here is similar to attribute recompilation, according to the classification given in [1]. Here attributes correspond to local type assumptions which can be automatically inferred when compiling a closed set of fragments; these assumptions can be used later for selective recompilation. However, in order to compile an open fragment in isolation, the user must specify the needed type assumptions.

Solutions allowing inference of local type assumption even in presence of open fragments imply substantial modifications in order to support type variable annotations in the bytecode. For instance, by introducing the notion of pre-bytecode and by requiring static linking of pre-bytecode fragments in order to obtain a standard executable bytecode (see for instance [11]). A more drastic solution consists in allowing the execution of bytecode annotated with type variables, but this might imply major changes to the virtual machine and introduce both semantic and performance challenging problems.

The implementation of the system proposed here is a prototype and, in fact, efficiency problems should be better analyzed and a more realistic language should be supported, by considering other features common to Java and C#, like fields, constructors, static members and so on.

Another point interesting for future development concerns the completeness of the entailment relation of type environments. Indeed, there exist entailment judgments which should intuitively hold, but cannot be proved in our system, like, for instance:

$$\mathit{Sel}(\mathtt{C},\mathtt{m},\mathtt{T}) = [\mathtt{C},\mathtt{T}_1,\mathtt{C}],[\mathtt{C},\mathtt{T}_2,\mathtt{C}],\ \mathit{Sel}(\mathtt{C},\mathtt{m},\mathtt{T}) = [\mathtt{C},\mathtt{T}_1,\mathtt{C}] \vdash \mathtt{T}_1 \leq \mathtt{T}_2.$$

This amounts to require a logics for our system able to formally capture the intuition by defining the usual notions of model and satisfaction relation.

References

1. Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.
2. 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.
3. D. Ancona, G. Lagorio, and E. Zucca. A formal framework for Java separate compilation. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 609–635. Springer, 2002.
4. D. Ancona, G. Lagorio, and E. Zucca. True separate compilation of Java classes. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'02)*, pages 189–200. ACM Press, 2002.
5. D. Ancona and E. Zucca. True modules for Java-like languages. In J.L. Knudsen, editor, *ECOOP'01 - European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 354–380. Springer, 2001.
6. L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, 1997.
7. Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible models for dynamic linking. Submitted for publication, 2002.
8. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification, Second Edition*. Addison-Wesley, 2000.

9. R. Harper, P. Lee, F. Pfenning, and E. Rollins. A compilation manager for standard ML of New Jersey. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, July 94.
10. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Second edition, 1999.
11. S. McDirmid, M.Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*. ACM Press, October 2001.
12. Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, October 1988.
13. Z. Shao and A.W. Appel. Smartest recompilation. In *ACM Symp. on Principles of Programming Languages 1993*, pages 439–450. ACM Press, 1993.
14. Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.
15. J.B. Wells. The essence of principal typing. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming (ICALP'02)*, number 2380 in Lecture Notes in Computer Science, pages 913–925. Springer, 2002.