Even more principal typings for Java-like languages

Even More Principal Typings for Java-like Languages*

Davide Ancona[1], Ferruccio Damiani[2], Sophia Drossopoulou[3], and Elena Zucca[1]

[1] DISI - Università di Genova
[2] Dipartimento di Informatica - Università di Torino
[3] Imperial College - London

Abstract. We propose a new type system for Java-like languages which allows compilation of a class in isolation, that is, in a context where *no information* is available on other classes. Indeed, by this type system it is possible to *infer* the assumptions guaranteeing type correctness of a class c, and generate (abstract) bytecode for c, by just inspecting the source code of c. Then, a collection of classes can be safely linked together without further inspection of the classes' code, provided that each class has been typechecked in isolation (*intra-checking*), and that the mutual class assumptions are satisfied (*inter-checking*). In other words, the type systems supports *compositional analysis*, as formally guaranteed by the fact that it has *principal typings*. We also develop an algorithm for inter-checking, and prove it correct.

1 Introduction

Successful *global* compilation of (i.e., of all) sources composing an application guarantees that the resulting target is "sound". Global compilation is often impractical (e.g., too many, or unavailable sources). Mainstream object-oriented programming languages, such as Java and C♯, support the following notion of *separate compilation* of fragments: a fragment (e.g., a single class) can be compiled in a context containing other classes *only in binary form.* Java and C♯ do not enforce the notion of "sound target code" in this case, instead target code is checked at run-time during loading and verification. A link related exception (e.g., NoSuchFieldError) is thrown if an assumption in a target fragment is not satisfied. Thus, end users may face perplexing linking errors.

Recent innovative type systems for Java-like languages [2, 1, 3], support separate compilation in a stronger sense [5], that is, a single source fragment can be compiled in isolation (intra-checked) in a context where *only type information* but no code is available on the fragments it depends on. Then, an executable application can be constructed by linking together a collection of fragments, provided that their types mutually satisfy the required type assumptions (inter-checking), without any need to reinspect their code. An obvious property we expect for inter-checking is *soundness*, i.e., that successful inter-checking implies successful global compilation. Ideally, we would like to have *completeness* as well, i.e., that failing inter-checking implies failing global compilation. Intuitively, this is guaranteed if we can associate with each fragment a set of type assumptions and a type (formally, a *typing*, in the terminology of [9]) which represents all other possible typings (*principal typing*), and hence can be used to check compatibility in *all possible contexts.*

In Java and C♯ completeness of inter-checking is hard to achieve. This is due to the tight connection between the compilation environment and the generated bytecode. 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 $\mathsf{md}_1^{\mathsf{b}}$ with annotations which reflect the classes where fields were found and their types, that is:[4]

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

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

[4] We use a very high level presentation of bytecode.

However, 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 $\mathsf{md}_2^{\mathsf{b}}$:

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

Formally, we can assign to the fragment containing md^{s} two different (incomparable) typings, corresponding to the compilation environments Δ_1 and Δ_2: the former has type constraints including[5] $\phi(\mathsf{B},\mathsf{f1},\mathsf{C})$, $\phi(\mathsf{C},\mathsf{f2},\mathsf{E})$, the latter has type constraints including $\phi(\mathsf{B},\mathsf{f1},\mathsf{D}), \phi(\mathsf{D},\mathsf{f2},\mathsf{F}), \mathsf{F} \leq \mathsf{E}$. Hence, if for the fragment containing md^{s} we derive the first typing (in the intra-checking phase), and then inter-check this fragment with the classes in Δ_2, inter-checking fails, even though global compilation would succeed. In [2, 1, 3] the problem is solved by considering binary as part of the term to be typed; thus, we get two different principal typings: One reflecting the minimal set of assumptions leading to the generation of $\mathsf{md}_1^{\mathsf{b}}$, the other reflecting the minimal set of assumptions leading to the generation of $\mathsf{md}_2^{\mathsf{b}}$. This corresponds to the selective recompilation view, where it makes sense for inter-checking to fail whenever global compilation would have generated *different* bytecode. It also corresponds to the execution view: indeed, execution of $\mathsf{md}_1^{\mathsf{b}}$ in environment Δ_1 succeeds (modulo null access errors), but execution of $\mathsf{md}_1^{\mathsf{b}}$ in Δ_2 fails (even though Δ_2 essentially *does* contain what is required by m, namely, the field accesses f1.f2 leading from class B to a subclass of E).

The approach in [2, 1, 3] works well for selective recompilation, where the type constraints can be generated the first time an application is globally compiled [7, 8], but does not support compilation of a single source fragment in a context where no information is available on the fragments it depends on.

In this paper, we propose a new approach which supports a stronger form of separate compilation: a single source fragment can be compiled in isolation (intra-checked) in a context where *no information* is available on the fragments it depends on. We formalize the new approach by means of a type system where bytecode is considered as part of the type. The key idea is having both polymorphic type constraints and bytecode. In this way, it is possible to infer from the source code the set of type constraints needed for compiling the method declaration, that is, $\phi(\mathsf{B},\mathsf{f1},\alpha), \phi(\alpha,\mathsf{f2},\beta), \beta \leq \mathsf{E}$, where α, β are *type variables*. Correspondingly, the following polymorphic bytecode md^{b} is generated:

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

In this type system, we can assign to each typable fragment a principal typing (actually, exactly one typing); for instance, in the (principal) typing for the fragment containing md^{s} the set of type constraints contains $\phi(\mathsf{B},\mathsf{f1},\alpha), \phi(\alpha,\mathsf{f2},\beta), \beta \leq \mathsf{E}$ and the polymorphic bytecode contains md^{b}.

The rest of the paper is organized as follows: in Sect.2 we introduce a general notion of type system for separate compilation and inter-checking for Java-like languages. In Sect.3 we present two type systems, corresponding to the separate compilation approach in [2, 1, 3] and to the new approach proposed in this paper, respectively. In Sect.4 we describe an algorithm which shows how inter-checking in the new approach can be effectively performed and state its correctness. We conclude by discussing some further work.

2 Type systems for separate compilation

In this section we define a schema of type system for separate compilation of Java-like languages, by listing the basic syntactic categories and judgments such a type system should define. The monomorphic and the polymorphic type systems from the next section are instances of this schema.

- Source class declarations (cd^{s}), binary class declarations (cd^{b}).
- Sequences of source class declarations (S), sequences of binary class declarations (B).

[5] A type constraint $\phi(\mathtt{t},\mathtt{f},\mathtt{t}')$ reads "$\mathtt{t}$ provides field $\mathtt{f}$ with type $\mathtt{t}'$".

- Class type environments (Δ), which are sequences of class type assignments (δ). A class type assignment is, roughly, 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 program deprived of bodies.
- Global compilation judgment $\vdash \mathsf{S}{:}\Delta \rightsquigarrow \mathsf{B}$: the program consisting of class declarations S has type Δ and compiles to B.
- Type constraint environments Γ, which are sequences of type constraints (γ).
- Separate compilation judgment $\vdash \mathsf{cd}^{\mathsf{s}}{:}\delta \rightsquigarrow \Gamma \mid \mathsf{cd}^{\mathsf{b}}$: the source class declaration cd^{s} has type δ and compiles to cd^{b} under the type constraints in Γ.
- Linking judgment $\Delta \vdash \Gamma | \mathsf{cd}^{\mathsf{b}} \rightsquigarrow \hat{\mathsf{cd}}^{\mathsf{b}}$: class type environment Δ satisfies the type constraints Γ, and in Δ binary class declaration cd^{b} becomes $\hat{\mathsf{cd}}^{\mathsf{b}}$.

There are two different approaches to compilation which can be both modelled by the ingredients from above.

The first approach compiles all class declarations together, as formalized by the global compilation judgment $\vdash \mathsf{S}{:}\Delta \rightsquigarrow \mathsf{B}$. The second approach compiles each class $\mathsf{cd}^{\mathsf{s}}_{\mathsf{i}}$ in isolation (*intra-checking*, following the terminology in [5]), as formalized by $\vdash \mathsf{cd}^{\mathsf{s}}_{\mathsf{i}}{:}\delta_i \rightsquigarrow \Gamma_{\mathsf{i}} \mid \mathsf{cd}^{\mathsf{b}}_{\mathsf{i}}$, and then checks whether a *linkset* (a successfully intra-checked collection of classes) *inter-checks*, that is, whether these classes' mutual requirements are satisfied, as formalized by $\delta_1 ... \delta_n \vdash \Gamma_1 ... \Gamma_{\mathsf{n}} | \mathsf{cd}^{\mathsf{b}}_{\mathsf{i}} \rightsquigarrow \hat{\mathsf{cd}}^{\mathsf{b}}{}_{\mathsf{i}}$. Notice that the check does *not* depend on the source code.

Definition 1. *Given a* linkset*, that is, a sequence* $\mathsf{L} = \vdash \mathsf{cd}^{\mathsf{s}}_{\mathsf{i}}{:}\delta_{\mathsf{i}} \rightsquigarrow \Gamma_{\mathsf{i}} \mid \mathsf{cd}^{\mathsf{b}^{\mathsf{i}\in 1..\mathsf{n}}}_{\mathsf{i}}$ *of valid separate compilation judgments, we say that* L inter-checks *producing binaries* $\hat{\mathsf{B}} = \hat{\mathsf{cd}}^{\mathsf{b}}{}_1 ... \hat{\mathsf{cd}}^{\mathsf{b}}{}_n$ *iff* $\delta_1 ... \delta_n \vdash \Gamma_{\mathsf{i}} | \mathsf{cd}^{\mathsf{b}}_{\mathsf{i}} \rightsquigarrow \hat{\mathsf{cd}}^{\mathsf{b}}{}_{\mathsf{i}}$ *holds for* $i \in 1..n$.

A type system supports compositional analysis if successful intra-checking and inter-checking phases produce the same result as global compilation. This is formalized below.

Definition 2. *We say that inter-checking is* sound *w.r.t. global compilation iff, for any linkset* $\mathsf{L} = \vdash \mathsf{cd}^{\mathsf{s}}_{\mathsf{i}}{:}\delta_{\mathsf{i}} \rightsquigarrow \Gamma_{\mathsf{i}} \mid \mathsf{cd}^{\mathsf{b}^{\mathsf{i}\in 1..\mathsf{n}}}_{\mathsf{i}}$, L *inter-checks producing* B *implies* $\vdash \mathsf{cd}^{\mathsf{s}}_1 ... \mathsf{cd}^{\mathsf{s}}_{\mathsf{n}}{:}\delta_1 ... \delta_n \rightsquigarrow \mathsf{B}$. *We say that inter-checking is* complete *w.r.t. global compilation iff, for any linkset* $\mathsf{L} = \vdash \mathsf{cd}^{\mathsf{s}}_{\mathsf{i}}{:}\delta_{\mathsf{i}} \rightsquigarrow \Gamma_{\mathsf{i}} \mid \mathsf{cd}^{\mathsf{b}^{\mathsf{i}\in 1..\mathsf{n}}}_{\mathsf{i}}$, $\vdash \mathsf{cd}^{\mathsf{s}}_1 ... \mathsf{cd}^{\mathsf{s}}_{\mathsf{n}}{:}\delta_1 ... \delta_n \rightsquigarrow \mathsf{B}$ *implies that* L *inter-checks producing* B.

The monomorphic and the polymorphic type systems define global compilation compositionally, by the following metarule, which appears both in the monomorphic and polymorphic flavour:

$$(program)\ \frac{\vdash \mathsf{cd}^{\mathsf{s}}_{\mathsf{i}}{:}\delta_i \rightsquigarrow \Gamma_{\mathsf{i}} \mid \mathsf{cd}^{\mathsf{b}}_{\mathsf{i}}\ \forall i \in 1..n \qquad \delta_1 ... \delta_n \vdash \Gamma_1 ... \Gamma_{\mathsf{n}} | \mathsf{cd}^{\mathsf{b}}_{\mathsf{i}} \rightsquigarrow \hat{\mathsf{cd}}^{\mathsf{b}}{}_{\mathsf{i}}\ \forall \mathsf{i} \in 1..\mathsf{n}}{\vdash \mathsf{cd}^{\mathsf{s}}_1 ... \mathsf{cd}^{\mathsf{s}}_{\mathsf{n}}{:}\delta_1 ... \delta_n \rightsquigarrow \hat{\mathsf{cd}}^{\mathsf{b}}{}_1 ... \hat{\mathsf{cd}}^{\mathsf{b}}{}_n}$$

With the rule from above, inter-checking is trivially sound. However, it is not necessarily complete. Indeed, assume that the program $\mathsf{cd}^{\mathsf{s}}_1 ... \mathsf{cd}^{\mathsf{s}}_{\mathsf{n}}$ successfully compiles to B. In general, we can derive many separate compilation judgments for a class, therefore, if we chose a "wrong" type constraint Γ_{j} for some class in the linkset $\mathsf{L} = \vdash \mathsf{cd}^{\mathsf{s}}_{\mathsf{i}}{:}\delta_{\mathsf{i}} \rightsquigarrow \Gamma_{\mathsf{i}} \mid \mathsf{cd}^{\mathsf{b}^{\mathsf{i}\in 1..\mathsf{n}}}_{\mathsf{i}}$ (formally, s.t. $\delta_1 ... \delta_n \not\vdash \Gamma_1 ... \Gamma_{\mathsf{n}} | \mathsf{cd}^{\mathsf{b}}_{\mathsf{j}} \rightsquigarrow \hat{\mathsf{cd}}^{\mathsf{b}}{}_{\mathsf{j}}$), inter-checking L would fail. In the following section, we show that inter-checking is not complete for the monomorphic type system, but it is complete for the polymorphic type system, since for each class we derive *exactly one* separate compilation judgment.

3 Two type systems for separate compilation

In this section, we present two type systems formalizing separate compilation of a small Java-like language, that is, the one considered in [3] where, for simplicity, we do not allow field hiding and method overloading. The system in Fig.4 is the same as that defined in [3] (modulo the differences

$$
\begin{array}{rcl}
\mathsf{S} & ::= & \mathsf{cd}^{\mathsf{s}}_1 \ldots \ \mathsf{cd}^{\mathsf{s}}_n \\
\mathsf{cd}^{\mathsf{s}} & ::= & \texttt{class c extends c}' \ \{ \ \mathsf{fds} \ \mathsf{mds}^{\mathsf{s}} \ \} \\
\mathsf{fds} & ::= & \mathsf{fd}_1 \ldots \ \mathsf{fd}_n \\
\mathsf{fd} & ::= & \mathsf{c} \ \mathsf{f}; \\
\mathsf{mds}^{\mathsf{s}} & ::= & \mathsf{md}^{\mathsf{s}}_1 \ldots \ \mathsf{md}^{\mathsf{s}}_n \\
\mathsf{md}^{\mathsf{s}} & ::= & \mathsf{mh} \ \{\texttt{return} \ \mathsf{e}^{\mathsf{s}};\} \\
\mathsf{mh} & ::= & \mathsf{c}_0 \ \mathsf{m}(\mathsf{c}_1 \ \mathsf{x}_1, \ldots, \mathsf{c}_n \ \mathsf{x}_n) \\
\mathsf{e}^{\mathsf{s}} & ::= & \mathsf{x} \mid \mathsf{e}^{\mathsf{s}}.\mathsf{f} \mid \mathsf{e}^{\mathsf{s}}_0.\mathsf{m}(\mathsf{e}^{\mathsf{s}}_1, \ldots, \mathsf{e}^{\mathsf{s}}_n) \mid \texttt{new c}(\mathsf{e}^{\mathsf{s}}_1 \ldots \mathsf{e}^{\mathsf{s}}_n) \mid (\mathsf{c})\mathsf{e}^{\mathsf{s}} \\
 & & \\
\mathsf{B} & ::= & \mathsf{cd}^{\mathsf{b}}_1 \ldots \ \mathsf{cd}^{\mathsf{b}}_n \\
\mathsf{cd}^{\mathsf{b}} & ::= & \texttt{class c extends c}' \ \{ \ \mathsf{fds} \ \mathsf{mds}^{\mathsf{b}} \ \} \\
\mathsf{mds}^{\mathsf{b}} & ::= & \mathsf{md}^{\mathsf{b}}_1 \ldots \ \mathsf{md}^{\mathsf{b}}_n \\
\mathsf{md}^{\mathsf{b}} & ::= & \mathsf{mh} \ \{\texttt{return} \ \mathsf{e}^{\mathsf{b}};\} \\
\mathsf{e}^{\mathsf{b}} & ::= & \mathsf{x} \mid \mathsf{e}^{\mathsf{b}}[\mathsf{t}.\mathsf{f} \ \mathsf{t}'] \mid \mathsf{e}^{\mathsf{b}}_0[\mathsf{t}.\mathsf{m}(\bar{\mathsf{t}})\mathsf{t}'](\mathsf{e}^{\mathsf{b}}_1, \ldots, \mathsf{e}^{\mathsf{b}}_n) \mid \texttt{new} \ [\mathsf{c} \ \bar{\mathsf{t}}](\mathsf{e}^{\mathsf{b}}_1 \ldots \mathsf{e}^{\mathsf{b}}_n) \mid \ (\mathsf{c})\mathsf{e}^{\mathsf{b}} \ \mid \ \ll \mathsf{c}, \alpha \gg \mathsf{e}^{\mathsf{b}} \\
\mathsf{t} & ::= & \mathsf{c} \mid \alpha \\
\bar{\mathsf{t}} & ::= & \mathsf{t}_1 \ldots \mathsf{t}_n
\end{array}
$$

Fig. 1. Syntax

in the language and in the notation). It is *monomorphic* and supports separate compilation of a class in a context where (only) type information on the classes it depends on is available. The system in Fig.5 is *polymorphic* and supports separate compilation of a class in a context where *no* information on the classes it depends on is available.

The syntax of the language is defined in Fig.1. It is basically Featherweight Java [6], 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. 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 allow simpler typing rules, we assume field names in fds, method names in $\mathsf{mds}^{\mathsf{s}}$, parameter names in mh to be distinct.

As already mentioned, 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 is annotated 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 the polymorphic type system, classes are separately compiled into bytecode annotated with type variables (that is, a meta-variable t denotes either a type variable or a class name), whereas in the monomorphic system bytecode can be only annotated with class names (that is, a meta-variable t denotes a class name). Note that polymorphic and monomorphic casting have a different form: $\ll \mathsf{c}, \alpha \gg \mathsf{e}^{\mathsf{b}}$ can be instantiated either into e^{b}, if α is substituted with c' s.t. $\mathsf{c}' \leq \mathsf{c}$ (casting-up), or into $(\mathsf{c})\mathsf{e}^{\mathsf{b}}$, if α is substituted with c' s.t. $\mathsf{c} \leq \mathsf{c}'$ (casting-down). For the casting annotation we use a different notation (double angle brackets rather than square brackets), since this annotation is only allowed at the polymorphic level.

Class type assignments and type constraints are defined in Fig.2. Type constraints are defined as in [3] (where they were called local type assumptions), except for $\mathsf{c} \sim \mathsf{t}$, introduced to deal with casting. As for bytecode, the meta-variable t will be instantiated with type variables in the polymorphic system, and with class names in the monomorphic system.

Type constraints have the following informal meaning:

- $\exists\, \mathsf{c}$ means "c must be defined";
- $\mathsf{t} \leq \mathsf{t}'$ means "t must be a subtype of t'";
- $\phi(\mathsf{t}, \mathsf{f}, \mathsf{t}')$ means "t provides field f with type t'";
- $\mu(\mathsf{t}, \mathsf{m}, \bar{\mathsf{t}}, (\mathsf{t}', \bar{\mathsf{t}}'))$ means "t provides method m applicable to arguments of type $\bar{\mathsf{t}}$, with parameters of type $\bar{\mathsf{t}}'$ and return type t'.

$$
\begin{array}{rl}
\delta & ::= (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \\
\gamma & ::= \exists\, \mathsf{c} \;\mid \mathtt{t} \leq \mathtt{t}' \;\mid \phi(\mathtt{t}, \mathsf{f}, \mathtt{t}') \mid \mu(\mathtt{t}, \mathsf{m}, \bar{\mathtt{t}}, (\mathtt{t}', \bar{\mathtt{t}}')) \;\mid \kappa(\mathsf{c}, \bar{\mathtt{t}}, \bar{\mathtt{t}}') \mid \mathsf{c}\text{☺}\mathsf{ms} \mid \mathsf{c}\text{☺}\mathsf{f} \mid \mathtt{t} \not< \mathtt{t}' \mid \mathsf{c} \sim \mathtt{t} \\
\\
\mathsf{fss} & ::= \{\mathsf{fs}_1, \ldots, \mathsf{fs}_n\} \\
\mathsf{fs} & ::= \mathsf{c}\ \mathsf{f} \\
\mathsf{mss} & ::= \{\mathsf{ms}_1, \ldots, \mathsf{ms}_n\} \\
\mathsf{ms} & ::= \mathsf{c}\ \mathsf{m}(\bar{\mathsf{c}})
\end{array}
$$

Fig. 2. Class type assignments and type constraints

- $\kappa(\mathsf{c}, \bar{\mathtt{t}}, \bar{\mathtt{t}}')$ means "c provides constructor applicable to arguments of type $\bar{\mathtt{t}}$, with parameters of type $\bar{\mathtt{t}}'$";
- c☺f and c☺ms means "c must be extensible with a subclass declaring f and ms, respectively";
- $\mathsf{c} \not< \mathsf{c}'$ means "c must not be a proper subtype of c'";
- $\mathsf{c} \sim \mathtt{t}$ means "c and $\mathtt{t}$ must be comparable".

In Fig.3 we define the two entailment relations $\vdash_m$ and $\vdash_p$. The former is used by the rule defining the linking judgment for the monomorphic system, whereas the latter is used for the same purpose in the polymorphic system. The only difference between the two, is that $\vdash_p$ also deals with the constraint $\mathsf{c} \sim \mathsf{c}'$.

Intuitively, if $\Delta \vdash \Gamma$ is derivable (either with m, or p subscript), then it means that, under the assumption that Δ is well-formed, all type constraints in Γ are satisfied by Δ. Note that entailments work only on ground type constraints. Type variables are managed by the rule defining the linking judgment in the polymorphic system.

The rules are intuitive and almost self-explanatory, however more comments can be found in other papers [3, 1].

In rules *(ϕ-2)* and *(☺-3)*, the side condition $\mathsf{f} \notin \mathsf{fss}$ means that f is not declared in fss; analogously, in rules *(μ-2)* and *(☺-4)*, $\mathsf{m} \notin \mathsf{mss}$ means that m is not declared in mss.

Rules for intra-checking a class declaration are defined in Fig.4 (monomorphic system) and Fig.5 (polymorphic system). The straightforward definition of the auxiliary function *type*, extracting type information from source fragments, is omitted (see [3]). The intuition behind the rules is the same in the two systems: they just extract all type constraints Γ necessary to compile a given source fragment into a certain binary fragment. For instance, if $\Pi \vdash \mathsf{e}^{\mathsf{s}}{:}\mathsf{c} \rightsquigarrow \Gamma \mid \mathsf{e}^{\mathsf{b}}$ is derivable, then, whenever the type constraints in Γ are satisfied, the expression e^{s} with variables described in Π has type c and can be compiled into e^{b}.

As already explained in the Introduction, the main difference between the two systems is that the polymorphic system has principal typings, since 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 c the type and the (polymorphic) bytecode of c, and the required type constraints. This is not possible for the monomorphic system, where one needs to know the environment where c is compiled [3, 1].

Both systems use the rule *(program)* defined in Sect.2 (which is repeated for completeness); however, the linking judgment needs to be defined.

For the monomorphic system, the linking judgment simply coincides with the entailment relation $\vdash_m$ (see [3]), whereas in the polymorphic system, in order to obtain monomorphic bytecode, we need to find σ, the right substitution mapping all the type variables into class names.

$$
\textit{(m-linking)}\frac{\Delta \vdash_m \Gamma}{\Delta \vdash_m \Gamma | \mathsf{cd}^{\mathsf{b}} \rightsquigarrow \mathsf{cd}^{\mathsf{b}}} \qquad \textit{(p-linking)}\frac{\Delta \vdash_p \sigma(\Gamma)}{\Delta \vdash_p \Gamma | \mathsf{cd}^{\mathsf{b}} \rightsquigarrow (\Delta, \sigma)(\mathsf{cd}^{\mathsf{b}})}
$$

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 cd^{b} w. r. t. Δ

$$(\epsilon)\ \frac{}{\Delta \vdash_m \epsilon} \qquad (and)\ \frac{\Delta \vdash_m \Gamma \qquad \Delta \vdash_m \gamma}{\Delta \vdash_m \Gamma, \gamma} \qquad (\exists)\ \frac{}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash_m \exists\, \mathsf{c}} \qquad (Obj)\ \frac{}{\Delta \vdash_m \exists\, \mathtt{Object}}$$

$$(refl)\ \frac{}{\Delta \vdash_m \mathsf{c} \leq \mathsf{c}} \qquad (trans)\ \frac{\Delta, (\mathsf{c}_1, \mathsf{c}_2, \mathsf{fss}, \mathsf{mss}) \vdash_m \mathsf{c}_2 \leq \mathsf{c}_3}{\Delta \vdash_m \mathsf{c}_1 \leq \mathsf{c}_3}$$

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

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

$$(\mu\text{-}2)\ \frac{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash_m \mu(\mathsf{c}', \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{c}'', \bar{\mathsf{c}}''))}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash_m \mu(\mathsf{c}, \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{c}'', \bar{\mathsf{c}}''))}\ \mathsf{m} \notin \mathsf{mss} \qquad (\kappa\text{-}1)\ \frac{}{\Delta \vdash_m \kappa(\mathtt{Object}, \epsilon, \epsilon)}$$

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

$$(\circledcirc\text{-}1)\ \frac{}{\Delta \vdash_m \mathtt{Object}\circledcirc\mathsf{f}} \qquad (\circledcirc\text{-}2)\ \frac{}{\Delta \vdash_m \mathtt{Object}\circledcirc\mathsf{ms}} \qquad (\circledcirc\text{-}3)\ \frac{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash_m \mathsf{c}'\circledcirc\mathsf{f}}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash_m \mathsf{c}\circledcirc\mathsf{f}}\ \mathsf{f} \notin \mathsf{fss}$$

$$(\circledcirc\text{-}4)\ \frac{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash_m \mathsf{c}'\circledcirc\mathsf{c}''\ \mathsf{m}(\bar{\mathsf{c}})}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash_m \mathsf{c}\circledcirc\mathsf{c}''\ \mathsf{m}(\bar{\mathsf{c}})}\ \mathsf{m} \notin \mathsf{mss} \vee \mathsf{c}''\ \mathsf{m}(\bar{\mathsf{c}}) \in \mathsf{mss} \qquad (not\text{-}sub\text{-}1)\ \frac{}{\Delta \vdash_m \mathtt{Object} \not\leq \mathsf{c}}$$

$$(not\text{-}sub\text{-}2)\ \frac{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash_m \mathsf{c}' \not\leq \mathsf{c}''}{\Delta, (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \vdash_m \mathsf{c} \not\leq \mathsf{c}''}\ \mathsf{c}' \neq \mathsf{c}'' \qquad (emb)\ \frac{\Delta \vdash_m \gamma}{\Delta \vdash_p \gamma} \qquad (\sim\text{-}1)\ \frac{\Delta \vdash_p \mathsf{c} \leq \mathsf{c}'}{\Delta \vdash_p \mathsf{c} \sim \mathsf{c}'} \qquad (\sim\text{-}2)\ \frac{\Delta \vdash_p \mathsf{c} \sim \mathsf{c}'}{\Delta \vdash_p \mathsf{c}' \sim \mathsf{c}}$$

Fig. 3. Rules for the entailment $\vdash_m$ and $\vdash_p$

and σ is denoted by $(\Delta, \sigma)(\mathsf{cd}^{\mathsf{b}})$; Δ is needed for dealing with the case $\ll \mathsf{c}, \alpha \gg \mathsf{e}^{\mathsf{b}}$:

$$(\Delta, \sigma)(\ll \mathsf{c}, \alpha \gg \mathsf{e}^{\mathsf{b}}) = \begin{cases} (\Delta, \sigma)(\mathsf{e}^{\mathsf{b}}) & \text{if } \sigma(\alpha) = \mathsf{c}' \text{ and } \Delta \vdash_p \mathsf{c}' \leq \mathsf{c} \\ (\mathsf{c})(\Delta, \sigma)(\mathsf{e}^{\mathsf{b}}) & \text{if } \sigma(\alpha) = \mathsf{c}' \text{ and } \Delta \vdash_p \mathsf{c} \leq \mathsf{c}' \\ \ll \mathsf{c}, \alpha \gg (\Delta, \sigma)(\mathsf{e}^{\mathsf{b}}) & \text{if } \alpha \text{ is not substituted by } \sigma \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

4 Implementation of the polymorphic type system

In this section we outline the algorithm for implementing the polymorphic type system defined in the previous section. Except for rule *(p-linking)*, all other rules in Fig.5 can be directly turned into an algorithm which, given a class declaration, returns its type, its polymorphic bytecode and the minimal type constraints needed for compiling it.

Implementing the linking judgment is not as straightforward, since the *(p-linking)* rule does not describe how to find a substitution σ s.t. $\Delta \vdash_p \sigma(\Gamma)$.[6] An algorithm for finding such a substitution is described by the pseudo-code in Fig. 6, with the function *entails*, which processes type constraints from Γ.

The function *entails* can process only *determined* type constraints. Intuitively, a type constraint γ is determined iff for all Δ there exists at most one substitution σ s.t. $\Delta \vdash_p \sigma(\gamma)$. Ground type constraints (that is, constraints without type variables) are trivially determined, and constraints of the form $\phi(\mathsf{c}, \mathsf{f}, \mathsf{t})$, $\kappa(\mathsf{c}, \bar{\mathsf{c}}, \bar{\mathsf{t}})$, $\mu(\mathsf{c}, \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{t}, \bar{\mathsf{t}}))$ are also determined. Consider, for instance, a type

[6] On the contrary, the entailment $\Delta \vdash_p \Gamma$ can be implemented almost directly [1].

$$\textit{(m-program)}\ \frac{\vdash_m \mathsf{cd}_i^{\mathsf{s}}{:}\delta_i \rightsquigarrow \Gamma_{\mathsf{i}} \mid \mathsf{cd}_i^{\mathsf{b}}\ \forall i \in 1..n \qquad \delta_1 ... \delta_n \vdash_m \Gamma_1 ... \Gamma_{\mathsf{n}} | \mathsf{cd}_{\mathsf{i}}^{\mathsf{b}} \rightsquigarrow \hat{\mathsf{cd}}^{\mathsf{b}}{}_{\mathsf{i}}\ \forall \mathsf{i} \in 1..\mathsf{n}}{\vdash_m \mathsf{cd}_1^{\mathsf{s}} ... \mathsf{cd}_{\mathsf{n}}^{\mathsf{s}} {:} \delta_1 ... \delta_n \rightsquigarrow \hat{\mathsf{cd}}^{\mathsf{b}}{}_1 ... \hat{\mathsf{cd}}^{\mathsf{b}}{}_n}$$

$$\textit{(m-class)}\ \frac{\vdash_m \mathsf{mds}^{\mathsf{s}} \rightsquigarrow \Gamma \mid \mathsf{mds}^{\mathsf{b}}}{\vdash_m\ \mathtt{class}\ \mathsf{c}\ \mathtt{extends}\ \mathsf{c}'\ \{\mathsf{fds}\ \mathsf{mds}^{\mathsf{s}}\}\ :\ (\mathsf{c}, \mathsf{c}', \mathsf{fss}, \mathsf{mss}) \rightsquigarrow \Gamma' \mid \mathtt{class}\ \mathsf{c}\ \mathtt{extends}\ \mathsf{c}'\ \{\mathsf{fds}\ \mathsf{mds}^{\mathsf{b}}\}}$$
$$\begin{array}{l} \mathit{type}(\mathsf{mds}^{\mathsf{s}}) = \mathsf{mss} = \{\mathsf{ms}_1, \ldots, \mathsf{ms}_{\mathsf{n}}\} \\ \mathit{type}(\mathsf{fds}) = \mathsf{fss} = \{\mathsf{c}_1\ \mathsf{f}_1, \ldots, \mathsf{c}_{\mathsf{m}}\ \mathsf{f}_{\mathsf{m}}\} \\ \Gamma' = \Gamma, \mathsf{c}' \textcircled{:} \mathsf{ms}_{\mathsf{i}}^{\mathsf{i} \in 1..\mathsf{n}}, \mathsf{c}' \textcircled{:} \mathsf{f}_{\mathsf{i}}^{\mathsf{i} \in 1..\mathsf{m}}, \mathsf{c}' \not\leq \mathsf{c} \end{array}$$

$$\textit{(m-methods)}\ \frac{\vdash_m\ \mathsf{md}_{\mathsf{i}}^{\mathsf{s}} \rightsquigarrow \Gamma_{\mathsf{i}} \mid \mathsf{md}_{\mathsf{i}}^{\mathsf{b}}\ \forall i \in 1..n}{\vdash_m \mathsf{md}_1^{\mathsf{s}}\ \ldots\ \mathsf{md}_{\mathsf{n}}^{\mathsf{s}} \rightsquigarrow \Gamma_1, \ldots, \Gamma_{\mathsf{n}} \mid \mathsf{md}_1^{\mathsf{b}} \ldots \mathsf{md}_{\mathsf{n}}^{\mathsf{b}}}$$

$$\textit{(m-method)}\ \frac{\mathsf{x}_1{:}\mathsf{c}_1 ... \mathsf{x}_n{:}\mathsf{c}_n \vdash_m \mathsf{e}^{\mathsf{s}}{:}\mathsf{c} \rightsquigarrow \Gamma \mid \mathsf{e}^{\mathsf{b}}}{\vdash_m\ \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1 ... \mathsf{c}_n\ \mathsf{x}_n)\ \{\mathtt{return}\ \mathsf{e}^{\mathsf{s}};\} \rightsquigarrow \Gamma, \mathsf{c} \leq \mathsf{c}_0, \exists\, \mathsf{c}_{\mathsf{i}}{}^{\mathsf{i} \in 1..\mathsf{n}} \mid \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1 ... \mathsf{c}_n\ \mathsf{x}_n)\ \{\mathtt{return}\ \mathsf{e}^{\mathsf{b}};\}}$$

$$\textit{(m-parameter)}\ \frac{\Pi \vdash_m \mathsf{x}{:}\mathsf{c}}{\Pi \vdash_m \mathsf{x}{:}\mathsf{c} \rightsquigarrow \epsilon \mid \mathsf{x}} \qquad \textit{(m-field access)}\ \frac{\Pi \vdash_m \mathsf{e}^{\mathsf{s}}{:}\mathsf{c} \rightsquigarrow \Gamma \mid \mathsf{e}^{\mathsf{b}}}{\Pi \vdash_m \mathsf{e}^{\mathsf{s}}.\mathsf{f}{:}\mathsf{c}' \rightsquigarrow \Gamma, \phi(\mathsf{c}, \mathsf{f}, \mathsf{c}') \mid \mathsf{e}^{\mathsf{b}}[\mathsf{c}.\mathsf{f}\ \mathsf{c}']}$$

$$\textit{(m-meth call)}\ \frac{\begin{array}{c}\Pi \vdash_m \mathsf{e}_0^{\mathsf{s}}{:}\mathsf{c}_0 \rightsquigarrow \Gamma_0 \mid \mathsf{e}_0^{\mathsf{b}} \\ \Pi \vdash_m \mathsf{e}_{\mathsf{i}}^{\mathsf{s}}{:}\mathsf{c}_i \rightsquigarrow \Gamma_{\mathsf{i}} \mid \mathsf{e}_{\mathsf{i}}^{\mathsf{b}}\ \forall i \in 1..n\end{array}}{\Pi \vdash_m \mathsf{e}_0^{\mathsf{s}}.\mathsf{m}(\mathsf{e}_1^{\mathsf{s}} ... \mathsf{e}_{\mathsf{n}}^{\mathsf{s}}){:}\mathsf{c} \rightsquigarrow \Gamma_0, \Gamma_1, \ldots, \Gamma_{\mathsf{n}}, \mu(\mathsf{c}_0, \mathsf{m}, \mathsf{c}_1 ... \mathsf{c}_{\mathsf{n}}, (\mathsf{c}, \bar{\mathsf{c}}')) \mid \mathsf{e}_0^{\mathsf{b}}[\mathsf{c}_0.\mathsf{m}(\bar{\mathsf{c}}')\mathsf{c}](\mathsf{e}_1^{\mathsf{b}} ... \mathsf{e}_{\mathsf{n}}^{\mathsf{b}})}$$

$$\textit{(m-new)}\ \frac{\Pi \vdash_m \mathsf{e}_{\mathsf{i}}^{\mathsf{s}}{:}\mathsf{c}_i' \rightsquigarrow \Gamma_{\mathsf{i}} \mid \mathsf{e}_{\mathsf{i}}^{\mathsf{b}}\ \forall i \in 1..n}{\Pi \vdash_m \mathtt{new}\ \mathsf{c}(\mathsf{e}_1^{\mathsf{s}} ... \mathsf{e}_{\mathsf{n}}^{\mathsf{s}}){:}\mathsf{c} \rightsquigarrow \Gamma_1, \ldots, \Gamma_{\mathsf{n}}, \kappa(\mathsf{c}, \mathsf{c}_1' ... \mathsf{c}_{\mathsf{n}}', \bar{\mathsf{c}}) \mid \mathtt{new}\ [\mathsf{c}\ \bar{\mathsf{c}}](\mathsf{e}_1^{\mathsf{b}} ... \mathsf{e}_{\mathsf{n}}^{\mathsf{b}})}$$

$$\textit{(m-down cast)}\ \frac{\Pi \vdash_m \mathsf{e}^{\mathsf{s}}{:}\mathsf{c}' \rightsquigarrow \Gamma \mid \mathsf{e}^{\mathsf{b}}}{\Pi \vdash_m (\mathsf{c})\mathsf{e}^{\mathsf{s}}{:}\mathsf{c} \rightsquigarrow \Gamma, \mathsf{c} \leq \mathsf{c}' \mid (\mathsf{c})\mathsf{e}^{\mathsf{b}}}\ \mathsf{c} \neq \mathsf{c}' \qquad \textit{(m-up cast)}\ \frac{\Pi \vdash_m \mathsf{e}^{\mathsf{s}}{:}\mathsf{c}' \rightsquigarrow \Gamma \mid \mathsf{e}^{\mathsf{b}}}{\Pi \vdash_m (\mathsf{c})\mathsf{e}^{\mathsf{s}}{:}\mathsf{c} \rightsquigarrow \Gamma, \mathsf{c}' \leq \mathsf{c}, \exists\, \mathsf{c} \mid \mathsf{e}^{\mathsf{b}}}$$

Fig. 4. Monomorphic separate compilation

constraint $\mu(\mathsf{c}, \mathsf{m}, \bar{\mathsf{c}}, (\mathsf{t}, \bar{\mathsf{t}}))$. Since the receiver and argument types are determined (indeed, they are class names and not variables), it is possible to directly check whether the method call specified by the constraint is correct w.r.t. Δ. If so, we need to match t and $\bar{\mathsf{t}}$ against the return type and the type of the parameters, respectively, of the method m found in Δ; clearly, such a matching can be satisfied by one substitution at most (see the straightforward definition of *match* in Fig. 7). The function *meth* performs standard method look-up, and checks whether the types of the arguments are compatible with the method found; therefore, the function can fail either if the method could not be found,[7] or if the types of the arguments are not compatible with the found method. Similar considerations apply to the other two forms of type constraints, *i.e.,* $\phi(\mathsf{c}, \mathsf{f}, \mathsf{t})$ and $\kappa(\mathsf{c}, \bar{\mathsf{c}}, \bar{\mathsf{t}})$.

When *entails* successfully processes a type constraint in Γ', it applies the corresponding computed substitution σ' to the rest of Γ', and merges it with the global substitution σ computed so far. Note that the domains of σ and σ' are disjoint, therefore $\sigma \cup \sigma'$ is always well defined. Indeed, the domain of σ and the set of type variables occurring in Γ' are always disjoint. When *entails* has successfully processed all constraints, then it succeeds and returns the computed substitution σ. The algorithm can fail in two cases: either there exists a determined type constraint γ s.t. $\mathit{process}(\gamma, \Delta)$ fails, or Γ' is not empty, and contains only undetermined constraints.[8]

The correctness of *entails* is formalized by the following claim.

[7] This can happen either if *meth* reaches the `Object` class, or (in case Δ is not well-formed) if it reaches an undefined class, or an already visited class.

[8] Note, that the latter case would not happen if *entails* were only "called" by rule *(program)*, since in this case, *entails* would only be applied to environments of the form $\Gamma_1, \ldots, \Gamma_{\mathsf{n}}$, where each Γ_{i} has been inferred by compiling a certain class declaration.

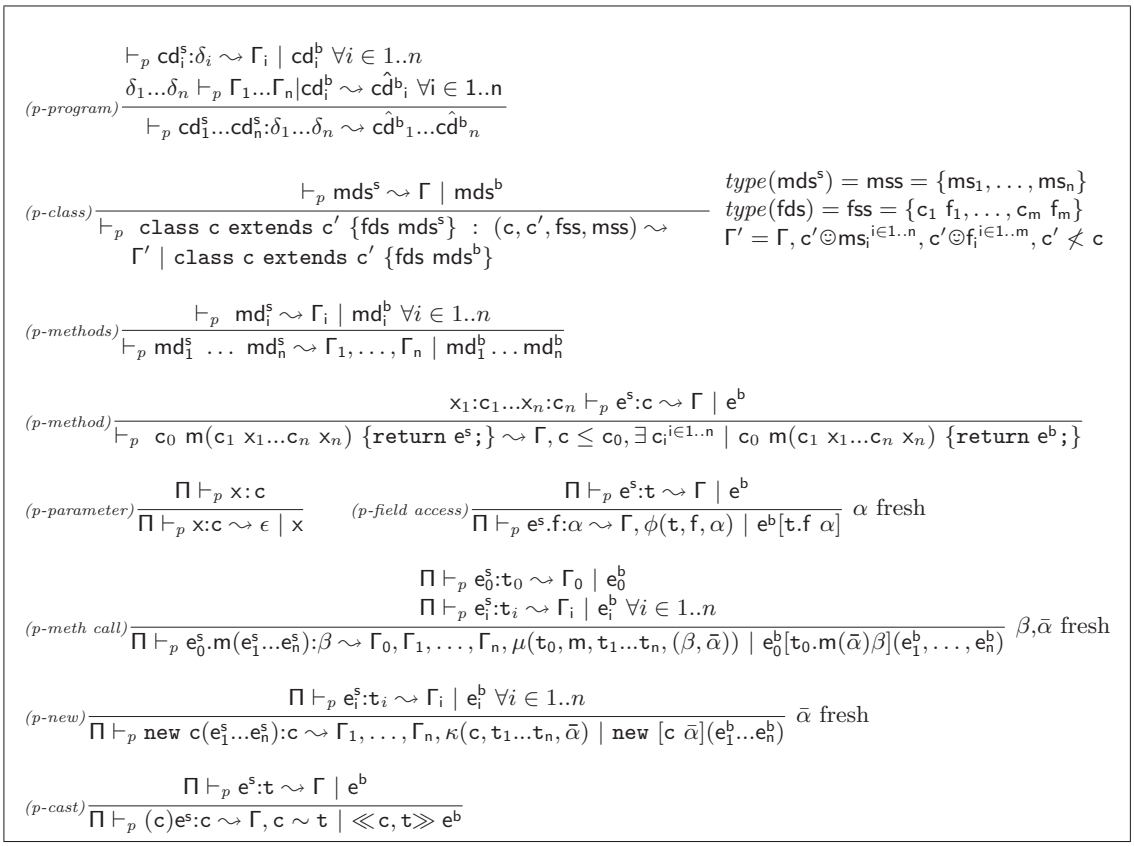

Fig. 5. Polymorphic separate compilation

Claim. Let $\vdash \mathsf{cd}_i^s{:}\delta_i \rightsquigarrow \Gamma_i \mid \mathsf{cd}_i^b$ be a valid judgment, for all $i \in 1..n$ and let $\Delta = \delta_1, \ldots, \delta_n$, and $\Gamma = \Gamma_1, \ldots, \Gamma_n$. Then,

1. the selection order of determined constraints in Γ does not affect the outcome of $entails(\Delta, \Gamma)$;
2. if $entails(\Delta, \Gamma) = \sigma$, then $\Delta \vdash_p \sigma(\Gamma)$ holds;
3. if $entails(\Delta, \Gamma)$ fails, then there exists no σ s. t. $\Delta \vdash_p \sigma(\Gamma)$ holds.

5 Conclusion

The main contribution of this paper is a type inference algorithm, that derives exact type requirements for inter-checking in a Java-like language. To our knowledge, ours is the first such algorithm. For simplicity, we have illustrated our approach on a simple language; however, the basic idea can be generalized with no substantial difficulty to other features, such as field shadowing and method overloading.

This result can be exploited in several, different ways. Firstly, it can be directly applied to the development of alternative compilation mechanisms for Java-like languages based on intra-checking and inter-checking phases. Such compilation mechanisms would support separate compilation in the *absence* of any information on the imported classes, whereas in the previous approach in [2, 3] type constraints had to be provided by the programmer. Secondly, it can be applied in selective recompilation mechanisms, in the same spirit of [7, 8], but with the difference that recompilation only amounts to bytecode instantiation. Finally, execution of bytecode containing type variables

Input: Δ class type environment, Γ type constraint environment
Output: either succeeds by finding a unique substitution σ s.t. $\Delta \vdash_p \sigma(\Gamma)$ holds, or fails
Pseudo-code:

```
entails(Δ, Γ){
  Γ' := Γ; σ := ε
  while Γ' ≠ ε {
    if ∃ γ ∈ Γ' s. t. γ is determined {
      σ' := process(γ, Δ)
      Γ' := σ'(Γ' \ {γ})
      σ := σ ∪ σ'
    }
    else fail
  }
  return σ
}
```

```
process(γ, Δ){
  if γ is ground and Δ ⊢_p γ holds
    return ε
  else if γ = φ(c, f, t) {
    c' := field(Δ, c, f); return match(c', t)
  }
  else if γ = κ(c, c̄, t̄) {
    c̄' := cons(Δ, c, c̄); return match(c̄', t̄)
  }
  else if γ = μ(c, m, c̄, (t, t̄)) {
    (c, c̄') := meth(Δ, c, m, c̄)
    return match((c', c̄'), (t, t̄))
  }
}
```

Fig. 6. Linking algorithm

```
match(c̄, t̄){
  c̄' := c̄; t̄' := t̄; σ := ε
  while c̄' = c, c̄'' and t̄' = t, t̄'' {
    if t is a variable α { t̄' := {α ↦ c}(t̄''); σ := σ ∪ {α ↦ c} }
    else if t = c {t̄' := t̄''}
    else fail
    c̄' := c̄''
  }
  if c̄' = ε and t̄' = ε return σ else fail
}
```

Fig. 7. Definition of the *match* function

could either replace all type variables first, in a step corresponding to inter-checking, or could substitute type variables lazily, during dynamic linking and loading; some initial exploration appears in the companion paper [4].

Further work also includes the obvious extensions of our polymorphic model, e.g., to encompass field hiding and overloading, and also, the extension of the source language so that it may contain type variables as well.

References

1. D. Ancona and G. Lagorio. Stronger Typings for Smarter Recompilation of Java-like Languages. *Journal of Object Technology*, 2004. To appear.
2. 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.
3. 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.

4. Alex Buckley and Sophia Drossopoulou. Flexible Dynamic Linking. In *6th Intl. Workshop on Formal Techniques for Java Programs 2004*, June 2004.
5. L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, 1997.
6. 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.
7. 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.
8. 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.
9. 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.