Polymorphic Bytecode: Compositional Compilation for Java-like Languages

Davide Ancona
DISI - Università di Genova
davide@disi.unige.it

Ferruccio Damiani
Dip. di Informatica - Università di Torino
damiani@di.unito.it

Sophia Drossopoulou
Dep. of Computing - Imperial College
sd@doc.ic.ac.uk

Elena Zucca
DISI - Università di Genova
zucca@disi.unige.it

ABSTRACT

We define *compositional compilation* as the ability to typecheck source code fragments in isolation, generate corresponding binaries, and link together fragments whose mutual assumptions are satisfied, without reinspecting the code. Even though compositional compilation is a highly desirable feature, in Java-like languages it can hardly be achieved. This is due to the fact that the bytecode generated for a fragment (say, a class) is not uniquely determined by its source code, but also depends on the compilation context.

We propose a way to obtain compositional compilation for Java, by introducing a *polymorphic form of bytecode* containing type variables (ranging over class names) and equipped with a set of constraints involving type variables. Thus, polymorphic bytecode provides a representation for all the (standard) bytecode that can be obtained by replacing type variables with classes satisfying the associated constraints.

We illustrate our proposal by developing a typing and a linking algorithm. The typing algorithm compiles a class in isolation generating the corresponding polymorphic bytecode fragment and constraints on the classes it depends on. The linking algorithm takes a collection of polymorphic bytecode fragments, checks their mutual consistency, and possibly simplifies and specializes them. In particular, linking a self-contained collection of fragments either fails, or produces standard bytecode (the same as what would have been produced by standard compilation of all fragments).

Categories and Subject Descriptors: D.3.3[Programming languages]: Language constructs and features—*classes and objects*; D.3.1[Programming languages]: Formal definitions and theory—*syntax, semantics*; D.3.4[Programming languages]: Processors—*incremental compilers*

General Terms: languages, theory, design

POPL'05, January 12–14, 2005, Long Beach, California, USA.

Keywords: type systems, compositional analysis

1. INTRODUCTION

Compilers have two main tasks: To check that the source code adheres to the language rules (which usually means that it typechecks), and to produce target code.

Originally, compilers would process the complete source of an application; thus they would apply *global compilation*. In strongly typed languages execution of a globally compiled application is guaranteed to be type safe.

In the 70s, inspired by Parnas's ideas about abstract data types, languages like Mesa, LIS, Modula-2, and Ada introduced *separate compilation*, whereby an application would consist of *fragments* (e.g., modules, packages, or classes).[1] A fragment would be compiled separately in the context of other, *used* fragments. The produced target code fragment would reflect these used fragments, that is, it would reflect the compilation environment in which it was created.

An application would be put together through *safe linking* of target fragments; linking of target fragments was legal only with target fragments which corresponded to the compilation environment in which the former was created. Thus, linking preserved a correspondence between the compilation and the execution environment, and the ensuing application would correspond to a globally compiled one, and have the same type safety guarantees.

In recent years, Java and C♯ have adopted the separate compilation approach, however combined with *dynamic linking*, whereby fragments (in this case, classes in binary form) are loaded lazily at run-time. Thus, dynamic linking does not attempt to preserve a correspondence between the compilation and the execution environment, nor does the ensuing application correspond to a globally compiled one; type safety can only be achieved through runtime verification checks.

Thus, we argue that in Java (and C♯ too) there is a clash of philosophy between compilation and execution. Namely, the adoption of separate compilation means that the target fragments reflect the compilation environment in which they

[1]Separate compilation should not be confused with *independent compilation*, introduced by FORTRAN II, which does not use type information from the used fragments, and thus does not guarantee type-safety [15].

were created, while the adoption of dynamic linking means that there is no correspondence between compilation and execution environment.

For example, compilation of the source method declaration md^{s}:

$$\mathsf{E\ m(B\ x)\{\ return\ x.f1.f2;\ \}}$$

in an environment Δ_1 containing class B with field f1 of type C, and class C with field f2 of type E, generates bytecode $\mathsf{md}^{\mathsf{b}}_1$ with annotations which reflect the classes where fields were found and their types, that is:

$$\mathsf{E\ m(B\ x)\{\ \ return\ x[B.f1\ C][C.f2\ E];\ \}}$$

On the other hand, compilation of md^{s} in an environment Δ_2 containing a class B with a field f1 of type D, and a class D with a field f2 of type F, for some F subclass of E, generates a different bytecode $\mathsf{md}^{\mathsf{b}}_2$:

$$\mathsf{E\ m(B\ x)\{\ \ return\ x[B.f1\ D][D.f2\ F]\ \}}$$

More importantly, execution of $\mathsf{md}^{\mathsf{b}}_1$ in the environment Δ_2 will throw a FieldNotFoundException, even though compilation and subsequent execution of md^{s} in the environment Δ_2 would be successful.[2]

Thus, given the lazy nature of Java dynamic linking, separate compilation is, in some sense, too eager, and too context dependent.

In this paper, we consider, instead, *compositional compilation*, whereby target fragments do *not* reflect the compilation environment in which they were created, while linking produces an application which corresponds to a globally compiled one, and which, therefore, has all its type safety guarantees.

We define compositional compilation to be the ability to typecheck source code fragments in isolation, to generate corresponding binaries, and to link together fragments whose mutual assumptions are satisfied, without reinspecting the code.

We illustrate a new approach to compilation and linking for Java-like languages, which will support compositional compilation. We propose a *polymorphic form of bytecode* containing *type variables* (ranging over class names) and equipped with a set of constraints involving type variables. Thus, polymorphic bytecode provides a representation for all the (standard Java) bytecode that can be obtained by replacing type variables with class names satisfying the associated constraints.

In terms of our example, md^{s} can be compiled in isolation; the set of polymorphic constraints associated with md^{s} is $\{\ \phi(\mathsf{B},\mathsf{f1},\alpha), \phi(\alpha,\mathsf{f2},\beta), \beta \leq \mathsf{E}\ \}$, where α, β are type variables, and a constraint of the shape $\phi(\mathtt{t},\mathtt{f},\mathtt{t}')$ expresses that type $\mathtt{t}$ is expected to declare or inherit a field $\mathtt{f}$ of type $\mathtt{t}'$. Furthermore, the following polymorphic bytecode md^{b} would be generated:

$$\mathsf{E\ m(B\ x)\{\ \ return\ x[B.f1}\ \alpha][\alpha\mathsf{.f2}\ \beta];\ \}$$

Our approach also supports linking, which checks whether the polymorphic bytecode of the various fragments satisfies each other's requirements, without inspecting the code itself. The process involves the replacement of some type variables by concrete class names. In our example, linking md^{b} in the environment Δ_1 leads to $\mathsf{md}^{\mathsf{b}}_1$, linking md^{b} in the environment Δ_2 leads to $\mathsf{md}^{\mathsf{b}}_2$.

The rest of the paper is organized as follows. In Section 2 we define a schema formalizing global and compositional compilation, introduce the notions of soundness and completeness of compositional compilation w.r.t. global compilation, and give sufficient conditions for guaranteeing them. These conditions place requirements on the linking process, and on the relation between global and compositional compilation of *one* class. In Sections 3 and 4 we instantiate the schema to model global compilation and compositional compilation for a small Java-like language [9]. In Section 5 we give an algorithm for linking which satisfies the properties required by the theorems in Section 2. Finally, in Section 6 we discuss related work and in the Conclusion applicability of our approach and further work.

A preliminary version of the material from this paper appeared in [1]. Proofs of the main results can be found in an extended version of this work [2].

2. FORMALIZING COMPILATION

In this section we define a schema formalizing both global and compositional compilation.

We start by listing the basic syntactic categories and judgments such a type system should define. We use a Java-oriented terminology, since a significant class of languages on which the schema could be instantiated are Java-like (in particular, in the next sections we present an instance which defines global and compositional compilation for Featherweight Java [9]). However, the schema is much more general, and is appropriate for any language where, roughly speaking, generated binary code is context-dependent. Hence, "class" below can be thought of, in general terms, as "language entity".

- Source and binary class declarations (s and b).
- Source fragments (S), which are sequences of source class declarations; and binary fragments (B), which are sequences of binary class declarations.
- Class type environments (Δ), which are sequences of class type assignments (δ). A class type assignment can be thought of as the type information which can be extracted from a class declaration (hence the metavariables Δ and δ); thus a class type environment corresponds to a sequence of source class declarations deprived of bodies.
- Global compilation (of a class), $\Delta \vdash_g \mathsf{s} : \delta \mid \mathsf{b}$, to be read: The source class declaration s has type δ and compiles to b in the class type environment Δ.
- Type constraint environments (Γ), which are sequences of type constraints (γ). A type constraint expresses expectations from used classes, e.g., that a given class has a field of a given type.
- Compositional compilation (of a class), $\vdash_c \mathsf{s} : \delta \mid \Gamma \mid \mathsf{b}$, to be read: The source class declaration s has type δ and compiles to b under the type constraints in Γ.

[2]This example shows, in a sense, the most simple case of dependency of generated bytecode on the context; Java features such as method overloading and field hiding complicate matters even further, but the essence of the problem remains the same.

- Linking, $\Delta \vdash \Gamma \mid \mathsf{B} \rightsquigarrow \Gamma' \mid \mathsf{B}'$, to be read: In the class type environment Δ the type constraints Γ are simplified into Γ', and the binary fragment B becomes B'.

Empty class type environments and empty type constraint environments will be denoted by Λ.

The ingredients from above model two different approaches to compilation.

In the first approach, code fragments are compiled in the context of full type information on other fragments (formalized by the class type environment Δ), as shown by the rules in Fig.1. The global compilation (of a fragment) judgment $\Delta' \vdash_G \mathsf{S} : \Delta \mid \mathsf{B}$ means that in Δ' the fragment S has type Δ and compiles to B. Here Δ' represents (full) type information related to classes which are not being compiled, such as libraries. In particular, $\Lambda \vdash_G \mathsf{S} : \Delta \mid \mathsf{B}$ models the compilation of a self-contained fragment.

$$(g\text{-}frag1)\ \frac{\Delta\, \delta \vdash_g \mathsf{s} : \delta \mid \mathsf{b}}{\Delta \vdash_G \mathsf{s} : \delta \mid \mathsf{b}}$$

$$(g\text{-}frag2)\ \frac{\Delta\, \Delta_1 \ldots \Delta_{i-1}\, \Delta_{i+1} \ldots \Delta_n \vdash_G \mathsf{S}_i : \Delta_i \mid \mathsf{B}_i\ \forall i \in 1..n}{\Delta \vdash_G \mathsf{S}_1 \ldots \mathsf{S}_n : \Delta_1 \ldots \Delta_n \mid \mathsf{B}_1 \ldots \mathsf{B}_n}\ n \geq 2$$

Figure 1: Global compilation of a fragment

In the second approach, code fragments can be compiled in isolation producing binary code equipped with type constraints (both the binary code and the type constraints might contain type variables denoting yet unknown classes). Then, it is possible to *link* together a successfully compiled collection of fragments, obtaining, if their mutual requirements are compatible, a new binary fragment with simplified type constraints. Notice that this check does *not* depend on the source code. This is shown by the rules in Fig.2. The compositional compilation (of a fragment) judgment $\vdash_C \mathsf{S} : \Delta \mid \Gamma \mid \mathsf{B}$ means that the source fragment S has type Δ and compiles to B under the type constraints in Γ.

$$(c\text{-}frag1)\ \frac{\vdash_c \mathsf{s} : \delta \mid \Gamma' \mid \mathsf{b}' \qquad \delta \vdash \Gamma' \mid \mathsf{b}' \rightsquigarrow \Gamma \mid \mathsf{b}}{\vdash_C \mathsf{s} : \delta \mid \Gamma \mid \mathsf{b}}$$

$$(c\text{-}frag2)\ \frac{\vdash_C \mathsf{S}_i : \Delta_i \mid \Gamma'_i \mid \mathsf{B}'_i\ \forall i \in 1..n \qquad \Delta_1 \ldots \Delta_n \vdash \Gamma' \mid \mathsf{B}' \rightsquigarrow \Gamma \mid \mathsf{B}}{\vdash_C \mathsf{S}_1 \ldots \mathsf{S}_n : \Delta_1 \ldots \Delta_n \mid \Gamma \mid \mathsf{B}}\ \begin{array}{l} n \geq 2 \\ (\Gamma', \mathsf{B}') = \overset{i \in 1..n}{\oplus} (\Gamma'_i, \mathsf{B}'_i) \end{array}$$

Figure 2: Compositional compilation of a fragment

In rule (c-frag2), we assume an operator $\oplus$ which, given a sequence of (at least two) pairs consisting of a type constraint environment and a binary fragment, gives a new pair, intuitively obtained by combining them avoiding interferences (typically, this operator will eliminate clashes in type variables, through α-renaming.)

Iterating this process, we will eventually obtain a fragment for which no type constraints are left, that is, a judgment $\vdash_C \mathsf{S} : \Delta \mid \Lambda \mid \mathsf{B}$. This means that we have obtained a self-contained fragment. In this case, we expect to have obtained the same result as global compilation, that is, compositional compilation to be *sound* w.r.t. global compilation, and conversely, that is, compositional compilation to be *complete* w.r.t. global compilation. As a first approximation, soundness and completeness could be expressed as follows:

$$\vdash_C \mathsf{S} : \Delta \mid \Lambda \mid \mathsf{B} \text{ if and only if } \Lambda \vdash_G \mathsf{S} : \Delta \mid \mathsf{B}.$$

However, since compositional compilation is obviously expected to be incremental, and since global compilation allows for the import of some library classes, whose binary may be available, but whose source may not be available, the claim above should be generalized in order to deal with open source fragments, that is, fragments where some needed class is missing.

We say that a class type environment Δ is *consistent* (w.r.t. global compilation) if $\Lambda \vdash_G \mathsf{S} : \Delta \mid \mathsf{B}$ for some S, B.[3]

Definition 1. *We say that compositional compilation is* sound *w.r.t. global compilation iff for consistent* Δ' Δ:
$\vdash_C \mathsf{S} : \Delta \mid \Gamma \mid \mathsf{B}$, *and* $\Delta' \Delta \vdash \Gamma \mid \mathsf{B} \rightsquigarrow \Lambda \mid \mathsf{B}'$
imply $\Delta' \vdash_G \mathsf{S} : \Delta \mid \mathsf{B}'$.

We now state sufficient conditions for soundness. These conditions (and analogously those which guarantee completeness later on) provide a general schema for proving that compositional compilation is sound (resp. complete). Indeed, they allow to split the proof in two parts: First, checking that compositional compilation of a single class is sound and complete — conditions (1) in Theorems 4 and 6. Second, checking that linking satisfies some requirements of well-behaviour — conditions (2) and (3) in Theorems 4 and 6. In particular, conditions (2) are related to vertical (sequential) composition and conditions (3) to horizontal composition of linking steps. Conditions (2) require that, whenever two linking steps in sequence allow to eliminate all type constraints, and the latter requires more type information on classes, then starting with this richer type information the two steps can be combined in just one step; conversely, given a linking step which allows to eliminate all type constraints under some type information on classes, if we start with only partial type information it is always possible to perform a partial simplification of type constraints. Conditions (3) require that, whenever a linking step allows to eliminate a combination of type constraint environments, it is always possible to eliminate each component, and conversely. Finally, condition (4) in Theorem 4 is just a trivial requirement needed to express soundness for closed fragments as a particular case of soundness.

Note that, in order to prove soundness and completeness, it is not necessary that linking is well-behaved for *all* type constraint environments, but only for those which can be obtained through the compilation and linking process. This is formalized by the following definition:

Definition 2. Relevant *type constraint environments are inductively defined by the following clauses:*

- *if* $\vdash_c \mathsf{s} : \delta \mid \Gamma \mid \mathsf{b}$ *holds for some* $\mathsf{s}, \delta, \mathsf{b}$, *then* Γ *is relevant;*

[3]This definition assumes the fact that libraries used in global compilation have been obtained by correctly compiling some code.

- *if* Γ *is relevant and* $\Delta \vdash \Gamma \mid \mathsf{B} \rightsquigarrow \Gamma' \mid \mathsf{B}'$ *holds for some* $\Delta, \mathsf{B}, \mathsf{B}'$*, then* Γ' *is relevant;*
- *if* $\Gamma_1, ..., \Gamma_n$ *are relevant and* $(\Gamma,\mathsf{B}) = \overset{i\in 1..n}{\oplus} (\Gamma_i, \mathsf{B}_i)$ *for some* $\mathsf{B}, \mathsf{B}_1, ..., \mathsf{B}_n$*, then* Γ *is relevant.*

Fact 3. *If* $\vdash_C \mathsf{S} \ : \ \Delta \mid \Gamma \mid \mathsf{B}$ *holds for some* $\mathsf{S}, \Delta, \mathsf{B}$*, then* Γ *is relevant.*

Theorem 4 (Sufficient conditions for soundness). *Compositional compilation is sound w.r.t. global compilation if the following conditions hold:*

1. $\vdash_c \mathsf{s} \ : \ \delta \mid \Gamma' \mid \mathsf{b}'$*, and* $\Delta\,\delta \vdash \Gamma' \mid \mathsf{b}' \rightsquigarrow \Lambda \mid \mathsf{b}$*, and* $\Delta\,\delta$ *consistent imply* $\Delta\,\delta \vdash_g \mathsf{s} \ : \ \delta \mid \mathsf{b}$.
2. Γ'' *relevant,* $\Delta_2 \vdash \Gamma'' \mid \mathsf{B}'' \rightsquigarrow \Gamma' \mid \mathsf{B}'$*, and* $\Delta_1\Delta_2 \vdash \Gamma' \mid \mathsf{B}' \rightsquigarrow \Lambda \mid \mathsf{B}$ *imply* $\Delta_1\,\Delta_2 \vdash \Gamma'' \mid \mathsf{B}'' \rightsquigarrow \Lambda \mid \mathsf{B}$.
3. $\Delta \vdash \Gamma \mid \mathsf{B} \rightsquigarrow \Lambda \mid \mathsf{B}''$*, and* $(\Gamma,\mathsf{B}) = \overset{i\in 1..n}{\oplus} (\Gamma_i, \mathsf{B}_i)$ *for some* $\mathsf{B}_1, ..., \mathsf{B}_n$ *and* $\Gamma_1, ..., \Gamma_n$ *relevant, implies* $\Delta \vdash \Gamma_i \mid \mathsf{B}_i \rightsquigarrow \Lambda \mid \mathsf{B}'_i$*, for some* B'_i*, for all* $i \in 1..n$*, and* $\mathsf{B}'' = \mathsf{B}'_1 \ldots \mathsf{B}'_n$.
4. $\Lambda \vdash \Lambda \mid \mathsf{B} \rightsquigarrow \Lambda \mid \mathsf{B}$ *for all* B.

Definition 5. *We say that compositional compilation is* complete *w.r.t. global compilation iff*
$\Delta' \vdash_G \mathsf{S} \ : \ \Delta \mid \mathsf{B}'$ *implies* $\exists\ \mathsf{B}, \Gamma$ *s.t.*
$\vdash_C \mathsf{S} \ : \ \Delta \mid \Gamma \mid \mathsf{B}$ *and* $\Delta' \Delta \vdash \Gamma \mid \mathsf{B} \rightsquigarrow \Lambda \mid \mathsf{B}'$.

Theorem 6 (Sufficient conditions for completeness). *Compositional compilation is complete w.r.t. global compilation if the following conditions hold:*

1. $\Delta\,\delta \vdash_g \mathsf{s} \ : \ \delta \mid \mathsf{b}$ *implies* $\exists\ \mathsf{b}', \Gamma'$ *s.t.* $\vdash_c \mathsf{s} \ : \ \delta \mid \Gamma' \mid \mathsf{b}'$ *and* $\Delta\,\delta \vdash \Gamma' \mid \mathsf{b}' \rightsquigarrow \Lambda \mid \mathsf{b}$.
2. Γ'' *relevant and* $\Delta_1\,\Delta_2 \vdash \Gamma'' \mid \mathsf{B}'' \rightsquigarrow \Lambda \mid \mathsf{B}$ *imply* $\exists\ \Gamma', \mathsf{B}'$ *s.t.* $\Delta_2 \vdash \Gamma'' \mid \mathsf{B}'' \rightsquigarrow \Gamma' \mid \mathsf{B}'$ *and* $\Delta_1\,\Delta_2 \vdash \Gamma' \mid \mathsf{B}' \rightsquigarrow \Lambda \mid \mathsf{B}$.
3. Γ_i *relevant and* $\Delta \vdash \Gamma_i \mid \mathsf{B}_i \rightsquigarrow \Lambda \mid \mathsf{B}'_i$*, for* $i \in 1..n$*, with* $\overset{i\in 1..n}{\oplus} (\Gamma_i, \mathsf{B}_i) = (\Gamma, \mathsf{B})$*, imply* $\Delta \vdash \Gamma \mid \mathsf{B} \rightsquigarrow \Lambda \mid \mathsf{B}'_1 \ldots \mathsf{B}'_n$.

Note that theorems 4 and 6 imply soundness and completeness for closed fragments. Namely, if we take $\Delta' = \Lambda = \Gamma$, applying (4) from theorem 4 we get soundness for closed fragments:

$$\vdash_C \mathsf{S} \ : \ \Delta \mid \Lambda \mid \mathsf{B} \text{ implies } \Lambda \vdash_G \mathsf{S} \ : \ \Delta \mid \mathsf{B}.$$

On the other hand, for $\Delta' = \Lambda$, if there exist B, Γ s.t. $\vdash_C \mathsf{S} \ : \ \Delta \mid \Gamma \mid \mathsf{B}$ and $\Delta \vdash \Gamma \mid \mathsf{B} \rightsquigarrow \Lambda \mid \mathsf{B}'$, we get $\vdash_C \mathsf{S} \ : \ \Delta \mid \Lambda \mid \mathsf{B}'$ by applying rule (c-frag2), and hence we obtain completeness for closed fragments:

$$\Lambda \vdash_G \mathsf{S} \ : \ \Delta \mid \mathsf{B} \text{ implies } \vdash_C \mathsf{S} \ : \ \Delta \mid \Lambda \mid \mathsf{B}.$$

3. FJ GLOBAL COMPILATION

In this section, we formalize global compilation of a small Java-like language. This models both standard type checking for Java-like languages (see, e.g., [9, 16]), as well as bytecode generation (as already in, e.g., [4, 8]).

The syntax of the (source) language is defined in Fig.3. It is basically Featherweight Java [9] (FJ in the sequel) hence a functional subset of Java with no primitive types, except for the minor difference that here class constructors are implicitly declared. Every class can contain instance field and method declarations and has only one constructor whose parameters correspond to all class fields (both inherited and declared) in the order of declaration. In class declarations we assume that the name of the class c cannot be `Object`. Method overloading and field hiding are not supported. Expressions are variables, field access, method invocation, instance creation and casting; the keyword `this` is considered a special variable. Finally, in order to simplify the presentation, we assume field names in fds, method names in $\mathsf{mds}^{\mathsf{s}}$, parameter names in mh to be distinct.

$$\begin{array}{rcl}
\mathsf{S} & ::= & \mathsf{s}_1 \ldots \mathsf{s}_n \\
\mathsf{s} & ::= & \texttt{class c extends c}' \ \{ \ \mathsf{fds}\ \mathsf{mds}^{\mathsf{s}} \ \} \quad (\mathsf{c} \neq \texttt{Object}) \\
\mathsf{fds} & ::= & \mathsf{c}_1\ \mathsf{f}_1\,; \ldots \mathsf{c}_n\ \mathsf{f}_n\,; \\
\mathsf{mds}^{\mathsf{s}} & ::= & \mathsf{md}^{\mathsf{s}}_1 \ldots \mathsf{md}^{\mathsf{s}}_n \\
\mathsf{md}^{\mathsf{s}} & ::= & \mathsf{mh}\ \{\texttt{return}\ \mathsf{e}^{\mathsf{s}}\,;\} \\
\mathsf{mh} & ::= & \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1 \ldots \mathsf{c}_n\ \mathsf{x}_n) \\
\mathsf{e}^{\mathsf{s}} & ::= & \mathsf{x} \mid \mathsf{e}^{\mathsf{s}}.\mathsf{f} \mid \mathsf{e}^{\mathsf{s}}_0.\mathsf{m}(\mathsf{e}^{\mathsf{s}}_1 \ldots \mathsf{e}^{\mathsf{s}}_n) \mid \\
 & & \texttt{new}\ \mathsf{c}(\mathsf{e}^{\mathsf{s}}_1 \ldots \mathsf{e}^{\mathsf{s}}_n) \mid (\mathsf{c})\mathsf{e}^{\mathsf{s}}
\end{array}$$

where field, method and parameter names in fds, $\mathsf{mds}^{\mathsf{s}}$ and mh are distinct

Figure 3: Source language

In Fig.4 we give the syntax of bytecode generated by global compilation — that is (an abstraction of) standard Java bytecode.

$$\begin{array}{rcl}
\mathsf{B} & ::= & \mathsf{b}_1 \ldots \mathsf{b}_n \\
\mathsf{b} & ::= & \texttt{class c extends c}' \ \{ \ \mathsf{fds}\ \mathsf{mds}^{\mathsf{b}} \ \} \\
\mathsf{mds}^{\mathsf{b}} & ::= & \mathsf{md}^{\mathsf{b}}_1 \ldots \mathsf{md}^{\mathsf{b}}_n \\
\mathsf{md}^{\mathsf{b}} & ::= & \mathsf{mh}\ \{\texttt{return}\ \mathsf{e}^{\mathsf{b}}\,;\} \\
\mathsf{e}^{\mathsf{b}} & ::= & \mathsf{x} \mid \mathsf{e}^{\mathsf{b}}[\mathsf{c.f}\ \mathsf{c}'] \mid \mathsf{e}^{\mathsf{b}}_0[\mathsf{c.m}(\bar{\mathsf{c}})\mathsf{c}'](\mathsf{e}^{\mathsf{b}}_1 \ldots \mathsf{e}^{\mathsf{b}}_n) \\
 & & \mid \texttt{new}\ [\mathsf{c}\ \bar{\mathsf{c}}](\mathsf{e}^{\mathsf{b}}_1 \ldots \mathsf{e}^{\mathsf{b}}_n) \mid (\mathsf{c})\mathsf{e}^{\mathsf{b}} \\
\bar{\mathsf{c}} & ::= & \mathsf{c}_1 \ldots \mathsf{c}_n
\end{array}$$

where fds and mh are defined in Fig.3 and method names in $\mathsf{mds}^{\mathsf{b}}$ are distinct

Figure 4: (Abstract) standard bytecode

Our notion of bytecode is abstract, since the only differences between source code and bytecode of interest here are the annotations needed by the JVM verifier — recall that in Java bytecode a field access is annotated with the static type of the receiver and the type of the field, a method invocation with the static type of the receiver, the type of the parameters and the return type, and an instance creation with the type of the parameters.

In Fig.5 we define class type assignments. A class type assignment collects the type information needed for compiling other classes which can be extracted from a class declaration; it is a 4-tuple consisting of the name of the class, the name of the parent class, the set of field signatures (type and name of declared fields) and the set of method signatures (return type, name and parameter types of declared methods).

We assume the existence of a function *type* extracting type

$$
\begin{array}{rcl}
\delta & ::= & (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \\
\mathsf{fss} & ::= & \{\mathsf{fs}_1, \ldots, \mathsf{fs}_n\} \\
\mathsf{fs} & ::= & \mathsf{c}\ \mathsf{f} \\
\mathsf{mss} & ::= & \{\mathsf{ms}_1, \ldots, \mathsf{ms}_n\} \\
\mathsf{ms} & ::= & \mathsf{c}\ \mathsf{m}(\bar{\mathsf{c}})
\end{array}
$$

Figure 5: Class type assignments

information from a source class declaration. So we will write *type*(fds) and *type*($\mathsf{mds}^{\mathsf{s}}$) to denote the set of field signatures and the set of method signatures extracted from the field declarations fds and from the method declarations $\mathsf{mds}^{\mathsf{s}}$, respectively. The straightforward definition of *type* has been omitted.

The typing rules defining global compilation of a class are given in Fig.6. They are standard rules analogous to those given in other type systems for Java-like languages [9, 8, 4]. We use the following auxiliary judgments:

- $\Delta; \mathsf{c} \vdash_g \mathsf{mds}^{\mathsf{s}} : \mathsf{mds}^{\mathsf{b}}$, meaning that method declaration(s) $\mathsf{mds}^{\mathsf{s}}$ in class type environment Δ and current class c (needed for assigning the right type to `this`) compile(s) to $\mathsf{mds}^{\mathsf{b}}$.
- $\Delta; \Pi \vdash_g \mathsf{e}^{\mathsf{s}} : \mathsf{c} \mid \mathsf{e}^{\mathsf{b}}$, meaning that expression e^{s} in class type environment Δ and local type environment Π (which maps `this` and method parameters to class names) has type c and compiles to e^{b}.

In rule (g-class), for compiling a class c in a class type environment Δ we check that Δ is well-formed (judgment $\vdash_g \Delta\diamond$), and compile each method body in Δ and current class c. A class type environment is well-formed if there are no multiple type assignments for the same class name, the inheritance relation is acyclic, each extended class is available, there is no field hiding, the Java rule on overriding is respected and there is no overloading (the last two conditions correspond to the requirement that a class may not declare a method with the same name and different return or parameter types as an inherited method). Note, however, that a class which is only used as a type may not have an assignment in Δ. This definition of well-formedness exactly models the checks which Java compilers perform on the needed compilation context.The formal definition of $\vdash_g \Delta\diamond$ can be found in [2].

It is easy to see that the notion of well-formed class type environment is the concrete version in the FJ case of the general notion of consistent class type environment introduced in Section 2 (see [2] for the details).

In rules for compiling expressions, we use an auxiliary judgment of the form $\Delta \vdash \gamma$, meaning that in the class type environment Δ the type constraint γ holds. Type constraints are listed in Fig.7. They have the following informal meaning:

- $\mathsf{c} \leq \mathsf{c}'$ means "c is a subtype of c'".
- $\exists\, \mathsf{c}$ means "c is defined".
- $\phi(\mathsf{c}, \mathsf{f}, \mathsf{c}')$ means "c provides field f with type c'".

$$
\begin{array}{rcl}
\gamma & ::= & \mathsf{c} \leq \mathsf{c}' \mid \exists\, \mathsf{c} \mid \phi(\mathsf{c}, \mathsf{f}, \mathsf{c}') \mid \mu(\mathsf{c}, \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{c}', \bar{\mathsf{c}}')) \\
 & & \mid \kappa(\mathsf{c}, \bar{\mathsf{c}}, \bar{\mathsf{c}}')
\end{array}
$$

Figure 7: Type constraints

- $\mu(\mathsf{c}, \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{c}', \bar{\mathsf{c}}'))$ means "c provides method m applicable to arguments of type $\bar{\mathsf{c}}$, with return type c' and parameters of type $\bar{\mathsf{c}}'$.
- $\kappa(\mathsf{c}, \bar{\mathsf{c}}, \bar{\mathsf{c}}')$ means "c provides constructor applicable to arguments of type $\bar{\mathsf{c}}$, with parameters of type $\bar{\mathsf{c}}'$".

Note that both the constraints $\mu(\mathsf{c}, \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{c}', \bar{\mathsf{c}}'))$ and $\kappa(\mathsf{c}, \bar{\mathsf{c}}, \bar{\mathsf{c}}')$ implicitly include the constraint $\bar{\mathsf{c}} \leq \bar{\mathsf{c}}'$.

The rules defining the judgment $\Delta \vdash \gamma$ (in Fig.8) are intuitive and almost self-explanatory. In rule *(ϕ-2)*, the side condition $\mathsf{f} \notin \mathsf{fss}$ means that f is not declared in fss; analogously, in rule *(μ-2)*, $\mathsf{m} \notin \mathsf{mss}$ means that m is not declared in mss. The type constraints in Fig.7 and the rules in Fig.8 are essentially a subset of those defined in [5] (where type constraints were called local type assumptions).

The rules for the compilation of a class in Fig.6, together with the general rules for the global compilation of a fragment, given in Fig.1, provide an instantiation to FJ of the global compilation schema introduced in Section 2.

4. FJ COMPOSITIONAL COMPILATION

In this section we formalize compositional compilation for the small Java-like language introduced in Section 3. Classes are compiled in isolation into polymorphic bytecode. That is, bytecode where the annotations may contain type variables denoting names of yet unknown classes. The syntax of polymorphic bytecode is described by the first four productions in Fig.4 (defining binary fragments, class declarations, method sequences, and methods, respectively) and by the productions in Fig.9 (defining binary expressions).

$$
\begin{array}{rcl}
\mathsf{e}^{\mathsf{b}} & ::= & \mathtt{x} \mid \mathsf{e}^{\mathsf{b}}[\mathtt{t.f}\ \mathtt{t}'] \mid \mathsf{e}^{\mathsf{b}}_0[\mathtt{t.m}(\bar{\mathtt{t}})\mathtt{t}'](\mathsf{e}^{\mathsf{b}}_1 \ldots \mathsf{e}^{\mathsf{b}}_n) \\
 & & \mid \mathtt{new}\ [\mathtt{c}\ \bar{\mathtt{t}}](\mathsf{e}^{\mathsf{b}}_1 \ldots \mathsf{e}^{\mathsf{b}}_n) \mid (\mathtt{c})\mathsf{e}^{\mathsf{b}} \mid \ll \mathtt{c}, \mathtt{t} \gg \mathsf{e}^{\mathsf{b}} \\
\mathtt{t} & ::= & \mathtt{c} \mid \alpha \\
\bar{\mathtt{t}} & ::= & \mathtt{t}_1 \ldots \mathtt{t}_n
\end{array}
$$

Figure 9: Polymorphic bytecode (expressions)

Besides the presence of type variables, the only difference between polymorphic and standard bytecode is the presence of the "polymorphic casting" annotated expression; if the type variable α is substituted with c', then $\ll \mathsf{c}, \alpha \gg \mathsf{e}^{\mathsf{b}}$ can be specialized either into e^{b}, if $\mathsf{c}' \leq \mathsf{c}$ holds (casting-up), or into $(\mathsf{c})\mathsf{e}^{\mathsf{b}}$, if $\mathsf{c} \leq \mathsf{c}'$ holds (casting-down), or into $\ll \mathsf{c}, \mathsf{c}' \gg \mathsf{e}^{\mathsf{b}}$ if nothing can be said about $\mathsf{c}' \leq \mathsf{c}$ and $\mathsf{c} \leq \mathsf{c}'$. For the "polymorphic casting" annotation we use a different notation (double angle brackets rather than parentheses) since this annotation is only allowed in polymorphic bytecode.

Polymorphic bytecode comes with a sequence of *polymorphic type constraints*, which involve type variables and class names. These are listed in Fig.10. As for polymorphic bytecode, the meta-variable $\mathtt{t}$ denotes either a type variable or a class name. Besides the presence of type variables, the only

$$\textit{(g-class)}\ \frac{\vdash_g \Delta\diamond \quad \Delta;\mathsf{c}\vdash_g \mathsf{mds}^{\mathsf{s}}\ :\ \mathsf{mds}^{\mathsf{b}}}{\Delta\vdash_g \texttt{class c extends c'}\ \{\mathsf{fds}\ \mathsf{mds}^{\mathsf{s}}\}\ :\ (\mathsf{c},\mathsf{c}',\mathsf{fss},\mathsf{mss})\mid \texttt{class c extends c'}\ \{\mathsf{fds}\ \mathsf{mds}^{\mathsf{b}}\}}\ \begin{array}{l}\mathit{type}(\mathsf{mds}^{\mathsf{s}})=\mathsf{mss}\\ \mathit{type}(\mathsf{fds})=\mathsf{fss}\end{array}$$

$$\textit{(g-methods)}\ \frac{\Delta;\mathsf{c}\vdash_g \mathsf{md}_i^{\mathsf{s}}\ :\ \mathsf{md}_i^{\mathsf{b}}\quad \forall i\in 1..n}{\Delta;\mathsf{c}\vdash_g \mathsf{md}_1^{\mathsf{s}}\ \ldots\ \mathsf{md}_n^{\mathsf{s}}\ :\ \mathsf{md}_1^{\mathsf{b}}\ldots\mathsf{md}_n^{\mathsf{b}}}\ n\neq 1$$

$$\textit{(g-method)}\ \frac{\Delta;\mathsf{x}_1{:}\mathsf{c}_1\ldots\mathsf{x}_n{:}\mathsf{c}_n,\texttt{this}{:}\mathsf{c}\vdash_g \mathsf{e}^{\mathsf{s}}\ :\ \mathsf{c}'\mid \mathsf{e}^{\mathsf{b}}}{\Delta;\mathsf{c}\vdash_g \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1\ldots\mathsf{c}_n\ \mathsf{x}_n)\ \{\texttt{return}\ \mathsf{e}^{\mathsf{s}}\texttt{;}\}\ :\ \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1\ldots\mathsf{c}_n\ \mathsf{x}_n)\ \{\texttt{return}\ \mathsf{e}^{\mathsf{b}}\texttt{;}\}}\ \begin{array}{l}\Delta\vdash \mathsf{c}'\leq \mathsf{c}_0\\ \Delta\vdash\exists\,\mathsf{c}_i\ \forall i\in 0..n\end{array}$$

$$\textit{(g-parameter)}\ \frac{\Pi\vdash \mathsf{x}:\mathsf{c}}{\Delta;\Pi\vdash_g \mathsf{x}\ :\ \mathsf{c}\mid \mathsf{x}}\qquad \textit{(g-field access)}\ \frac{\Delta;\Pi\vdash_g \mathsf{e}^{\mathsf{s}}\ :\ \mathsf{c}\mid \mathsf{e}^{\mathsf{b}}}{\Delta;\Pi\vdash_g \mathsf{e}^{\mathsf{s}}.\mathsf{f}\ :\ \mathsf{c}'\mid \mathsf{e}^{\mathsf{b}}[\mathsf{c}.\mathsf{f}\ \mathsf{c}']}\ \Delta\vdash\phi(\mathsf{c},\mathsf{f},\mathsf{c}')$$

$$\textit{(g-meth call)}\ \frac{\begin{array}{c}\Delta;\Pi\vdash_g \mathsf{e}_0^{\mathsf{s}}\ :\ \mathsf{c}_0\mid \mathsf{e}_0^{\mathsf{b}}\\ \Delta;\Pi\vdash_g \mathsf{e}_i^{\mathsf{s}}\ :\ \mathsf{c}_i\mid \mathsf{e}_i^{\mathsf{b}}\quad \forall i\in 1..n\end{array}}{\Delta;\Pi\vdash_g \mathsf{e}_0^{\mathsf{s}}.\mathsf{m}(\mathsf{e}_1^{\mathsf{s}},\ldots,\mathsf{e}_n^{\mathsf{s}})\ :\ \mathsf{c}\mid \mathsf{e}_0^{\mathsf{b}}[\mathsf{c}_0.\mathsf{m}(\bar{\mathsf{c}})\mathsf{c}](\mathsf{e}_0^{\mathsf{b}},\ldots,\mathsf{e}_n^{\mathsf{b}})}\ \Delta\vdash\mu(\mathsf{c}_0,\mathsf{m},(\mathsf{c}_1,\ldots,\mathsf{c}_n),(\mathsf{c},\bar{\mathsf{c}}))$$

$$\textit{(g-new)}\ \frac{\Delta;\Pi\vdash_g \mathsf{e}_i^{\mathsf{s}}\ :\ \mathsf{c}_i\mid \mathsf{e}_i^{\mathsf{b}}\quad \forall i\in 1..n}{\Delta;\Pi\vdash_g \texttt{new}\ \mathsf{c}(\mathsf{e}_1^{\mathsf{s}}\ldots\mathsf{e}_n^{\mathsf{s}})\ :\ \mathsf{c}\mid \texttt{new}\ [\mathsf{c}\ \bar{\mathsf{c}}](\mathsf{e}_1^{\mathsf{b}}\ldots\mathsf{e}_n^{\mathsf{b}})}\ \Delta\vdash\kappa(\mathsf{c},\mathsf{c}_1\ldots\mathsf{c}_n,\bar{\mathsf{c}})$$

$$\textit{(g-downcast)}\ \frac{\Delta;\Pi\vdash_g \mathsf{e}^{\mathsf{s}}\ :\ \mathsf{c}'\mid \mathsf{e}^{\mathsf{b}}}{\Delta;\Pi\vdash_g (\mathsf{c})\mathsf{e}^{\mathsf{s}}\ :\ \mathsf{c}\mid (\mathsf{c})\mathsf{e}^{\mathsf{b}}}\ \Delta\vdash\mathsf{c}\leq\mathsf{c}'\qquad \textit{(g-upcast)}\ \frac{\Delta;\Pi\vdash_g \mathsf{e}^{\mathsf{s}}\ :\ \mathsf{c}'\mid \mathsf{e}^{\mathsf{b}}}{\Delta;\Pi\vdash_g (\mathsf{c})\mathsf{e}^{\mathsf{s}}\ :\ \mathsf{c}\mid \mathsf{e}^{\mathsf{b}}}\ \Delta\vdash\mathsf{c}'\leq\mathsf{c}$$

Figure 6: Global compilation

$$\gamma\ ::=\ \mathsf{t}\leq\mathsf{t}'\ \mid\exists\,\mathsf{c}\ \mid\phi(\mathsf{t},\mathsf{f},\mathsf{t}')\mid\mu(\mathsf{t},\mathsf{m},\bar{\mathsf{t}},(\mathsf{t}',\bar{\mathsf{t}}'))\ \mid\kappa(\mathsf{c},\bar{\mathsf{t}},\bar{\mathsf{t}}')\mid\mathsf{c}\sim\mathsf{t}$$

Figure 10: Polymorphic type constraints

difference between the polymorphic type constraints and the type constrains listed in Fig.7 is the presence of the last constraint in Fig.10, whose informal meaning is the following:

- $\mathsf{c}\sim\mathsf{t}$ means "c and t are comparable" (this constraint will be generated when compiling a cast).

Polymorphic type constraints not containing type variables will be called *monomorphic type constraints*.

The rules defining the judgement for compositional compilation of classes are given in Fig.11. We use the following auxiliary judgments:

- $\mathsf{c}\vdash_c \mathsf{mds}^{\mathsf{s}}\ :\ \Gamma\mid\mathsf{mds}^{\mathsf{b}}$, meaning that method declaration(s) $\mathsf{mds}^{\mathsf{s}}$ in current class c (needed for assigning the right type to `this`) compile to $\mathsf{mds}^{\mathsf{b}}$ under the polymorphic type constraints in Γ.
- $\Pi\vdash_c \mathsf{e}^{\mathsf{s}}\ :\ \mathsf{t}\mid\Gamma\mid\mathsf{e}^{\mathsf{b}}$, meaning that expression e^{s} in local type environment Π (which maps `this` and method parameters to class names) has type t and compiles to e^{b} under the polymorphic type constraints in Γ.

The intuition behind the compositional compilation rules is that they extract the polymorphic type constraints Γ necessary to compile a given source fragment into a certain polymorphic binary fragment.

However, note that the rules do not check whether the inferred collection of constraints Γ is actually satisfiable; indeed, for any fragment it is possible to derive a judgment, even for those that are not statically correct. Consistency checks are performed by the rule for the linking judgment (see below). This approach has the advantage that the typing rules for separate compilation are very simple and can be implemented in a straightforward way.

Note also that in the type system a unique judgment can be derived for any class declaration (the proof is immediate); therefore, we can easily define a *type inference* algorithm, that is, an effective way for deducing just from the single declaration of a class c the type and the (polymorphic) bytecode of c, and the required type constraints. This is not possible for the systems in [4, 5, 3], where one needs to know the environment where c is compiled.

We now define the linking judgment $\Delta\vdash\Gamma\mid\mathsf{B}\rightsquigarrow\Gamma'\mid\mathsf{B}'$. Linking a fragment of polymorphic bytecode B, equipped with polymorphic type constraints Γ, in a given class type environment Δ, amounts to finding a suitable substitution σ mapping the type variables into class names. The substitution σ instantiates some polymorphic type constraints in Γ into monomorphic type constraints that hold in Δ, and instantiates variables in B correspondingly; these constraints can be eliminated, leaving only the constraints in Γ'. In particular, when all constraints are eliminated, we obtain a fragment of standard bytecode (like the one produced by global compilation).

Instantiation of Γ w.r.t. substitution σ is denoted by $\sigma(\Gamma)$; we have omitted the trivial inductive definition which coincides with conventional variable substitution. Instantiation of B w.r.t. Δ and σ is denoted by $\mathbf{I}_\Delta^\sigma(\mathsf{B})$; Δ is needed for

$$(\Gamma)\frac{\Delta \vdash \gamma \ \forall \gamma \in \Gamma}{\Delta \vdash \Gamma} \qquad (\exists)\frac{}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \exists\, \mathsf{c}} \qquad (\exists\text{-}Obj)\frac{}{\Delta \vdash \exists\, \mathtt{Object}}$$

$$(\leq\text{-}refl)\frac{}{\Delta \vdash \mathsf{c} \leq \mathsf{c}} \qquad (\leq\text{-}trans)\frac{\Delta, (\mathsf{c}_1, \mathsf{c}_2, \mathsf{fss}, \mathsf{mss}) \vdash \mathsf{c}_2 \leq \mathsf{c}_3}{\Delta \vdash \mathsf{c}_1 \leq \mathsf{c}_3} \qquad (\leq\text{-}Obj)\frac{}{\Delta \vdash \mathsf{c} \leq \mathtt{Object}}$$

$$(\phi\text{-}1)\frac{}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \phi(\mathsf{c}, \mathsf{f}, \mathsf{c}'')}\ \mathsf{c}'' \ \mathsf{f} \in \mathsf{fss} \qquad (\phi\text{-}2)\frac{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \phi(\mathsf{c}', \mathsf{f}, \mathsf{c}'')}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \phi(\mathsf{c}, \mathsf{f}, \mathsf{c}'')}\ \mathsf{f} \notin \mathsf{fss}$$

$$(\mu\text{-}1)\frac{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \mathsf{c}_i \leq \mathsf{c}_i'' \ \forall\, i \in 1..n}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \mu(\mathsf{c}, \mathsf{m}, \mathsf{c}_1 \ldots \mathsf{c}_n, (\mathsf{c}'', \mathsf{c}_1'' \ldots \mathsf{c}_n''))}\ \mathsf{c}''\ \mathsf{m}(\mathsf{c}_1'' \ldots \mathsf{c}_n'') \in \mathsf{mss}$$

$$(\mu\text{-}2)\frac{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \mu(\mathsf{c}', \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{c}'', \bar{\mathsf{c}}''))}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \mu(\mathsf{c}, \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{c}'', \bar{\mathsf{c}}''))}\ \mathsf{m} \notin \mathsf{mss}$$

$$(\kappa\text{-}1)\frac{}{\Delta \vdash \kappa(\mathtt{Object}, \epsilon, \epsilon)} \qquad (\kappa\text{-}2)\frac{\begin{array}{c}\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \kappa(\mathsf{c}', \mathsf{c}_1' \ldots \mathsf{c}_k', \mathsf{c}_1 \ldots \mathsf{c}_k) \\ \Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \mathsf{c}_i' \leq \mathsf{c}_i \ \forall\, i \in k+1..n\end{array}}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash \kappa(\mathsf{c}, \mathsf{c}_1' \ldots \mathsf{c}_n', \mathsf{c}_1 \ldots \mathsf{c}_n)}\ \mathsf{fss} = \{\mathsf{c}_{k+1}\ \mathsf{f}_{k+1}, \ldots, \mathsf{c}_n\ \mathsf{f}_n\}$$

Figure 8: Entailment judgement $\Delta \vdash \gamma$ (rules for the type constraints in Fig.7)

$$(\sim\text{-}1)\frac{\Delta \vdash \mathsf{c} \leq \mathsf{c}'}{\Delta \vdash \mathsf{c} \sim \mathsf{c}'} \qquad (\sim\text{-}2)\frac{\Delta \vdash \mathsf{c} \sim \mathsf{c}'}{\Delta \vdash \mathsf{c}' \sim \mathsf{c}}$$

Figure 12: Entailment judgement $\Delta \vdash \gamma$ (rules for the type constraint $\mathsf{c}_1 \sim \mathsf{c}_2$)

dealing with the case $\ll \mathsf{c}, \mathsf{t} \gg \mathsf{e}^{\mathsf{b}}$:

$$\mathbf{I}^{\sigma}_{\Delta}(\ll \mathsf{c}, \mathsf{t} \gg \mathsf{e}^{\mathsf{b}}) = \begin{cases} \mathbf{I}^{\sigma}_{\Delta}(\mathsf{e}^{\mathsf{b}}) & \text{if } \sigma(\mathsf{t}) = \mathsf{c}' \text{ and } \Delta \vdash \mathsf{c}' \leq \mathsf{c} \\ (\mathsf{c})\mathbf{I}^{\sigma}_{\Delta}(\mathsf{e}^{\mathsf{b}}) & \text{if } \sigma(\mathsf{t}) = \mathsf{c}' \text{ and } \Delta \vdash \mathsf{c} \leq \mathsf{c}' \\ \ll \mathsf{c}, \sigma(\mathsf{t}) \gg \mathbf{I}^{\sigma}_{\Delta}(\mathsf{e}^{\mathsf{b}}) & \text{otherwise.} \end{cases}$$

In all other cases instantiation of polymorphic bytecode corresponds to variable substitution.

Remark. Note that the ability of instantiating polymorphic bytecode into different kinds of standard bytecode instructions is a key feature for dealing with important issues, such as resolution of Java syntax ambiguities (see the Conclusion for an example of such ambiguities) and, in a wider context, code optimization.

The fact that the monomorphic type constraint γ holds in the class type environment Δ is expressed by the judgement $\Delta \vdash \gamma$, which is defined by the rules in Fig.8 and in Fig.12.

$$(c\text{-}linking)\frac{\begin{array}{c}\vdash_c \Delta\diamond \\ \Delta \vdash \Gamma \rightarrow_{ls} \sigma \mid \Gamma'\end{array}}{\Delta \vdash \Gamma \mid \mathsf{B} \rightsquigarrow \Gamma' \mid \mathbf{I}^{\sigma}_{\Delta}(\mathsf{B})}$$

Figure 13: Linking

The linking judgement is defined by rule (c-linking) in Fig.13, which is parameterized w.r.t. a *linking-simplification relation* $\rightarrow_{ls}$. A linking-simplification relation models a particular way of finding suitable substitutions for simplifying type constraints w.r.t. class type environments.

The formal definition of $\vdash_c \Delta\diamond$ (well-formed type environments for compositional compilation) can be found in [2]. The judgment $\vdash_c \Delta\diamond$ is more liberal than $\vdash_g \Delta\diamond$, since it allows extended classes to be undefined in Δ. Indeed, the check $\vdash_g \Delta\diamond$ in global compilation (rule (g-class)) models the behaviour of standard Java compilers. On the other hand, the premise $\vdash_c \Delta\diamond$ in rule (c-linking) corresponds to an intrinsic notion of consistency, and expresses that some additional constraints on each compiled class c are satisfied in Δ, that is, that there is no other declaration for c, that the parent class does not have conflicting field/method declarations, and that the parent class is not a subtype of c. These constraints could be equivalently generated during compositional compilation of class c, as done in [4, 5, 3]; here we preferred a more compact and efficient formulation. Note that for FJ the check of $\exists\, \mathsf{c}$ constraints could also be moved to be part of well-formedness of class type environment, since used classes are always mentioned in field or method declarations; but this is not true in general, for instance if we have local variables.

The rules for the compilation of a class in Fig.11 and the rule for linking in Fig.13, together with the general rules for the compositional compilation of a fragment, given in Fig.2, provide an instantiation to FJ of the compositional compilation schema introduced in Section 2. In this case the operator $\oplus$ used in rule (c-frag2) in Fig.2 in Section 2 just corresponds to pairwise concatenation of sequences (of constraints and binary fragments, respectively) with proper α-renaming in order to avoid conflicts of type variables.

Note that all the type variables occurring in a compositional compilation judgement are (implicitly) universally quantified. In fact, in a class compilation judgement $\vdash_c \mathsf{s} : \delta \mid \Gamma \mid \mathsf{b}$, the triple $\delta \mid \Gamma \mid \mathsf{b}$ represents all the standard binary class declarations of the form $\mathbf{I}^{\sigma}_{\Delta' \Delta}(\mathsf{b})$, for some class

$$\textit{(c-class)}\ \frac{\mathsf{c} \vdash_c \mathsf{mds^s} \ : \ \Gamma \mid \mathsf{mds^b}}{\vdash_c \mathtt{class\ c\ extends\ c'\ \{fds\ mds^s\}} \ : \ (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \mid \Gamma, \exists\, \mathsf{c}' \mid \mathtt{class\ c\ extends\ c'\ \{fds\ mds^b\}}}\ \begin{array}{l} \textit{type}(\mathsf{mds^s}) = \mathsf{mss} \\ \textit{type}(\mathsf{fds}) = \mathsf{fss} \end{array}$$

$$\textit{(c-methods)}\ \frac{\mathsf{c} \vdash_c \ \mathsf{md}_i^{\mathsf{s}} \ : \ \Gamma_i \mid \mathsf{md}_i^{\mathsf{b}} \qquad \forall i \in 1..n}{\mathsf{c} \vdash_c \mathsf{md}_1^{\mathsf{s}} \ \ldots \mathsf{md}_n^{\mathsf{s}} \ : \ \Gamma_1 \ldots \Gamma_n \mid \mathsf{md}_1^{\mathsf{b}} \ldots \mathsf{md}_n^{\mathsf{b}}}\ n \neq 1$$

$$\textit{(c-method)}\ \frac{\mathsf{x}_1{:}\mathsf{c}_1 \ldots \mathsf{x}_n{:}\mathsf{c}_n, \mathtt{this}{:}\mathsf{c} \vdash_c \mathsf{e^s} \ : \ \mathtt{t} \mid \Gamma \mid \mathsf{e^b}}{\mathsf{c} \vdash_c \ \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1 \ldots \mathsf{c}_n\ \mathsf{x}_n)\ \{\mathtt{return\ e^s;}\} \ : \ \Gamma, \mathtt{t} \leq \mathsf{c}_0, \exists\, \mathsf{c}_i{}^{i \in 0..n} \mid \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1 \ldots \mathsf{c}_n\ \mathsf{x}_n)\ \{\mathtt{return\ e^b;}\}}$$

$$\textit{(c-parameter)}\ \frac{\Pi \vdash \mathsf{x}{:}\mathsf{c}}{\Pi \vdash_c \mathsf{x} \ : \ \mathsf{c} \mid \Lambda \mid \mathsf{x}} \qquad \textit{(c-field access)}\ \frac{\Pi \vdash_c \mathsf{e^s} \ : \ \mathtt{t} \mid \Gamma \mid \mathsf{e^b}}{\Pi \vdash_c \mathsf{e^s.f} \ : \ \alpha \mid \Gamma, \phi(\mathtt{t}, \mathsf{f}, \alpha) \mid \mathsf{e^b}[\mathtt{t.f}\ \alpha]}\ \alpha \text{ fresh}$$

$$\textit{(c-meth call)}\ \frac{\begin{array}{l}\Pi \vdash_c \mathsf{e}_0^{\mathsf{s}} \ : \ \mathtt{t}_0 \mid \Gamma_0 \mid \mathsf{e}_0^{\mathsf{b}} \\ \Pi \vdash_c \mathsf{e}_i^{\mathsf{s}} \ : \ \mathtt{t}_i \mid \Gamma_i \mid \mathsf{e}_i^{\mathsf{b}} \quad \forall i \in 1..n\end{array}}{\Pi \vdash_c \mathsf{e}_0^{\mathsf{s}}.\mathsf{m}(\mathsf{e}_1^{\mathsf{s}} \ldots \mathsf{e}_n^{\mathsf{s}}) \ : \ \beta \mid \Gamma_0 \Gamma_1 \ldots \Gamma_n, \mu(\mathtt{t}_0, \mathsf{m}, \mathtt{t}_1 \ldots \mathtt{t}_n, (\beta, \bar{\alpha})) \mid \mathsf{e}_0^{\mathsf{b}}[\mathtt{t}_0.\mathsf{m}(\bar{\alpha})\beta](\mathsf{e}_1^{\mathsf{b}}, \ldots, \mathsf{e}_n^{\mathsf{b}})}\ \beta,\bar{\alpha} \text{ fresh}$$

$$\textit{(c-new)}\ \frac{\Pi \vdash_c \mathsf{e}_i^{\mathsf{s}} \ : \ \mathtt{t}_i \mid \Gamma_i \mid \mathsf{e}_i^{\mathsf{b}}\ \forall i \in 1..n}{\Pi \vdash_c \mathtt{new\ c}(\mathsf{e}_1^{\mathsf{s}} \ldots \mathsf{e}_n^{\mathsf{s}}) \ : \ \mathsf{c} \mid \Gamma_1 \ldots \Gamma_n, \kappa(\mathsf{c}, \mathtt{t}_1 \ldots \mathtt{t}_n, \bar{\alpha}) \mid \mathtt{new}\ [\mathsf{c}\ \bar{\alpha}](\mathsf{e}_1^{\mathsf{b}} \ldots \mathsf{e}_n^{\mathsf{b}})}\ \bar{\alpha} \text{ fresh}$$

$$\textit{(c-cast)}\ \frac{\Pi \vdash_c \mathsf{e^s} \ : \ \mathtt{t} \mid \Gamma \mid \mathsf{e^b}}{\Pi \vdash_c (\mathsf{c})\mathsf{e^s} \ : \ \mathsf{c} \mid \Gamma, \mathsf{c} \sim \mathtt{t} \mid \ll \mathsf{c}, \mathtt{t} \gg \mathsf{e^b}}$$

Figure 11: Compositional compilation

type environment Δ' and substitution σ such that $\vdash_c \Delta' \Delta\diamond$ and $\Delta' \Delta \vdash \sigma(\Gamma)$.

We prove now that, under suitable hypothesis on the linking-simplification relation, the compositional compilation which we defined for FJ can be safely used in place of global compilation, that is, is sound and complete w.r.t. global compilation in the sense of Def.1 and Def.5 in Section 2.

We start by stating some properties of entailment which will be needed in the proof and hold independently of the linking-simplification relation we choose. Let σ and σ' be two substitutions, that is, finite mappings from type variables to class names. Then $\sigma\,\sigma'$ denotes the unique substitution s.t. $(\sigma\,\sigma')(\Gamma) = \sigma(\sigma'(\Gamma))$ for all Γ; the empty substitution is denoted by ϵ. Finally, $\Gamma \setminus \Gamma'$ denotes the sequence of constraints obtained by removing from Γ all constraints in Γ', while $\mathit{Vars}(\Gamma)$ (resp. $\mathit{Vars}(\mathsf{b})$) denotes the set of type variables appearing in Γ (resp. in b).

Proposition 7. *If* $\vdash_g \Delta\,\delta\diamond$, $\Delta\,\delta \vdash \sigma(\Gamma)$, *then for all* s, b :

$$\vdash_c \mathsf{s} \ : \ \delta \mid \Gamma \mid \mathsf{b} \ \textit{implies}\ \Delta\,\delta \vdash_g \mathsf{s} \ : \ \delta \mid \mathbf{I}^{\sigma}_{\Delta\,\delta}(\mathsf{b}).$$

Proposition 8. *If* $\Delta\,\delta \vdash_g \mathsf{s} \ : \ \delta \mid \mathsf{b}$, *then there exist* $\sigma, \Gamma, \mathsf{b}'$ *s.t.* $\mathit{dom}(\sigma) \subseteq \mathit{Vars}(\Gamma)$, *and*

$$\vdash_c \mathsf{s} \ : \ \delta \mid \Gamma \mid \mathsf{b}',\ \Delta\,\delta \vdash \sigma(\Gamma),\ \mathsf{b} = \mathbf{I}^{\sigma}_{\Delta\,\delta}(\mathsf{b}').$$

Proposition 9 (Weakening of entailment). $\Delta \vdash \Gamma$ *implies* $\Delta\,\Delta' \vdash \Gamma$.

Proposition 10. $\vdash_c \Delta_1\Delta_2\diamond$ *implies* $\vdash_c \Delta_1\diamond$.

Note that Prop.10 does not hold for the judgment $\vdash_g \Delta\diamond$ defining well-formed type environments for global compilation; indeed, $\vdash_g \Delta\diamond$ holds only if the class hierarchy in Δ is (upward) complete.

In order to establish assumptions in Theorem 4 and Theorem 6, we require the simplification relation to satisfy the properties listed in Theorem 11 below, which have the following informal meaning.

$\rightarrow_{ls}$ **-sound** guarantees that the simplification step is sound, in the sense that, after applying the given substitution, it actually eliminates only type constraints which hold in the current class type environment.

$\rightarrow_{ls}$ **-complete-1** guarantees that, if Δ contains enough type information to satisfy all type constraints in Γ, then this simplification step must be possible in $\rightarrow_{ls}$.

$\rightarrow_{ls}$ **-complete-2** handles the case in which there is not enough type information in a class type environment, say Δ_2, to guarantee that all constraints in Γ hold. However, if it is possible to eliminate these type constraints in a larger class type environment $\Delta_1\,\Delta_2$, then it must be possible to partially simplify Γ in Δ_2, obtaining Γ'.

Note that the last requirement allows many different "strategies" for $\rightarrow_{ls}$ — including the strategy where simplification just does nothing ($\Gamma' = \Gamma$ in the above) until the class type environment contains sufficient information to simplify *all* type constraints, in which case it applies *all* the substitutions. However, algorithms implementing $\rightarrow_{ls}$ should attempt to find as many simplifications as possible at any step. We will present such an algorithm in the next section.

Theorem 11. *If the linking-simplification relation* $\rightarrow_{ls}$ *satisfies the following properties:*

$\rightarrow_{ls}$ **-sound**
Γ *relevant and* $\vdash_c \Delta\diamond$ *and* $\Delta \vdash \Gamma \rightarrow_{ls} \sigma \mid \Gamma'$ *imply* $\mathit{dom}(\sigma) \subseteq \mathit{Vars}(\Gamma)$ *and* $\Gamma' \subseteq \sigma(\Gamma)$ *and* $\Delta \vdash \sigma(\Gamma) \setminus \Gamma'$.

$\rightarrow_{ls}$ **-complete-1**
Γ *relevant and* $\vdash_c \Delta\diamond$ *and* $\Delta \vdash \sigma(\Gamma)$ *and* $dom(\sigma) \subseteq Vars(\Gamma)$ *imply* $\Delta \vdash \Gamma \rightarrow_{ls} \sigma \mid \Lambda$.

$\rightarrow_{ls}$ **-complete-2**
Γ *relevant and* $\vdash_c \Delta_1\,\Delta_2\diamond$ *and* $\Delta_1\,\Delta_2 \vdash \Gamma \rightarrow_{ls} \sigma \mid \Lambda$ *imply* $\exists\ \Gamma', \sigma', \sigma''$ *s.t. (1)* $\Delta_2 \vdash \Gamma \rightarrow_{ls} \sigma' \mid \Gamma'$*, (2)* $\Delta_1\,\Delta_2 \vdash \Gamma' \rightarrow_{ls} \sigma'' \mid \Lambda$*, and (3)* $\sigma = \sigma'\,\sigma''$.

Then, compositional compilation of FJ is sound and complete w.r.t. global compilation.

5. A LINKING ALGORITHM

In this section we describe a particular linking algorithm, thus making rule *c-linking* effective, and we sketch a proof that this algorithm is a correct implementation of the $\rightarrow_{ls}$ relation. We start with some basic definitions which specify the problem we are aiming to solve.

5.1 Basic definitions

Unless specified, in this section we will only consider type environments Δ s.t. $\vdash_c \Delta\diamond$, that is, possibly open environments with no multiple type assignments for the same class name, no inheritance cycles, no field hiding and no bad method overriding. Furthermore, when more convenient, we will treat Δ and Γ as the sets of their elements (regardless possible repetitions in Γ), and use the standard set theory notations.

As a first, rather imprecise attempt, the problem could be informally stated as follows: given a type environment Δ and a sequence of constraints Γ, find the possibly maximal $\Gamma' \subseteq \Gamma$ s.t. Δ satisfies Γ'; note that, since we are interested in incremental linking, Δ might not satisfy the whole environment Γ.

A first problem with the above statement is that satisfaction of type constraints is under-specified; e.g., for $\Delta = (\mathsf{c}, \mathtt{Object}, \emptyset, \emptyset)$ and $\Gamma = \phi(\mathsf{c}_1, \mathsf{f}, \alpha), \alpha \leq \mathsf{c}$ (with $\mathsf{c} \neq \mathsf{c}_1$), one might be tempted to assert that $\alpha \leq \mathsf{c}$ is satisfied by Δ with $\alpha = \mathsf{c}$. Nevertheless, even though $\alpha = \mathsf{c}$ originally seems like the only possible solution, it cannot be considered valid because it is sensitive to extensions to Δ. For instance, if we take $\Delta' = \Delta, (\mathsf{c}_1, \mathsf{c}, \{\mathsf{c}_1\ \mathsf{f}\}, \emptyset)$ then we discover that $\alpha = \mathsf{c}$ is no longer a valid solution, and that the whole Γ is satisfied by Δ' with $\alpha = \mathsf{c}_1$. In terms of the linking process, this means that we would need to backtrack from $\alpha = \mathsf{c}$ when adding a class c_1 as specified by Δ'.

We now formalize the above reasoning. First, a solution is a substitution σ for type variables s.t. Δ satisfies $\sigma(\Gamma)$, that is, $\Delta \vdash \sigma(\Gamma)$ is valid. Second, in order to avoid backtracking, σ must be the unique possible choice (up to inclusion of maps) for all extensions of Δ.

Definition 12. *For substitutions* σ, σ'*, we say* $\sigma \subseteq \sigma'$ *iff* $dom(\sigma) \subseteq dom(\sigma')$ *and for all* $\alpha \in dom(\sigma)$, $\sigma(\alpha) = \sigma'(\alpha)$.

Definition 13. *A sequence of constraints* Γ has solution σ *w.r.t. a given* Δ *iff*

1. $\Delta \vdash \sigma(\Gamma)$*;*
2. $\forall \sigma', \Delta'$*: if* $\Delta \subseteq \Delta'$ *and* $\Delta' \vdash \sigma'(\Gamma)$*, then* $\sigma \subseteq \sigma'$.

Fact 14. *A* Γ *has at most one solution w.r.t. a given* Δ.

Now that we have formalized the notion of solution, we can consider in more detail the situation where Γ has no solution w.r.t. a given Δ; as already said, this situation occurs quite naturally when considering incremental linking, because Δ is likely to be incomplete, and thus may contain insufficient information to compute a solution for Γ. Nevertheless, the algorithm should be able to identify a subsequence $\Gamma'' \subseteq \Gamma$ s.t. Γ'' has a solution σ w.r.t. Δ, and perform a simplification step: the constraints Γ'' are removed in order to avoid unnecessary checks in further linking steps, whereas σ is applied to the remaining constraints $\Gamma \setminus \Gamma''$, thus returning a sequence Γ'. Note that if the algorithm was smart enough, then Γ'' would be maximal, that is, there would not exist a Γ''' s.t. $\Gamma'' \subset \Gamma''' \subseteq \Gamma$ and Γ''' has solution w.r.t. Δ.

We distinguish two possible situations w.r.t. remaining constraints Γ': If there exists no extension of Δ which gives a solution for Γ', then the algorithm should detect a linking error. In this case, we say that Γ is *inconsistent* w.r.t. Δ.

Otherwise, linking succeeds, but the constraints Γ' still need to be satisfied, therefore the obtained fragment needs to be linked further before execution. In this case, we say that Γ'' is *undetermined* w.r.t. Δ.

Definition 15. *A sequence of constraints* Γ *is inconsistent w.r.t. a given* Δ *iff for all* Δ' *if* $\Delta \subseteq \Delta'$*, there is no* σ *s.t.* $\Delta' \vdash \sigma(\Gamma)$*; it is consistent (w.r.t.* Δ*) otherwise.*

A sequence of constraints Γ which is inconsistent w.r.t. Λ (that is, for all Δ) is called inconsistent. Conversely, a sequence of constraints Γ which is consistent w.r.t. some Δ (hence, w.r.t. Λ as well) is called consistent.

Definition 16. *A sequence of constraints* Γ *is* determined *w.r.t.* Δ *iff* Γ *either has a solution or is inconsistent w.r.t.* Δ*; it is* undetermined *(w.r.t.* Δ*) otherwise.*

Definition 17. *A sequence of constraints* Γ *is* determined *iff for all* Δ*, there exists* Δ' *s.t.* $\Delta \subseteq \Delta'$ *and* Γ *is determined w.r.t.* Δ'*; and is* undetermined *otherwise.*

Proposition 18. *The constraint* γ *is* determined *iff it matches one of the following patterns:* $\exists\, \mathsf{c}$, $\mathsf{c} \leq \mathsf{c}'$, $\phi(\mathsf{c}, \mathsf{f}, \mathsf{t})$, $\mu(\mathsf{c}, \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{t}, \bar{\mathsf{t}}))$, $\kappa(\mathsf{c}, \bar{\mathsf{c}}, \bar{\mathsf{t}})$, $\mathsf{c} \sim \mathsf{c}'$.

Finally, in order to prove that our algorithm implements a $\rightarrow_{ls}$ relation satisfying properties $\rightarrow_{ls}$ -sound, $\rightarrow_{ls}$ -complete-1, and $\rightarrow_{ls}$ -complete-2 of Theorem 11 — hence, soundness and completeness of compositional compilation w.r.t. global compilation is guaranteed — we show that each relevant set of constraints satisfies a quite intuitive syntactic property, and that for constraint sets which satisfy this syntactic property we can define an algorithm which satisfies the conditions of Theorem 11.

This syntactic property is based on a topological relation $\prec$ between constraints, with the following intuitive meaning: if $\gamma \prec \gamma'$, then γ has to be processed prior to γ'. For this we define the predicates $\mathcal{I}(\alpha, \gamma)$, expressing that α appears in an "in-position" in γ that is, needs to have been substituted before γ can be processed, and $\mathcal{O}(\alpha, \gamma)$ expressing that α appears in an "out-position" in γ, that is, processing γ will provide a substitution for this variable. $\mathcal{I}(\gamma)$, $\mathcal{I}(\Gamma)$, $\mathcal{O}(\gamma)$, $\mathcal{O}(\Gamma)$ have the obvious meaning, that is, the set of $\mathcal{I}$ variables in γ, or Γ, and the set of $\mathcal{O}$ variables in γ, or Γ.

Input:
this: a well-formed Γ
argument: a well-formed Δ
Output:
if it fails then Γ is inconsistent w.r.t. Δ
else returns σ and transforms Γ into Γ' s.t.
– $\Gamma' = \sigma(\Gamma \setminus \Gamma'')$
– $\Gamma'' \subseteq \Gamma$
– σ is the solution of Γ'' w.r.t. Δ
– for all $\gamma \in \Gamma'$, γ has no solution w.r.t. Δ
Pseudo-code:

```
Subs solve(Env Δ) throws fail{
  σ = ϵ
  this.topsort() // Γ must be topologically sorted
  for each γ ∈ this { // according to the order
    try{
      γ.apply(σ)
      σ' = γ.entailedBy(Δ)
      this.remove(γ)
      σ.update(σ')
    }
    catch(undetermined){}
  }
  return σ
}
```

Figure 14: Constraint solving algorithm *solve*

Definition 19. *For any Γ, α, and γ we define:*

1. $\mathcal{I}(\alpha,\gamma)$ *iff* $\gamma = \alpha \leq \mathtt{t}$, *or* $\gamma = \mathtt{t} \leq \alpha$, *or* $\gamma = \mathtt{c} \sim \alpha$, *or* $\gamma = \phi(\alpha, \mathsf{f}, \mathtt{t})$, *or* $\gamma = \mu(\mathtt{t}, \mathsf{m}, \bar{\mathtt{t}}, (\mathtt{t}', \bar{\mathtt{t}}'))$ *with* $\alpha = \mathtt{t}$ *or* $\alpha \in \bar{\mathtt{t}}$, *or* $\gamma = \kappa(\mathtt{c}, \bar{\mathtt{t}}, \bar{\mathtt{t}}')$ *with* $\alpha \in \bar{\mathtt{t}}$.
2. $\mathcal{O}(\alpha,\gamma)$ *iff* $\gamma = \phi(\mathtt{t}, \mathsf{f}, \alpha)$, *or* $\gamma = \mu(\mathtt{t}, \mathsf{m}, \bar{\mathtt{t}}, (\mathtt{t}', \bar{\mathtt{t}}'))$ *with* $\alpha \in \bar{\mathtt{t}}'$ *or* $\alpha = \mathtt{t}'$, *or* $\gamma = \kappa(\mathtt{c}, \bar{\mathtt{t}}, \bar{\mathtt{t}}')$ *with* $\alpha \in \bar{\mathtt{t}}'$.
3. $\mathcal{I}(\gamma) = \{\, \alpha \,|\, \mathcal{I}(\alpha,\gamma) \,\}$, $\quad \mathcal{I}(\Gamma) = \cup_{\gamma\in\Gamma}\mathcal{I}(\gamma)$,
   $\mathcal{O}(\gamma) = \{\, \alpha \,|\, \mathcal{O}(\alpha,\gamma) \,\}$, $\quad \mathcal{O}(\Gamma) = \cup_{\gamma\in\Gamma}\mathcal{O}(\gamma)$.

Definition 20. *The following topological relation is defined on type constraints: $\gamma \prec \gamma'$ iff $\mathcal{O}(\gamma) \cap \mathcal{I}(\gamma') \neq \emptyset$.*

Definition 21. *Let $\prec^+$ denote the transitive closure of $\prec$. Then a set of constraints Γ is well-formed iff*

1. $\forall \gamma \in \Gamma : \mathcal{I}(\alpha,\gamma) \implies \exists \gamma' \in \Gamma : \mathcal{O}(\alpha,\gamma')$,
2. $\nexists \gamma \in \Gamma : \gamma \prec^+ \gamma$,
3. $\forall \gamma,\gamma' \in \Gamma : \mathcal{O}(\alpha,\gamma)$ *and* $\mathcal{O}(\alpha,\gamma') \implies \gamma = \gamma'$

5.2 Description of the algorithm

The Java pseudo-code of the main algorithm is defined in Figure 14. The method *solve* takes a class type environment Δ and returns a substitution; it is declared in the class implementing sequences of constraints, therefore *this* denotes a certain Γ. When invoked, method *solve* either throws *fail* or returns a substitution σ and simplify Γ into Γ'.

The invocation $\gamma.entailedBy(\Delta)$, will throw the exception *fail* if a $\gamma \in \Gamma$ is found to be inconsistent w.r.t. Δ (and hence the whole Γ is inconsistent as well).

The method *solve* returns a σ, if it has found a subsequence $\Gamma'' \subset \Gamma$ which has solution σ w.r.t. Δ, and has modified Γ to Γ' by removing Γ'' and applying σ. We can see that Γ'' is maximal, by proving that for all $\gamma \in \Gamma'$, γ has no solution w.r.t. Δ. However, while all possible simplifications are always performed, some inconsistencies may be discovered later on, when some new fragment is linked, mainly because each constraint $\gamma \in \Gamma$ is checked separately.

For instance, given $\Delta = (\mathtt{c}, \mathtt{Object}, \emptyset, \emptyset)$ and $\Gamma = \mathtt{c}_1 \leq \mathtt{c}_2, \mathtt{c}_2 \leq \mathtt{c}_1$ (with all class names distinct), $\Gamma.solve(\Delta)$ returns the empty substitution and does not modify Γ, even though Γ is clearly inconsistent; however, such inconsistency can be captured when performing further linking steps. For instance, if we take $\Delta' = \Delta, (\mathtt{c}_1, \mathtt{Object}, \emptyset, \emptyset)$ then $\Gamma.solve(\Delta')$ throws *fail* since $\mathtt{c}_1 \leq \mathtt{c}_2$ is clearly inconsistent w.r.t. Δ'.

The constraints are processed respecting the topological order given in Definition 20, so that it is possible to scan Γ only once without failing to simplify some constraints. To see that the topological sorting is needed, consider for instance $\Gamma = \alpha \leq \mathtt{c}, \phi(\mathtt{c}, \mathsf{f}, \alpha)$, which is not topologically sorted, and $\Delta = (\mathtt{c}, \mathtt{Object}, \{\mathtt{c}\ \mathsf{f}\}, \emptyset)$. The constraint $\alpha \leq \mathtt{c}$ is processed first and kept, since is undetermined w.r.t. Δ. Then the constraint $\phi(\mathtt{c}, \mathsf{f}, \alpha)$ is removed (since it has solution $\alpha = \mathtt{c}$); however, the already examined constraint $\alpha \leq \mathtt{c}$ cannot be further instantiated into $\mathtt{c} \leq \mathtt{c}$, and, therefore, is not removed. Therefore *solve* fails to perform a simplification step. Finally, note that since Γ is assumed to be well-formed, it can always be topologically sorted.

Before being processed, each γ is instantiated w.r.t. the current substituion σ. Then, the method *entailedBy* checks whether γ has solution w.r.t. Δ; if so, it returns the corresponding substitution σ', otherwise it throws either *fail*, if γ is inconsistent w.r.t. Δ, or undetermined if γ is undetermined w.r.t. Δ. As already explained, *fail* is propagated by *solve*, whereas undetermined is captured; in this way, the constraint is not removed, the current substitution is not updated, and *solve* continues with the next constraint.

Figure 15 contains pseudo-code for *entailedBy* in the class representing constraints of the form $\mathtt{t} \leq \mathtt{t}'$ (the other cases can be found in [2]).

If either the left or right hand side of the constraint is a variable, then the constraint is undetermined, therefore the corresponding exception is thrown. Otherwise the constraint is ground and we can check whether it is satisfied by Δ. If the trivial cases do not apply (reflection and top type), then we perform an inheritance graph traversal from c_1 up to `Object` which can terminate in three different ways: if c_2 is found, then the constraint is satisfied and we return the empty substitution; if `Object` is reached without finding c_2 then the constraint is inconsistent and *fail* is thrown; otherwise the traversal stops because method *superclass* throws *undetermined* since some superclass of c_1 (c_1 included) could not be found in Δ. In the last case, the exception must be caught since the constraint still could be inconsistent if there exists c s.t. $c_2 \leq c$ and $c \leq c_1$. For this reason a new traversal is started from c_2 looking for a superclass of c_2 (c_2 included) contained in the set S of all superclasses of c_1 (c_1 included) collected during the first traversal. If such a class is found, then *fail* is thrown, otherwise (if either *superclass* throws *undetermined*, or `Object` is reached) *undetermined* is thrown.

5.3 Correctness of the algorithm

In the sequel, let $\rightarrow_{ls}$ denote the relation implemented by *solve*, that is, the relation defined as follows:

```
Subs entailedBy(Env Δ) throws undetermined, fail{
// Δ assumed to be well-formed
  if this.lhs().var() ∨ this.rhs().var() throw undetermined
  c1 = this.lhs()
  c2 = this.rhs()
  if c1.equals(c2) ∨ c2.equals(Object) return ϵ
  c = c1
  S = {c1} // will contain all c s.t. c1 ≤ c
  try {
    while c ≠ Object{
      c = Δ.superclass(c)
      S.add(c)
      if c.equals(c2) return ϵ
    }
    throw fail // chain complete up to Object
  }
  catch(undetermined){}
  c = c2 // still could fail
  while c ≠ Object{
    c = Δ.superclass(c)
    if S.contains(c) throw fail
  }
  throw undetermined
}
```

Figure 15: Definition of *entailedBy* for $\mathtt{t} \leq \mathtt{t}'$

$\Delta \vdash \Gamma \rightarrow_{ls} \sigma \mid \Gamma'$ iff $\Gamma.solve(\Delta)$ returns σ and transforms Γ into Γ'.

The following propositions ensure that $\rightarrow_{ls}$ leads to a sound and complete compositional compilation of FJ w.r.t. global compilation.

Proposition 22. *Any relevant Γ is well-formed.*

Proposition 23. *The relation $\rightarrow_{ls}$ satisfies properties $\rightarrow_{ls}$-sound, $\rightarrow_{ls}$-complete-1, and $\rightarrow_{ls}$-complete-2 of Theorem 11 for any well-formed Γ.*

Finally, note that Prop. 22 ensures also that *solve* is only invoked for sequences of constraints which can be sorted w.r.t. the topological order $\prec$ defined in Definition 20.

6. RELATED WORK

No formal model for separate compilation and linking had been developed until Cardelli's seminal work in 1997 [7], which can be considered a milestone in the area. It develops a simple formal framework for separate compilation (which, for the sake of simplification, is considered to consist only of typechecking), and linking. This framework can be considered to embody the notion of compositional compilation formalized in the present paper. The main differences are that we are also interested in code generation, and that the general framework in Section 2 abstracts from a particular programming language, and is parametric in the definition of separate compilation and linking judgments. Instead, [7] illustrates the concepts on a simple lambda-calculus, and linking just amounts in checking that each used entity has the required type and replacing its name by its definition.

Moreover, [7] did not consider the issue of which properties compositional compilation should satisfy in order to produce the same final result as global compilation. These properties correspond to some form of soundness and completeness as described in the present paper. They have been firstly introduced and formalized in [5], within a general framework for separate compilation and linking especially suited for Java-like languages similar to that introduced here. However, the framework in [5] did not take into account possible specialization of bytecode during linking, hence linking was reduced to *inter-checking*, that is, checking that mutual assumptions on fragments are satisfied. As a consequence, in the instantation of the framework presented in [5] (again on Featherweight Java) the code generated by separate compilation was standard Java bytecode, rather than the polymorphic bytecode introduced here; hence, since generated bytecode is context-dependent, separate compilation of a class was only possible under some given type constraints (or, equivalently, for each possible resulting bytecode).

The result in [5] was nevertheless important, since a type system for a Java-like language was firstly formally proved to guarantee sound and complete inter-checking (or, from another point of view, to support *principal typings*, see below). On the practical side, type systems as that in [5] (see also [4, 3]) constitute a good basis for selective recompilation, as exploited for full Java in [11, 12, 14], since it is possible to check whether a change in a source fragment affects other fragments by simply checking that their requirements still hold, but not for type inference, since type constraints needed for compiling a class cannot be inferred by just inspecting its source code.

In other words, compositional compilation as meant in this paper (compilation of a fragment in isolation) is not possible; type constraints can only be inferred for a given resulting bytecode.

The difference between the present paper and [5] becomes clearer by discussing the relation of our approach with the notion of *principal typing*. According to Wells [17]:

- a typing for a term T is the collection of all the information other than T which appears in the final judgement of a proof derivation showing that T is typable, and
- (roughly speaking) a typing for a term T is *principal* if it somehow represents all the typings for T.

The paper [17] (see also [10]) pointed out that the principal typing property (that is, every typable term has a principal typing) plays a key role in compositional type inference.

In this paper, as in [5], we have formalized compilation (including code generation) by means of a type system. Therefore, in our framework, performing compilation amounts to performing type inference.

In the type system in [5], the principal typing property holds by taking as terms pairs consisting of a source and a binary fragment (in other words, bytecode is considered part of the term). Instead, in the type system for compositional compilation proposed in this paper, $\vdash_C$, the principal typing property holds by taking as terms source fragments S: a typing is a triple $\Delta \mid \mathsf{B} \mid \Gamma$ (note that bytecode is considered part of the typing) and the system has the principal typing property since, for any source fragment S, at most one typing (modulo renaming of type variables) can be derived.

7. CONCLUSION

In this paper we addressed the problem of supporting compositional compilation for languages (like Java and C♯) where the binary code depends on the compilation context. For this, we defined a schema formalizing global and compositional compilation for such languages, and instantiated it by providing algorithms supporting compositional compilation for Featherweight Java. To the best of our knowledge, this is the first compositional compilation procedure for a Java-like language.

We believe that the results in this paper can be exploited at least in two different ways.

Firstly, they can be directly applied to the development of a new generation of compilers/interpreters/linkers (supporting compositional compilation) for real languages like Java and C♯. In this approach, polymorphic bytecode would be instantiated eagerly, in a step corresponding to static-linking. Such compilers would naturally support selective recompilation mechanisms, in the same spirit of [11, 12, 14]. In this respect, note that the application of our approach to, e.g., full Java, does not pose substantially new problems, though obviously more and more involved forms of type constraints, as we briefly discuss below. Type constraints modeling field hiding and method overloading were already presented (in the non polymorphic approach) in [4, 5, 3], and are omitted in this paper for simplicity. Roughly speaking, allowing method overloading simply leads to a different interpretation of the type constraint $\mu(\mathsf{c}, \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{c}', \bar{\mathsf{c}}'))$, which becomes: "for a call of method m with receiver of type c and arguments of type $\bar{\mathsf{c}}$ the most specific method has return type c' and parameters of type $\bar{\mathsf{c}}'$" (and analogously for field hiding). Hence, rule (c-meth-call) in Fig.11 is not affected, whereas rules defining the entailment judgment for this type constraint, that is, (μ-1) and (μ-2) in Fig.8, must change in order to reflect that the type constraint holds when c' and $\bar{\mathsf{c}}'$ are the return and parameter types of the most specific method for the call in the given class type environment. The linking simplification algorithm must change accordingly. Type constraints modeling most other relevant Java features, such as checked exceptions, accessibility levels, unreachable code, compile-time constants, are presented in [11, 12, 14]. Finally, in [13] it is shown how to deal with another Java peculiar feature, that is, the ambiguity in an expression such as A.B.C.[4]

Secondly, the results in this paper could lead to the development of a more flexible run-time support for Java-like languages, allowing execution of bytecode containing type variables. In this approach, polymorphic bytecode would be instantiated lazily, during dynamic linking and loading — some initial exploration appears in [6].

Further work includes extensions of our polymorphic model to other Java features, adapting to the polymorphic case the results mentioned above, and, more interestingly, to F-founded polymorphic methods and classes as introduced by GJ and Java 1.5. We also plan to investigate the extension of the source language so that it may contain type variables as well.

[4]For instance, A could be a class, B a static field, C a field, or A could be a class, B a nested class, C a static fied, or A could be a package, B a class, and C a field.

Acknowledgements. We are grateful to the anonymous referees for detailed and insightful comments and suggestions, and to to Alex Buckley for feedback. This work has been partially supported by Dynamic Assembly, Reconfiguration and Type-checking - EC project IST-2001-33477, and by APPSEM II - Thematic network IST-2001-38957.

8. REFERENCES

[1] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Even more principal typings for Java-like languages. In *6th Intl. Workshop on Formal Techniques for Java Programs 2004*, June 2004.

[2] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, November 2004. Extended verson with proofs, available at `ftp://ftp.disi.unige.it/pub/person/AnconaD/PBCCJL.pdf`.

[3] D. Ancona and G. Lagorio. Stronger Typings for Smarter Recompilation of Java-like Languages. *Journal of Object Technology*, 3(6):5–25, June 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.

[4] D. Ancona, G. Lagorio, and E. Zucca. True separate compilation of Java classes. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'02)*, pages 189–200. ACM Press, 2002.

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

[6] Alex Buckley and Sophia Drossopoulou. Flexible Dynamic Linking. In *6th Intl. Workshop on Formal Techniques for Java Programs 2004*, June 2004.

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

[8] Sophia Drossopoulou and Susan Eisenbach. Is the Java Type System Sound? In *11th European Conference on Object Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 389–418. Springer-Verlag, June 1997.

[9] 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.

[10] T. Jim. What are principal typings and what are they good for? In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–53. ACM Press, 1996.

[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] G. Lagorio. Capturing ghost dependencies in Java sources. *Journal of Object Technology*, 2004. To appear.

[14] G. Lagorio. *Type systems for Java separate compilation and selective recompilation*. PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, May 2004.

[15] Robert Sebesta. *Concepts of Programming Languages*. Addison-Wesley, 2004. Seventh edition.

[16] D. von Oheimb and T. Nipkow. Java is Type Safe - Definitely. In *ACM Symp. on Principles of Programming Languages 1998*, ACM Press, 1998.

[17] 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.