Stronger Typings for Separate Compilation of Java-like Languages (Extended Abstract)*

Davide Ancona and Giovanni Lagorio

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

Abstract. We define a formal system supporting separate compilation for a small but significant Java-like language.
This system is 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.

1 Introduction

Separate compilation of statically typed languages is an important feature of modern systems, since it promotes software reuse while retaining type safety and semantic consistency of programs.
As pointed out by Cardelli [7], separate compilation essentially corresponds to a typing judgment which has the general form $\Gamma \vdash \mathtt{S}{:}\tau$, where Γ is a type environment containing all necessary information needed for compiling in isolation the source fragment `S`, and τ is the inferred type of `S`. Using the terminology introduced in [7], we say that fragment `S` *intrachecks* in Γ and has type τ.
For instance, in the SDK systems, that is, those specifying the SDK Java[1] compiler [3, 9, 12], `S` corresponds to the declaration of a class `C`, Γ_s is a *standard* type environment containing all the information on the classes needed by `C` and having the general form $\mathtt{C}_1{:}\tau_1, \ldots, \mathtt{C}_n{:}\tau_n$, and τ and τ_i, for $i = 1..n$, are *class*

* Partially supported by Dynamic Assembly, Reconfiguration Type-checking - EC project IST-2001-33477, APPSEM II - Thematic network IST-2001-38957, and Murst NAPOLI - Network Aware Programming: Oggetti, Linguaggi, Implementazioni.

[1] Throughout this paper we will only mention Java for brevity, but all claims about Java can be replaced with analogous claims about C#.

types obtained by extracting all type annotations in the code, that is, the direct superclass, the headers of methods and so forth.

Another crucial notion for separate compilation is *interchecking* [7]; in general, the fact that two fragments $\mathtt{S}_1$ and $\mathtt{S}_2$, named $\mathtt{C}_1$ and $\mathtt{C}_2$, respectively, intracheck does not guarantee that the program obtained by merging $\mathtt{S}_1$ with $\mathtt{S}_2$ is statically correct. To ensure this we need to either recompile $\mathtt{S}_1$ and $\mathtt{S}_2$ as a whole[2] (but this solution obviously contradicts separate compilation), or just verify that $\mathtt{S}_1$ and $\mathtt{S}_2$ intercheck by proving that the type environment $\Gamma_s = \mathtt{C}_1{:}\tau_1, \mathtt{C}_2{:}\tau_2$ is *stronger* than both the environments Γ_1 and Γ_2 used for intrachecking $\mathtt{S}_1$ and $\mathtt{S}_2$, respectively.

In order to have an effective interchecking process, the notion of "stronger type environment" need to be syntactically captured by an *entailment* relation $\vdash$ between type environments, which must be at least sound: $\Gamma_1 \vdash \Gamma_2$ implies that Γ_1 is stronger than Γ_2, that is, for all $\mathtt{S}$ and τ, if $\Gamma_2 \vdash \mathtt{S}{:}\tau$ then $\Gamma_1 \vdash \mathtt{S}{:}\tau$. In the simplest cases the entailment relation reduces to type environment inclusion, but is not so simple in the system presented here.

The notion of interchecking can be fruitfully used for enhancing *selective recompilation* [1], which is the ability of avoiding unnecessary recompilation steps after code modification, while retaining both type safety and semantic consistency of programs. As an example, let us consider the following two classes, assuming that they are defined in separate fragments:

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

```
class P extends Object{
 int f(Object o){return 1;}
}
```

Classes `H` and `P` both intracheck and intercheck. Now let us change the definition of `P`; obviously `P` needs to be recompiled (let us assume that such a recompilation is successful), but what about `H`? In Java three different cases may occur:

Case 1. The modification of `P` invalidates neither the type safety nor the semantic consistency of `H`: if we recompile `H`, then we get no error and the same bytecode. This happens, for instance, if we add to `P` a new method `int h()`.

Case 2. The modification of `P` does not invalidate the type safety, but does compromise the semantic consistency of `H`: if we recompile `H` we get no error but we obtain a different bytecode. For instance, if the parameter type of `f` becomes `P`, then the bytecode for `H` must change correspondingly, since in the Java bytecode method calls are annotated with the types of the parameters of the resolved method.

Case 3. The modification of `P` compromises the type safety of `H`: if we recompile `H` we get an error; for instance, this happens if we change into `int` the parameter type of `f`.

In case 1 we would like to avoid an unnecessary recompilation of class `H`, while in case 2 we do want to force recompilation of `H`; however, this is not possi-

[2] For simplicity, we only consider interchecking of closed programs, even though this notion could be easily extended to open programs.

ble with the SDK compiler[3], where the user can choose either to recompile all classes (avoiding type unsafety and semantic inconsistency, but not unnecessary recompilation), or to recompile only modified classes (avoiding unnecessary recompilation, but not type unsafety and semantic inconsistency). For what concerns case 3, the best solution would consist in detecting the problem without reinspecting the source code for `H`, but since this task turns out to be quite hard, in the system presented in this paper cases 2 and 3 cannot be distinguished.
From the example above we can draw the following important conclusions:

– Case 2 shows that some class modification can affect the bytecode of other unchanged classes, because of the type annotations in the bytecode needed for run-time resolution of methods and verification of classes [14, 10]. Therefore, in order to capture case 2, the intrachecking judgment needs to be extended to take into account the generated code. In this paper we use an intrachecking judgment of the form $\Gamma \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$ with the meaning "in the type environment Γ the source fragment `S` intrachecks, has type τ, and compiles to the binary fragment `B`".
– In order to avoid an unnecessary recompilation in case 1, we can just redo interchecking for class `H` by using the entailment relation between type environments. Let Γ_H denote the type environment used for intrachecking class `H` before `P` was modified, so that $\Gamma_H \vdash \mathtt{S}_H : \tau_H \leadsto \mathtt{B}_H$ (for the appropriate $\mathtt{S}_H$, τ_H and $\mathtt{B}_H$), and let τ'_P denote the new type of `P` after its modification. If we can prove $\mathtt{H}{:}\tau_H, \mathtt{P}{:}\tau'_P \vdash \Gamma_H$, then we can conclude that $\mathtt{H}{:}\tau_H, \mathtt{P}{:}\tau'_P \vdash \mathtt{S}_H : \tau_H \leadsto \mathtt{B}_H$ holds, and, therefore, recompilation of `H` is unnecessary.
– In order to minimize the number of unnecessary recompilations, intrachecking typings should be as *strong* as possible. Adapting the formal definitions given by Wells [18] to our intrachecking judgments, a *typing* t is a pair $<\Gamma, \tau>$ and $\mathsf{Terms}(t)$ denotes the set of all pairs $<\mathtt{S}, \mathtt{B}>$ s.t. $\Gamma \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$ is provable in the system. Then, a typing t_1 is *stronger* than a typing t_2 if and only if $\mathsf{Terms}(t_1) \subseteq \mathsf{Terms}(t_2)$ (*strictly stronger* when the inclusion is proper). Intuitively, if t_1 is stronger than t_2 and $\mathtt{S} \in \mathsf{Terms}(t_1)$, then 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 and semantically consistent way without recompiling it.

Ideally, the best situation would be a type system with principal typings, where for each correct fragment there would exist the strongest typing [18]. However, in this paper we do not propose to solve the problem of principality for Java-like languages. Instead, we formally investigate a system where typings are strictly stronger than those of the SDK systems. The proof that our system has principal typings can be found in [5].
Indeed, as already argued in a previous paper [4], the intrachecking typings $<\Gamma, \tau>$ in the SDK systems are too weak. For instance, in order to successfully compile class `H`, the SDK compiler retrieves all the type information on class `P`,

[3] Throughout the paper we refer to the compiler of Java 2 SDK, version 1.4.1.

including its direct superclass, its method headers, and so on; therefore, class `H` is intrachecked under the assumptions that class `P` extends `Object` and its body exactly contains just one method, named `f`, with one parameter of type `Object`, and return type `int`. However, these type assumptions for intrachecking `H` are far from minimal. We would like to be able to express less restrictive assumptions like, for instance, "the invocation `p.f(new H())` can be resolved to a method `f`, with parameter type `Object`, and return type `int`".

While in functional programming, typings are made stronger by extending systems with more accurate types (for instance, intersection types) [18], here we take a dual approach by extending the SDK systems with "more accurate" type environments, that is, type environments able to express weaker requirements on classes. This is due to the fact that some Java features, like method overloading, require global analysis and thus conflict with modularity [7], therefore standard type environments assigning types to single classes are not expressive enough.

We extend standard type environments with local type assumptions [4], e.g., $\mathtt{C1} \leq \mathtt{C2}$, which requires class `C1` to be a subtype of `C2`, but says nothing, for instance, about the methods of `C1`; however, for Java other kinds of local type assumptions are needed.

In this paper we focus on two related issues. First, the system presented here is an evolution of the system defined in a previous paper [4], where two important properties are proved: *compositionality*, which does not hold for our previous system, and the fact that typings are strictly stronger than those of the SDK systems. Second, we show how this system can be effectively used for enhancing Java selective recompilation [8].

Compositionality is an expected property of separate compilation defined by Cardelli [7] as follows: "The linked program should have the same effect as a program obtained by merging all the sources together and compiling the result in a single step". In our system this amounts to requiring 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 is that our system is effectively "stronger" than the SDK ones: we prove that if Γ_s is a standard type environment, and $\Gamma_s \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$, then there exists a strictly weaker type environment Γ which yields a stronger typing $<\Gamma, \tau>$ for `S` and `B`, and, more interestingly, can be effectively constructed by collecting all local assumptions needed to prove $\Gamma_s \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$.

This last result shows that, during the compilation of a closed program `P` (that is, a self-contained set of code fragments) it is possible to infer for each fragment of `P` the set of local type assumptions (that is, a type environment) which is really needed by the compiler for that particular fragment; the compiler can take advantage of these automatically generated type environments for enhancing Java selective recompilation. In particular, we show how this idea can be used for enhancing *Javamake*, the only Java-specific make technology we are aware of (at least in form of a publication) [8, 13].

The rest of the paper is structured as follows. Section 2 is a gentle introduction to the system, whereas Section 3 discusses the most important related work [8] and shows how our system can be used in practice for enhancing *Javamake*. Finally, Section 4 contains pointers to other related work and some conclusions. All the formal definitions and results can be found in the extended version of this paper [2].

2 An Informal Presentation

This section is a gentle introduction to the system formally defined in the extended version of this paper [2]. More precisely, the two basic notions of *local* type assumption and *entailment* relation between type environments are informally presented and motivated.

Local Type Assumptions Let us consider a little bit more involved version of the class `H` mentioned in the Introduction:

```
class H extends P {
 int g(P p) {return p.f(new H());}
 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 would need 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_s which contains standard type assumptions on the classes `P`, `U`, `X` and `Y`. For instance, if Γ_s is defined by:

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

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. An environment like Γ_s containing standard type assumptions only is called a *standard type environment*.
Under the assumptions contained in Γ_s class `H` can be successfully compiled to the following binary fragment $\texttt{B}_\texttt{h}$:

```
class H extends P {
 int g(P p) {return p.<<P.f(Object)int>>(new H());}
 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 [14] 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.
Let us now try to relax the strong assumptions in Γ_s 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 containing `H` must provide a definition for the four classes which `H` depends on. In our system this is expressed by a local assumption of the form $\exists\,\mathtt{C}$, therefore Γ_l will contain at least the 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` (into $\mathtt{B_h}$) the following additional assumptions on class `P` must be added to Γ_l:

- $\mathtt{P} \not\leq \mathtt{H}$: `P` cannot be a subtype of `H` since inheritance cannot be cyclic.
- $\mathtt{P}☺\mathtt{int\ g(P)}$: `P` can be correctly extended with method `int g(P)`; indeed, according to Java rules on method overriding, if `P` has a method `g(P)`, then `g` must have the same return type `int` as declared in `H`. Analogous requirements are needed for the other methods declared in `H`.
- $\mathtt{P.f(H)} \overset{\mathtt{res}}{\to} \mathtt{<Object, int>}$: invocation of method `f`, for an object of type `P` and with an argument of type `H`, is successfully resolved to a method with a parameter of type `Object` and return type `int`. 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). Note that we do not need to know the class where the method is declared, since the bytecode is annotated with the type of the receiver.

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 $\mathtt{Y} \leq \mathtt{X}$.
In conclusion, class `H` can be successfully compiled and produce $\mathtt{B_h}$ in the environment Γ_l defined by:

$$\begin{aligned}\Gamma_l = {} & \exists\,\mathtt{P}, \exists\,\mathtt{U}, \exists\,\mathtt{X}, \exists\,\mathtt{Y}, \mathtt{P} \not\leq \mathtt{H}, \mathtt{Y} \leq \mathtt{X}, \mathtt{P}☺\mathtt{int\ g(P)},\\ & \mathtt{P}☺\mathtt{int\ m()}, \mathtt{P}☺\mathtt{U\ id(U)}, \mathtt{P}☺\mathtt{X\ em(Y)}, \mathtt{P.f(H)} \overset{\mathtt{res}}{\to} \mathtt{<Object, int>}\end{aligned}$$

Furthermore, Γ_l is weaker than Γ_s; for instance, class `U` must extend `Object` and be empty in Γ_s, while in Γ_l it can extend any class and declare any method. The notion of stronger type environment is syntactically captured by an entailment relation on type environments.

Entailment of Type Environments Referring to the previous example, in our system the fact that Γ_l is weaker than Γ_s is formalized by the following property: for all $\mathtt{S}, \tau, \mathtt{B}$ if $\Gamma_l \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$ is provable, then $\Gamma_s \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$ is provable as well. However, since the definition above cannot be directly checked in an effective way, the notion of stronger type environment needs to be captured by an *entailment* relation (that is, a computable relation) between type environments.
For instance, in our system $\Gamma_s \vdash \Gamma_l$ can be proved. Furthermore, the entailment relation is proved to be *sound*, that is, if $\Gamma_1 \vdash \Gamma_2$ can be proved and Γ_1 is consistent (in the sense that it does not contain contradictory assumptions[4]), then Γ_1 is stronger than Γ_2. In the particular example, we can go further, by showing that Γ_l is actually strictly weaker than Γ_s.
Let us add in `H` the new method `int one(){return 1;}`. After this change, the new code for class `H` still intrachecks in Γ_s, whereas intrachecking of the same code in Γ_l fails, otherwise the system would not be compositional. To see this, let us consider the following new declaration for class `P`:

```
class P extends Object{
 int f(Object o){return 1;}
 P one(){return new P();}
}
```

The reader can easily verify that each type assumption in Γ_l about `P` is satisfied by the new version of `P` above, however if we put all classes together we obtain a statically incorrect program, since method `one` is redefined in `H` with a different return type. Therefore Γ_s is strictly stronger than Γ_l; from this last claim and from the soundness of the entailment we can deduce $\Gamma_l \nvdash \Gamma_s$.
Finally, we end this section with another example of provable entailment, by showing that Γ_l contains redundant assumptions which, in fact, can be removed without affecting the outcome of the compilation of class `H`.
Let us consider the type environment Γ_l' obtained from Γ_l by removing the two assumptions $\exists\,\mathtt{P}$ and $\exists\,\mathtt{Y}$. Then, both the entailments $\Gamma_l \vdash \Gamma_l'$ and $\Gamma_l' \vdash \Gamma_l$ can be proved, hence Γ_l and Γ_l' are equivalent. The first entailment is trivial to prove, since Γ_l' is included in Γ_l; the proof of the second entailment relies on the validity of the following two entailments:

$$\mathtt{P}☺\mathtt{int}\ \mathtt{g}(\mathtt{P}) \vdash \exists\,\mathtt{P} \qquad\qquad \mathtt{Y} \leq \mathtt{X} \vdash \exists\,\mathtt{Y}$$

Intuitively, these two entailments must be provable because assumptions $\mathtt{P}☺\mathtt{int}\ \mathtt{g}(\mathtt{P})$ and $\mathtt{Y} \leq \mathtt{X}$ can be verified only in presence of a definition for `P` and `Y`, respectively. On the other hand, $\exists\,\mathtt{X}$ cannot be entailed from $\mathtt{Y} \leq \mathtt{X}$; 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 $\exists\,\mathtt{X}$.

[4] Consistency can be checked by a polynomial time algorithm [2].

3 Selective Recompilation

While several papers have been written on the subject of selective recompilation (see Section 4), to our best knowledge only Dmitriev [8] and Lagorio [13] has focused on Java. Dmitriev's paper describes a make technology, based on smart dependency checking, that aims to keep a project (that is, a set of source and binary fragments) consistent while reducing the number of files to be recompiled. A project is said to be consistent when all its sources can be recompiled producing the same binaries as before. The main idea is to catalog all possible changes to a source code (as, for instance, adding/removing methods) establishing a criterion for finding a subset of dependent classes that have to be recompiled. A freely downloadable tool, *Javamake*, is based on such a paper and implements the selective recompilation for Java upon any Java compiler. This tool stores some type information for each project in database files which are used to determine which changes have been made to the sources with respect to the previous (consistent) version. Unfortunately, as pointed out by the author too, the approach is not based on a theoretical foundation, and therefore there is no proof of soundness. So, it might happen that *Javamake* fails to force the recompilation of some classes which is actually needed for ensuring the consistency of the project. Furthermore, *Javamake* cannot avoid a considerable amount of unnecessary recompilations. The main advantages of *Javamake* are that it is well documented and in practice its implementation can work upon any Java compiler.
Our system can be exploited for extending *Javamake* in order to sensibly decrease the number of unnecessary recompilations: a set of local type assumptions for a source fragment `C` can be automatically inferred when compiling `C` starting from a standard environment. These local type assumptions describe in a precise way what `C` needs in order to be recompiled into the same binary. These assumptions can be used to decide whether an unchanged class needs to be recompiled. In fact, if a new global environment still entails the local assumptions for a fragment `S`, then there is no need to recompile `S`. From the complexity point of view, checking whether a local assumption is entailed by a standard environment requires a polynomial algorithm that can be implemented efficiently [2]. Without going into the details, let us show the idea on the the example already discussed in Section 2:

```
class P extends Object {
 int f(Object o) { return 0 ; }
 // int f(int i) { return i ; }
}
class U extends Object {}
class Y extends X {}
class X extends Object {}
```

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

These classes compile successfully, and they form our example project. The compiler would generate the following type assumption for `H`:

$$\Gamma_l = \exists\,\mathtt{P}, \exists\,\mathtt{U}, \exists\,\mathtt{X}, \exists\,\mathtt{Y}, \mathtt{P} \not\leq \mathtt{H}, \mathtt{P}☺\mathtt{int\ g}(\mathtt{P}), \mathtt{P}☺\mathtt{int\ m}(),$$
$$\mathtt{P}☺\mathtt{U\ id}(\mathtt{U}), \mathtt{P}☺\mathtt{X\ em}(\mathtt{Y}), \mathtt{P.f}(\mathtt{H}) \overset{\mathtt{res}}{\to} <\mathtt{Object}, \mathtt{int}>, \mathtt{Y} \leq \mathtt{X}$$

If we add a method `f(int)` in `P` (that is, if we remove the comment in the previous listing), then class `H` would still call the same method as before, because the new method is not even applicable to the call with an argument of type `P`. These considerations are formally captured by the entailment relation: the new standard environment (that can be extracted from the new source for `P` and the old binaries of the other classes) still entails Γ_l therefore there is no need to recompile `H`. For instance, the reader can verify that $\texttt{P.f(H)} \overset{\texttt{res}}{\rightarrow} <\texttt{Object}, \texttt{int}>$ can still be entailed. On the other hand, whereas *Javamake*[5] is able to detect that classes `U`, `X` and `Y` need not to be recompiled, since they do not use `P` at all, it cannot distinguish between changes to a set of overloaded methods that alter the resolution of a particular call and changes that do not. So, *Javamake* would unnecessarily recompile class `H`, because it contains a call to `P.f`, producing the same binary as before.

4 Conclusion and Other Related Work

We have presented a new system for Java-like languages for better supporting separate compilation; the system relies on the notions of local type environment and entailment of type environments. In comparison with the standard type environments adopted by SDK systems, local type environments allow specification of weaker type assumptions for compiling in isolation a code fragment.
The system has been proved to be compositional and strictly stronger (and, therefore, more accurate) than SDK systems [2], and to have principal typings [5]. Furthermore, local type assumptions can be automatically generated the first time a program is compiled and can be used later to avoid unnecessary recompilations; based on these last considerations, in Section 3 we have sketched an algorithm for improving the Java-specific make technology *Javamake* [8].
Separate compilation is an issue that has been deeply studied for programming languages, especially in the context of *selective recompilation* [1] which seeks to reduce rebuilding time due to source modifications.
Besides the paper on *Javamake* [8] already discussed in Section 3, there is a number of papers on selective recompilation for several languages. According to the classification given in [1], [11] adopts for ML an approach which involves both cut-off elimination and smart recompilation, while [16] 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 [17, 15].
Unfortunately, very little has been done on this side for Java-like languages. This paper, together with [6, 3, 4, 8, 13], is a step towards a better understanding of separate compilation of Java-like languages.
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.

[5] Version 1.3.1, the latest available at the moment of writing this paper.

Acknowledgements We warmly thank Sophia Drossopoulou and Elena Zucca for their useful suggestions and corrections to previous versions of this paper.

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. Stronger typings for separate compilation of Java-like languages. Technical report, DISI, March 2003.
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. Principal typings for Java-like languages. To appear as DISI technical report.
6. 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.
7. L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, 1997.
8. M. Dmitriev. Language-specific make technology for the Java programming language. *ACM SIGPLAN Notices*, 37(11):373–385, 2002.
9. S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer, 1999.
10. S. Drossopoulou, G. Lagorio, and S. Eisenbach. Flexible models for dynamic linking. In *European Symposium on Programming 2003*, 2003.
11. 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.
12. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.
13. G. Lagorio. Towards a smart compilation manager for Java. In *Italian Conf. on Theoretical Computer Science 2003*, Lecture Notes in Computer Science. Springer, 2003. To appear.
14. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Second edition, 1999.
15. Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, October 1988.
16. Z. Shao and A.W. Appel. Smartest recompilation. In *ACM Symp. on Principles of Programming Languages 1993*, pages 439–450. ACM Press, 1993.
17. Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.
18. J.B. Wells. The essence of principal typings. 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.