Another Step Towards a Smart Compilation Manager for Java*

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

Categories and Subject Descriptors

D.3.4 [**Language Constructs and Features**]: Compilers; D.3.4 [**Language Constructs and Features**]: Incremental Compilers

General Terms

Languages

Keywords

Java, separate compilation

ABSTRACT

In a recent work we have proposed a compilation strategy (that is, a way to decide which unchanged sources have to be recompiled) for a substantial subset of Java which has been shown to be *sound* and *minimal*. That is, an unchanged source is recompiled if and only if its recompilation produces a different binary or an error. However, that model does not handle two features of Java, namely, compile-time constant fields (`static final` fields initialized by a compile-time constant of a primitive type or `String`) and unreachable code, which turn out to be troublesome for having a sound and minimal compilation strategy. To our best knowledge these two features, probably because of their low-level nature, have been omitted in all models of Java separate compilation written so far. Yet, a compilation strategy for full Java has to deal with them. Thus, in this paper we analyze the implications of handling compile-time constant fields and unreachable code, and extend our previous model in order to handle these two features as well.

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

SAC'04 March 14-17, 2004, Nicosia, Cyprus

1. INTRODUCTION

The intuitive idea behind smart (re)compilation mechanisms is to be able to recompile only a part of a program after having changed some sources, with the implicit requirement that the result of such a recompilation should be equivalent to recompiling all the sources, but cheaper (in time). Note that recompiling only the changed sources is, in most cases, not enough, since these changes could affect the compatibility with other sources. On the other hand, recompiling a source `S`, whose corresponding binary `B` is already present, to obtain the same binary `B` would be a waste of time. In a recent work [8] we have proposed a compilation strategy (that is, a way to decide which unchanged sources have to be recompiled) for a substantial subset of Java which has been shown to be *sound* and *minimal*. That is, an unchanged source is recompiled if and only if its recompilation produces a different binary (or an error). However, that model does not handle two features of Java, namely, compile-time constant fields (`static final` fields initialized by a compile-time constant of a primitive type or `String`) and unreachable code, which turn out to be troublesome for having a sound and minimal compilation strategy. To our best knowledge these two features, probably because of their low-level nature, have been omitted in all models of Java separate compilation written so far. Yet, a compilation strategy for the full Java has to deal with them. Thus, we first discuss the implications of handling compile-time constant fields and unreachable code and then we extend our previous model in order to handle these two features as well. Section 2 discusses, by means of some examples, why the way Java handles compile-time constant fields and unreachable code affects selective recompilation. Section 3 extends the model we have presented in [8]. Finally, Section 4 discusses related and further work.

2. AN INFORMAL PRESENTATION

Because of lack of space we cannot describe in details how the model we extend works, so we recall here the basic idea but refer to the original paper [8] for an extended description. The intuitive idea behind our smart recompilation mechanism is to collect, while compiling a source fragment `S` into a binary `B`, a set of type assumptions which represent the *minimal* requirements on the other fragments which have been used to compile `S` into `B`. So, the result of each successful compilation is a binary, as usual, plus a set of assumptions. After a change to some sources is made, the requirements of all *unchanged* fragments are checked against

the new type environment[1] (the changed sources have to be recompiled anyway, so their assumptions are not checked). All unchanged sources whose type assumptions still hold in the new environment need not to be recompiled, since they would be recompiled to the same binaries. Instead, all other fragments have to be recompiled because either they compile to different binaries or they do not successfully compile anymore.

As said in the introduction, two features (not considered in the original model) heavily affect the smart recompilation mechanism: compile-time constant fields and unreachable code. The former is discussed in Section 2.1 and the latter in Section 2.2.

2.1 Compile-time Constant Fields

Compile-time constant fields are `static final` fields, of a primitive type or `String`, which are initialized by a compile-time constant. For the sake of brevity we will call them *ctc-fields* from now on. Because the value of these fields is a compile-time constant (see 15.28 of [7]), the bytecode generated for an access to a ctc-field is the same that would be generated if the literal corresponding to the value of the field was used in its place.

Consider, for instance, the following declarations (assuming that each class is declared in a separate file).

```
class A {
    static final int CONST_A = 1 ;
}
class B {
    static final int CONST_B = 1 ;
}
class FirstClient {
    int ma() {
        return A.CONST_A ;
    }
    int mb() {
        return B.CONST_B ;
    }
}
class SecondClient {
    int m() {
        return A.CONST_A+B.CONST_B ;
    }
}
```

These sources are compiled to the following binaries[2]:

```
class A {
    static final int CONST_A = 1 ;
}
class B {
    static final int CONST_B =1 ;
}
class FirstClient {
    int ma() { return 1 ; }
    int mb() { return 1 ; }
}
class SecondClient {
    int m() { return 2 ; }
}
```

As shown in the example, in the binaries every symbolic reference to ctc-fields has disappeared. Indeed, there is no need to calculate at run-time a value which is a compile-time constant. This means that every time the value of a ctc-field is changed and the class where it has been declared is recompiled, its clients might need to be recompiled as well, in order to have a sound compilation strategy. Indeed, the bytecode corresponding to the method `ma` declared in class `FirstClient` changes each time the value of `A.CONST_A` changes. So, we need to keep track of the dependencies between the definition of ctc-fields and their uses. For this reason, we introduce a new kind of type assumption. However, the naive approach of adding assumptions of the form "field name=value" is sound but not minimal. As the example shows, client classes `FirstClient` and `SecondClient` use both constants, but the former needs both `A.CONST_A` and `B.CONST_B` to be equal to `1` to be recompiled to the same bytecode, while the latter just needs their sum to be equal to `2`. Thus, if we change, say, `CONST_A` to `0` and `CONST_B` to `2`, then the first client has to be recompiled, while the second has not. Indeed, after the change, the expression `A.CONST_A+B.CONST_B` has still the same value it had before, so class `SecondClient` would be recompiled to the same existing binary. Hence, to achieve minimality we use type assumptions of the form "expression=value" where the expression is built on values and references to fields (which must be constant). Moreover, we need assumptions of the form "`f` $=\perp$" which express the fact that a field `f` *is not* a ctc-field. This is needed, in this case, because every access to `f` is compiled to a field-access instruction in the bytecode but, if `f` became a ctc-field, then the compile-time value of `f` would be directly used instead.

Note that compiling a single source expression may lead to several assumptions; for instance, the source expression `A.CONST_A+aMethod(A.CONST_A>=B.CONST_B)`, is compiled[3] to `1+aMethod(true)` and generates two type assumptions:

1. `A.CONST_A=1` and
2. `A.CONST_A>=B.CONST_B = true`.

These assumptions model the fact that this source expression is compiled to the same bytecode whenever `A.CONST_A` is equal to `1` (because this is the value of the left operand of the sum) and `B.CONST_B` is any integer less or equal to `1` (that is, the value `A.CONST_A` is constrained to be by the first assumption). In some cases these assumptions could be simplified; for instance, the second one can be simplified, as just noted, to `1>=B.CONST_B = true`. In the general case, anyway, this would require an analysis of assumptions deduced from different expressions and we think it is not worth the effort.

2.2 Unreachable Code

The way Java handles unreachable code is peculiar: what is a warning in most languages is a compile-time error in Java, see 14.20 of [7]. The relation between this choice and selective recompilation lies in the fact that a change in some source may make unreachable a piece of code, contained

[1] A type environment is a map from class/interface names to their types.

[2] As in [8], we use a very abstract view of the bytecode. In a real `.class` file the initializer expressions for all ctc-fields are not present (see 13.4.8 of [7]) and their values are stored as attributes of type `ConstantValue` (see 4.7.2 of [9]).

[3] Ignoring method call annotations, which are of no interest in the context of this example.

in an unchanged source, which was reachable before. So, a source which was perfectly legal is not correct anymore. In this case, a sound compilation strategy must trigger the compilation on that source, in order to raise the error (unreachable code) that a global recompilation would raise.

Two ways to make unreachable a piece of code, which was reachable, without changing its source are: changing the value of a ctc-field and changing the exception specification of a method. Examples of both cases are shown below, starting with the simpler one: changing the value of a ctc-field (making it a member of the infamous "inconstant constants[4]" club ☺). For instance, consider:

```
class ThirdClient {
    void m() {
        int x = 0 ;
        while (A.CONST_A==B.CONST_B)
            ++x ;
    }
}
```

This client[5] class can be compiled only when both ctc-fields `A.CONST_A` and `B.CONST_B` have the same value; when they do not, the increment of `x` can never be reached.

The following example shows that changing an exception specification can do the trick too.

```
class E extends Exception { ... }
class Foo {
    Foo m() throws E { ... }
    Foo m2() { ... }
}
class Client {
    ...
    try   {
        new Foo().m().m2() ;
    } catch (E e) {
        ++x ;
    }
    ...
}
```

If `E` is removed from the exception specification of method `m`, then the increment of `x` inside the `catch` becomes unreachable. As for ctc-fields, we introduce a new assumption to model the requirements which guarantee a piece of code to be reachable. Again, a naive assumption like "method `m` throws `E`" is not enough to obtain a minimal, although sound, compilation strategy: indeed, if we move `E` from the exception specification of `m` to the exception specification of `m2`, then the code inside the `catch` remains reachable. The nesting of try statements complicates the matter further; consider, for instance, the following code:

```
try {
    m1() ;
    try {
        m2() ;
    } catch (E1 e1) { }
} catch (E2 e2) { ++x ; }
```

[4]See 13.4.8 of [7].

[5]of classes `A` and `B` which have been defined in Section 2.1.

In this example the increment of `x` is reachable if and only if:

- method `m1` has an exception specification which includes a subclass of `E2` (which can, of course, be `E2` itself) *or*
- method `m2` has an exception specification which includes a subclass of `E2` which is not a subclass of `E1` (otherwise such an exception would be captured by the `catch` clause of the inner `try`).

Using $\varepsilon(\texttt{m})$ to indicate the exception specification of the method `m` we can express the above assumptions as shown below:

$$(\varepsilon(\texttt{m1}) \cup_{\texttt{Exc}} (\varepsilon(\texttt{m2}) \setminus_{\texttt{Exc}} \{\texttt{E1}\})) \supseteq_{\texttt{Exc}} \{\texttt{E2}\}$$

The operators $\cup_{\texttt{Exc}}$, $\setminus_{\texttt{Exc}}$ and $\supseteq_{\texttt{Exc}}$ are discussed in the next section.

3. EXTENSIONS TO THE MODEL

Albeit the model we have presented in [8] handles a substantial subset of Java, we need to extend it in order to model ctc-fields and unreachable code. In this paper we only describe the extension and briefly recall the original definitions where needed to understand the new parts. Figure 1 summarizes the changes.

- The syntax of source expressions $\texttt{E}^s$ and binary expressions $\texttt{E}^b$ has been enriched in order to include arithmetic expressions in the language; this change is necessary to show how constant expressions are compiled.
- The keyword `break` has been added to the syntax of source statements $\texttt{STMT}^s$ and binary statements $\texttt{STMT}^b$ because, albeit this is not strictly necessary, it makes the typing rules for the statement `while` a bit more interesting when dealing with unreachable code.
- As before, type environments Γ map a class/interface name to its type. These types contain, among other information, the signatures of the declared fields, which have been enriched to describe also ctc-fields. A field signature `FS` for a ctc-field consists of: the access modifier `AM`, the kind[6], the type `T`, the name `f` and the compile-time value ν. This change implies that discarding bodies from sources and binaries, as described in [8], is no longer enough for extracting a type environment from a compilation environment (a collection of sources and binaries); now the initialization expressions for the ctc-fields declared in the source fragments must be evaluated as well.
- Exception specifications `ES`, as in the previous model, are sets of class names (for simplicity we allow any class to be used as an exception). In this new model we introduce exception expressions `EE`, which describe a set of exceptions in an abstract way. The exception expressions $\varepsilon(\texttt{C}(\bar{\texttt{T}}))$ and $\varepsilon(\texttt{RT.m}(\bar{\texttt{T}}))$ represent, respectively, the exception specification of the constructor for class `C` with parameter types $\bar{\texttt{T}}$ and of the method named `m`, with parameter types $\bar{\texttt{T}}$, declared in the reference type `RT`. These abstract sets of exceptions, and

[6]The kind is always `static` for ctc-fields, but the kind is kept in the signature for uniformity.

$$
\begin{array}{rcl}
\mathtt{E}^s & ::= & \ldots \mid \mathtt{E}^s_1 + \mathtt{E}^s_2 \mid \mathtt{E}^s_1 - \mathtt{E}^s_2 \mid \ldots \\
\mathtt{E}^b & ::= & \ldots \mid \mathtt{E}^b_1 + \mathtt{E}^b_2 \mid \mathtt{E}^b_1 - \mathtt{E}^b_2 \mid \ldots \\
\mathtt{STMT}^s & ::= & \ldots \mid \mathtt{break}\ ; \\
\mathtt{STMT}^b & ::= & \ldots \mid \mathtt{break}\ ; \\
\\
\Gamma & ::= & \gamma_1 \ldots \gamma_n \\
\gamma & ::= & \mathtt{C} \mapsto [\ldots,\ \mathtt{FSS} = \mathtt{FS}_1 \ldots \mathtt{FS}_n,\ \ldots] \mid \\
 & & \mathtt{I} \mapsto [\ldots,\ \mathtt{FSS} = \mathtt{FS}_1 \ldots \mathtt{FS}_n,\ \ldots] \\
\\
\beta & ::= & \mathtt{true} \mid \mathtt{false} \\
\iota & ::= & 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots \\
\nu & ::= & \beta \mid \iota \\
\mathtt{FS} & ::= & \ldots \mid \mathtt{AM\ static\ T\ f}\ =\ \nu \\
\\
\mathtt{ES} & ::= & \{\mathtt{C}_1, \ldots, \mathtt{C}_n\} \\
\mathtt{EE} & ::= & \varepsilon(\mathtt{C}(\overline{\mathtt{T}})) \mid \varepsilon(\mathtt{RT.m}(\overline{\mathtt{T}})) \mid \mathtt{ES} \mid \\
 & & \mathtt{EE}_1 \cup_{\mathtt{Exc}} \mathtt{EE}_2 \mid \mathtt{EE} \setminus_{\mathtt{Exc}} \mathtt{ES} \\
\\
\lambda & ::= & \ldots \mid \mathtt{RT} \not\lhd \mathtt{E}^s = \nu \mid \mathtt{RT.f} = \bot \mid \mathtt{EE} \supseteq_{\mathtt{Exc}} \mathtt{ES}
\end{array}
$$

Figure 1: Extensions to the previous model

also the actual sets of exceptions ES, can be combined together using the operators "union" $\cup_{\mathtt{Exc}}$ and "minus" $\setminus_{\mathtt{Exc}}$ which, in first approximation, can be thought as the corresponding set operations. However, they have to deal with the fact that any exception C actually stands for "C or any subclass". Because of lack of space we do not give here the formal semantics of these operators. An in-depth discussion of these issues can be found in [4].

- Three new kinds of type assumptions λ have been added. The first two are needed to deal with constant expressions, the last one with code reachability. The assumption $\mathtt{RT} \not\lhd \mathtt{E}^s = \nu$ has the meaning: "if the code contained in RT evaluates $\mathtt{E}^s$ it gets the value ν"; $\mathtt{RT.f} = \bot$ has the meaning: "the field f declared in RT is not a ctc-field"; finally, $\mathtt{EE} \supseteq_{\mathtt{Exc}} \mathtt{ES}$ has the meaning: "the (abstract) set of exceptions EE contains the (actual) set ES".

Compilation of expressions is expressed by the following judgment:

$$\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{T}\ \textbf{throws:}\ \mathtt{EE}$$

with the meaning "expression $\mathtt{E}^s$ has type T, throws exceptions EE and compiles to binary expression $\mathtt{E}^b$ when contained in type RT, in a local environment Π, in a context where exceptions ES can be thrown and in a type environment Γ". Type RT is needed to model the access control; for instance, `private` methods of RT can be invoked only by expressions inside RT. The local environment Π maps parameter names and `this` to their respective types (`this` is undefined when typing expressions contained in static contexts). The only difference with respect to the previous model is the presence of EE, which is needed for expressing the requirements on code reachability (see below). Figure 2 shows some selected rules defining this judgment. The first two rules model the compilation of a compile-time constant expression of type, respectively, `int` and `bool`. As explained before, these expressions are compiled directly to their corresponding value and, obviously, their evaluation throw no exceptions (so EE is the empty set). The third rule models the compilation of an expression which accesses the static field f of the type RT′. In this case the field is not a ctc-field (because of the premise $\Gamma \vdash \mathtt{RT}'.\mathtt{f} = \bot$) so the field-access expression is compiled to a field-access binary expression (which is annotated with the type of f). In the last rule two expressions, which are not both constant, are combined using an operator Θ: the result is a non constant expression that can throw any exception its operands can throw.

Compilation of statements is expressed by the following judgment:

$$\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{STMT}^s \rightsquigarrow \mathtt{STMT}^b\ \textbf{throws:}\ \mathtt{EE}\ \textbf{ccn:}\ \beta$$

with the meaning "statement $\mathtt{STMT}^s$ is compiled to $\mathtt{STMT}^b$, throws exceptions EE and can complete normally β when contained in type RT, in a local environment Π, in a context where exceptions ES can be thrown and in a type environment Γ". Figure 3 shows some selected rules defining this judgment. The difference with respect to the previous model is the presence of EE and the flag β, both required to model unreachable code. The intuitive idea is that a reachable statement can complete normally (so the flag β is true) when, at run-time, the flow of execution may continue beyond such a statement. For instance, a sequence of statements can complete normally if and only if all statements it consists of can complete normally (see the first two rules of Figure 3), the evaluation of a statement expression $\mathtt{SE}^b$ always complete normally (third rule), while a statement `break` cannot ever complete normally (fourth rule). These notions are described in detail in 14.20 of [7]. Figure 3 contains an apparent asymmetry in how the statements `if` and `while` are handled. That is, `while (false) {...}` is not allowed, because its body would be unreachable, but the similar cases `if (false) {...} else {...}` and `if (true) {...} else {...}` are ok (even though in both cases one of the two branches is indeed unreachable). The typing rules reflect the language specification which allows this use of statement `if` as a way to express conditional compilation. The last two rules in Figure 3 deal with throwing and catching exceptions. A statement `throw`, whose expression $\mathtt{E}^s$ has type C, can throw all the exceptions the evaluation of $\mathtt{E}^s$ can throw, plus, of course, C itself. A statement `throw` never completes normally. The last rule is the trickiest of all: a statement `try` can throw the exceptions its body can throw $\mathtt{EE}_0$, minus all the caught ones $\{\mathtt{C}_1, \ldots \mathtt{C}_n\}$, plus the ones that can be thrown by statement inside its `finally` clause, $\mathtt{EE}_{n+1}$. Moreover, the statement `try` can complete normally only when at least its body $\mathtt{STMT}^s_0$ or one of the catch clauses $\mathtt{STMT}^s_i$ can complete normally *and* the `finally` clause can complete normally.

4. RELATED AND FURTHER WORK

This paper describes an extension of the model presented in [8], which, in turn, is based on ideas recently discussed with Ancona [2, 3] and has been inspired by Dmitriev's paper [6]. The solution presented here is similar to attribute recompilation, according to the classification given in [1] - here attributes correspond to assumptions.

The initial goal was to provide a compilation strategy able to recompile the minimum number of source fragments still guaranteeing the soundness of the approach (that is, guaranteeing that the result of a smart recompilation is equivalent to recompiling all the sources). The idea of having some

$$\frac{\Gamma \vdash \mathtt{RT} \not\downarrow \mathtt{E}^s = \iota}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{E}^s \rightsquigarrow \iota : \mathtt{int}\ \textbf{throws:}\ \emptyset}$$

$$\frac{\Gamma \vdash \mathtt{RT} \not\downarrow \mathtt{E}^s = \beta}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{E}^s \rightsquigarrow \beta : \mathtt{bool}\ \textbf{throws:}\ \emptyset}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{RT}'.\mathtt{f} = \bot \\ \Gamma \vdash \mathtt{RT} \not\downarrow \mathit{Fld}(\mathtt{RT}', \mathtt{f}) = [\textsc{final}{=}\_, \textsc{fk}{=}\mathtt{static}, \textsc{t}{=}\mathtt{T}]\end{array}}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{RT}'.\mathtt{f} \rightsquigarrow \ll \mathtt{RT}'.\mathtt{T} \gg_{\mathtt{sf}} \mathtt{f} : \mathtt{T}\ \textbf{throws:}\ \emptyset}$$

$$\frac{\begin{array}{l}\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{E}_1^s \rightsquigarrow \mathtt{E}_1^b : \mathtt{int}\ \textbf{throws:}\ \mathtt{EE}_1 \\ \mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{E}_2^s \rightsquigarrow \mathtt{E}_2^b : \mathtt{int}\ \textbf{throws:}\ \mathtt{EE}_2\end{array}}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{E}_1^s \ominus \mathtt{E}_2^s \rightsquigarrow \mathtt{E}_1^b \ominus \mathtt{E}_2^b : \mathtt{int}\ \textbf{throws:}\ \mathtt{EE}_1 \cup_{\mathtt{Exc}} \mathtt{EE}_2} \quad \begin{array}{l}\ominus \in \{+, -, \ldots\} \\ \mathtt{E}_1^b \text{ or } \mathtt{E}_2^b \text{ is not a value}\end{array}$$

Figure 2: Selected expression typing rules

$$\frac{}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \{\} \rightsquigarrow \{\}\ \textbf{throws:}\ \emptyset\ \textbf{ccn:}\ \mathtt{true}}$$

$$\frac{\begin{array}{l}i \in 1..(n-1)\ \mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{STMT}_i^s \rightsquigarrow \mathtt{STMT}_i^b\ \textbf{throws:}\ \mathtt{EE}_i\ \textbf{ccn:}\ \mathtt{true} \\ \mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{STMT}_n^s \rightsquigarrow \mathtt{STMT}_n^b\ \textbf{throws:}\ \mathtt{EE}_n\ \textbf{ccn:}\ \beta\end{array}}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \{\mathtt{STMT}_1^s \ldots \mathtt{STMT}_n^s\} \rightsquigarrow \{\mathtt{STMT}_1^b \ldots \mathtt{STMT}_n^b\}\ \textbf{throws:}\ \mathtt{EE}_1 \cup_{\mathtt{Exc}} \ldots \cup_{\mathtt{Exc}} \mathtt{EE}_n\ \textbf{ccn:}\ \beta}$$

$$\frac{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{SE}^s \rightsquigarrow \mathtt{SE}^b : \mathtt{T}\ \textbf{throws:}\ \mathtt{EE}}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{SE}^s\ ; \rightsquigarrow \mathtt{SE}^b\ ;\ \textbf{throws:}\ \mathtt{EE}\ \textbf{ccn:}\ \mathtt{true}}$$

$$\frac{}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{break}\ ; \rightsquigarrow \mathtt{break}\ ;\ \textbf{throws:}\ \emptyset\ \textbf{ccn:}\ \mathtt{false}}$$

$$\frac{\begin{array}{l}\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{bool}\ \textbf{throws:}\ \mathtt{EE}_0 \\ i \in 1..2\ \mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{STMT}_i^s \rightsquigarrow \mathtt{STMT}_i^b\ \textbf{throws:}\ \mathtt{EE}_i\ \textbf{ccn:}\ \beta_i\end{array}}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{if}\ (\mathtt{E}^s)\ \mathtt{STMT}_1^s\ \mathtt{else}\ \mathtt{STMT}_2^s \rightsquigarrow \mathtt{if}\ (\mathtt{E}^b)\ \mathtt{STMT}_1^b\ \mathtt{else}\ \mathtt{STMT}_2^b\ \textbf{throws:}\ \mathtt{EE}_0 \cup_{\mathtt{Exc}} \mathtt{EE}_1 \cup_{\mathtt{Exc}} \mathtt{EE}_2\ \textbf{ccn:}\ \beta_1 \vee \beta_2}$$

$$\frac{\begin{array}{l}\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{bool}\ \textbf{throws:}\ \mathtt{EE}_0 \\ \mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{STMT}^s \rightsquigarrow \mathtt{STMT}^b\ \textbf{throws:}\ \mathtt{EE}_1\ \textbf{ccn:}\ \_\end{array}}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{while}\ (\mathtt{E}^s)\ \mathtt{STMT}^s \rightsquigarrow \mathtt{while}\ (\mathtt{E}^b)\ \mathtt{STMT}^b\ \textbf{throws:}\ \mathtt{EE}_0 \cup_{\mathtt{Exc}} \mathtt{EE}_1\ \textbf{ccn:}\ \mathtt{true}} \quad \mathtt{E}^b \text{ is not a value}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{RT} \not\downarrow \mathtt{E}^s = \mathtt{true} \\ \mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{STMT}^s \rightsquigarrow \mathtt{STMT}^b\ \textbf{throws:}\ \mathtt{EE}\ \textbf{ccn:}\ \_\end{array}}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{while}\ (\mathtt{E}^s)\ \mathtt{STMT}^s \rightsquigarrow \mathtt{while}\ (\mathtt{true})\ \mathtt{STMT}^b\ \textbf{throws:}\ \mathtt{EE}\ \textbf{ccn:}\ \beta} \quad \begin{array}{ll}\beta = & \text{there is a break for this} \\ & \text{while inside } \mathtt{STMT}^s\end{array}$$

$$\frac{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{C}\ \textbf{throws:}\ \mathtt{EE}}{\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{throw}\ \mathtt{E}^s\ ; \rightsquigarrow \mathtt{throw}\ \mathtt{E}^b\ ;\ \textbf{throws:}\ \mathtt{EE} \cup_{\mathtt{Exc}} \{\mathtt{C}\}\ \textbf{ccn:}\ \mathtt{false}}$$

$$\frac{\begin{array}{l}\mathtt{RT}; \Pi; \mathtt{ES} \cup_{\mathtt{Exc}} \{\mathtt{C}_1, \ldots \mathtt{C}_n\}; \Gamma \vdash \mathtt{STMT}_0^s \rightsquigarrow \mathtt{STMT}_0^b\ \textbf{throws:}\ \mathtt{EE}_0\ \textbf{ccn:}\ \beta_0 \\ i \in 1..n\ \mathtt{RT}; \Pi[\mathtt{x}_i \mapsto \mathtt{C}_i]; \mathtt{ES}; \Gamma \vdash \mathtt{STMT}_i^s \rightsquigarrow \mathtt{STMT}_i^b\ \textbf{throws:}\ \mathtt{EE}_i\ \textbf{ccn:}\ \beta_i \\ \mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash \mathtt{STMT}_{n+1}^s \rightsquigarrow \mathtt{STMT}_{n+1}^b\ \textbf{throws:}\ \mathtt{EE}_{n+1}\ \textbf{ccn:}\ \beta_{n+1} \\ i \in 1..n\ \Gamma \vdash (\mathtt{EE}_0 \setminus_{\mathtt{Exc}} \{\mathtt{C}_1, \ldots, \mathtt{C}_{i-1}\}) \supseteq_{\mathtt{Exc}} \{\mathtt{C}_i\}\end{array}}{\begin{array}{l}\mathtt{RT}; \Pi; \mathtt{ES}; \Gamma \vdash\ \mathtt{try}\ \{\ \mathtt{STMT}_0^s\ \}\ \mathtt{catch}\ (\mathtt{C}_1\ \mathtt{x}_1)\ \{\ \mathtt{STMT}_1^s\ \}\ \ldots\ \mathtt{catch}\ (\mathtt{C}_n\ \mathtt{x}_n)\ \{\ \mathtt{STMT}_n^s\ \}\ \mathtt{finally}\ \{\ \mathtt{STMT}_{n+1}^s\ \} \rightsquigarrow \\ \quad \mathtt{try}\ \{\ \mathtt{STMT}_0^b\ \}\ \mathtt{catch}\ (\mathtt{C}_1\ \mathtt{x}_1)\ \{\ \mathtt{STMT}_1^b\ \}\ \ldots\ \mathtt{catch}\ (\mathtt{C}_n\ \mathtt{x}_n)\ \{\ \mathtt{STMT}_n^b\ \}\ \mathtt{finally}\ \{\ \mathtt{STMT}_{n+1}^b\ \} \\ \quad \textbf{throws:}\ (\mathtt{EE}_0 \setminus_{\mathtt{Exc}} \{\mathtt{C}_1, \ldots, \mathtt{C}_n\}) \cup_{\mathtt{Exc}} \mathtt{EE}_{n+1}\ \textbf{ccn:}\ (\beta_0 \vee \ldots \vee \beta_n) \wedge \beta_{n+1}\end{array}}$$

Figure 3: Selected statement typing rules

type assumptions which describe *exactly* what is needed for a certain source fragment to be compiled to a specific binary fragment is very similar to have a type system with principal typings [10]. These ideas have been formalized for a Java-like language by Ancona and Zucca [5].

The intuition behind our work is to shorten the recompilation time by recompiling less fragments, even though, from a practical point of view, this might not be the case: the time required to decide which files have to be recompiled has to be taken in account as well. Evaluating that time is a difficult task which heavily depends on how much the compilation manager and the compiler are integrated. In a production system they have to be tightly integrated, but in a prototype implementation they would probably consist of two different programs: the compilation manager and a standard Java compiler slightly modified (in order to produce the type assumptions in addition to the standard bytecode). Unfortunately, in order to deal with ctc-fields and unreachable code we had to add some new type assumptions which are rather complex.

We think that a contribution of this paper is showing clearly how some Java features were not designed with compositional compilation in mind. While the way ctc-fields are handled could be explained for efficiency reasons, the rationale in banning some[7] unreachable code is not clear: a warning would be enough to help the programmer in spotting a potential problem in the code, whereas an error is just annoying when the programmer knows what she/he is doing. Albeit we do not like how these two features have been designed, our goal is to analyze Java as an example of a real-world mainstream language, so we have to bear with them.

Because much of the complexity of the new type assumptions stems from avoiding unnecessary recompilations, it could make sense to trade ease of implementation with minimality if this can also help in gaining a speed-up in checking whether a fragment has to be recompiled. Of course, we do *not* mean to give up on minimality altogether, because that would mean to compile all the sources each time and this contradicts the whole idea of smart recompilation. A compromise could consist in "relaxing" some of the assumptions so that they continue to work for the most common cases and "fail" (so an unnecessary recompilation may be triggered) on some, hopefully rare, cases.

The next step is to implement the ideas presented here, experimenting with "less precise" type assumptions and comparing the various strategies in a quantitative way as it has been done with a number of different mechanisms in [1].

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

5. 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 and G. Lagorio. Stronger Typings for Separate Compilation of Java-like Languages (Extended Abstract). In *5th Intl. Workshop on Formal Techniques for Java Programs 2003*, July 2003.

[4] D. Ancona, G. Lagorio, and E. Zucca. A core calculus for Java exceptions. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*, SIGPLAN Notices. ACM Press, October 2001.

[5] D. Ancona and E. Zucca. Principal typings for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2004*, January 2004.

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

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

[8] 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.

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

[10] J.B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming 2002*, number 2380 in Lecture Notes in Computer Science, pages 913–925. Springer, 2002.

[7] It is of course undecidable to determine whether a statement is reachable in the general case.