Towards a Smart Compilation Manager for Java* (Extended Abstract)

Giovanni Lagorio

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

Abstract. It is often infeasible to recompile all the sources an application consists of each time a change is made. Yet, a recompilation strategy which does not guarantee the same outcome of an entire recompilation is not useful: why wasting time in debugging a program (a set of `.class` files in the Java case) which might behave *differently* from the program obtained recompiling all the sources from scratch?
We say that a compilation strategy is *sound* if it recompiles, besides the changed sources, all the unchanged sources whose new binary, produced by the overall recompilation, would differ from the existing one (if any) *and* all the sources for which the recompilation would be undefined: indeed, when the entire compilation fails, so should do the partial recompilation.
We say that a compilation strategy is *minimal* if it never recompiles an unchanged source whose new binary would be equal to the existing one.
In this paper we present a compilation strategy for a substantial subset of Java which is proved to be *sound* and *minimal*.

1 Introduction

When dealing with large applications it is infeasible to recompile all the sources each time a change is made. Of course, separate compilation is the answer to such a problem, but a key point has to be considered: in addition to the modified sources, which files have to be recompiled as well? A recompilation strategy which does not guarantee the same outcome of an entire recompilation is not useful: why wasting time in debugging a program (a set of `.class` files in the Java case) which might behave *differently* from the program obtained recompiling all the sources from scratch? It is known that, using the most common Java compilers (including the standard one), a program which is the result of a successful recompilation may throw linking related exceptions at runtime even if these errors could have been detected at compile time [3]. This is not in contrast

* 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.

with the soundness for Java investigated by other papers [6, 7] because they did not take separate compilation into account.

Two contrasting requirements have to be considered: on the one hand recompilations can be rather expensive (in time), hence they should be avoided when possible. More precisely, they are useless when the recompilation of an unchanged (with respect to the previous compilation) source fragment `S`, whose corresponding binary fragment `B` is already present, would produce a binary equal to `B`. On the other hand, a recompilation strategy which saves time not recompiling a fragment `S`, with a corresponding binary fragment `B`, whose recompilation would produce a new binary `B′` different from `B`, could cost a lot of wasted time in debugging an *inconsistent* application, that is, an application that *cannot* be rebuilt by recompiling all the sources.

Albeit some Java IDEs support smart or incremental compilation, to our knowledge there are no publications which explain in detail the inner working of such recompilation strategies. In this paper we chose to analyze Java, a mainstream language, and model its peculiar features, because our final goal is to implement a compilation manager, for a widespread language, whose correctness and minimality (in the sense explained below) can be formally proved.

We say that a compilation strategy is *sound* if it recompiles all the changed sources and the unchanged sources whose new binary, produced by the overall recompilation, would differ from the existing one (if any) *and* all the unchanged sources for which the recompilation would be undefined. This latter requirement is very important: indeed, when the entire recompilation is not defined, so should be the partial recompilation.

Of course, a strategy which recompiles all the sources each time a change is made is trivially sound and, obviously, totally useless in practice. We say that a compilation strategy is *minimal* if it never recompiles an unchanged fragment whose new binary would be equal to the existing one.

In this paper we present a compilation strategy for a substantial subset of Java which is proved to be *sound* and *minimal*.

Section 2 presents the formal framework, Section 3 explains the ideas behind our compilation strategy and Section 4 discusses related and future work.

2 Formalization

2.1 The Language

In this paper we model a substantial subset of Java at both source (Figure 1) and binary (Figure 2) level. Our model of bytecode is rather abstract: it is basically source code enriched with some annotations (discussed below). However, the same source level expression can be compiled to different binary level expressions, as it happens in Java. For instance, a method invocation `x.m()` can be translated to a virtual method invocation or an interface method invocation depending on the static type of `x`.

With the exception of arrays and inner-classes we model all the major features of Java: classes (including abstract classes), interfaces, primitive types, access

$$
\begin{array}{rcl}
\texttt{S} & ::= & \texttt{AM CK class C extends C}' \texttt{ implements } \texttt{I}_1 \ldots \texttt{I}_n \ \{\ \texttt{KDS}^s\ \texttt{FDS}^s\ \texttt{MDS}^s\ \} \mid \\
 & & \texttt{AM interface I extends } \texttt{I}_1 \ldots \texttt{I}_n \ \{\ \texttt{FDS}^s\ \texttt{MDS}^s\ \} \\
\texttt{AM} & ::= & \texttt{public} \mid \texttt{protected} \mid \epsilon \mid \texttt{private} \\
\texttt{CK} & ::= & \epsilon \mid \texttt{abstract} \\
\texttt{KDS}^s & ::= & \texttt{KD}_1^s \ldots\ \texttt{KD}_n^s \\
\texttt{FDS}^s & ::= & \texttt{FD}_1^s \ldots\ \texttt{FD}_n^s \\
\texttt{MDS}^s & ::= & \texttt{MD}_1^s \ldots\ \texttt{MD}_n^s \\
\texttt{KD}^s & ::= & \texttt{AM KH} \ \{\ \texttt{super}(\texttt{E}_1^s, \ldots, \texttt{E}_n^s);\ \texttt{STMTS}^s\ \} \\
\texttt{FD}^s & ::= & \texttt{AM FINAL FK T f} = \texttt{E}^s\ ; \\
\texttt{MD}^s & ::= & \texttt{AM MK MH} \ \{\ \texttt{STMTS}^s \texttt{ return E}^s;\ \} \mid \texttt{AM abstract MH}\ ; \\
\texttt{KH} & ::= & (\texttt{T}_1\ \texttt{x}_1, \ldots, \texttt{T}_n\ \texttt{x}_n) \texttt{ throws ES} \\
\texttt{MK} & ::= & \epsilon \mid \texttt{static} \mid \texttt{abstract} \\
\texttt{MH} & ::= & \texttt{T m}(\texttt{T}_1\ \texttt{x}_1, \ldots, \texttt{T}_n\ \texttt{x}_n) \texttt{ throws ES} \\
\texttt{FINAL} & ::= & \epsilon \mid \texttt{final} \\
\texttt{FK} & ::= & \epsilon \mid \texttt{static} \\
\texttt{T} & ::= & \texttt{RT} \mid \texttt{int} \mid \texttt{bool} \\
\texttt{RT} & ::= & \texttt{C} \mid \texttt{I} \\
\texttt{ES} & ::= & \texttt{C}_1, \ldots, \texttt{C}_n \\
\texttt{E}^s & ::= & \texttt{PRIMARY}^s \mid \texttt{ASSIGN}^s \mid \texttt{N} \mid \texttt{true} \mid \texttt{false} \\
\texttt{PRIMARY}^s & ::= & \texttt{null} \mid \texttt{this} \mid \texttt{NEW}^s \mid \texttt{x} \mid \texttt{INVOKE}^s \mid \texttt{super.f} \mid \texttt{PRIMARY}^s\texttt{.f} \mid \texttt{RT.f} \\
\texttt{ASSIGN}^s & ::= & \texttt{x} = \texttt{E}^s \mid \texttt{PRIMARY}^s\texttt{.f} = \texttt{E}^s \mid \texttt{super.f} = \texttt{E}^s \mid \texttt{RT.f} = \texttt{E}^s \\
\texttt{NEW}^s & ::= & \texttt{new C}(\texttt{E}_1^s, \ldots, \texttt{E}_n^s) \\
\texttt{INVOKE}^s & ::= & \texttt{PRIMARY}^s\texttt{.m}(\texttt{E}_1^s, \ldots, \texttt{E}_n^s) \mid \texttt{super.m}(\texttt{E}_1^s, \ldots, \texttt{E}_n^s) \mid \texttt{RT.m}(\texttt{E}_1^s, \ldots, \texttt{E}_n^s) \\
\texttt{STMTS}^s & ::= & \texttt{STMT}_1^s \ldots \texttt{STMT}_n^s \\
\texttt{STMT}^s & ::= & \{\texttt{STMTS}^s\} \mid \texttt{SE}^s\ ; \mid \texttt{if } (\texttt{E}^s)\ \texttt{STMT}_1^s \texttt{ else } \texttt{STMT}_2^s \mid \texttt{while } (\texttt{E}^s)\ \texttt{STMT}^s \mid \\
 & & \texttt{try } \{\texttt{STMTS}^s\ \}\ \texttt{CATCHES}^s\ \texttt{finally } \{\ \texttt{STMTS}_1^s\ \} \mid \texttt{throw E}^s \\
\texttt{SE}^s & ::= & \texttt{ASSIGN}^s \mid \texttt{INVOKE}^s \mid \texttt{NEW}^s \\
\texttt{CATCHES}^s & ::= & \texttt{CATCH}_1^s \ldots \texttt{CATCH}_n^s \\
\texttt{CATCH}^s & ::= & \texttt{catch } (\texttt{C x}) \ \{\ \texttt{STMTS}^s\ \}
\end{array}
$$

Assumptions:

- interface names in S are distinct;
- field names in $\texttt{FDS}^s$ are distinct;
- method/constructor signatures in $\texttt{MDS}^s$/$\texttt{KDS}^s$ are distinct;
- parameter and exception names in both KH and MH are distinct;
- class names in $\texttt{CATCHES}^s$ are distinct.

Fig. 1. Syntax - Sources

modifiers (including packages, but without the `import` directive), constructors, (instance/static) fields (both in classes and interfaces), (instance/static/abstract) methods, super field accesses and method invocations, exceptions. The treatment of arrays and inner-classes would complicate the model without apparently giving further insights.

Figure 1 gives the syntax of the language. A source fragment `S` can be a class declaration or an interface declaration. In the former case it consists of: an access modifier `AM`, a class kind `CK` (either ϵ or `abstract`), the name of the class, the name of the superclass, the list of the implemented interfaces and the declaration of constructors, fields and methods. Analogously, an interface declaration consists of an access modifier, the name of the interface, the name of the superinterfaces and the declaration of fields and methods.

A constructor declaration $\texttt{KD}^s$ consists of an access modifier `AM`, a constructor header `KH`, the invocation of a superclass's constructor[1] and a sequence of statements $\texttt{STMTS}^s$. A constructor header consists of the sequence of parameters and the exception specification `ES`. In this paper we assume for simplicity that any class can be an exception, that is, we do not model the predefined class `Throwable`. So, an exception specification is just a sequence of class names.

A field declaration $\texttt{FD}^s$ consists of an access modifier `AM`, an optional modifier `FINAL`, a field kind `FK`, a type `T`, the name of the field `f` and the initialization expression $\texttt{E}^s$.

A method declaration $\texttt{MD}^s$ can be either concrete or abstract. In the former case it consists of an access modifier `AM`, a method kind `MK`, a method header `MH`, a sequence of statements $\texttt{STMTS}^s$ and a return expression $\texttt{E}^s$. In the latter case it just consists of an access modifier `AM`, the keyword `abstract` and a method header. A type `T` can be a reference type `RT` or a primitive type (`int` or `bool`). We distinguish between class names `C` and interface names `I` for clarity only, even though they actually range over the same set of names.

An expression $\texttt{E}^s$ can be: a primary expression, an assignment expression, an integer literal `N` or a boolean literal. Some expressions, $\texttt{SE}^s$, can be used as statements; they are: assignment $\texttt{ASSIGN}^s$, method invocation $\texttt{INVOKE}^s$ and instance creation $\texttt{NEW}^s$.

While Java permits accessing a static member of a class/interface `RT` via both the type name `RT` or any expression which has static type `RT`, here we allow only the former kind of access (because allowing both kinds of access would require additional, uninteresting, typing rules).

Figure 2 gives the syntax of the binary language. As already said, it mostly mimics the source language, except for it is enriched with some annotations enclosed between "$\ll$" and "$\gg$".

For example, the instance creation expression $\texttt{NEW}^s$ is translated to $\texttt{NEW}^b$, which contains, as annotation, the tuple of types describing the constructor which has been found as most specific at compile time. Analogously, method

[1] Invocations of a constructor of the same class (using `this`) are not considered since they are simply syntactic shortcuts (recursive invocations are not allowed - see 8.8.5 of [8]).

$$
\begin{array}{rcl}
\texttt{B} & ::= & \texttt{AM CK class C extends C}' \texttt{ implements } \texttt{I}_1 \ldots \texttt{I}_n \ \{\ \texttt{KDS}^b\ \texttt{FDS}^b\ \texttt{MDS}^b\ \} \mid \\
& & \texttt{AM interface I extends } \texttt{I}_1 \ldots \texttt{I}_n \ \{\ \texttt{FDS}^b\ \texttt{MDS}^b\ \} \\
\texttt{KDS}^b & ::= & \texttt{KD}_1^b \ldots\ \texttt{KD}_n^b \\
\texttt{MDS}^b & ::= & \texttt{MD}_1^b \ldots\ \texttt{MD}_n^b \\
\texttt{FDS}^b & ::= & \texttt{FD}_1^b \ldots\ \texttt{FD}_n^b \\
\texttt{KD}^b & ::= & \texttt{AM KH}\ \{\ \texttt{super}(\texttt{E}_1^b, \ldots, \texttt{E}_n^b) \ll \bar{\texttt{T}} \gg_{\texttt{c}};\ \texttt{STMTS}^b\ \} \\
\texttt{FD}^b & ::= & \texttt{AM FINAL FK T f} = \texttt{E}^b \\
\texttt{MD}^b & ::= & \texttt{AM MK MH}\ \{\ \texttt{STMTS}^b\ \texttt{return E}^b;\ \} \mid \texttt{AM abstract MH ;} \\
\texttt{E}^b & ::= & \texttt{PRIMARY}^b \mid \texttt{ASSIGN}^b \mid \texttt{N} \mid \texttt{true} \mid \texttt{false} \\
\texttt{PRIMARY}^b & ::= & \texttt{null} \mid \texttt{this} \mid \texttt{NEW}^b \mid \texttt{x} \mid \texttt{INVOKE}^b \\
& & \texttt{PRIMARY}^b\texttt{.} \ll \texttt{C.T} \gg_{\texttt{if}} \texttt{f} \mid \ll \texttt{RT.T} \gg_{\texttt{sf}} \texttt{f} \\
\texttt{ASSIGN}^b & ::= & \texttt{x} = \texttt{E}^b \mid \texttt{PRIMARY}^b\texttt{.} \ll \texttt{C.T} \gg_{\texttt{if}} \texttt{f} = \texttt{E}^b \mid \ll \texttt{RT.T} \gg_{\texttt{sf}} \texttt{f} = \texttt{E}^b \\
\texttt{NEW}^b & ::= & \texttt{new C} \ll \bar{\texttt{T}} \gg_{\texttt{c}} (\texttt{E}_1^b, \ldots, \texttt{E}_n^b) \\
\texttt{INVOKE}^b & ::= & \texttt{PRIMARY}^b\texttt{.} \ll \texttt{C.m}(\bar{\texttt{T}})\texttt{T} \gg_{\texttt{vrt}} \texttt{m}(\texttt{E}_1^b, \ldots, \texttt{E}_n^b) \mid \texttt{this.} \ll \texttt{C.m}(\bar{\texttt{T}})\texttt{T} \gg_{\texttt{spr}} \texttt{m}(\texttt{E}_1^b, \ldots, \texttt{E}_n^b) \mid \\
& & \texttt{PRIMARY}^b\texttt{.} \ll \texttt{C.m}(\bar{\texttt{T}})\texttt{T} \gg_{\texttt{stt}} \texttt{m}(\texttt{E}_1^b, \ldots, \texttt{E}_n^b) \mid \texttt{PRIMARY}^b\texttt{.} \ll \texttt{I.m}(\bar{\texttt{T}})\texttt{T} \gg_{\texttt{int}} \texttt{m}(\texttt{E}_1^b, \ldots, \texttt{E}_n^b) \\
\texttt{STMTS}^b & ::= & \texttt{STMT}_1^b \ldots \texttt{STMT}_n^b \\
\texttt{STMT}^b & ::= & \{\texttt{STMTS}^b\} \mid \texttt{SE}^b\ ; \mid \texttt{if}\ (\texttt{E}^b)\ \texttt{STMT}_1^b\ \texttt{else}\ \texttt{STMT}_2^b \mid \texttt{while}\ (\texttt{E}^b)\ \texttt{STMT}^b \\
& & \texttt{try}\ \{\texttt{STMTS}^b\ \}\ \texttt{CATCHES}^b\ \texttt{finally}\ \{\ \texttt{STMTS}_1^b\ \} \mid \texttt{throw}\ \texttt{E}^s \\
\texttt{SE}^b & ::= & \texttt{ASSIGN}^b \mid \texttt{INVOKE}^b \mid \texttt{NEW}^b \\
\texttt{CATCHES}^b & ::= & \texttt{CATCH}_1^b \ldots \texttt{CATCH}_n^b \\
\texttt{CATCH}^b & ::= & \texttt{catch}\ (\texttt{C x})\ \{\ \texttt{STMTS}^b\ \}
\end{array}
$$

Assumptions:

- interface names in $\texttt{B}$ are distinct;
- field names in $\texttt{FDS}^b$ are distinct;
- method/constructor signatures in $\texttt{MDS}^b$/$\texttt{KDS}^b$ are distinct;
- class names in $\texttt{CATCHES}^b$ are distinct.

Fig. 2. Syntax - Binaries

invocation expressions are annotated with the signature of the most specific method found at compile time and the static type of the receiver. There are four kinds of (binary) method invocation expressions $\texttt{INVOKE}^b$: virtual (instance method invocation), super (invocation via `super`), static and interface.

2.2 Type Environments

Type environments Γ are defined in Figure 3. A type assignment γ maps a class/interface name to its type.

The assignment $\texttt{C} \mapsto [\textsc{am}{=}\texttt{AM}, \textsc{ck}{=}\texttt{CK}, \textsc{parent}{=}\texttt{C}', \textsc{is}{=}\texttt{I}_1 \ldots \texttt{I}_n, \textsc{kss}{=}\texttt{KSS}, \textsc{fss}{=}\texttt{FSS}, \textsc{mss}{=}\texttt{MSS}]$ has the meaning "the class C has access modifier AM and kind CK, extends C', implements $\texttt{I}_1 \ldots \texttt{I}_n$ and has constructor signatures KSS, field signatures FSS and method signatures MSS". Analogously, $\texttt{I} \mapsto [\textsc{am}{=}\texttt{AM}, \textsc{is}{=}\texttt{I}_1 \ldots \texttt{I}_n, \textsc{fss}{=}\texttt{FSS}, \textsc{mss}{=}\texttt{MSS}]$ has the meaning "the interface I has access modifier AM, extends $\texttt{I}_1 \ldots \texttt{I}_n$, and has field signatures FSS and method signatures MSS".

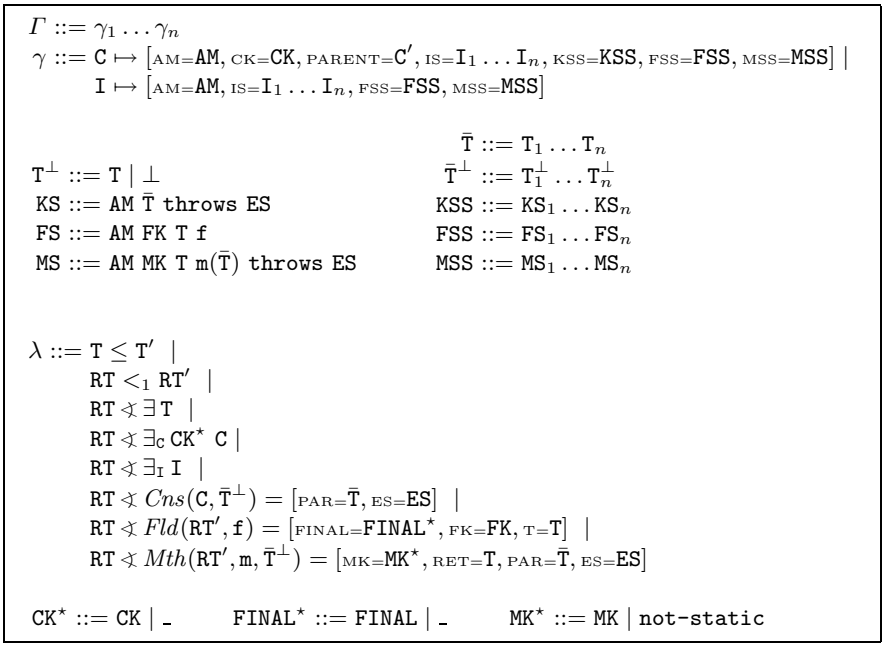

Fig. 3. Type environments and Type assumptions

Type assumptions λ, also defined in Figure 3, describe fine-grained requirements; they are:

- $\mathtt{T} \leq \mathtt{T'}$ with the meaning "T is a subtype of T'";
- $\mathtt{RT} <_1 \mathtt{RT'}$ with the meaning "RT *directly* extends RT'";
- $\mathtt{RT} \lhd \exists\, \mathtt{T}$ with the meaning "type T exists and is accessible from code contained in type RT"[2];
- $\mathtt{RT} \lhd \exists_{\mathtt{C}}\, \mathtt{CK}^{\star}\ \mathtt{C}$ with the meaning "class C, with kind $\mathtt{CK}^{\star}$, exists and is accessible from code contained in RT". $\mathtt{CK}^{\star} = \_$ means "any kind", that is, we do not care whether the class is abstract or not;
- $\mathtt{RT} \lhd \exists_{\mathtt{I}}\, \mathtt{I}$ with the meaning "interface I exists and is accessible from code contained in RT";
- $\mathtt{RT} \lhd \mathit{Cns}(\mathtt{C}, \bar{\mathtt{T}}^{\perp}) = [\text{PAR}=\bar{\mathtt{T}}, \text{ES}=\mathtt{ES}]$ with the meaning "the most specific constructor for class C and parameter types $\bar{\mathtt{T}}^{\perp}$, invoked from code contained in RT, has parameter types $\bar{\mathtt{T}}$ and can throw exceptions which are compatible with the exception specification ES"[3];
- $\mathtt{RT} \lhd \mathit{Fld}(\mathtt{RT'}, \mathtt{f}) = [\text{FINAL}=\mathtt{FINAL}, \text{FK}=\mathtt{FK}, \text{T}=\mathtt{T}]$ with the meaning "if code contained in RT looks up a field named f in type RT', then it finds a field with

[2] If you think the symbol "$\lhd$" as an eye, then you can interpret any assumption of the form "RT $\lhd \ldots$" as: "RT *sees* ..." and this is supposed to help ☺.

[3] That is, any exception C such that a C's superclass is contained in ES - see 11.2 of [8].

a final modifier $\texttt{FINAL}^\star$, kind $\texttt{FK}$ and type $\texttt{T}$". $\texttt{FINAL}^\star = \_$ means we do not care whether the field is final or not;

- $\texttt{RT} \lhd \mathit{Mth}(\texttt{RT}', \texttt{m}, \bar{\texttt{T}}^\perp) = [\text{MK}{=}\texttt{MK}^\star, \text{RET}{=}\texttt{T}, \text{PAR}{=}\bar{\texttt{T}}, \text{ES}{=}\texttt{ES}]$ with the meaning "the most specific method, invoked from code contained in $\texttt{RT}$, for a method named $\texttt{m}$, with parameter types $\bar{\texttt{T}}^\perp$ on a receiver with static type $\texttt{RT}'$ is a method which has kind $\texttt{MK}^\star$, return type $\texttt{T}$, parameter types $\bar{\texttt{T}}$ and can throw exceptions which are compatible with the exception specification $\texttt{ES}$"[3].

These assumptions can be thought as "the minimal pieces of information" needed to compile a certain source to a certain bytecode, as we will detail in the sequel. The rules defining the judgment $\Gamma \vdash \lambda$ are omitted for lack of space. A forthcoming extended work, containing all the rules, will be available at `http://www.disi.unige.it/person/LagorioG/publications.html` shortly. Also, the rules for a small subset of Java can be found in [2].

Next subsection shows how and when these assumptions are used in the process of compilation, while Section 3 explain how to exploit type assumptions in order to obtain a recompilation strategy which is both sound and minimal.

2.3 Compilation

Compilation of expressions is expressed by the following judgment:

$$\texttt{RT}; \Pi; \texttt{ES}; \Gamma \vdash \texttt{E}^s \rightsquigarrow \texttt{E}^b : \texttt{T}$$

with the meaning "expression $\texttt{E}^s$ has type $\texttt{T}$ and compiles to binary expression $\texttt{E}^b$ when contained in type $\texttt{RT}$, in a local environment Π, in a context where exceptions $\texttt{ES}$ can be thrown and in a type environment Γ". Type $\texttt{RT}$ is needed to model the access control; for instance, $\texttt{RT}$'s `private` methods can be invoked only by expressions inside $\texttt{RT}$. The local environment Π maps parameter names and `this` to their respective types (`this` is undefined when typing expressions contained in static contexts). Figure 4 show some selected rules defining this judgment.

Compilation of statements is expressed by the following judgment:

$$\texttt{RT}; \Pi; \texttt{ES}; \Gamma \vdash \texttt{STMT}^s \rightsquigarrow \texttt{STMT}^b$$

with the meaning "statement $\texttt{STMT}^s$ is compiled to $\texttt{STMT}^b$ when contained in type $\texttt{RT}$, in a local environment Π, in a context where exceptions $\texttt{ES}$ can be thrown and in a type environment Γ". The rules defining this judgment are omitted because of lack of space.

Before describing the compilation of fragments, we need to introduce the notion of *compilation environment*. A compilation environment *ce* maps fragment names to the corresponding fragment.

$$ce : \texttt{RT} \rightharpoonup \texttt{S} \cup \texttt{B}$$

Note that *ce* models a compilation environment from a compiler's point of view; that is, for each fragment name *ce* returns either a source or a binary

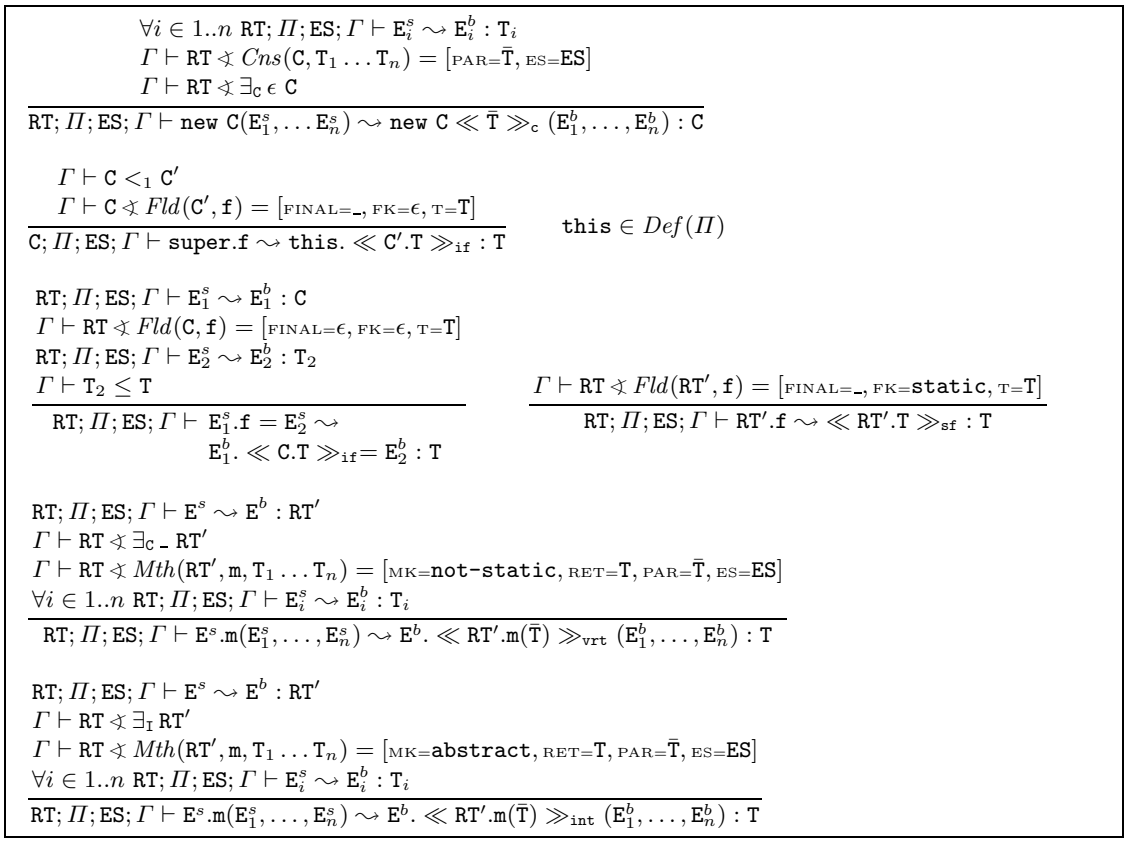

Fig. 4. Selected expression typing rules

fragment, but not both. In fact, even if both are present, only one is considered by the compiler, usually the most up-to-date according to the file's attributes. As a consequence, a type environment Γ can be extracted from a compilation environment *ce* by disregarding the code while retaining signatures and information about type hierarchy. Let us assume the function $extractEnv : ce \to \Gamma$ does this job.

Assume we have a *consistent* compilation environment *ce*, that is, if we compile all the sources in *ce* we obtain, for the sources which have been recompiled, the same binaries already present in *ce*. Assume, then, to change a bunch of sources, say $\mathtt{RT}_1, \ldots, \mathtt{RT}_n$, obtaining $ce_{\mathtt{new}}$ (and a corresponding $\Gamma_{\mathtt{new}} = extractEnv(ce_{\mathtt{new}})$). What do we have to do in order to obtain a new *consistent* compilation environment?

First of all, we have to check that the environment $\Gamma_{\mathtt{new}}$ is well-formed, for instance, it must not contain a cycle in the type hierarchy. We formally capture this notion with the judgment $\vdash \Gamma_{\mathtt{new}}\diamond$. One could argue that we do not want

to check the well-formedness of the *whole* $\Gamma_{\texttt{new}}$ after having changed, say, a *tiny detail* in one source. Right, we do not want to; yet, we need to *unless* we know something more (see Section 3).

After having checked the environment $\Gamma_{\texttt{new}}$, we need to decide which fragments to (re)compile. Of course we have to recompile those which have been changed, that is, $\texttt{RT}_1 \ldots \texttt{RT}_n$ but what else? Are you tempted to answer "`make clean; make all` will do"? A lot of real-world programmers would do just that. When dealing with a small/medium application and a quite powerful computer you can do that. And, it works. It works *fine* actually, but what if you *can not* do that? Next section addresses this crucial point. For now, just assume we have to compile $\texttt{RT}_1, \ldots, \texttt{RT}_m$ (obviously $m \geq n$).

When we have a well-formed $\Gamma_{\texttt{new}}$ and do know which fragments to recompile we can go for it: in our model "running the compilation" on each fragment $\texttt{RT}_i$ amounts to prove the following judgments:

$$\texttt{RT}_i; \Gamma_{\texttt{new}} \vdash ce_{\texttt{new}}(\texttt{RT}_i) \rightsquigarrow \texttt{B}_i$$

Figure 5 shows some selected rules defining this judgment. In defining the compilation of a set of classes we assume to compile them one by one, that is, we assume that no global optimizations take place. This reflects the fact that in languages with dynamic linking like Java, the concept of "program" is only significant at runtime so it is safer to leave cross class optimizations to virtual machines like HotSpot [9].

3 A Smart Strategy

When dealing with an updated compilation context $ce_{\texttt{new}}$ (in respect to a previous $ce_{\texttt{old}}$) there are two steps to perform: checking whether the corresponding new type environment $\Gamma_{\texttt{new}}$ is well-formed and, when it is, decide which (unchanged) source fragments have to be recompiled besides the changed ones. Some information gathered during a previous compilation can be used to speedup both these steps. We first show how we can check only the "updated part" of an environment $\Gamma_{\texttt{new}}$ when a previous well-formed environment $\Gamma_{\texttt{old}}$ is known. Then, we show how the type assumptions used to compile a fragment can be used later to decide whether it has to be recompiled.

We define $\texttt{leaves}_{\Gamma}(\texttt{RT}) = \{\texttt{RT}' | \Gamma \vdash \texttt{RT}' \leq \texttt{RT} \ \wedge \ \forall \texttt{RT}'' \ \Gamma \vdash \texttt{RT}'' \leq \texttt{RT}' \implies \texttt{RT}'' = \texttt{RT}'\}$ and the judgment $\Gamma \vdash \texttt{okOvr RT}$ with the meaning "RT correctly extends its parent types (up to Object) in Γ". That is, RT's hierarchy is acyclic and the Java rules on method overriding/hiding are respected. The rules defining such a judgment are omitted for lack of space.

Definition 1. *A type environment $\Gamma_{\texttt{new}}$ is well-formed w.r.t. another type environment $\Gamma_{\texttt{old}}$ iff the following conditions hold:*

$$\textit{[add]} \ \texttt{RT} \in \mathit{Def}(\Gamma_{\texttt{new}}) \setminus \mathit{Def}(\Gamma_{\texttt{old}}) \implies \begin{cases} \Gamma_{\texttt{new}} \vdash \texttt{okOvr RT} \\ \texttt{used}_{\Gamma_{\texttt{new}}}(\texttt{RT}) \subseteq \mathit{Def}(\Gamma_{\texttt{new}}) \end{cases}$$

These metarules assume Γ to be well-formed, that is, $\vdash \Gamma\diamond$.
Anyway, we do *not* want $\vdash \Gamma\diamond$ to be a premise of any of them (see the text for full details).

$$\frac{\mathtt{C};\Gamma \vdash \mathtt{KDS}^s \rightsquigarrow \mathtt{KDS}^b \quad \mathtt{C};\Gamma \vdash \mathtt{FDS}^s \rightsquigarrow \mathtt{FDS}^b \quad \mathtt{C};\Gamma \vdash \mathtt{MDS}^s \rightsquigarrow \mathtt{MDS}^b}{\begin{array}{l}\Gamma \vdash \mathtt{AM\ CK\ class\ C\ extends\ C'\ implements\ I}_1,\ldots,\mathtt{I}_m\ \{\mathtt{KDS}^s\ \mathtt{FDS}^s\ \mathtt{MDS}^s\} \rightsquigarrow \\ \quad \mathtt{AM\ CK\ class\ C\ extends\ C'\ implements\ I}_1,\ldots,\mathtt{I}_m\ \{\mathtt{KDS}^b\ \mathtt{FDS}^b\ \mathtt{MDS}^b\}\end{array}} \quad \mathtt{AM} \in \{\epsilon, \mathtt{public}\}$$

$$\frac{\mathtt{I};\Gamma \vdash \mathtt{FDS}^s \rightsquigarrow \mathtt{FDS}^b \quad \mathtt{I};\Gamma \vdash \mathtt{MDS}^s \rightsquigarrow \mathtt{MDS}^b}{\begin{array}{l}\Gamma \vdash \mathtt{AM\ interface\ I\ extends\ I}_1 \ldots \mathtt{I}_m\ \{\ \mathtt{FDS}^s\ \mathtt{MDS}^s\ \} \rightsquigarrow \\ \quad \mathtt{AM\ interface\ I\ extends\ I}_1 \ldots \mathtt{I}_m\ \{\ \mathtt{FDS}^b\ \mathtt{MDS}^b\ \}\end{array}} \quad \begin{array}{l}\mathtt{MDS}^s = \mathtt{MD}^s_1..\mathtt{MD}^s_n \\ \forall i \in 1..n\ \mathtt{MD}^s_i = \mathtt{public\ abstract}\ldots \\ \mathtt{FDS}^s = \mathtt{FD}^s_1..\mathtt{FD}^s_m \\ \forall i \in 1..m\ \mathtt{FD}^s_i = \mathtt{public\ static\ final}\ldots \\ \mathtt{AM} \in \{\epsilon, \mathtt{public}\}\end{array}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{C} <_1 \mathtt{C'} \\ \forall i \in 1..n\ \mathtt{C};\Pi;\mathtt{ES};\Gamma \vdash \mathtt{E}^s_i \rightsquigarrow \mathtt{E}^b_i : \mathtt{T}_i \\ \mathtt{C};\Pi;\mathtt{ES};\Gamma \vdash \mathtt{STMTS}^s \rightsquigarrow \mathtt{STMTS}^b \\ \Gamma \vdash \mathtt{C} \not< \mathit{Cns}(\mathtt{C'}, \mathtt{T}_1 \ldots \mathtt{T}_n) = [\textsc{par}{=}\bar{\mathtt{T}}, \textsc{es}{=}\mathtt{ES}]\end{array}}{\begin{array}{l}\mathtt{C};\Gamma \vdash\ \mathtt{AM}\ (\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws\ ES}\ \{\ \mathtt{super}(\mathtt{E}^s_1,\ldots,\mathtt{E}^s_n);\ \mathtt{STMTS}^s\ \} \rightsquigarrow \\ \quad \mathtt{AM}\ (\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws\ ES}\ \{\ \mathtt{super}(\mathtt{E}^b_1,\ldots,\mathtt{E}^b_n) \ll \bar{\mathtt{T}} \gg_{\mathtt{c}};\ \mathtt{STMTS}^b\ \}\end{array}} \quad \begin{array}{l}\Pi = \{\, x_1 \mapsto \mathtt{T}_1, \\ \qquad \ldots, \\ \qquad \mathtt{x}_n \mapsto \mathtt{T}_n, \\ \qquad \mathtt{this} \mapsto \mathtt{C}\}\end{array}$$

$$\frac{\begin{array}{l}\mathtt{RT};\Pi;\emptyset;\Gamma \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{T'} \\ \Gamma \vdash \mathtt{T'} \leq \mathtt{T}\end{array}}{\mathtt{RT};\Gamma \vdash\ \mathtt{AM\ FINAL\ FK\ T\ f} = \mathtt{E}^s\ ; \rightsquigarrow \mathtt{AM\ FINAL\ FK\ T\ f} = \mathtt{E}^b\ ;} \quad \Pi = \mathit{This}(\mathtt{FK}, \mathtt{C})$$

$$\frac{\begin{array}{l}\mathtt{C};\Pi;\mathtt{ES};\Gamma \vdash \mathtt{STMTS}^s \rightsquigarrow \mathtt{STMTS}^b \\ \mathtt{C};\Pi;\mathtt{ES};\Gamma \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{T'} \\ \Gamma \vdash \mathtt{T'} \leq \mathtt{T}\end{array}}{\begin{array}{l}\mathtt{C};\Gamma \vdash\ \mathtt{AM\ MK\ T\ m}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws\ ES}\ \{\ \mathtt{STMTS}^s\ \mathtt{return\ E}^s;\ \} \rightsquigarrow \\ \quad \mathtt{AM\ MK\ T\ m}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws\ ES}\ \{\ \mathtt{STMTS}^b\ \mathtt{return\ E}^b;\ \}\end{array}} \quad \begin{array}{l}\Pi = \{\, \mathtt{x}_1 \mapsto \mathtt{T}_1, \\ \qquad \ldots, \\ \qquad \mathtt{x}_n \mapsto \mathtt{T}_n\} \cup \\ \qquad \mathit{This}(\mathtt{MK}, \mathtt{C}) \\ \mathtt{MK} \neq \mathtt{abstract}\end{array}$$

$\mathit{This}(\_, \mathtt{I}) = \emptyset$
$\mathit{This}(\mathtt{static}, \mathtt{C}) = \emptyset;$
$\mathit{This}(\epsilon, \mathtt{C}) = \{\mathtt{this} \mapsto \mathtt{C}\}$

Fig. 5. Selected compilation rules

$$\textit{[rmv]}\ \ Def(\Gamma_{\texttt{old}}) \setminus Def(\Gamma_{\texttt{new}}) \neq \emptyset \implies \forall \texttt{RT} \in Def(\Gamma_{\texttt{new}})\ \texttt{used}_{\Gamma_{\texttt{new}}}(\texttt{RT}) \subseteq Def(\Gamma_{\texttt{new}})$$

$$\textit{[cng]}\ \ \texttt{RT} \in Def(\Gamma_{\texttt{old}}) \cap Def(\Gamma_{\texttt{new}}), \Gamma_{\texttt{old}}(\texttt{RT}) \neq \Gamma_{\texttt{new}}(\texttt{RT}) \implies \begin{cases} \forall \texttt{RT}' \in \texttt{leaves}_{\Gamma_{\texttt{new}}}(\texttt{RT})\ \Gamma_{\texttt{new}} \vdash \texttt{okOvr RT}' \\ \texttt{used}_{\Gamma_{\texttt{new}}}(\texttt{RT}) \subseteq Def(\Gamma_{\texttt{new}}) \end{cases}$$

where "used by RT in Γ", $\texttt{used}_{\Gamma}(\texttt{RT})$, means all types directly referenced by RT.

Theorem 1. *If $\vdash \Gamma_{\texttt{old}}\diamond$ holds, and $\Gamma_{\texttt{new}}$ is well-formed w.r.t. $\Gamma_{\texttt{old}}$ then, $\vdash \Gamma_{\texttt{new}}\diamond$ holds.*

Proof Two requirements have to be met:

- $\Gamma_{\texttt{new}}$ must be closed, that is, $\texttt{used}_{\Gamma_{\texttt{new}}}(\texttt{RT}) \subseteq Def(\Gamma_{\texttt{new}})$ for all $\texttt{RT} \in Def(\Gamma_{\texttt{new}})$;
- overriding rules must be satisfied, that is, for any $\texttt{RT} \in Def(\Gamma_{\texttt{new}})$ the judgment $\Gamma_{\texttt{new}} \vdash \texttt{okOvr RT}$ must be valid.

The former requirement is met by hypothesis when some type defined in $\Gamma_{\texttt{old}}$ has been removed from $\Gamma_{\texttt{new}}$, see *[rmv]* in Definition 1. When no types have been removed, the unchanged types cannot, trivially, refer to undefined types because, by hypothesis, $\Gamma_{\texttt{old}}$ is closed. By *[add]* and *[cng]* of Definition 1 new and updated types refer only to types defined in $\Gamma_{\texttt{new}}$.

The latter requirement can be proved by case analysis. Consider $Def(\Gamma_{\texttt{new}})$ as the union of three disjoint sets: U, C and N. These sets contain, respectively, the unchanged, changed and new types in $\Gamma_{\texttt{new}}$ with respect to $\Gamma_{\texttt{old}}$. Formally:

$$\begin{aligned} U &= \{\texttt{RT} | \texttt{RT} \in Def(\Gamma_{\texttt{new}}) \cap Def(\Gamma_{\texttt{old}}), \Gamma_{\texttt{new}}(\texttt{RT}) = \Gamma_{\texttt{old}}(\texttt{RT})\} \\ C &= \{\texttt{RT} | \texttt{RT} \in Def(\Gamma_{\texttt{new}}) \cap Def(\Gamma_{\texttt{old}}), \Gamma_{\texttt{new}}(\texttt{RT}) \neq \Gamma_{\texttt{old}}(\texttt{RT})\} \\ N &= Def(\Gamma_{\texttt{new}}) \setminus Def(\Gamma_{\texttt{old}}) \end{aligned}$$

Let us consider $N \cup C$ first. For any new type RT, contained in N, the judgment $\Gamma_{\texttt{new}} \vdash \texttt{okOvr RT}$ is valid by the hypothesis, see *[add]* of Definition 1. For any changed type RT, contained in C, the judgment $\Gamma_{\texttt{new}} \vdash \texttt{okOvr RT}$ is valid because of *[cng]* of Definition 1.

It remains to prove that the judgment holds for the unchanged types, contained in U. Since they are unchanged the *direct* supertypes of any type in U are the same in $\Gamma_{\texttt{old}}$ and $\Gamma_{\texttt{new}}$. Furthermore, each direct supertype of RT must be contained in $Def(\Gamma_{\texttt{old}})$ *and* in $Def(\Gamma_{\texttt{new}})$ because of, respectively, the fact that $\vdash \Gamma_{\texttt{old}}\diamond$ holds and the fact that $\Gamma_{\texttt{new}}$ is closed (which we have proved before). Hence, these direct supertypes must be contained in $Def(\Gamma_{\texttt{old}}) \cap Def(\Gamma_{\texttt{new}})$ which, by definition, is equal to: $U \cup C$. If a direct supertype is in U, then the same reasoning can be applied; so, for any $\texttt{RT} \in U$, only two cases are possible:

- *all* supertypes of RT are in U; then, the hierarchy of RT has not changed and by the hypothesis $\vdash \Gamma_{\texttt{old}}\diamond$ is valid, so the judgment $\Gamma_{\texttt{new}} \vdash \texttt{okOvr RT}$ is valid too;
- there exists a supertype $\texttt{RT}'$ of RT which is in C, whose subtypes till RT are in U. Then, by *[cng]* there exists a type $\texttt{RT}''$ such that $\texttt{RT}'' \leq \texttt{RT}$ and $\Gamma_{\texttt{new}} \vdash \texttt{okOvr RT}''$ holds. So, $\Gamma_{\texttt{new}} \vdash \texttt{okOvr RT}$ must be valid too. □

In summary, as long as we keep trace of a previous well-formed environment $\Gamma_{\texttt{old}}$ we can check for the well-formedness of any new environment $\Gamma_{\texttt{new}}$ by examining the changes w.r.t. $\Gamma_{\texttt{old}}$. The most expensive check must be performed when some classes are removed; this is acceptable because classes are added/changed more often than removed in the usual software development cycle. The requirement of having a previous well-formed environment available may seem restrictive, but it is not, since one can always use the empty environment (which is trivially well-formed) as $\Gamma_{\texttt{old}}$ when starting to use our strategy from scratch.

Assume we have a compilation environment $ce_{\texttt{old}}$, a corresponding well-formed type environment $\Gamma_{\texttt{old}} = \mathit{extractEnv}(ce_{\texttt{old}})$, and we can prove:

$$\texttt{RT}; \Gamma_{\texttt{old}} \vdash ce_{\texttt{old}}(\texttt{RT}) \rightsquigarrow \texttt{B}$$

The proof tree for this judgment can be proved to be unique, and contains a set of assumptions $\Lambda = \lambda_1, \ldots, \lambda_n$. We call it the *requirements* for RT in $\Gamma_{\texttt{old}}$ and write

$$\mathit{Reqs}(\texttt{RT}, \Gamma_{\texttt{old}}) = \Lambda$$

Assume to change $ce_{\texttt{old}}$ leaving the fragment for RT untouched. That is, assume to have another compilation environment $ce_{\texttt{new}}$, with a corresponding well-formed environment $\Gamma_{\texttt{new}}$, such that $ce_{\texttt{new}}(\texttt{RT}) = ce_{\texttt{old}}(\texttt{RT})$.

Does RT compile in $ce_{\texttt{new}}$? If it does, can we say something about the corresponding binary?

The following theorem states that, if $\Gamma_{\texttt{new}}$ still satisfy the requirements of RT in $\Gamma_{\texttt{old}}$, then we do know that RT compiles in $\Gamma_{\texttt{new}}$ and it compiles to the *same* binary.

Theorem 2. *If $ce_{\texttt{old}}$ and $ce_{\texttt{new}}$ are two compilation environments which share the same source for RT, that is, $ce_{\texttt{old}}(RT) = ce_{\texttt{new}}(RT)$, the corresponding type environments $\Gamma_{\texttt{old}} = \mathit{extractEnv}(ce_{\texttt{old}})$ and $\Gamma_{\texttt{new}} = \mathit{extractEnv}(ce_{\texttt{new}})$ are well-formed, and the judgment $RT; \Gamma_{\texttt{old}} \vdash ce_{\texttt{old}}(RT) \rightsquigarrow B$ can be proved, then:*

$$RT; \Gamma_{\texttt{new}} \vdash ce_{\texttt{new}}(RT) \rightsquigarrow B \iff \Gamma_{\texttt{new}} \vdash \mathit{Reqs}(RT, \Gamma_{\texttt{old}})$$

Proof (sketch) If $\Gamma_{\texttt{new}} \vdash \mathit{Reqs}(\texttt{RT}, \Gamma_{\texttt{old}})$, that is, $\Gamma_{\texttt{new}} \vdash \lambda$ holds for all $\lambda \in \mathit{Reqs}(\texttt{RT}, \Gamma_{\texttt{old}})$, the proof for $\texttt{RT}; \Gamma_{\texttt{old}} \vdash ce_{\texttt{old}}(\texttt{RT}) \rightsquigarrow \texttt{B}$ is a proof for $\texttt{RT}; \Gamma_{\texttt{new}} \vdash ce_{\texttt{new}}(\texttt{RT}) \rightsquigarrow \texttt{B}$ too. On the other hand, if $\Gamma_{\texttt{new}} \not\vdash \mathit{Reqs}(\texttt{RT}, \Gamma_{\texttt{old}})$, then there exists at least one $\lambda \in \mathit{Reqs}(\texttt{RT}, \Gamma_{\texttt{old}})$ such that $\Gamma_{\texttt{new}} \not\vdash \lambda$. Since the proof tree for a compilation judgment is unique, either there is no proof tree for compiling RT in $\Gamma_{\texttt{new}}$ or the compilation produces another binary $\texttt{B}'$ which differs from B. □

When the compilation for a fragment RT is undefined, or the binary produced by the compilation differs from the existing one, a sound strategy requires the compilation of RT.

To sum up, our compilation strategy is quite simple: *a fragment has to be recompiled if and only if the new environment does not entail its requirements.* This strategy is both *sound* and *minimal*.

So far so good: if we keep trace of the requirements for a fragment when we compile it we can apply a sound and minimal strategy. However, from a practical point of view there is another point to ponder: the cost of checking whether the requirements of a fragment are entailed by a type environment. If this checking costed more than compiling the source fragment, then all the reasoning so far would be useless. Luckily, this is not the case. In typechecking a source fragment a compiler must necessarily perform all the steps which are necessary to check the validity of the entailment. In addition, a compiler must, of course, parse the source and generate the code. So, checking whether the entailment holds is definitely faster than recompiling, and we expect this to be much faster. Of course, the global cost of using a smart strategy is not easy to determine because it depends on the particular compiler and the compilation context. That is, on the one hand there is the additional cost of checking the entailment, on the other hand there is the saving in not compiling the fragments whose requirements are entailed by the new environment. In a sense, the time used to find that a fragment's requirements are *not* entailed may appear "wasted", because that fragment has to be recompiled. However, if the compilation manager and the compiler are tightly integrated, the compiler's typechecking step can use the results of the previous entailment checking step as a sort of cache and skip many checks when recompiling the fragment. Using this trick, the "wasted" time is extremely small: it just consists in finding that a single requirement λ does not hold.

We can model an *extended* compilation environment as follows:

$$ce_+ : \langle \Gamma, \mathtt{RT} \rightharpoonup \mathtt{S} \cup \mathtt{B} \cup (\mathtt{B} \times \Lambda) \rangle$$

That is, a pair consisting of the previous (well-formed) environment Γ and a function which, for each fragment name, returns either: the source only, the binary only (when the source is not available) or a pair consisting of the binary and the requirements (the idea is that they are the result of a previous compilation).

4 Related and Further Work

This paper, together with [3, 4], can be considered a step towards a better support for separate compilation of Java-like languages.

The solution presented here, which extends the ideas in [2], is similar to attribute recompilation, according to the classification given in [1] - here attributes correspond to assumptions.

An inspiring source of our work has certainly been Dmitriev's paper [5], which describes a make technology, based on smart dependency checking, that aims to keep a project consistent while reducing the number of files to be recompiled. A freely downloadable tool, *Javamake*, is based on such a paper and implements the selective recompilation upon any Java compiler. Unfortunately, as pointed out by the author himself, there is no proof of the correctness of the approach, which is not based on theoretical foundations. 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. Conversely, *Javamake* cannot avoid a considerable amount of unnecessary recompilations. Hence, it is neither sound nor minimal.

The final goal of the work presented in this paper is implementing a smart compilation manager for the whole Java language, à la Javamake, but based on a formal model on which the correctness can be actually proved. To achieve this result, there are some subtle and Java-peculiar features which must be addressed.

- *Unreachable code* is not just a bad idea: it is forbidden (see 14.20 of [8]) - this is the most challenging issue to be tackled.
- *Final methods*: an invocation of a final instance method is compiled to different bytecode w.r.t. an invocation of a non-final instance method (non-virtual invocation vs virtual one, see 15.12.3 of [8]). This issue should not pose major problems.
- *Accessing a* `final` *field initialized by a constant expression* is a constant expression (see 15.28 of [8]) and so should be compiled directly to the corresponding value. This issue should not pose problems too, as long as an assumption keeps track that such a constant is really an access to a static final field.

Acknowledgements We warmly thank Elena Zucca, Davide Ancona and Sophia Drossopoulou for their useful suggestions and feedback.

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. M. Dmitriev. Language-specific make technology for the Java programming language. *ACM SIGPLAN Notices*, 37(11):373–385, 2002.
6. 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.
7. S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Dept. of Computing - Imperial College of Science, Technology and Medicine, September 2000.
8. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification, Second Edition*. Addison-Wesley, 2000.
9. SUN Microsystems. The Java HotSpot Virtual Machine, 2001. Technical White Paper.