JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 6
Special issue: ECOOP 2003 workshop on FTfJP

Stronger Typings for Smarter Recompilation of Java-like Languages

Davide Ancona, DISI, University of Genova, Italy
Giovanni Lagorio, DISI, University of Genova, Italy

We define an algorithm for smarter recompilation of a small but significant Java-like language; such an algorithm is inspired by a type system previously defined by Ancona and Zucca.
In comparison with all previous type systems for Java-like languages, this system enjoys the principal typings property, and is based on the two novel notions of *local type assumption* and *entailment of type environments*. The former allows the user to specify minimal requirements on the source fragments which need to be compiled in isolation, whereas the latter syntactically captures the concept of stronger type assumption.
One of the most important practical advantages of this system is a better support for selective recompilation; indeed, it is possible to define an algorithm directly driven by the typing rules which is able to avoid the unnecessary recompilation steps which are usually performed by the Java compilers.
The algorithm is smarter in the sense that it never forces useless recompilations, that is, recompilations which would generate the same binary fragment obtained from the previous compilation of the same source fragment.
Finally, we show that the algorithm can actually speed up the overall recompilation process, since checking for recompilation is always strictly less expensive than recompiling the same fragment.

1 INTRODUCTION

Selective recompilation [1] is the ability to avoid unnecessary recompilation steps after code modification, while retaining both type safety and semantic consistency of programs. Despite Java[1] compilers allow separate compilation, the support for selective recompilation is rather poor; as a consequence, for maintaining the consistency of large projects, users are forced to spend a considerable amount of time in recompiling software in reaction to source modifications.

As a starting example, let us consider the following two classes defined in two separate files:

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

Cite this article as follows: Davide Ancona, Giovanni Lagorio: "Stronger Typings for Smarter Recompilation of Java-like Languages", in *Journal of Object Technology*, vol. 3, no. 6, June 2004, Special issue: ECOOP 2003 workshop on FTfJP, pp. 5–25, http://www.jot.fm/issues/issue_2004_06/article1

```
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 compile successfully. 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:

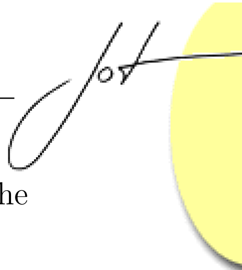
> Case 1. The modification of `P` invalidates neither the type safety nor the semantic consistency of `H`: if we recompile `H`, then we get no error and the same bytecode. This happens, for instance, if we add to `P` a new method `int h()`.
>
> Case 2. The modification of `P` does not invalidate the type safety of `H`, but does compromise its semantic consistency: 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 the parameter type of `f` to be `int`.

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 possible with the SDK compiler[2], where the user can in fact choose either to recompile all classes (avoiding type unsafety and semantic inconsistency, but not unnecessary recompilations), or to recompile only modified classes (avoiding unnecessary recompilations, but not type unsafety and semantic inconsistency).

The only Java-specific approach to selective recompilation we are aware of (at least in form of a publication) is *Javamake* [6]. *Javamake* is a make technology based on a quite elaborate dependency analysis which allows avoidance of useless recompilations. However this approach still allows some useless recompilations in the situations depicted by case 1, and (what is perhaps worst) does not guarantee that recompilation is always forced in the situations corresponding to case 2; this is mainly due to the fact that the algorithm is quite involved, and not based on a formal model.

On the contrary, the algorithm presented in this paper is directly driven by a formal type system for a small but significant subset of Java, which is just a simplified version of the system which has been proved to have principal typings by Ancona and Zucca [4]. Principality and soundness of the system guarantee that

[2] Throughout the paper we refer to the compiler of Java 2 SDK, version 1.4.2.



our algorithm is smarter, in the sense that recompilations are always avoided in the situations captured by case 1, and always forced in those depicted by case 2.

However, our algorithm is not the smartest, since for what concerns case 3 of our starting example, the best solution would consist in detecting the problem without re-inspecting the source code for `H`, but since this task turns out to be quite hard, our algorithm deals with cases 2 and 3 in the same way, by always forcing recompilation of the involved fragments.

The algorithm is heavily based on the formal system and indeed implements the two most peculiar notions: *local type assumptions*, and *entailment* between type environments.

In the SDK type systems, that is, those modeling typechecking of the SDK Java compiler [2, 7, 10], typings cannot be principal because the type environments express too strong requirements on classes [3]. For instance, in our starting example class `H` is successfully compiled by the SDK compiler 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`. This assumption, which we call standard, specifies the *global* type of `P` which is far from minimal for successfully compiling `H`. On the other hand, local - as opposite to global - type assumptions can express minimal requirements on classes as, for instance, "the invocation `p.f(new H())` can be resolved to a method `f`, with parameter type `Object`, and return type `int`".

As a consequence, our algorithm uses several kinds of local type assumptions; for instance, besides the conventional assumption $\mathtt{C1} \leq \mathtt{C2}$ requiring class `C1` to be a subtype of `C2`, the local assumption $\mathtt{C.m}(\bar{\mathtt{T}}) \stackrel{res}{\rightarrow} <\bar{\mathtt{T}}', \mathtt{T}>$ requires that the invocation of method `m` of an object of type `C` with arguments of type $\bar{\mathtt{T}}$ is successfully resolved to a method (obviously named `m`) with parameters of type $\bar{\mathtt{T}}'$ and return type `T`.

In order to avoid unnecessary recompilations, the algorithm needs to compare the minimal assumptions (that is, a type environment Γ_1) required for compiling a given class into a given binary, with the standard type assumptions extracted from the modified code (that is, a type environment Γ_2), in order to decide whether the assumptions in Γ_2 implies those in Γ_1. In other words, the algorithm implements an *entailment relation* $\vdash$ between type environments which is the same defined in the formal system, so that it is possible to check whether $\Gamma_2 \vdash \Gamma_1$ holds.

Recalling our starting example, let Γ_H denote the type environment used for compiling class `H` before `P` was modified, and let τ_H and τ_P denote the types of `H` and `P` extracted from the code after `P` was modified. Then the algorithm forces the recompilation of `H` if and only if the entailment relation $\mathtt{H}{:}\tau_H, \mathtt{P}{:}\tau_P \vdash \Gamma_H$ does not hold.

The paper is structured as follows. Section 2 is a gentle introduction to the formal system, whereas Section 3 gives the formal definitions. Section 4 makes a comparison with *Javamake*, introduces the basic ideas of our approach and then describes the algorithm and, finally, makes some considerations on the costs of smarter

selective recompilation. Section 5 contains pointers to other related work and some conclusions.

2 AN INFORMAL PRESENTATION

This section is a gentle introduction to the 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.

The language used in the examples is a basic subset of Java, where classes can only declare instance methods (possibly overloaded) and have just the implicit default (parameterless) constructor.

Local Type Assumptions Let us consider a slightly more complex version of the class H defined 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.

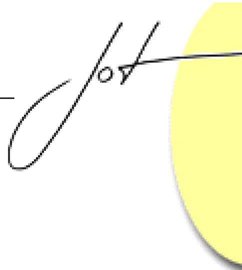
In our system this corresponds to compiling H in a type environment Γ_s which contains standard type assumptions on the classes P, U, X and Y. For instance, if Γ_s is defined by:

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

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 only standard type assumptions 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;}
}
```

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 [13] 5.1). Indeed, from our perspective the most critical difference between source and binary fragments is type annotations in method invocations, since it makes the problem of separate compilation (that is, separate typechecking plus code generation) substantially different from that of separate typechecking.

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:

- $\mathtt{P} \not< \mathtt{H}$: P cannot be a proper subtype of H since inheritance cannot be cyclic.
- P☺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.
- $\mathtt{P.f(H)} \overset{\mathtt{res}}{\rightarrow} \mathtt{<Object, int>}$: invocation of method f of an object of type P 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 to $\mathtt{B_h}$ in the environment Γ_l defined by:

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

Furthermore, Γ_s is stronger than Γ_l; 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 In our system the intuition that Γ_s is stronger than Γ_l 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 (see Figure 4 for the formal definition).

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, then Γ_1 is stronger than Γ_2. In the particular example, we can go further, by showing that Γ_s is actually strictly stronger than Γ_l.

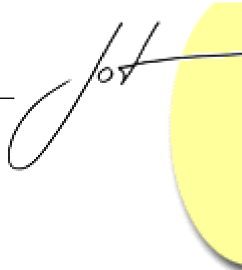
Let us add in H the new method `int one(){return 1;}`. After this change, the new code for class H still compiles in Γ_s, whereas the compilation of the same code in Γ_l fails. To see this, let us consider the following new declaration for class P:

```
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 $\Gamma_l \not\vdash \Gamma_s$.

3 FORMALIZATION

In this section we define the type system informally presented in the previous section. The system is just a slightly simplified version of the system defined by Ancona and Zucca [4].



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

Implicit assumptions:

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

Figure 1: Syntax

Definition of the System

The language we consider is a rather small but significant subset of Java; indeed, it includes one of the most critical features for separate compilation, which is Java *static overloading* ([5] – see the end of page 1).

The syntax of the language is defined in Figure 1, and type environments are defined in Figure 2; 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 $\texttt{MDS}^s$. A method declaration $\texttt{MD}^s$ 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.

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

$$
\begin{array}{rcl@{\qquad}rcl}
\texttt{MS} & ::= & \texttt{T m}(\bar{\texttt{T}}) & \Gamma_l & ::= & \gamma^l_1 \ldots \gamma^l_n \\
\texttt{MSS} & ::= & \{\texttt{MS}_1, \ldots, \texttt{MS}_n\} & \gamma^l & ::= & \exists\,\texttt{T} \mid \\
\mu & ::= & <\texttt{C}, \bar{\texttt{T}}, \texttt{T}> & & & \texttt{T} \leq \texttt{T}' \mid \\
\mu s & ::= & \{\mu_1, \ldots, \mu_n\} & & & \texttt{C.m}(\bar{\texttt{T}}) \stackrel{res}{\rightarrow} <\bar{\texttt{T}}', \texttt{T}> \mid \\
 & & & & & \texttt{C}☺\texttt{MS} \mid \\
\Gamma_s & ::= & \gamma^s_1 \ldots \gamma^s_n & & & \texttt{C} \not< \texttt{C}' \mid \\
\gamma^s & ::= & \texttt{C}{:}\tau & \Gamma & ::= & \gamma_1 \ldots \gamma_n \\
\tau & ::= & <\texttt{C}, \texttt{MSS}> & \gamma & ::= & \gamma^s \mid \gamma^l \\
\bar{\tau} & ::= & \langle \tau_1 \ldots \tau_n \rangle & & &
\end{array}
$$

Figure 2: Type environments

method to be invoked (see Section 2).

A standard type environment Γ_s is a possibly empty sequence of type assumptions of the form $\texttt{C}{:}<\texttt{C}', \texttt{MSS}>$ with the meaning "C extends C′ and declares exactly all methods[3] specified by MSS". We denote the empty sequence by Λ and use the notation $\mathit{def}(\Gamma_s)$ for the set $\{\texttt{C} \mid \texttt{C}{:}\tau \in \Gamma_s\}$ (where $\gamma \in \Gamma$ means Γ contains the assumption γ).

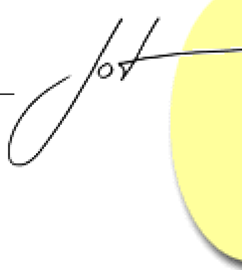
A local type environment Γ_l is a possibly empty sequence of local type assumptions of the following kinds:

- $\exists\,\texttt{T}$ with the meaning "T is defined";
- $\texttt{T} \leq \texttt{T}'$ with the meaning "T is a subtype of T′";
- $\texttt{C.m}(\bar{\texttt{T}}) \stackrel{res}{\rightarrow} <\bar{\texttt{T}}', \texttt{T}>$ with the meaning "the invocation of method m of an object of type C with arguments of type $\bar{\texttt{T}}$, is successfully resolved to a method (obviously named m) with parameters of type $\bar{\texttt{T}}'$ and return type T.
- $\texttt{C}☺\texttt{T m}(\texttt{T}_1 \ldots \texttt{T}_n)$ with the meaning "C can be extended by a subclass having method $\texttt{T m}(\texttt{T}_1 \ldots \texttt{T}_n)$ without breaking the Java rule on method overriding";
- $\texttt{C} \not< \texttt{C}'$ with the meaning "C is not a proper subtype of C′";

Finally, type environments Γ used for separate compilation can contain both standard and local type assumptions; standard type assumptions are needed for dealing with mutual recursion between classes (see rule for separate compilation of programs in Figure 3) and for compatibility with the SDK systems.

Typing rules for separate compilation are defined in Figure 3. The top-level rule defines the compilation of a program P, whose type is $\langle \tau_1 \ldots \tau_n \rangle$, into a set of binary fragments $\langle \texttt{B}_1 \ldots \texttt{B}_n \rangle$. The provided environment, Γ, is enriched with the type of the fragments to compile to deal with mutual recursion. The resulting environment Γ'

[3]For simplicity, in our language instance methods are the only members a class can contain.



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

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{MD}^s_i \leadsto \mathtt{MD}^b_i \ \forall\ i \in 1,\dots,n \\ \Gamma \vdash \mathtt{C}' \circledcirc \mathtt{MS}_i \ \forall\ i \in 1,\dots,n \\ \Gamma \vdash \mathtt{C}' \not\leq \mathtt{C}\end{array}}{\begin{array}{ll}\Gamma \vdash & \mathtt{class\ C\ extends\ C'}\ \{\mathtt{MDS}^s\} : \langle \mathtt{C}', \mathtt{MSS}\rangle \leadsto \\ & \mathtt{class\ C\ extends\ C'}\ \{\mathtt{MDS}^b\}\end{array}} \quad \begin{array}{l} \mathtt{MDS}^s = \mathtt{MD}^s_1 \dots \mathtt{MD}^s_n \\ \mathtt{MSS} = \{\mathtt{MS}_1, \dots \mathtt{MS}_n\} \\ \mathit{type}(\mathtt{MD}^s_i) = \mathtt{MS}_i \\ \forall\ i \in 1,\dots,n \end{array}$$

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

$$\frac{}{\Gamma; \Pi \vdash \mathtt{x} : \mathtt{T} \leadsto \mathtt{x}} \quad \Pi(\mathtt{x}) = \mathtt{T} \qquad \frac{\Gamma \vdash \exists\, \mathtt{C}}{\Gamma; \Pi \vdash \mathtt{new\ C} : \mathtt{C} \leadsto \mathtt{new\ C}}$$

$$\frac{\begin{array}{l}\Gamma; \Pi \vdash \mathtt{E}^s_0 : \mathtt{C} \leadsto \mathtt{E}^b_0 \\ \Gamma; \Pi \vdash \mathtt{E}^s_i : \mathtt{T}_i \leadsto \mathtt{E}^b_i \ \forall i \in 1,\dots,n \\ \Gamma \vdash \mathtt{C.m}(\mathtt{T}_1 \dots \mathtt{T}_n) \stackrel{res}{\to} <\bar{\mathtt{T}}', \mathtt{T}'>\end{array}}{\Gamma; \Pi \vdash \mathtt{E}^s_0.m(\mathtt{E}^s_1, \dots, \mathtt{E}^s_n) : \mathtt{T}' \leadsto \mathtt{E}^b_0 \ll \mathtt{C}.m(\bar{\mathtt{T}}')\mathtt{T}' \gg\ (\mathtt{E}^b_1, \dots, \mathtt{E}^b_n)}$$

$$\begin{array}{l}\mathit{type}(\mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \}) = \langle \mathtt{C}', \mathit{type}(\mathtt{MDS}^s)\rangle \\ \mathit{type}(\mathtt{MDS}^s) = \mathit{type}(\mathtt{MD}^s_1) \dots \mathit{type}(\mathtt{MD}^s_n) \\ \mathit{type}(\mathtt{MH}\ \{\ \mathtt{return\ E}^s;\ \}) = \mathit{type}(\mathtt{MH}) \\ \mathit{type}(\mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \dots, \mathtt{T}_n\ \mathtt{x}_n)) = \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1 \dots \mathtt{T}_n)\end{array}$$

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

Figure 3: Separate compilation

must be well-formed and the compilation of all the fragments $\mathtt{S}_i$ must be derivable in Γ'. The functions *name* and *type* extract from a class declaration the name and the type of the class, respectively.

The rule which defines the compilation of a single fragment for class C checks that all methods can be compiled, that the superclass $\mathtt{C}'$ can be safely extended with the methods declared in C and that there are no cycles involving C and $\mathtt{C}'$ (existence of the superclass is guaranteed by this last check). The function *type* extract from a method declaration the return type and the signature of the method.

The rule for compiling a method declaration checks that the body can be compiled and that the return type and the types of the parameters are defined.

The typing rule for method invocation checks that all sub-expressions can be compiled and that the method can be successfully resolved.

The entailment of type environments is given in Figure 4.

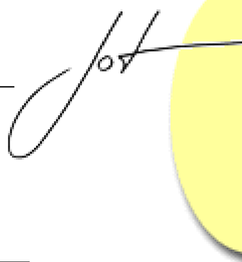
Rule (*well-def*) defines well-formed type environments, that is, consistent environments, and relies on the definition of well-formed standard environments. A standard type environment is well-formed if the inheritance relation is acyclic, all used classes are defined and the Java rules on overriding are respected (that is, a class cannot declare a method with the same name and parameter types of an inherited method and different return type). The rules for well-formed standard type environments can be found in Figure 5.

Rules (*empty*), (*conc*), and (*singleton*) ensure the basic properties expected by an entailment relation. Note that $\Gamma_1 \vdash \Gamma_2$ is provable only if Γ_1 is well-formed.

A class type C is defined in Γ if C is declared in Γ (*def*), whereas types Object and int are always defined (rules (*Object*) and (*int*)).

Rules for subtyping are standard.

Rule (*exact res*) deals with the situation where there exists a method in a superclass of the type of the receiver with parameters of the same type of the arguments; clearly, in this case invocation will be always resolved to that method, despite the other type assumptions contained in Γ. Rule (*match res*) requires more type assumptions than the previous rule in order to be applicable: the standard type assumptions of all superclasses of the type C of the receiver - C included - must be in Γ. Then it is possible to compute all applicable methods (function *applAll*) and to verify that each applicable method is matchable (function *matchAll*) as well. Finally, the set of all applicable methods must contain the most specific method (function *mostSpec*). A method $\mathtt{T}'\ \mathtt{m}'(\bar{\mathtt{T}}')$ is applicable to the invocation $\mathtt{C.m}(\bar{\mathtt{T}})$ in Γ if $\mathtt{m}' = \mathtt{m}$ and $\bar{\mathtt{T}} \leq \bar{\mathtt{T}}'$ can be proved in Γ, while it is matchable if $\mathtt{m} = \mathtt{m}'$ and there exists a type environment where $\bar{\mathtt{T}} \leq \bar{\mathtt{T}}'$ can be proved. The definitions of the auxiliary functions are shown in Figure 6. Rule (*complete res*) can be applied if the standard type assumptions of all superclasses of the types C and $\bar{\mathtt{T}}$ - C and $\bar{\mathtt{T}}$ included - of the receiver and of the arguments ($\Gamma \vdash \bar{\mathtt{T}} \Uparrow$) can be found in Γ; in this case the set of all applicable methods is the same in any environment entailed by Γ, therefore there is no need to compute



$$(\textit{well-def})\quad \frac{\Gamma_s \vdash \Gamma \quad \vdash \Gamma_s \diamond}{\vdash \Gamma \diamond}$$

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

$$(\textit{def})\quad \frac{\Gamma \vdash \mathtt{C}{:}{<}\mathtt{C}', \mathtt{MSS}{>}}{\Gamma \vdash \exists \mathtt{C}} \qquad (\textit{Object})\quad \frac{\vdash \Gamma \diamond}{\Gamma \vdash \exists \mathtt{Object}} \qquad (\textit{int})\quad \frac{\vdash \Gamma \diamond}{\Gamma \vdash \exists \mathtt{int}}$$

$$(\textit{refl})\quad \frac{\Gamma \vdash \exists \mathtt{T}}{\Gamma \vdash \mathtt{T} \leq \mathtt{T}} \qquad (\textit{trans})\quad \frac{\Gamma \vdash \mathtt{C}_1 \leq \mathtt{C}_2 \quad \Gamma \vdash \mathtt{C}_2{:}{<}\mathtt{C}_3, \mathtt{MSS}{>}}{\Gamma \vdash \mathtt{C}_1 \leq \mathtt{C}_3}$$

$$(\textit{top})\quad \frac{\Gamma \vdash \exists \mathtt{C}}{\Gamma \vdash \mathtt{C} \leq \mathtt{Object}} \qquad (\textit{vector})\quad \frac{\Gamma \vdash \mathtt{T}_i \leq \mathtt{T}'_i \; \forall\, i \in 1, \ldots, n}{\Gamma \vdash \mathtt{T}_1 \ldots \mathtt{T}_n \leq \mathtt{T}'_1 \ldots \mathtt{T}'_n}$$

$$(\textit{exact res})\quad \frac{\Gamma \vdash \mathtt{C}'{:}{<}\mathtt{C}'', \mathtt{MSS}{>} \quad \Gamma \vdash \mathtt{C} \leq \mathtt{C}'}{\Gamma \vdash \mathtt{C.m}(\bar{\mathtt{T}}) \stackrel{res}{\rightarrow} {<}\bar{\mathtt{T}}, \mathtt{T}{>}} \qquad \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{MSS}$$

$$(\textit{match res})\quad \frac{\begin{array}{l} \textit{applAll}(\Gamma, \mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu s \\ \textit{matchAll}(\Gamma, \mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu s \\ \textit{mostSpec}(\Gamma, \mu s) = {<}\bar{\mathtt{T}}', \mathtt{T}'{>} \end{array}}{\Gamma \vdash \mathtt{C.m}(\bar{\mathtt{T}}) \stackrel{res}{\rightarrow} {<}\bar{\mathtt{T}}', \mathtt{T}'{>}}$$

$$(\textit{complete res})\quad \frac{\begin{array}{l} \textit{applAll}(\Gamma, \mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu s \\ \Gamma \vdash \bar{\mathtt{T}} \Uparrow \\ \textit{mostSpec}(\Gamma, \mu s) = {<}\bar{\mathtt{T}}', \mathtt{T}'{>} \end{array}}{\Gamma \vdash \mathtt{C.m}(\bar{\mathtt{T}}) \stackrel{res}{\rightarrow} {<}\bar{\mathtt{T}}', \mathtt{T}'{>}}$$

$$(\smiley\, \textit{obj})\quad \frac{\vdash \Gamma \diamond}{\Gamma \vdash \mathtt{Object} \smiley \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})}$$

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

$$(\textit{not sub})\quad \frac{\Gamma \vdash \mathtt{C} \Uparrow \quad \mathtt{C}' \notin \textit{supertypes}(\Gamma, \mathtt{C})}{\Gamma \vdash \mathtt{C} \not\leq \mathtt{C}'}$$

Figure 4: Type environments entailment

$$\frac{\Gamma_s \vdash \Gamma \quad \vdash \Gamma_s\diamond}{\vdash \Gamma\diamond} \qquad \frac{\Gamma_s \vdash \Gamma_s\diamond}{\vdash \Gamma_s\diamond} \qquad \frac{}{\Gamma_s \vdash \Lambda\diamond} \qquad \frac{}{\Gamma_s \vdash \mathtt{T}\diamond_{\mathtt{type}}} \quad \begin{array}{l}\mathtt{T} = \mathtt{int} \vee \\ \mathtt{T} = \mathtt{Object} \vee \\ \mathtt{T} \in \mathit{def}(\Gamma_s)\end{array}$$

$$\frac{\Gamma_s \vdash \mathtt{T}_i \diamond_{\mathtt{type}} \ \forall i \in 0,\ldots,n}{\Gamma_s \vdash \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1..\mathtt{T}_n)\diamond_{\mathtt{MS}}} \qquad \frac{\Gamma_s \vdash \mathtt{MS}_i \diamond_{\mathtt{MS}} \ \forall i \in 1,\ldots,n}{\Gamma_s \vdash \mathtt{MS}_1 \ldots \mathtt{MS}_n\diamond_{\mathtt{MSS}}}$$

$$\frac{}{\Gamma_s \vdash \mathtt{Object}{:}\emptyset} \qquad \frac{\begin{array}{c}\Gamma_s \vdash \mathtt{C}'{:}\mathtt{MSS}' \\ \Gamma_s \vdash \mathtt{MSS}\diamond_{\mathtt{MSS}}\end{array}}{\Gamma_s \vdash \mathtt{C}{:}\mathtt{MSS} \cup \mathtt{MSS}'} \quad \begin{array}{l}\Gamma_s(\mathtt{C}) = \langle\mathtt{C}', \mathtt{MSS}\rangle \\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{MSS}, \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{MSS}' \implies \mathtt{T} = \mathtt{T}' \\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}), \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{MSS} \implies \mathtt{T} = \mathtt{T}'\end{array}$$

$$\frac{\Gamma_s \vdash \Gamma_s^1\diamond \quad \Gamma_s \vdash \mathtt{C}{:}\tau}{\Gamma_s \vdash \Gamma_s^1, \mathtt{C}{:}\tau\diamond} \qquad \mathtt{C}{:}\tau' \in \Gamma_s^1 \implies \tau' = \tau$$

Figure 5: Well-formed standard type environments

the matchable methods.

Rule (☺*obj*) states that `Object` can be safely extended by any method[4] (that is, without breaking the Java rule on overriding); in all other cases, if a class $\mathtt{C}'$ can be safely extended by a method $\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})$ then any direct subclass `C` of $\mathtt{C}'$ can be safely extended by the same method, providing that `C` does not contain a method with the same name, same types as parameters but *different* return type (rule (☺*down*)).

Finally, rule (*not sub*) is applicable only if Γ contains the standard type assumptions of all supertypes of `C` - `C` included ($\Gamma \vdash \mathtt{C} \Uparrow$) - and $\mathtt{C}'$ is not in the set of such supertypes (function *supertypes*).

Properties

Since the type system presented here is just a simplification of that defined by Ancona and Zucca [4], the same properties can be proved and all proofs can be immediately adapted.

The two main properties we are interested in for selective recompilation are soundness of entailment, and existence of principal typings.

Soundness of the entailment corresponds to the following claim: if $\Gamma_1 \vdash \Gamma_2$, then for all `P`, $\bar{\tau}$, $\bar{\mathtt{B}}$, if $\Gamma_2 \vdash \mathtt{P}{:}\bar{\tau} \leadsto \bar{\mathtt{B}}$ holds, then $\Gamma_1 \vdash \mathtt{P}{:}\bar{\tau} \leadsto \bar{\mathtt{B}}$ holds as well. Since entailment is used by our algorithm to check for recompilation (see also Section 4), this property guarantees that the algorithm always forces recompilations which are really needed, that is, which either fail or generate different binaries.

Principality says that if a program `P` can be compiled to a binary $\bar{\mathtt{B}}$, then there

[4]For simplicity, we ignore all the predefined methods of `Object`, defined in 4.3.2 of [8].

$$\frac{}{\Gamma \vdash \texttt{Object} \Uparrow} \qquad \frac{\Gamma \vdash \texttt{C}' \Uparrow}{\Gamma \vdash \texttt{C} \Uparrow} \quad \Gamma(\texttt{C}) = \texttt{<}\texttt{C}', _\texttt{>} \qquad \frac{\Gamma \vdash \texttt{T}_1 \Uparrow \ldots \Gamma \vdash \texttt{T}_n \Uparrow}{\Gamma \vdash \texttt{T}_1 \ldots \texttt{T}_n \Uparrow}$$

$$\Gamma \vdash \texttt{int} \Uparrow$$

$$supertypes(\Gamma, \texttt{C}) = \begin{cases} \{\texttt{Object}\} & \text{if } \texttt{C} = \texttt{Object} \\ \{\texttt{C}\} \cup supertypes(\Gamma, \texttt{C}') & \text{if } \Gamma(\texttt{C}) = \texttt{<}\texttt{C}', _\texttt{>} \\ \bot & \text{otherwise} \end{cases}$$

$$appl(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \begin{cases} \{\texttt{<}\texttt{C}, \bar{\texttt{T}}', \texttt{T}\texttt{>} \mid \texttt{T}\ \texttt{m}(\bar{\texttt{T}}') \in \texttt{MSS}, \Gamma \vdash \bar{\texttt{T}} \leq \bar{\texttt{T}}'\ \} & \text{if } \Gamma(\texttt{C}) = \texttt{<}_, \texttt{MSS}\texttt{>} \\ \bot & \text{otherwise} \end{cases}$$

$$applAll(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \begin{cases} \emptyset & \text{if } \texttt{C} = \texttt{Object} \\ \mu s_1 \cup \mu s_2 & \text{if } \ \Gamma(\texttt{C}) = \texttt{<}\texttt{C}', _\texttt{>} \\ & \quad appl(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \mu s_1, \\ & \quad applAll(\Gamma, \texttt{C}', \texttt{m}, \bar{\texttt{T}}) = \mu s_2 \\ \bot & \text{otherwise} \end{cases}$$

$$match(\texttt{T}_1 \ldots \texttt{T}_m, \texttt{T}'_1 \ldots \texttt{T}'_n) \iff m = n \wedge \forall i \in 1..n\ (\texttt{T}_i = \texttt{int}) \iff (\texttt{T}'_i = \texttt{int})$$

$$match(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \begin{cases} \{\texttt{<}\texttt{C}, \bar{\texttt{T}}', \texttt{T}\texttt{>} \mid \texttt{T}\ \texttt{m}(\bar{\texttt{T}}') \in \texttt{MSS}, match(\bar{\texttt{T}}, \bar{\texttt{T}}')\} & \text{if } \Gamma(\texttt{C}) = \texttt{<}_, \texttt{MSS}\texttt{>} \\ \bot & \text{otherwise} \end{cases}$$

$$matchAll(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \begin{cases} \emptyset & \text{if } \texttt{C} = \texttt{Object} \\ \mu s_1 \cup \mu s_2 & \text{if } \ \Gamma(\texttt{C}) = \texttt{<}\texttt{C}', _\texttt{>} \\ & \quad match(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \mu s_1, \\ & \quad matchAll(\Gamma, \texttt{C}', \texttt{m}, \bar{\texttt{T}}) = \mu s_2 \\ \bot & \text{otherwise} \end{cases}$$

$$mostSpec(\Gamma, \mu s) = \begin{cases} \texttt{<}\texttt{C}, \bar{\texttt{T}}, \texttt{T}\texttt{>} & \text{if } \ \texttt{<}\texttt{C}, \bar{\texttt{T}}, \texttt{T}\texttt{>} \in \mu s \text{ and} \\ & \quad \Gamma \vdash \texttt{C} \leq \texttt{C}', \\ & \quad \Gamma \vdash \bar{\texttt{T}} \leq \bar{\texttt{T}}' \text{ for all } \texttt{<}\texttt{C}', \bar{\texttt{T}}', \texttt{T}'\texttt{>} \in \mu s \\ \bot & \text{otherwise} \end{cases}$$

Figure 6: Auxiliary judgment and functions

exists a minimal Γ s.t. $\Gamma \vdash \texttt{P}{:}\bar{\tau} \leadsto \bar{\texttt{B}}$ holds, where "minimal" means that for all Γ', if $\Gamma' \vdash \texttt{P}{:}\bar{\tau} \leadsto \bar{\texttt{B}}$ holds, then $\Gamma' \vdash \Gamma$ holds as well. From the point of view of smarter recompilation, this means that our algorithm never forces useless recompilations which would generate the same binary.

4 SELECTIVE RECOMPILATION

Why making Java selective recompilation smarter?

Although several papers have been written on the subject of selective recompilation (see Section 5), Dmitriev's approach [6] is the only other Java specific proposal we are aware of. 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 that 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. Even though this approach has the advantage of being the only one to be both well documented and fully implemented, unfortunately, it is not based on a theoretical foundation, as pointed out by the author himself. As a consequence, no clue of the fact that the algorithm is correct is provided, and there is no guarantee that *Javamake* always forces the recompilation of a class when needed for ensuring the consistency of the project. Furthermore, *Javamake* cannot avoid a considerable amount of unnecessary recompilations.

Our approach

Our algorithm, on the other hand, is defined on top of a good theoretical basis. Soundness of entailment ensures that the algorithm is correct, in the sense that all recompilations needed for maintaining the project consistent are always forced. The principal typings property guarantees that the algorithm is smarter, since it never forces useless recompilations: whenever a class is recompiled, either recompilation fails, or generates a different binary.

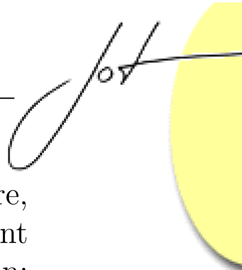
One of the key point is that, during compilation of a class `C` to a binary `B`, it is possible to infer the minimal type assumptions Γ needed for compiling `C` to `B` [11, 4]. If the new standard environment obtained after some modification to other classes still entails Γ, then there is no need to recompile `C`. Before sketching the whole algorithm, 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 minimal environment for `H`:

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



If we add a method `f(int)` in `P`, then class `H` still invokes the same method as before, because the new method is not even applicable to the invocation 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 binary of `H`) still entails Γ_l therefore there is no need to recompile `H`. For instance, the reader can verify that $\texttt{P.f(H)} \overset{\texttt{res}}{\rightarrow} \texttt{<Object, int>}$ 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.

Description of the algorithm

For sake of simplicity we assume that all project files reside in the same directory and contain just one class declaration each. The input of the algorithm is the project directory. The algorithm computes the set S of source project files which need to be recompiled, and then typechecks all files in S and, if no error is detected, generates for each file in S the corresponding binary file (as usual, in a `.class` file) and the set of minimal type assumptions needed for generating that binary (in a separate `.mta` file).

The algorithm consists of the following three main components: *extract*, *select*, and *typecheck*.

Component *extract* generates a well-formed standard type environment by extracting the global types of all classes from the sources and binaries of the project. If a class has an up-to-date binary file, then its type information is extracted from there, otherwise it is extracted from the source file. If the extracted type environment is not well-formed, then *extract* fails.

Component *select* produces the set of files that need to be recompiled, and its behavior is specified by the following pseudo-code:

select $(\Gamma)\{$
let $S = \{C.java \mid$ the binary of $C.java$ is either absent or not up-to-date$\}$
let $T = \{C.java \mid C.java$ is a project file$\} \setminus S$
for all $C.java \in T$ if not $entails(\Gamma, C.mta)$ then $S = S \cup \{C.java\}$
return S $\}$

We assume the standard Java name conventions for files; binary files are not up-to-date if the corresponding source file has been changed but not yet recompiled (hence

[5] Version 1.3.1, the latest available when we run this test.

it has a more recent time-stamp). Since `.class` and `.mta` files are simultaneously generated by *typecheck*, for simplicity we assume that they are always both present or absent, and that they always have the same time-stamp. The actual parameter of *select* is the standard type environment returned by *extract*, whereas *entails* is the subcomponent at the heart of *select* which implements the entailment relation. It takes a well-formed standard type environment Γ, a local type environment Γ_l, and returns *true* if and only if $\Gamma \vdash \Gamma_l$ holds.

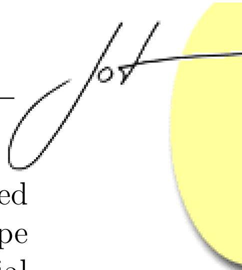
```
entails (Γ,Γ_l){
 for all ta in Γ_l if not holds (Γ,ta) return false
 return true }
```

The subcomponent *holds* just performs some of the standard checks done by conventional Java compilers:

```
holds (Γ,ta){
 switch(ta){
  case ∃T:
   return T == int or T defined in Γ
  case C.m(T̄) -res-> <T̄', T>:
   return T m(T̄') is the most specific method applicable to the call C.m(T̄)
 ... } }
```

Finally, the main component *typecheck* takes a well-formed standard type environment (produced by *extract*) and a source file *C.java*, and either fails (if some error is detected), or produces the file *C.mta* of minimal type assumptions and the binary file *C.class*. The most interesting subcomponent of *typecheck* is the one which typechecks expressions and, in case of success, returns the type and the minimal assumptions of the expression (for sake of simplicity we omit bytecode generation).

```
typecheckExp (Γ,e){
 switch(e){
  case e_0.m(e_1, ..., e_n):
   for i=0...n let (T_i, Γ_i) = typecheckExp(Γ, e_i)
   if ∃T', T'_1, ..., T'_n s.t. holds(Γ, T_0.m(T_1, ..., T_n) -res-> <T̄', T'>)
   then return (T', (Γ_0, ..., Γ_n, T_0.m(T_1, ..., T_n) -res-> <T̄', T'>))
   else fails
 ... } }
```



The most interesting case is method call. First, all sub-expressions are typechecked from left to right; if all sub-expressions are successfully typechecked, then the type $\mathtt{T}_i$ and minimal assumptions Γ_i of each e_i have been computed. The existential condition of the conditional statement can be implemented in a straightforward way, since corresponds to method resolution performed by conventional Java compilers. Finally, if resolution succeeds, then the whole typechecking succeeds as well, and the type of the expression is the return type $\mathtt{T}'$ of the most specific method, while the minimal assumptions are the collection of all minimal assumptions for sub-expressions plus the additional assumption $\mathtt{T}_0.\mathtt{m}(\mathtt{T}_1, \ldots, \mathtt{T}_n) \stackrel{res}{\rightarrow} <\bar{\mathtt{T}}', \mathtt{T}'>$.

The cost of selective recompilation

The real goal of smart recompilation is not avoiding useless recompilations, but rather making recompilation as fast as possible, while retaining consistency of the project. Clearly, avoiding useless recompilations can be an effective way for speeding up recompilation, but only if the costs of actually doing a recompilation outweigh the costs for checking for recompilation.

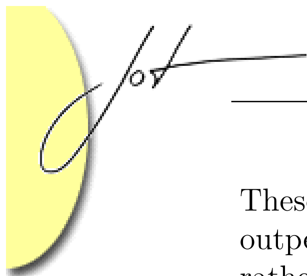
Notice that in the extreme case when all sources have to be recompiled, every approach different than blindly recompiling all sources is slower, due to the overhead necessary to detect which files have to be recompiled. Anyway, modifications to large projects are likely to affect only a small subset of the files.

So, in evaluating the cost of recompiling a source S, let us compare the direct recompilation of S with respect to checking whether the type assumptions of S still hold, followed by the recompilation of S if needed. An important thing to note is that there is some overlapping in what a compiler needs to do in order to compile a given source S, and in checking whether the type assumptions for S still hold in a new environment.

After some changes have been made, only two scenarios are possible:

- *Scenario n.1: the source* S *needs not to be recompiled, as all its assumptions are satisfied in the new environment.* In order to typecheck S the compiler must check exactly the same assumptions plus, of course, parsing the source and generating the bytecode. So, in this case, we can easily conclude that running the compiler on S is more expensive than discovering that it does not need to be recompiled with our method.

- *Scenario n.2: the source* S *needs to be recompiled, as at least one of its assumptions is not satisfied by the new environment.* Obviously, in this case the cost of checking for recompilation is an overhead. However, such an overhead is small since all assumptions which have been already successfully checked can be cached in order to be reused during recompilation. As a consequence, the overhead reduces to the time needed for checking the first single assumption that does not hold and which triggers the recompilation.



These last considerations strongly suggest that in the average case our strategy outperforms the blind recompilation. Moreover, the overhead in the worst cases is rather small.

For what concerns space costs there is always an overhead, which, however, can be kept small. However, type assumptions generated by our algorithm are stored in separate `.mta` files, in order to prevent `.class` from growing. This is important, as binary fragments are usually transmitted through the network.

In a Java implementation of our algorithm, type assumptions can be implemented by objects which are stored and retrieved using standard Java serialization mechanism. As this standard mechanism is known to be rather inefficient, the size of `.mta` files could be easily reduced by using optimized serialization mechanisms [14]. Moreover, most of the type information stored in a `.mta` file is a replication of what can be found in the corresponding `.class` file, therefore a considerable amount of space could be saved by using pointers to the `.class` files.

For instance, the space cost for storing the type assumption $\mathtt{T.m}(\bar{\mathtt{T}}) \stackrel{res}{\rightarrow} <\bar{\mathtt{T}}', \mathtt{T}'>$ for a method invocation amounts to the cost required for storing the static types $\bar{\mathtt{T}}$ of the arguments, plus a pointer to an entry of the constant pool in the `.class` file containing all the other needed information (the static type of the receiver, the name, return type and parameter types). Finally, note that different expressions can generate the same assumptions which, obviously, can be stored just once.

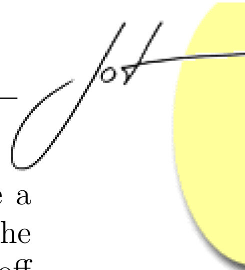
5 CONCLUSION AND OTHER RELATED WORK

We have presented an algorithm for smarter recompilation of a small subset of Java, based on a formal type system enjoying the principal typings property.

The properties of the type system guarantee that the algorithm is smarter, in the sense that it always forces recompilations which are really needed, but never forces useless recompilations which would lead to the same binary.

We have compared our algorithm with *Javamake* and shown that our approach is cleaner (since it is based on theoretical foundations) and leads to a smarter recompilation strategy. However, much work still remains to be done, since *Javamake* is a real tool running for the entire Java language (and not just a small subset). We are currently implementing in Java the algorithm we have presented in this paper and, in the meantime, we are studying how our approach can be scaled to the entire Java language [11, 12] by directly modifying the SDK compiler. Since some Java features badly interacts with separate compilation, we expect our tool for selective recompilation of Java to be still smarter than *Javamake*, but less smart that the algorithm presented here, in order to keep the costs of checking for recompilation less expensive than the costs of actually doing a recompilation.

Selective recompilation [1] is an issue that has been deeply studied for programming languages, in order to reduce rebuilding time due to source modifications.



Besides the paper on *Javamake* [6] already discussed in Section 4, there are a number of papers on selective recompilation for several languages. According to the classification given in [1], [9] 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. 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.

6 ACKNOWLEDGEMENTS

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

Partially supported by Dynamic Assembly, Reconfiguration and Type-checking - EC project IST-2001-33477, and APPSEM II - Thematic network IST-2001-38957.

REFERENCES

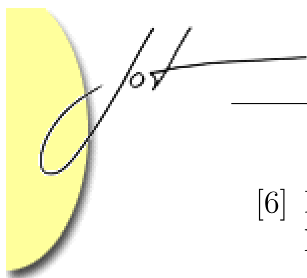
[1] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

[2] D. Ancona, 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.

[3] 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.

[4] D. Ancona and E. Zucca. Principal typings for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2004*, pages 306–317. ACM Press, January 2004.

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



[6] M. Dmitriev. Language-specific make technology for the Java programming language. *ACM SIGPLAN Notices*, 37(11):373–385, 2002.

[7] 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.

[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] 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.

[11] G. Lagorio. Towards a smart compilation manager for Java. In Blundo and Laneve, editors, *Italian Conf. on Theoretical Computer Science 2003*, number 2841 in Lecture Notes in Computer Science, pages 302–315. Springer, October 2003.

[12] G. Lagorio. Another step towards a smart compilation manager for Java. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *ACM Symp. on Applied Computing (SAC 2004), Special Track on Object-Oriented Programming Languages and Systems*, pages 1275–1280. ACM Press, March 2004.

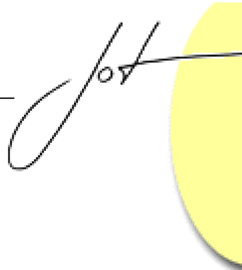
[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Second edition, 1999.

[14] Michael Philippsen and Bernhard Haumacher. More efficient object serialization. In *IPPS/SPDP Workshops*, pages 718–732, 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.



ABOUT THE AUTHORS

Davide Ancona took a Ph.D. in Computer Science at the University of Pisa on 1998, and since 2000 is assistant professor at the department of Computer Science of the University of Genova (DISI). His research interests are in object oriented programming, module systems, and type systems. He can be reached at davide@disi.unige.it. See also http://www.disi.unige.it/person/AnconaD/.

Giovanni Lagorio is a PhD student at the department of Computer Science of the University of Genova (DISI), University of Genova. His research interests are in the area of programming languages; in particular, design and foundations of modular and object-oriented languages and systems. He can be reached at lagorio@disi.unige.it. See also http://www.disi.unige.it/person/LagorioG/.