Principal Typings for Java-like Languages*

Davide Ancona
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
davide@disi.unige.it

Elena Zucca
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
zucca@disi.unige.it

Abstract

The contribution of the paper is twofold. First, we define a general notion of type system equipped with an entailment relation between type environments; this generalisation serves as a pattern for instantiating type systems able to support separate compilation and inter-checking of Java-like languages, and allows a formal definition of soundess and completeness of inter-checking w.r.t. global compilation. These properties are important in practice since they allow selective recompilation. In particular, we show that they are guaranteed when the type system has principal typings and provides sound and complete entailment relation between type environments.

The second contribution is more specific, and is an instantiation of the notion of type system previously defined for Featherweight Java with method overloading and field hiding. The aim is to show that it is possible to define type systems for Java-like languages, which, in contrast to those used by standard compilers, have principal typings, hence can be used as a basis for selective recompilation.

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

Keywords: principal typings, selective recompilation, Java-like languages

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

POPL'04, January 14–16, 2004, Venice, Italy.

1 Introduction

The fact that *separate compilation* is a highly desirable property is a generally accepted principle. However, as pointed out in the seminal Cardelli's paper [4], even though module mechanisms have received considerable theoretical attention, the notion of separate compilation and the associated notion of linking have not been emphasized, and there is little work on formal models for them; as a consequence, despite of the popularity of the word, it is often difficult to establish in a precise way whether a programming environment actually supports separate compilation or not.

The mentioned paper [4] can be considered a milestone in this direction and is based on the definition of a simple formal framework where separate compilation, which is there simplified to typechecking, is modeled by a judgment $\Gamma \vdash \mathtt{s} : \tau$. The intended meaning is that $\mathtt{s}$ is a source fragment assumed to be open, that is, to contain references to names defined in other fragments, τ is the resulting type, and Γ is a type environment intuitively containing all the assumptions on other fragments needed to typecheck $\mathtt{s}$. In this paper we are also interested in code generation, since, as we will show later, in Java-like languages different bytecode is produced under different assumptions in Γ; hence, we model separate compilation by a judgment $\Gamma \vdash \mathtt{s} : \tau \leadsto \mathtt{b}$ where $\mathtt{b}$ is the binary fragment generated by the compilation of $\mathtt{s}$.

A source fragment is a compilation unit, and exports one or more names to other fragments. For instance, in the case of Java-like languages, the most elementary (non-empty) compilation unit corresponds just to a class declaration, but several class declarations can be part of the same compilation unit as well, as happens for all Java systems. Hence, in general, $\mathtt{s}$ will be a sequence of declarations (e.g., class declarations in Java) and τ a sequence of types for the declared class names.

At this point, given a collection of successfully compiled fragments, it is possible to test whether they successfully *inter-check*, that is, the mutual assumptions between fragments are satisfied. Formally, we have a *linkset* $\Gamma_i \vdash \mathtt{s}_i : \tau_i \leadsto \mathtt{b}_i{}^{i \in 1..n}$ and we have to check that, for each $i \in 1..n$, assumptions Γ_i required by $\mathtt{s}_i$ are matched by other fragments[1], in a sense to be made precise depending on the nature of the type assumptions.

For instance, in the simple case in which a type environment is just

[1]Here we simplify the presentation by considering only self-contained linksets; in the following, linksets will be possibly open, that is, they will also include a type environment containing assumptions on the external fragments.

a sequence of pairs $c : \tau$ meaning that the entity (e.g., class) named c should have type τ, inter-checking just amounts to check that, for each $i, j \in 1..n$, if $c : \tau$ appears in Γ_j, then τ must be the same type c has in τ_i (as it is in [4]). However, a type environment could in general contain other kinds of type assumptions, such as subtyping assumptions $c_1 \leq c_2$ or others depending on the language we are considering. Hence, we need a definition of inter-checking which abstracts from the particular form of type assumptions.

In this paper we provide such a definition (Definition 8), based on the idea that an effective inter-checking procedure can be modeled by an *entailment* relation $\vdash$ on type environments, so that inter-checking succeeds if $c_1 : \tau_1, \ldots c_m : \tau_m \vdash \Gamma_i$ holds for all $i \in 1..n$, where $c_1, \ldots, c_m$ are all the classes declared in the linkset and, for all $j \in 1..m$, τ_j is the type derived for c_j in the linkset. Intuitively, this means that it is possible to prove all the required type assumptions whenever fragment types are those in the linkset.

The advantages of separate compilation plus inter-checking w.r.t. global compilation of $s_1, \ldots, s_n$ altogether are clear. Each fragment can be compiled without inspecting the fragments it depends on; then, a collection of fragments can be put together to form an executable application by just considering the type information (type environment and type) of fragments, without any need of re-inspecting code. However, in order to really offer these advantages, inter-checking should satisfy some properties which ensure that it can actually replace global compilation. This issue was not considered in [4] and its formalization in the abstract framework for inter-checking described above is the first main contribution of this paper.

Of course, inter-checking should at least be *sound* w.r.t. global compilation, in the sense that, if for some linkset $\Gamma_i \vdash s_i : \tau_i \rightsquigarrow b_i^{\,i \in 1..n}$ inter-checking succeeds, then we can be sure that compiling altogether $s_1, \ldots, s_n$ we successfully get the same binary fragments. This is a minimal property which we expect to be always satisfied by separate compilation (Definition 10), and which is guaranteed under the hypothesis that the entailment relation is sound (Theorem 11).

Consider now the situation in which inter-checking fails. This means that there is some type assumption in some Γ_i which is not satisfied by the types of other fragments. However, this does not necessarily mean that the fragments cannot be safely linked. Indeed, in general for each pair (s,b), many judgments $\Gamma \vdash s : \tau \rightsquigarrow b$ can be derived, and for some fragment we could have taken a too restrictive type environment (that is, containing unnecessary type assumptions).

We can be sure that this is not the case only if the typing (Γ, τ) gives *all* the type information about (s,b), that is, represents *all* possible typings of (s,b); in other words, Γ contains only the type assumptions that are strictly needed for compiling s generating b. This property can be expressed by saying that (Γ, τ) is a *principal typing* of (s,b), and has been recently formalized in a general setting (that is, independent of the particular type system we are considering) in [14]. If (Γ_1, τ_1), ..., (Γ_n, τ_n) are principal typings for (s_1, b_1), ..., (s_n, b_n), respectively, then we can be sure that no further type information about fragments can be obtained by re-inspecting the code, hence failure of inter-checking was not due to our particular choice of typings. Hence, we can conclude that global compilation would either fail as well, or would produce different binary fragments.

We will call *completeness* of inter-checking w.r.t. global compilation the fact that, with a suitable choice of typings in linksets, failure of inter-checking guarantees that we could not generate the binary fragments in the linkset by global compilation (Definition 12). We show that a sufficient condition for this is that the type system supports principal typings and complete environment entailment relation (Theorem 14).

The second contribution of this paper is more specific, and is an instantiation of the notion of type system previously defined for Featherweight Java [8] enriched by overloading and hiding. The aim is to show that it is possible to define type systems for Java-like languages, which, in contrast to those used by standard compilers, have principal typings, hence can be used as a basis for selective recompilation.

We briefly explain the essence of the problem of finding principal typings for Java-like languages on a simple example (more extended discussions and other examples can be found in [3, 2]).

Consider the following class:

```
class C extends Parent {
...
    Type1 m (Type2 x) { return new Used().g(x);}
}
```

Let us wonder which is the minimal type information on other classes needed for typechecking the class and generating the corresponding bytecode.

For classes `Type1` and `Type2`, since they are just used as pure types, it is enough to assume that they exist (we will model this in our type system by type assumptions $\exists$`Type1`, $\exists$`Type2`).

Looking at the method call, we can say that the class `C` can be typechecked in any type environment where a class `Used` is available which provides, besides the default constructor, a method (either directly declared or inherited) with name `g` and one parameter of a supertype of `Type2`; moreover we have the constraint that its return type must be a subtype of `Type1`. For instance, class `C` can be typechecked in the following context (1):

```
class Parent{}
class Type1{}
class Type2 extends Type1{}
class Type3 extends Type2{}
class UsedParent { Type3 g(Type1 x) { ...}}
class Used extends UsedParent {}
```

and also in this context (2):

```
class Parent{}
class Type1{}
class Type2 extends Type1{}
class Used {
    Type2 g(Type2 x) {...}
    int f() {...}
}
```

Hence, we would be tempted to express this by a type assumption expressing that class `Used` must have a method `g` with one parameter of a supertype of `Type2` and return type subtype of `Type1`. However, in order to produce the corresponding bytecode, a Java compiler must know *exactly* which are the parameter and return type of the method which will be selected, since they appear as annotations in bytecode (see Section 3 for more details). In the example, `C` can be typechecked, e.g., in a type environment Γ_1 where the method `g`

selected for method invocations with receiver type `Used` and argument type `Type2` has return type `Type3`, and parameter type `Type1`, as in environment (1); this constraint is formalized by the judgment $\Gamma_1 \vdash \texttt{Used.g(Type2)} \overset{\texttt{m-res}}{\rightarrow} (\texttt{Type1},\texttt{Type3})$.

As well, `C` can be typechecked in a type environment Γ_2 where the selected method has return and parameter type `Type2`, as in environment (2); this constraint is formalized by the judgment $\Gamma_2 \vdash \texttt{Used.g(Type2)} \overset{\texttt{m-res}}{\rightarrow} (\texttt{Type2},\texttt{Type2})$.

The example clearly shows that, in order to get a principal typing property (in particular, a "minimal" type environment), we must type pairs consisting of a source and a binary fragment.

The rest of the paper is organized as follows: in Section 2 we define the formal notions of type system for separate compilation and inter-checking and soundness and completeness of inter-checking. In Section 3 we define an instantiation of the notion of type system defined in the previous section for Featherweight Java [8] with method overloading and field hiding. In Section 4 we prove that this type system satisfies the hypotheses which guarantee soundness and completeness of inter-checking. We also prove that it has principal typings and that the environment entailment is complete. Finally in the Conclusion we summarize the contribution of the paper and draw some direction for further work.

2 Type systems for separate compilation

In this section we define a general notion of type system for separate compilation.

The main motivation is reuse: this general notion of type system serves as a pattern to be instantiated by a "real" type system where all definitions and details which have been intentionally omitted here are provided (including, e.g., the syntax of terms and types, and the typing rules for judgments). However, each correct instantiation (as we will see, there are some basic properties expected to hold) is guaranteed to support well selective recompilation [1]. We will denote by `T` a generic instantiation of our general notion of type system.

Even though in this paper we define just one instantiation (for Featherweight Java [8] enriched by overloading and hiding, see Section 3), we expect our general notion of type system to be useful for a number of other possible instantiations including both more significant subsets of Java and C# and toy languages defined for studying the interaction of Java or C# with advanced features like, for instance, generic types that will be soon included in Java and have been formally studied with GJ [8].

2.1 Basic notions

We start by listing the basic syntax categories and typing judgments that are expected to be defined by every instantiation.

Basic syntax categories

Each instantiation should at least define the following sets (metavariables used for the elements of such sets are shown in parentheses):

- Class names (`c`).
- (Sequences of) source class declarations (`s`).
- (Sequences of) binary class declarations (`b`).
- (Sequences of) class types (τ).
- Type assumptions (γ). They always include the type assumptions of the form $\texttt{c}{:}\tau$ which are called *standard*.
- Type environments (Γ). An environment is just a possibly empty sequence of type assumptions $\gamma_1,\ldots,\gamma_n$.

We assume that each (source/binary) class declaration introduces a class name `c` that can be extracted by a function *name* mapping a sequence of source or binary class declarations to the sequence of their corresponding names. As already explained in the Introduction, binaries are needed for modeling the situation where some source class modification can change the binary generated from other source classes.

Notation for sequences

We denote by $|\sigma|$ the length of a sequence σ, by σ_1,σ_2 the concatenation of the two sequences σ_1 and σ_2. A sequence is written either $e_1,\ldots,e_n$ or $e_i^{i\in 1..n}$; however, the first notation is only used when there is no ambiguity with concatenation.

Basic judgments

Each instantiation should at least define the following two judgments:

- $\Gamma\vdash \texttt{s}{:}\tau\rightsquigarrow\texttt{b}$: source class declarations `s` *compile* to `b` and have type τ in Γ. We assume that if $\Gamma\vdash \texttt{s}{:}\tau\rightsquigarrow\texttt{b}$ is valid, then $|\texttt{s}| = |\tau| = |\texttt{b}| = n$, $\mathit{name}(\texttt{s}) = \mathit{name}(\texttt{b}) = \texttt{c}_1,\ldots,\texttt{c}_n$, with $\texttt{c}_i \neq \texttt{c}_j$ for all $i,j \in 1..n$, $i \neq j$. Note that for Java-like languages the information about the inferred type τ is, in a sense, redundant, since it does not depend on Γ, but it is a function of just the source `s`; nevertheless, we have preferred to leave this information in the judgment for readability.
- $\Gamma_1 \vdash \Gamma_2$: Γ_1 *entails* Γ_2, that is, Γ_1 enforces stronger type requirements than those of Γ_2.

Intuitively, the notion of entailment should correspond to a computable relation (at least) sound w.r.t. the notion of stronger environment (see Definition 4 in Section 2.2).

A basic expected property of the compilation judgment is *compositionality*.

Let the expression $\mathit{env}(\texttt{s}{:}\tau)$ denote the type environment: $\texttt{c}_1{:}\tau_1,\ldots,\texttt{c}_n{:}\tau_n$ if $\mathit{name}(\texttt{s}) = \texttt{c}_i^{i\in 1..n}$, $\tau = \tau_i^{i\in 1..n}$ and `s` does not contain class name conflicts (that is, $\texttt{c}_i = \texttt{c}_j$ implies $i = j$, for all $i,j \in 1..n$) otherwise, it is undefined[2].

Def. 1 (COMPOSITIONALITY). *We say that `T` is* compositional *iff for all* Γ, $s = s_1,\ldots,s_n$, $\tau = \tau_1,\ldots,\tau_n$, $b = b_1,\ldots,b_n$: $\Gamma\vdash s{:}\tau\rightsquigarrow b \Leftrightarrow \Gamma,\mathit{env}(s{:}\tau)\vdash s_i{:}\tau_i\rightsquigarrow b_i$, *for all* $i \in 1..n$.

2.2 Principal typings

The system independent definition of principal typing given by Wells [14] fits well our general notion of type system. We recall

[2] Hence a judgment of the form $\Gamma,\mathit{env}(\texttt{s}{:}\tau)\vdash\ldots$ is valid if and only if $\mathit{env}(\texttt{s}{:}\tau)$ is defined and denotes a type environment Γ' s.t. $\Gamma,\Gamma'\vdash\ldots$ is valid.

here the basic notions and notations on principal typings inspired by Wells and adapted to our purposes.

Def. 2 (TYPING). *If* $\Gamma \vdash s{:}\tau \rightsquigarrow b$ *holds, then we say that the pair* (Γ,τ) *is a* typing *of* (s,b). *We say that* (s,b) *is* typable *iff it has a typing.*

Note that we could have adopted Well's definition of typing by considering binary sequences as part of the type so that $(\Gamma,(\tau,\mathtt{b}))$ is a typing of $\mathtt{s}$ if $\Gamma \vdash \mathtt{s}{:}\tau \rightsquigarrow \mathtt{b}$ holds. However, this definition would lead to a rather strong definition of principal typing for Java-like languages that, in fact, would not be satisfied by any system adopting the usual notion of bytecode (see the Conclusion).

Def. 3 (CONSISTENT ENVIRONMENT). *An environment is* consistent *iff there exist* s, τ, *and* b *s.t.* $\Gamma \vdash s{:}\tau \rightsquigarrow b$.

Def. 4 (STRONGER ENVIRONMENT). *An environment* Γ_1 *is* stronger *than* Γ_2 *(written* $\Gamma_1 \leq \Gamma_2$*) iff* Γ_2 *is consistent and for all* s, τ, *and* b, *if* $\Gamma_2 \vdash s{:}\tau \rightsquigarrow b$ *holds, then* $\Gamma_1 \vdash s{:}\tau \rightsquigarrow b$ *holds as well.*

Note that the relation of Definition 4 is a pre-order, but, in general, is not a partial order.

Def. 5 (STRONGER TYPING). *A typing* (Γ_1,τ_1) *is* stronger *than* (Γ_2,τ_2) *(written* $(\Gamma_1,\tau_1) \leq (\Gamma_2,\tau_2)$*) iff* $\Gamma_2 \leq \Gamma_1$ *and* $\tau_1 = \tau_2$.

The definition of stronger typing given here differs from Well's definition in two respects:

- Well's definition does not require that if (Γ_1,τ_1) is stronger than (Γ_2,τ_2), then Γ_2 is stronger than Γ_1 and τ_1 equals τ_2. However, this stronger property clearly holds in the setting of Java-like languages where the type of a class is uniquely determined by the annotations contained in its body. Under this property, the notion of stronger typing can be simply captured by the notion of entailment between environments (see Theorem 14).
- In Well's definition there is no notion of consistent type/environment. However, if non-consistent types and environments were not ruled out from Definition 4, then some expected completeness property would not hold, like, for instance, $\Gamma_1 \leq \Gamma_2 \Rightarrow \Gamma_1 \vdash \Gamma_2$; indeed, we may not want a system where $\Gamma_1 \vdash \Gamma_2$ is provable for any Γ_2, just because Γ_1 is not consistent. On the other hand, we would like to consider concatenation of environments as a total function, therefore non-consistent environments cannot simply ruled out from all definitions.

Def. 6 (PRINCIPAL TYPING). *A* principal typing *of* (s,τ) *is a typing of* (s,τ) *which is stronger than all typings of* (s,τ).
We say that T has principal typings *iff all typable* (s,b) *have a principal typing.*

Finally, the definition of principal typing given here is strictly stronger than Well's definition; indeed, our definition could be regarded as a refinement of Well's principality suitable for type systems in the Church style (that is, with explicitly typed terms).

2.3 Linksets

Selective recompilation tries to minimize compilation steps after changes to a certain software configuration. Software configurations can be modeled by the notion of *linkset*, which was firstly introduced by Cardelli [4].

Def. 7 (LINKSET). *A* linkset *is a pair, written*

$$\Gamma \mid \Gamma_i \vdash s_i{:}\tau_i \rightsquigarrow b_i{}^{i \in 1..n}$$

consisting of a type environment and a (possibly empty) sequence of valid compilation judgments s.t. $s = s_1,\ldots,s_n$ *does not contain class name conflicts.*

Intuitively, the environment Γ contains the type assumptions on the external classes (that is, not defined in the linkset), whereas for all $i \in 1..n$ the judgment $\Gamma_i \vdash \mathtt{s}_i{:}\tau_i \rightsquigarrow \mathtt{b}_i$ corresponds to the successful compilation of a single compilation unit $\mathtt{s}_i$ to $\mathtt{b}_i$ in the type environment Γ_i.

Since here the emphasis is on inter-checking, the definition of linksets assumes that the compilation judgments are valid, hence our linksets correspond to *intra-checked* Cardelli's linksets [4]. Moreover, in [4] type environments are just sequences of standard type assumptions $\mathtt{c} : \tau$, and class names in Γ need to be different from those in each Γ_i. Indeed, typechecking of a single fragment $\mathtt{s}_i$ is performed in the type environment Γ,Γ_i containing type assumptions on external classes and classes in the linkset, respectively. In our notion of linkset, instead, type environments contain arbitrary type assumptions, each one possibly involving more than one class, and typechecking of $\mathtt{s}_i$ is performed in the type environment Γ_i which contains type assumptions on both external classes and classes in the linkset. Thus, Γ,Γ_i can contain redundant assumptions, even though intuitively the best situation occurs when Γ_i contains exactly the minimal type assumptions on other classes needed to compile $\mathtt{s}_i$ and Γ contains exactly the minimal type assumptions on external classes needed to compile all $\mathtt{s}_i$.

Finally, judgments are not named as in Cardelli's linksets, since the type environment exported by any compilation unit $\Gamma \vdash \mathtt{s}{:}\tau \rightsquigarrow \mathtt{b}$ is simply obtained via the *name* function.

The definition of inter-checking is a generalization of that given by Cardelli.

Def. 8 (LINKSET INTER-CHECKING). *Let*

$$L = \Gamma \mid \Gamma_i \vdash s_i{:}\tau_i \rightsquigarrow b_i{}^{i \in 1..n}$$

be a linkset and set $s = s_1,\ldots,s_n$, $\tau = \tau_1,\ldots,\tau_n$. *We say that* L inter-checks *(written* $\vdash L \diamond$*) iff* $\Gamma, env(s{:}\tau) \vdash \Gamma_i$ *holds for all* $i \in 1..n$.

2.4 Sound and complete inter-checking

As already explained, the inter-checking procedure allows separate compilation of the units which need to be assembled in the linkset, and prevents code inspection and recompilation, since the overall consistency of the linkset is checked via the entailment relation on environments which completely relies on unit interfaces. On the other hand, one could always adopt a "brute force" algorithm by (re)compiling all units as a whole. We model global (re)-compilation by a judgment $\Gamma \vdash \mathtt{s}_1,\ldots,\mathtt{s}_n \overset{G}{\rightsquigarrow} \mathtt{b}_1,\ldots,\mathtt{b}_n$, expressing that source fragments $\mathtt{s}_1,\ldots,\mathtt{s}_n$ are compiled altogether, generating binary fragments $\mathtt{b}_1,\ldots,\mathtt{b}_n$ in the type environment Γ (see Def. 9).

Def. 9 (GLOBAL COMPILATION). *For all* Γ, s, b, *the judgment* $\Gamma \vdash s \overset{G}{\rightsquigarrow} b$ *is valid iff* $\Gamma \vdash s{:}\tau \rightsquigarrow b$ *can be proved for some* τ.

Of course we expect separate compilation plus inter-checking to produce the same binaries as we would have got from global com-

pilation; if so, we say that inter-checking is sound w.r.t. global compilation.

Def. 10 (SOUND INTER-CHECKING). *Inter-checking is* sound *w.r.t. global compilation iff for all linksets* $L = \Gamma|\Gamma_i \vdash s_i{:}\tau_i \rightsquigarrow b_i{}^{i\in 1..n}$ *if* $\vdash L\diamond$ *then* $\Gamma \vdash s_1,\ldots,s_n \overset{G}{\rightsquigarrow} b_1,\ldots,b_n$.

Soundness of inter-checking is guaranteed under some reasonable conditions: the type system should be compositional, and the entailment judgment should be sound with respect to the relation of stronger environment.

Theorem 11 (SOUNDNESS OF INTER-CHECKING). *Let* T *be a compositional type system satisfying the following additional property :*

$(*)$ $\Gamma_1 \vdash \Gamma_2 \Rightarrow \Gamma_1 \leq \Gamma_2$ *for all* Γ_1, Γ_2 *(entailment is sound).*

Then, inter-checking is sound w.r.t. global compilation.

PROOF. Let $\mathtt{L}$ be $\Gamma|\Gamma_i \vdash \mathtt{s}_i{:}\tau_i \rightsquigarrow \mathtt{b}_i{}^{i\in 1..n}$ be a linkset s.t. $\vdash \mathtt{L}\diamond$ holds and set $\mathtt{s} = \mathtt{s}_1,\ldots,\mathtt{s}_n$, $\tau = \tau_1,\ldots,\tau_n$. Then, by Definition 8, $\Gamma, env(\mathtt{s}{:}\tau) \vdash \Gamma_i$ holds for all $i \in 1..n$. By hypothesis $(*)$, $\Gamma, env(\mathtt{s}{:}\tau) \leq \Gamma_i$, therefore $\Gamma, env(\mathtt{s}{:}\tau) \vdash \mathtt{s}_i{:}\tau_i \rightsquigarrow \mathtt{b}_i$ for all $i \in 1..n$. Finally, by compositionality, $\Gamma \vdash \mathtt{s}_1,\ldots,\mathtt{s}_n \overset{\mathtt{G}}{\rightsquigarrow} \mathtt{b}_1,\ldots,\mathtt{b}_n$. □

From the point of view of selective recompilation, soundness of inter-checking ensures that recompilation steps are really unnecessary in case of successful inter-checking since they would lead to the same result. On the other hand, we would like to be sure that if inter-checking fails, then some recompilation step is really needed, so that it never happens that a recompilation step turns out to be useless. This happens if inter-checking is complete w.r.t. global compilation.

Def. 12 (COMPLETE INTER-CHECKING). *Inter-checking is* complete *w.r.t. global compilation iff, for all typable* (s,b)*, we can select a typing* $(\Gamma^{(s,b)}, \tau^{(s,b)})$ *of* (s,b) *s.t.*

for all linksets $L = \Gamma|\Gamma_i \vdash s_i{:}\tau_i \rightsquigarrow b_i{}^{i\in 1..n}$,
with $(\Gamma_i,\tau_i) = (\Gamma^{(s_i,b_i)}, \tau^{(s_i,b_i)})$, $i \in 1..n$,
if $\Gamma \vdash s_1,\ldots,s_n \overset{G}{\rightsquigarrow} b_1,\ldots,b_n$ *holds, then* $\vdash L\diamond$ *holds.*

Note that the property above is weaker than the opposite implication of Def. 10, which does not hold; indeed, for an arbitrary linkset, inter-checking could fail since for some fragment we have taken a too restrictive type environment. However, completeness as stated above requires that for each fragment we can select *a priori* a typing s.t., for any possible future context, failure of linking will ensure that we could not get the same binary fragments by global compilation.

Prop. 13 (COMPLETENESS OF INTER-CHECKING). *Let* T *be a compositional type system satisfying the following additional property:*

for all typable (s,b)*, there exists a* provably principal *typing of* (s,b)*, that is a typing* (Γ,τ) *of* (s,b) *s.t. for all typings* (Γ',τ') *of* (s,b), $\Gamma' \vdash \Gamma$ *and* $\tau = \tau'$.

Then, inter-checking is complete *w.r.t. global compilation.*

PROOF. Let us take, for all $(\mathtt{s},\mathtt{b})$, $(\Gamma^{(\mathtt{s},\mathtt{b})}, \tau^{(\mathtt{s},\mathtt{b})})$ a provably principal typing of $(\mathtt{s},\mathtt{b})$.

Let $\mathtt{L}$ be $\Gamma|\Gamma_i \vdash \mathtt{s}_i{:}\tau_i \rightsquigarrow \mathtt{b}_i{}^{i\in 1..n}$ with (Γ_i,τ_i) provably principal for $(\mathtt{s}_i,\mathtt{b}_i)$, $i \in 1..n$, and s.t. $\Gamma \vdash \mathtt{s} \overset{\mathtt{G}}{\rightsquigarrow} \mathtt{b}$ holds.

By compositionality:

for all $i \in 1..n$, $\Gamma, env(\mathtt{s}_1,\ldots,\mathtt{s}_n{:}\tau'_1,\ldots,\tau'_n) \vdash \mathtt{s}_i{:}\tau'_i \rightsquigarrow \mathtt{b}_i$,
for some $\tau'_1,\ldots,\tau'_n$.

Therefore, since (Γ_i,τ_i) is provably principal for $(\mathtt{s}_i,\mathtt{b}_i)$, $\tau_i = \tau'_i$ and $\Gamma, env(\mathtt{s}_1,\ldots,\mathtt{s}_n{:}\tau_1,\ldots,\tau_n) \vdash \Gamma_i$ hold for all $i \in 1..n$, hence $\vdash \mathtt{L}\diamond$ by Definition 8. □

The following is just a corollary of Theorem 13 stating that completeness of inter-checking holds whenever $\mathtt{T}$ is compositional, has principal typings and the entailment relation is complete.

Theorem 14. *Let* T *be a compositional type system with principal typings, satisfying the following additional property:*

$(**)$ $\Gamma_1 \leq \Gamma_2 \Rightarrow \Gamma_1 \vdash \Gamma_2$ *for all* Γ_1, Γ_2 *(entailment is complete).*

Then, inter-checking is complete w.r.t. global compilation.

PROOF. Let $(\mathtt{s},\mathtt{b})$ be typable; then by hypothesis $(\mathtt{s},\mathtt{b})$ has a principal typing (Γ,τ). By definition of principal typing, for all (Γ',τ') of $(\mathtt{s},\mathtt{b})$, $\tau = \tau'$ and $\Gamma' \leq \Gamma$, hence by hypothesis $(**)$, $\Gamma' \vdash \Gamma$. Finally, theorem 13 can be applied. □

2.5 Selective recompilation

In this section we illustrate more in detail the role of soundness and completeness of inter-checking for selective recompilation. Assume that in a compositional system $\mathtt{T}$ some of a successfully inter-checked linkset has been modified and recompiled, obtaining the new linkset $\Gamma|\Gamma_i \vdash \mathtt{s}_i{:}\tau_i \rightsquigarrow \mathtt{b}_i{}^{i\in 1..n}$. Of course, this change could have affected compatibility with some other fragment, therefore further recompilation steps could be required in principle. However, to avoid a pointless recompilation, we can use Definition 8; if all checks are passed, then by soundness we are sure that the modification did not affect any other fragment, hence any further recompilation step would be useless.

On the other hand, if inter-checking is not passed, and typings in the linkset are those selected according to Definition 12, then, by completeness, we know that for $\mathtt{s} = \mathtt{s}_1,\ldots,\mathtt{s}_n$, $\mathtt{b} = \mathtt{b}_1,\ldots,\mathtt{b}_n$, $\Gamma \vdash \mathtt{s} \overset{\mathtt{G}}{\rightsquigarrow} \mathtt{b}$ does not hold, hence, by completeness,

- either we simply introduced some name conflict, hence we obtain an ill-formed linkset;
- or $\exists i \in 1..n$ s.t. $\Gamma, env(\mathtt{s}{:}\tau) \vdash \mathtt{s}_i{:}\tau_i \rightsquigarrow \mathtt{b}_i$ is not valid.

In this latter case we recompile the *i*-th unit, since we are sure we will obtain either a different binary or a compilation error, but not the same result as before.

Note that it would be even better to be able to infer, in case of failure of inter-checking, whether recompilation would generate a different binary or a compilation error; indeed in this way we could avoid recompilation in the latter case and get an optimal procedure of selective recompilation. In languages where, differently from what happens in Java, changes to a fragment cannot affect other binary fragments, this is always the case since the former possibility does not hold. For Java-like languages, the same result could be achieved by introducing two different judgments, one for type-checking (not taking into account code generation) and one for compilation (that

$$
\begin{array}{rcl}
\mathtt{s} & ::= & \mathtt{CD}^s_1 \ldots \mathtt{CD}^s_n \\
\mathtt{CD}^s & ::= & \mathtt{class\ c\ extends\ c'\ \{\ FDS\ MDS}^s\ \} \\
\mathtt{FDS} & ::= & \mathtt{FD}_1 \ldots \mathtt{FD}_n \\
\mathtt{FD} & ::= & \mathtt{c\ f;} \\
\mathtt{MDS}^s & ::= & \mathtt{MD}^s_1 \ldots \mathtt{MD}^s_n \\
\mathtt{MD}^s & ::= & \mathtt{MH}\,\{\mathtt{return\ E}^s\mathtt{;}\} \\
\mathtt{MH} & ::= & \mathtt{c}_0\ \mathtt{m}(\mathtt{c}_1\ \mathtt{x}_1, \ldots, \mathtt{c}_n\ \mathtt{x}_n) \\
\mathtt{E}^s & ::= & \mathtt{x} \mid \mathtt{E}^s.\mathtt{f} \mid \mathtt{E}^s_0.\mathtt{m}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n) \\
 & & \mid \mathtt{new\ c}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n) \mid (\mathtt{c})\mathtt{E}^s \\
 & & \\
\mathtt{b} & ::= & \mathtt{CD}^b_1 \ldots \mathtt{CD}^b_n \\
\mathtt{CD}^b & ::= & \mathtt{class\ c\ extends\ c'\ \{\ FDS\ MDS}^b\ \} \\
\mathtt{MDS}^b & ::= & \mathtt{MD}^b_1 \ldots \mathtt{MD}^b_n \\
\mathtt{MD}^b & ::= & \mathtt{MH}\,\{\mathtt{return\ E}^b\mathtt{;}\} \\
\mathtt{E}^b & ::= & \mathtt{x} \mid \mathtt{E}^b \ll \mathtt{c.f\ c'} \gg \\
 & & \mathtt{E}^b_0 \ll \mathtt{c.m}(\overline{\mathtt{c}})\mathtt{c'} \gg (\mathtt{E}^b_1, \ldots, \mathtt{E}^b_n) \\
 & & \mid \mathtt{new} \ll \mathtt{c}\ \overline{\mathtt{c}} \gg (\mathtt{E}^b_1, \ldots, \mathtt{E}^b_n) \mid (\mathtt{c})\mathtt{E}^b \\
\overline{\mathtt{c}} & ::= & \mathtt{c}_1, \ldots, \mathtt{c}_n
\end{array}
$$

Implicit assumptions:

- field names in FDS are distinct
- parameter types of methods with the same name are distinct in $\mathtt{MDS}^s$ and $\mathtt{MDS}^b$
- parameter names in MH are distinct

Figure 1. Syntax

introduced in next section). See the Conclusion for more on this point.

3 Separate compilation for FJ

3.1 Syntax

The language we consider at the source level is an extended version of Featherweight Java [8], shortly FJ in the following. More precisely, we keep the same syntax, but take a more liberal type system allowing *field hiding* (a heir class can declare a field already present in the parent; the new field hides the inherited field, which can only be recovered by an up cast[3]) and *method overloading* (a class can have many methods, either directly declared or inherited, with the same name and different parameter types; they are considered as different methods and the right method associated to an invocation, if any, is determined by the rules for *overloading resolution*, see in the sequel).

We include these features from full Java since they are significant for the problem we are studying. Indeed, in both cases, the type which can be assigned to an expression in a fragment and the corresponding generated bytecode cannot be determined by simply inspecting the fragment, but depend on the context, as explained at the end of the Introduction.

The syntax of the language is defined in Figure 1; metavariables c, f, m and x range over sets of class, field, method and parameter names, respectively.

A source fragment s is a sequence of class declarations, each one consisting of the name of the class, the name of the superclass, a sequence of field declarations FDS and a sequence of method declarations $\mathtt{MDS}^s$. If $\mathtt{c}'$ is the superclass of c, then we also say that c *(directly) extends* $\mathtt{c}'$, and *extends* any class $\mathtt{c}''$ which $\mathtt{c}'$ (directly) extends. We assume a distinguished class name Object, denoting the root of the inheritance hierarchy, which cannot be declared.

A field declaration FD consists of the type and the name of the declared field. A method declaration $\mathtt{MD}^s$ consists of a method header and a method body (an expression). A method header MH consists of a (return) type, a method name and a sequence of parameter types and names. There are five kinds of expression: parameter name, field access, method invocation, instance creation and cast. Types of expressions are class names, and c is a subtype of $\mathtt{c}'$ iff either c extends $\mathtt{c}'$ or $\mathtt{c} = \mathtt{c}'$.

In FJ, any class c is assumed to have exactly one constructor $\mathtt{K}_\mathtt{c}$, which takes a canonical form explained below.

Let us define the sequence of the fields of c as follows: the sequence of the fields of Object is empty; if c directly extends $\mathtt{c}'$, then the sequence of the fields of c is obtained appending to the sequence of the inherited fields (that is, the fields of $\mathtt{c}'$) the sequence of the fields directly declared in c, in the given order.

Then

$$
\begin{array}{rcl}
\mathtt{K_c} & ::= & \mathtt{c}(\mathtt{c}_1\ \mathtt{f}_1, \ldots, \mathtt{c}_{n+m}\ \mathtt{f}_{n+m})\{\ \mathtt{KE_c;}\ \} \\
\mathtt{KE_c} & ::= & \mathtt{super}(\mathtt{f}_1, \ldots, \mathtt{f}_n); \\
 & & \mathtt{this.f}_{n+1} = \mathtt{f}_{n+1}; \ldots \mathtt{this.f}_{n+m} = \mathtt{f}_{n+m};
\end{array}
$$

where $\mathtt{c}_1\mathtt{f}_1, \ldots, \mathtt{c}_{n+m}\mathtt{f}_{n+m}$, for $n, m \geq 0$, are the fields of c and, in particular, $\mathtt{c}_{n+1}\mathtt{f}_{n+1}, \ldots, \mathtt{c}_{n+m}\mathtt{f}_{n+m}$ are the directly declared fields (hence $\mathtt{c}_1\mathtt{f}_1, \ldots, \mathtt{c}_n\mathtt{f}_n$ are the inherited fields).

Note that, if the whole FJ program is available, then the canonical constructor for a class c is completely determined by the inheritance hierarchy of c, hence it is immaterial to either explicitly write its declaration in the class or not. However, this consideration does not apply to separate compilation; indeed, if constructors are explicit, then compilation of a class requires the existence of all its ancestors, since we must check that the constructor matches inherited and declared fields. On the other hand, if constructors are implicit, then the availability of all ancestors is not required for compiling a class.

Here, we have chosen the second alternative, which allows a more modular type-checking. Another alternative would consist in allowing arbitrary constructors as in full Java. Here we preferred to keep the simpler FJ choice, since the problem of constructor overloading is basically an easier version of method overloading [10].

As already mentioned, the bytecode language we define for FJ differs from the source code only for field accesses, which contain a symbolic reference $\ll \mathtt{c.f\ c'} \gg$ to the field to be selected, method invocations, which contain a symbolic reference $\ll \mathtt{c.}m(\overline{\mathtt{c}})\mathtt{c'} \gg$ to the method to be invoked, and instance creation expressions, which contain a symbolic reference $\ll \mathtt{c}\ \overline{\mathtt{c}} \gg$ to the canonical constructor.

Type assumptions are defined in Figure 2.

A standard type assumption has form $\mathtt{c}{:}(\mathtt{c'}, \mathtt{FS}, \mathtt{MSS})$ with the meaning "c extends $\mathtt{c}'$ and declares exactly all fields specified by FS and all methods specified by MSS", where FS is a set of fields (field type and field name) and MSS is a set of method *signatures* (return type, method name and parameter types).

[3] Or, in full Java, by super.

$$
\begin{array}{rcl}
\gamma & ::= & \mathtt{c}{:}\tau \mid \exists \mathtt{c} \mid \mathtt{c} \leq \mathtt{c}' \mid \mathtt{c}.\mathbf{f} \stackrel{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}' \mid \mathtt{c}.\mathtt{m}(\overline{\mathtt{c}}) \stackrel{\mathtt{m\text{-}res}}{\rightarrow} (\overline{\mathtt{c}}', \mathtt{c}) \mid \mathtt{c} \stackrel{\mathtt{k\text{-}res}}{\rightarrow} \overline{\mathtt{c}} \mid \mathtt{c} \textcircled{:} \mathtt{MS} \mid \mathtt{c} \not\leq \mathtt{c}' \\
\\
\tau & ::= & (\mathtt{c}, \mathbf{FS}, \mathtt{MSS}) \\
\mathbf{FS} & ::= & \{\mathbf{F}_1, \ldots, \mathbf{F}_n\} \\
\mathbf{F} & ::= & \mathtt{c}\ \mathbf{f} \\
\mathtt{MSS} & ::= & \{\mathtt{MS}_1, \ldots, \mathtt{MS}_n\} \textit{ (where parameter types of methods with the same name are distinct)} \\
\mathtt{MS} & ::= & \mathtt{c}\ \mathtt{m}(\overline{\mathtt{c}})
\end{array}
$$

Figure 2. Type environments

Other forms of type assumptions are listed below:

- $\exists \mathtt{c}$ with the meaning "$\mathtt{c}$ is declared";
- $\mathtt{c} \leq \mathtt{c}'$ with the meaning "$\mathtt{c}$ is a subtype of $\mathtt{c}'$";
- $\mathtt{c}.\mathbf{f} \stackrel{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ with the meaning "the access to field $\mathbf{f}$ of an object of type $\mathtt{c}$ is successfully resolved to a field (obviously named $\mathbf{f}$) with type $\mathtt{c}'$".
- $\mathtt{c}.\mathtt{m}(\overline{\mathtt{c}}) \stackrel{\mathtt{m\text{-}res}}{\rightarrow} (\overline{\mathtt{c}}', \mathtt{c}')$ with the meaning "the invocation of method $\mathtt{m}$ of an object of type $\mathtt{c}$ and with arguments of types $\overline{\mathtt{c}}$ is successfully resolved to a method (obviously named $\mathtt{m}$) with parameters of types $\overline{\mathtt{c}}'$ and return type $\mathtt{c}'$".
- $\mathtt{c} \stackrel{\mathtt{k\text{-}res}}{\rightarrow} \overline{\mathtt{c}}$ with the meaning "the canonical constructor of $\mathtt{c}$ has parameter types $\overline{\mathtt{c}}$";
- $\mathtt{c} \textcircled{:} \mathtt{c}'\ \mathtt{m}(\overline{\mathtt{c}})$ with the meaning "$\mathtt{c}$ can be extended by a subclass with a method $\mathtt{c}'\ \mathtt{m}(\overline{\mathtt{c}})$ without breaking the Java rule on method overriding";
- $\mathtt{c} \not\leq \mathtt{c}'$ with the meaning "$\mathtt{c}$ is not a subtype of $\mathtt{c}'$";

Typing rules for separate compilation are given in Figure 3.

The top-level rule (*fragment*) defines the compilation of a sequence of source class declarations $\mathtt{s}_1, \ldots, \mathtt{s}_n$, whose type is $\tau_1, \ldots, \tau_n$, into a sequence of binary class declarations $\mathtt{b}_1, \ldots, \mathtt{b}_n$. The provided environment, Γ, is enriched with the standard type assumptions assigning to classes their declared types to deal with mutual recursion. The resulting environment must be well-formed; the definition of well-formed type environments (typing rules for the judgment $\vdash \Gamma\diamond$ are given in Figure 4) relies on that for well-formed standard type environments (that is, those which only contain standard type assumptions, ranged over by Γ^s). That is, Γ is well-formed iff there exists a well-formed standard type environment Γ^s which entails Γ. A standard type environment is well-formed if the inheritance relation is acyclic, for each class all its ancestor classes are available and the Java rules on overriding are respected (that is, a class cannot declare a method with the same name and parameter types of an inherited method and different return type).This is modeled by means of a judgment $\Gamma^s \vdash \mathtt{c}{:}\mathtt{MSS}\diamond$ which holds if we can calculate in Γ^s the set of all method signatures of class $\mathtt{c}$ (empty for `Object`).

Rule (*class*), which defines the compilation of a single source class declaration for class $\mathtt{c}$, checks that the superclass $\mathtt{c}'$ can be safely extended with the methods declared in $\mathtt{c}$ and that there are no cycles involving $\mathtt{c}$ and $\mathtt{c}'$ (existence of the superclass is guaranteed by this last check). Note that there is no constraint on the fields declared in $\mathtt{c}$ since we allow field hiding.

Compilation of a sequence of method declarations consists in compilation of each method declaration.

Rule (*method*) for compiling a single method declaration checks the existences of the return type and of the types of the parameters and, moreover that, under the type assumptions for parameters in the method header, the body is a well-formed expression of a subtype of the method return type.

The judgment for separate compilation of expressions takes an additional Π which is a mapping from variables to class names.

Rule (*field access*) for compiling a field access $\mathtt{E}^s.\mathbf{f}$ checks that $\mathtt{E}^s$ is a well-formed expression of some type $\mathtt{c}$, and an access to field $\mathbf{f}$ of an object of type $\mathtt{c}$ is successfully resolved to a field of some type $\mathtt{c}'$, which is the resulting type of the field access. Moreover, the corresponding binary field access is annotated with the static type of $\mathtt{E}^s$ and the type of the selected field.

Rule (*meth call*) for compiling a method invocation $\mathtt{E}^s_0.\mathtt{m}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n)$ checks that $\mathtt{E}^s_0$ is a well-formed expression of some type $\mathtt{c}$, all arguments are well-formed expressions of some types $\overline{\mathtt{c}}$, and an invocation of method $\mathtt{m}$ of an object of type $\mathtt{c}$ and with arguments of types $\overline{\mathtt{c}}$ is successfully resolved to a method, whose return type is the resulting type of the method invocation. Moreover, the corresponding binary method invocation is annotated with the static type of the receiver and the parameter types and return type of the selected method.

Rule (*new*) for compiling an instance creation $\mathtt{new}\ \mathtt{c}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n)$ checks that all arguments are well-formed expressions, an invocation of the canonical constructor of $\mathtt{c}$ is successfully resolved and the argument types are subtypes of the corresponding parameter types of the canonical constructor. Moreover, the corresponding binary instance creation is annotated with the parameter types of the canonical constructor.

There are two typing rules for compiling a cast expression $(\mathtt{c})\mathtt{E}^s$, which both check that $\mathtt{E}^s$ is a well-formed expression of some type $\mathtt{c}'$; then, the cast expression is well-formed and has type $\mathtt{c}$ if either $\mathtt{c}$ is a subtype of $\mathtt{c}'$ (down cast) or conversely (up cast). Note that up casts are removed from the binaries. In [8], also casts between classes which are not in the subtyping relation (*stupid* casts) are allowed, in order to get the subject reduction property[4], but no user-defined expression can be typed by using the corresponding rule. Here we do not deal with the reduction semantics of FJ, hence we do not include stupid casts.

The typing rules for entailment of type environments are given in Figure 5. The first three rules for environment entailment simply say that $\Gamma_1 \vdash \Gamma_2$ is valid if each type assumption γ contained in Γ_2 is entailed by Γ_1; the remaining rules cover the cases when Γ_2 is a single atomic type assumption γ.

There are two rules which deal with resolution of fields. Rule

[4] Since, e.g., $(\mathtt{B})(\mathtt{Object})\mathtt{new}\ \mathtt{A}()$ reduces to $(\mathtt{B})\mathtt{new}\ \mathtt{A}()$ for $\mathtt{A}$, $\mathtt{B}$ heirs of `Object`.

$$\textit{(fragment)}\ \frac{\vdash \Gamma, \mathit{env}(\mathtt{s}{:}\tau)\diamond \quad \Gamma, \mathit{env}(\mathtt{s}{:}\tau)\vdash \mathtt{s}_i{:}\tau_i{\rightsquigarrow}\mathtt{b}_i\ \forall i \in 1..n}{\Gamma\vdash \mathtt{s}{:}\tau{\rightsquigarrow}\mathtt{b}} \quad \mathtt{s}=\mathtt{s}_1,\ldots,\mathtt{s}_n, \tau=\tau_1,\ldots,\tau_n, \mathtt{b}=\mathtt{b}_1,\ldots,\mathtt{b}_n,\ \forall i\in 1..n.\tau_i=\mathit{type}(\mathtt{s}_i)$$

$$\textit{(class)}\ \frac{\begin{array}{l}\Gamma\vdash \mathtt{MDS}^s{\rightsquigarrow}\mathtt{MDS}^b \\ \Gamma\vdash \mathtt{c}'\circledcirc\mathtt{MS}_i\ \forall\, i\in 1..n \\ \Gamma\vdash \mathtt{c}'\not\leq \mathtt{c}\end{array}}{\Gamma\vdash \begin{array}{l}\mathtt{class\ c\ extends\ c}'\ \{\mathtt{FDS\ MDS}^s\} : (\mathtt{c}', \mathtt{FS}, \{\mathtt{MS}_1,\ldots,\mathtt{MS}_n\}) \rightsquigarrow \\ \mathtt{class\ c\ extends\ c}'\ \{\mathtt{FDS\ MDS}^b\}\end{array}} \quad \begin{array}{l}\mathit{type}(\mathtt{MDS}^s)=\{\mathtt{MS}_1,\ldots,\mathtt{MS}_n\} \\ \mathit{type}(\mathtt{FDS})=\mathtt{FS}\end{array}$$

$$\textit{(methods)}\ \frac{\Gamma\vdash \mathtt{MD}^s_i{\rightsquigarrow}\mathtt{MD}^b_i\ \forall i\in 1..n}{\Gamma\vdash \mathtt{MD}^s_1\ \ldots\ \mathtt{MD}^s_n{\rightsquigarrow}\mathtt{MD}^b_1\ldots\mathtt{MD}^b_n} \qquad \textit{(method)}\ \frac{\begin{array}{l}\Gamma;\mathtt{x}_1{:}\mathtt{c}_1,\ldots,\mathtt{x}_n{:}\mathtt{c}_n\vdash \mathtt{E}^s{:}\mathtt{c}{\rightsquigarrow}\mathtt{E}^b \\ \Gamma\vdash \mathtt{c}\leq \mathtt{c}_0 \\ \Gamma\vdash \exists\, \mathtt{c}_i\ \forall\, i\in 0..n\end{array}}{\Gamma\vdash \begin{array}{l}\mathtt{c}_0\ \mathtt{m}(\mathtt{c}_1\ \mathtt{x}_1,\ldots,\mathtt{c}_n\ \mathtt{x}_n)\ \{\mathtt{return\ E}^s\mathtt{;}\}\rightsquigarrow \\ \mathtt{c}_0\ \mathtt{m}(\mathtt{c}_1\ \mathtt{x}_1,\ldots,\mathtt{c}_n\ \mathtt{x}_n)\ \{\mathtt{return\ E}^b\mathtt{;}\}\end{array}}$$

$$\textit{(parameter)}\ \frac{\vdash \Gamma\diamond \quad \Pi\vdash \mathtt{x}:\mathtt{c}}{\Gamma;\Pi\vdash \mathtt{x}{:}\mathtt{c}{\rightsquigarrow}\mathtt{x}} \qquad \textit{(field access)}\ \frac{\Gamma;\Pi\vdash \mathtt{E}^s{:}\mathtt{c}{\rightsquigarrow}\mathtt{E}^b \quad \Gamma\vdash \mathtt{c.f}\overset{\mathtt{f\text{-}res}}{\rightarrow}\mathtt{c}'}{\Gamma;\Pi\vdash \mathtt{E}^s.\mathtt{f}{:}\mathtt{c}'{\rightsquigarrow}\mathtt{E}^b \ll \mathtt{c.f\ c}'\gg}$$

$$\textit{(meth call)}\ \frac{\begin{array}{l}\Gamma;\Pi\vdash \mathtt{E}^s_0{:}\mathtt{c}{\rightsquigarrow}\mathtt{E}^b_0 \\ \Gamma;\Pi\vdash \mathtt{E}^s_i{:}\mathtt{c}_i{\rightsquigarrow}\mathtt{E}^b_i\ \forall i\in 1..n \\ \Gamma\vdash \mathtt{c.m}(\mathtt{c}_1,\ldots,\mathtt{c}_n)\overset{\mathtt{m\text{-}res}}{\rightarrow}(\overline{\mathtt{c}'},\mathtt{c}')\end{array}}{\Gamma;\Pi\vdash \mathtt{E}^s_0.\mathtt{m}(\mathtt{E}^s_1,\ldots,\mathtt{E}^s_n){:}\mathtt{c}'{\rightsquigarrow}\mathtt{E}^b_0 \ll \mathtt{c.m}(\overline{\mathtt{c}'})\mathtt{c}'\gg (\mathtt{E}^b_1,\ldots,\mathtt{E}^b_n)}$$

$$\textit{(new)}\ \frac{\begin{array}{l}\Gamma;\Pi\vdash \mathtt{E}^s_i{:}\mathtt{c}'_i{\rightsquigarrow}\mathtt{E}^b_i\ \forall i\in 1..n \\ \Gamma\vdash \mathtt{c}\overset{\mathtt{k\text{-}res}}{\rightarrow}\mathtt{c}_1,\ldots,\mathtt{c}_n \\ \Gamma\vdash \mathtt{c}'_i\leq \mathtt{c}_i\ \forall i\in 1..n\end{array}}{\Gamma;\Pi\vdash \mathtt{new\ c}(\mathtt{E}^s_1,\ldots,\mathtt{E}^s_n){:}\mathtt{c}{\rightsquigarrow}\mathtt{new}\ \ll \mathtt{c}\ \mathtt{c}_1,\ldots,\mathtt{c}_n\gg (\mathtt{E}^b_1,\ldots,\mathtt{E}^b_n)}$$

$$\textit{(down cast)}\ \frac{\Gamma;\Pi\vdash \mathtt{E}^s{:}\mathtt{c}'{\rightsquigarrow}\mathtt{E}^b \quad \Gamma\vdash \mathtt{c}\leq \mathtt{c}'}{\Gamma;\Pi\vdash (\mathtt{c})\mathtt{E}^s{:}\mathtt{c}{\rightsquigarrow}(\mathtt{c})\mathtt{E}^b} \qquad \textit{(up cast)}\ \frac{\begin{array}{l}\Gamma;\Pi\vdash \mathtt{E}^s{:}\mathtt{c}'{\rightsquigarrow}\mathtt{E}^b \\ \Gamma\vdash \mathtt{c}'\leq \mathtt{c} \\ \Gamma\vdash \exists\, \mathtt{c}\end{array}}{\Gamma;\Pi\vdash (\mathtt{c})\mathtt{E}^s{:}\mathtt{c}{\rightsquigarrow}\mathtt{E}^b}$$

$\mathit{type}(\mathtt{CD}^s_1\ldots\ \mathtt{CD}^s_n) = \mathit{type}(\mathtt{CD}^s_1),\ldots,\mathit{type}(\mathtt{CD}^s_n)$
$\mathit{type}(\mathtt{class\ c\ extends\ c}'\ \{\ \mathtt{FDS\ MDS}^s\ \}) = (\mathtt{c}', \mathit{type}(\mathtt{FDS}), \mathit{type}(\mathtt{MDS}^s))$
$\mathit{type}(\mathtt{FD}_1\ldots\mathtt{FD}_n) = \{\mathit{type}(\mathtt{FD}_1),\ldots,\mathit{type}(\mathtt{FD}_n)\}$
$\mathit{type}(\mathtt{c\ f;}) = \mathtt{c\ f}$
$\mathit{type}(\mathtt{MD}^s_1\ldots\mathtt{MD}^s_n) = \{\mathit{type}(\mathtt{MD}^s_1),\ldots,\mathit{type}(\mathtt{MD}^s_n)\}$
$\mathit{type}(\mathtt{MH}\ \{\ \mathtt{return\ E}^s\mathtt{;}\ \}) = \mathit{type}(\mathtt{MH})$
$\mathit{type}(\mathtt{c}_0\ \mathtt{m}(\mathtt{c}_1\ \mathtt{x}_1,\ldots,\mathtt{c}_n\ \mathtt{x}_n)) = \mathtt{c}_0\ \mathtt{m}(\mathtt{c}_1..\mathtt{c}_n)$
$\mathit{name}(\mathtt{CD}^s_1\ldots\ \mathtt{CD}^s_n) = \mathit{name}(\mathtt{CD}^s_1),\ldots,\mathit{name}(\mathtt{CD}^s_n)$
$\mathit{name}(\mathtt{class\ c\ extends\ c}'\ \{\ \mathtt{FDS\ MDS}^s\ \}) = \mathtt{c}$

Figure 3. Separate compilation

$$\textit{(non standard)}\ \frac{\Gamma^s\vdash\Gamma \qquad \vdash\Gamma^s\diamond}{\vdash\Gamma\diamond}$$

$$\textit{(standard)}\ \frac{\mathtt{c}_1{:}\tau_1,\ldots,\mathtt{c}_n{:}\tau_n\vdash \mathtt{c}_i{:}\mathtt{MSS}_i\diamond\ \forall i\in 1,\ldots,n}{\vdash \mathtt{c}_1{:}\tau_1,\ldots,\mathtt{c}_n{:}\tau_n\diamond} \quad n\geq 0,\ \mathtt{c}_i=\mathtt{c}_j \implies \tau_i=\tau_j\ \forall i,j\in 1,\ldots,n$$

$$\textit{(msigs obj)}\ \frac{}{\Gamma^s\vdash \mathtt{Object}{:}\emptyset\diamond} \qquad \textit{(msigs down)}\ \frac{\Gamma^s\vdash \mathtt{c}'{:}\mathtt{MSS}'\diamond}{\Gamma^s\vdash \mathtt{c}{:}\mathtt{MSS}\cup\mathtt{MSS}'\diamond} \quad \Gamma^s(\mathtt{c})=(\mathtt{c}',-,\mathtt{MSS}),\ \mathtt{c}_1\ \mathtt{m}(\overline{\mathtt{c}})\in\mathtt{MSS}, \mathtt{c}_2\ \mathtt{m}(\overline{\mathtt{c}})\in\mathtt{MSS}'\implies \mathtt{c}_1=\mathtt{c}_2$$

Figure 4. Well-formed type environments

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

$$\textit{(def class)}\frac{\Gamma\vdash \mathtt{c}:\tau}{\Gamma\vdash\exists\,\mathtt{c}} \qquad \textit{(def obj)}\frac{\vdash\Gamma\diamond}{\Gamma\vdash\exists\mathtt{Object}}$$

$$\textit{(refl)}\frac{\vdash\Gamma\diamond}{\Gamma\vdash\mathtt{c}\le\mathtt{c}} \qquad \textit{(trans)}\frac{\Gamma\vdash\mathtt{c}_1\le\mathtt{c}_2 \qquad \Gamma\vdash\mathtt{c}_2:(\mathtt{c}_3,-,-)}{\Gamma\vdash\mathtt{c}_1\le\mathtt{c}_3} \qquad \textit{(}\le\textit{obj)}\frac{\vdash\Gamma\diamond}{\Gamma\vdash\mathtt{c}\le\mathtt{Object}}$$

$$\textit{(}\le\textit{vector)}\frac{\Gamma\vdash\mathtt{c}_i\le\mathtt{c}'_i\forall\, i\in 1..n}{\Gamma\vdash\mathtt{c}_1,\ldots,\mathtt{c}_n\le\mathtt{c}'_1,\ldots,\mathtt{c}'_n}$$

$$\textit{(direct field res)}\frac{\Gamma\vdash\mathtt{c}:(-,\mathtt{FS},-)}{\Gamma\vdash\mathtt{c}.\mathtt{f}\overset{\mathtt{f\text{-}res}}{\to}\mathtt{c}'}\ \mathtt{c}'\ \mathtt{f}\in\mathtt{FS} \qquad \textit{(inh field res)}\frac{\Gamma\vdash\mathtt{c}_1:(\mathtt{c}_2,\mathtt{FS},-) \qquad \Gamma\vdash\mathtt{c}_2.\mathtt{f}\overset{\mathtt{f\text{-}res}}{\to}\mathtt{c}}{\Gamma\vdash\mathtt{c}_1.\mathtt{f}\overset{\mathtt{f\text{-}res}}{\to}\mathtt{c}}\ \nexists\mathtt{c}'.\mathtt{c}'\ \mathtt{f}\in\mathtt{FS}$$

$$\textit{(exact meth res)}\frac{\Gamma\vdash\mathtt{c}_2:(-,-,\mathtt{MSS}) \qquad \Gamma\vdash\mathtt{c}_1\le\mathtt{c}_2}{\Gamma\vdash\mathtt{c}_1.\mathtt{m}(\overline{\mathtt{c}})\overset{\mathtt{m\text{-}res}}{\to}(\overline{\mathtt{c}},\mathtt{c})}\ \mathtt{c}\ \mathtt{m}(\overline{\mathtt{c}})\in\mathtt{MSS}$$

$$\textit{(complete meth res)}\frac{\begin{array}{l}\textit{applAll}(\Gamma,\mathtt{c},\mathtt{m},\overline{\mathtt{c}})=\mu s\\ \Gamma\vdash\overline{\mathtt{c}}\Uparrow\\ \textit{mostSpec}(\Gamma,\mu s)=(\overline{\mathtt{c}'},\mathtt{c}')\end{array}}{\Gamma\vdash\mathtt{c}.\mathtt{m}(\overline{\mathtt{c}})\overset{\mathtt{m\text{-}res}}{\to}(\overline{\mathtt{c}'},\mathtt{c}')} \qquad \textit{(match meth res)}\frac{\begin{array}{l}\textit{applAll}(\Gamma,\mathtt{c},\mathtt{m},\overline{\mathtt{c}})=\mu s\\ \textit{matchAll}(\Gamma,\mathtt{c},\mathtt{m},n)=\mu s\\ \textit{mostSpec}(\Gamma,\mu s)=(\overline{\mathtt{c}'},\mathtt{c}')\end{array}}{\Gamma\vdash\mathtt{c}.\mathtt{m}(\overline{\mathtt{c}})\overset{\mathtt{m\text{-}res}}{\to}(\overline{\mathtt{c}'},\mathtt{c}')}$$

$$\textit{(K obj)}\frac{\vdash\Gamma\diamond}{\Gamma\vdash\mathtt{Object}\overset{\mathtt{k\text{-}res}}{\to}\Lambda} \qquad \textit{(K up)}\frac{\Gamma\vdash\mathtt{c}:(\mathtt{c}',\mathtt{FS},-) \qquad \Gamma\vdash\mathtt{c}'\overset{\mathtt{k\text{-}res}}{\to}\overline{\mathtt{c}}}{\Gamma\vdash\mathtt{c}\overset{\mathtt{k\text{-}res}}{\to}\overline{\mathtt{c}},\mathtt{c}_1,\ldots,\mathtt{c}_n}\ \mathtt{FS}=\mathtt{c}_1\ \mathtt{f}_1\ldots\mathtt{c}_n\ \mathtt{f}_n$$

$$\textit{(}\Subset\textit{obj)}\frac{\vdash\Gamma\diamond}{\Gamma\vdash\mathtt{Object}\Subset\mathtt{c}\ \mathtt{m}(\overline{\mathtt{c}})} \qquad \textit{(}\Subset\textit{down)}\frac{\begin{array}{l}\Gamma\vdash\mathtt{c}_1:(\mathtt{c}_2,-,\mathtt{MSS})\\ \Gamma\vdash\mathtt{c}_2\Subset\mathtt{c}\ \mathtt{m}(\overline{\mathtt{c}})\end{array}}{\Gamma\vdash\mathtt{c}_1\Subset\mathtt{c}\ \mathtt{m}(\overline{\mathtt{c}})}\ (\mathtt{c}'\ \mathtt{m}(\overline{\mathtt{c}})\in\mathtt{MSS})\implies\mathtt{c}'=\mathtt{c}$$

$$\textit{(}\not\le\textit{)}\frac{\textit{notSub}(\Gamma,\mathtt{c},\mathtt{c}')}{\Gamma\vdash\mathtt{c}\not\le\mathtt{c}'}$$

Figure 5. Type environments entailment

$$\Gamma\vdash\mathtt{c}_1,\ldots,\mathtt{c}_n\Uparrow=\Gamma\vdash\mathtt{c}_1\Uparrow\wedge\ldots\wedge\Gamma\vdash\mathtt{c}_n\Uparrow$$

$$\Gamma\vdash\mathtt{c}\Uparrow=\begin{cases}\mathtt{true} & \text{if } \mathtt{c}=\mathtt{Object}\\ \Gamma\vdash\mathtt{c}'\Uparrow & \text{if } \Gamma\vdash\mathtt{c}:(\mathtt{c}',-,-)\\ \mathtt{false} & \text{otherwise}\end{cases}$$

$$\textit{notSub}(\Gamma,\mathtt{c},\mathtt{c}')=\Gamma\vdash\mathtt{c}\Uparrow\wedge\mathtt{c}'\notin\textit{supertypes}(\Gamma,\mathtt{c})$$

$$\textit{supertypes}(\Gamma,\mathtt{c})=\begin{cases}\{\mathtt{Object}\} & \text{if } \mathtt{c}=\mathtt{Object}\\ \{\mathtt{c}\}\cup\textit{supertypes}(\Gamma,\mathtt{c}') & \text{if } \Gamma\vdash\mathtt{c}:(\mathtt{c}',-,-)\\ \bot & \text{otherwise}\end{cases}$$

$$\textit{appl}(\Gamma,\mathtt{c},\mathtt{m},\overline{\mathtt{c}})=\begin{cases}\{<\mathtt{c},\overline{\mathtt{c}'},\mathtt{c}'> \mid \mathtt{c}'\ \mathtt{m}(\overline{\mathtt{c}'})\in\mathtt{MSS},\Gamma\vdash\overline{\mathtt{c}}\le\overline{\mathtt{c}'}\ \} & \text{if } \Gamma\vdash\mathtt{c}:(-,-,\mathtt{MSS})\\ \bot & \text{otherwise}\end{cases}$$

$$\textit{applAll}(\Gamma,\mathtt{c},\mathtt{m},\overline{\mathtt{c}})=\begin{cases}\emptyset & \text{if } \mathtt{c}=\mathtt{Object}\\ \mu s_1\cup\mu s_2 & \text{if } \textit{appl}(\Gamma,\mathtt{c},\mathtt{m},\overline{\mathtt{c}})=\mu s_1,\ \Gamma\vdash\mathtt{c}:(\mathtt{c}',-,-),\ \textit{applAll}(\Gamma,\mathtt{c}',\mathtt{m},\overline{\mathtt{c}})=\mu s_2\\ \bot & \text{otherwise}\end{cases}$$

$$\textit{match}(\Gamma,\mathtt{c},\mathtt{m},n)=\begin{cases}\{<\mathtt{c},\mathtt{c}'_1,\ldots,\mathtt{c}'_n,\mathtt{c}'> \mid \mathtt{c}'\ \mathtt{m}(\mathtt{c}'_1,\ldots,\mathtt{c}'_n)\in\mathtt{MSS} & \text{if } \Gamma\vdash\mathtt{c}:(-,-,\mathtt{MSS})\\ \bot & \text{otherwise}\end{cases}$$

$$\textit{matchAll}(\Gamma,\mathtt{c},\mathtt{m},n)=\begin{cases}\emptyset & \text{if } \mathtt{c}=\mathtt{Object}\\ \mu s_1\cup\mu s_2 & \text{if } \textit{match}(\Gamma,\mathtt{c},\mathtt{m},n)=\mu s_1,\ \Gamma\vdash\mathtt{c}:(\mathtt{c}',-,-),\ \textit{matchAll}(\Gamma,\mathtt{c}',\mathtt{m},n)=\mu s_2\\ \bot & \text{otherwise}\end{cases}$$

$$\textit{mostSpec}(\Gamma,\mu s)=\begin{cases}<\mathtt{c}_0,\overline{\mathtt{c}}_0,\mathtt{c}'_0> & \text{if } <\mathtt{c}_0,\overline{\mathtt{c}}_0,\mathtt{c}'_0>\in\mu s \text{ and } \Gamma\vdash\mathtt{c}_0,\overline{\mathtt{c}}_0\le\mathtt{c},\overline{\mathtt{c}}, \text{ for all } <\mathtt{c},\overline{\mathtt{c}},\mathtt{c}'>\in\mu s\\ \bot & \text{otherwise}\end{cases}$$

Figure 6. Auxiliary functions

(*direct field res*) states that an access to field `f` for an object of type `c` can be successfully resolved if class `c` directly declares a field named `f`. Rule (*inh field res*) states that an access to field `f` for an object of type c_1 can be successfully resolved if it can be resolved for an object of the parent type c_2 and c_1 declares no fields named `f` (which would hide the inherited field).

There are three rules which deal with resolution of methods.

Rule (*exact meth res*) covers the simple case where there exists an applicable method which perfectly matches the invocation, hence can be directly selected.

Otherwise, all applicable methods must be collected, and then the most specific method (if any) is selected [7].

The set $applAll(\Gamma, c, m, \overline{c})$, formally defined in Figure 6, contains *all* methods of `c` (either directly declared or inherited) named `m` whose parameter types are supertypes of $\overline{c}$ in Γ (note that for calculating this set all ancestors of `c` are needed).

However, in general we cannot be sure that this set actually contains *all* the applicable methods, since there could exist some `m` in `c` for which we do not have in Γ the type assumptions stating that the parameter types are supertypes of $\overline{c}$. We can conclude that we have collected all applicable methods (hence the most specific, if any, can be selected) only in two cases: if we have all ancestors of the argument types ($\Gamma \vdash \overline{c}\Uparrow$ in rule *complete meth res*), or if the set $applAll(\Gamma, c, m, \overline{c})$ coincides with the set $matchAll(\Gamma, c, m, n)$ of all methods `m` of class `c` whose number of parameters matches the number of arguments in the invocation (rule *match meth res*), hence the set of all applicable methods cannot be larger.

Rules (*K obj*) and (*K up*) deal with resolution of constructors. The former states that the canonical constructor of class `c` is `Object` has no parameters (Λ denotes the empty sequence). The latter states that the canonical constructor of a class `c` which extends c' has as sequence of parameter types the sequence of the parameter types of the canonical constructor of c', followed by the types of the fields directly declared in `c`.

Rule (☺*obj*) states that `Object` can be extended by any method.

Rule (☺*down*) states that if we know that a certain class c_2 can be extended by a method $c\ m(\overline{c})$, then a heir class c_1 which does not define the method with a different return type can be extended with the same method as well.

Finally, the last rule states that we can conclude that `c` is not a subtype of c' if all ancestors of `c` are available and c' is not among them (see the definition of *notSub* in Figure 6).

4 Results

In this section we prove that the type system T^{FJ} for FJ extended with overloading and hiding defined in the previous section satisfies the hypotheses of Theorem 11 and 13, hence supports sound and complete inter-checking. We also prove that it has principal typings and that the environment entailment is complete.

In order to prove these results, we need the following lemmas, stating that entailment is a pre-order on well-formed type environments and that well-formedness actually coincides with consistency.

Lemma 15.

1. *If* $\Gamma_1 \vdash \Gamma_2$ *holds, then* $\vdash \Gamma_1\diamond$ *holds.*
2. *If* $\vdash \Gamma_1, \Gamma_2\diamond$ *holds, then* $\Gamma_1, \Gamma_2 \vdash \Gamma_1$ *holds.*
3. *If* $\Gamma_1 \vdash \Gamma_2$ *and* $\Gamma_2 \vdash \Gamma_3$ *hold, then* $\Gamma_1 \vdash \Gamma_3$ *holds.*
4. *If* $\Gamma_1 \vdash \Gamma_2$ *holds, then* $\vdash \Gamma_2\diamond$ *holds.*

Lemma 16. Γ *is consistent iff* $\vdash \Gamma\diamond$.

Fact 17 (COMPOSITIONALITY OF T^{FJ}). *The type system* T^{FJ} *is compositional, that is, for all* Γ, $s = s_1, \ldots, s_n$, $\tau = \tau_1, \ldots, \tau_n$, $b = b_1, \ldots, b_n$*:*
$\Gamma \vdash s{:}\tau \rightsquigarrow b \Leftrightarrow \Gamma, env(s{:}\tau) \vdash s_i{:}\tau_i \rightsquigarrow b_i$*, for all* $i \in 1..n$.

Theorem 18. *The environment entailment is sound, that is, for all* Γ_1, Γ_2, $\Gamma_1 \vdash \Gamma_2 \Rightarrow \Gamma_1 \leq \Gamma_2$.

Theorem 19. *In the type system* T^{FJ} *inter-checking is sound w.r.t. global compilation.*

Theorem 20. *For all typable* (s, b)*, there exists a typing*

$$(\Gamma^{(s,b)}, \tau^{(s,b)})$$

of (s, b) *s.t. for all typings* (Γ, τ) *of* (s, b), $\Gamma \vdash \Gamma^{(s,b)}$ *and* $\tau^{(s,b)} = \tau$.

Theorem 21. *In the type system* T^{FJ} *inter-checking is complete w.r.t. global compilation.*

Theorem 22. *The type system* T^{FJ} *has principal typings.*

Theorem 23. *The environment entailment is complete, that is, for all* Γ, Γ', $\Gamma \leq \Gamma' \Rightarrow \Gamma \vdash \Gamma'$.

5 Conclusion

We have defined an abstract framework for modeling separate compilation and inter-checking, provided a formal definition of soundness and completeness of inter-checking, and proved that these properties can be guaranteed when the type system has principal typings and provides sound and complete entailment relation between type environments.

The fact that a type system has principal typings is often claimed to be a highly desirable feature since they allow *compositional type analysis* in the sense that the procedure of finding types for a term uses only the analysis results for its immediate subterms, which can be analyzed independently in any order [14] and *never need to be inspected again* (see also [9, 5]). Perhaps the most important result of this paper on the foundational side is that we are able to formally express this property in the context of separate compilation and linking and to prove that it is actually guaranteed by principal typings.

On the side of application to Java-like languages, this paper is part of a stream of work [3, 2, 10, 11] on alternative type systems for Java.

In [3] we firstly realized that, despite the fact that Java is considered a paradigmatic example of language supporting separate compilation, compilation as performed by standard Java compilers and

modeled in current Java formal definitions is not truly separate in the sense made precise in [4] and in this paper. Indeed, each class is typechecked against the same "global" type environment, that is, that extracted from a particular program context. Hence, a different type system for a subset of Java has been designed, introducing type assumptions expressing minimal requirements needed for typechecking a class in isolation, similar to those shown in this paper.

In [2] it is shown that this type system actually allows *stronger* typings w.r.t. to standard type systems for Java and that it can be the basis for selective recompilation of Java applications.

Concerning the applicability of these results to the whole Java language, [10] and [11] outline the extension of this type system to a large Java subset (including, e.g., constructors, access modifiers, static members, throws clauses and unreachable code) and the development of a corresponding smart compiler. The reader interested into aspects related to Java and selective recompilation can refer to these papers.

Here, our aim was to formally prove a result of existence of principal typings for a Java-like language (the first to our knowledge), hence we have preferred to consider a simple and clean language as Featherweight Java, enriched with the features which pose the main problems in the Java type system. We believe a nice side-contribution of this paper is that we have "exported" the notion of principal typing, more familiar in the community of functional languages, to a completely different context, and show that, whereas in classical type systems with principal typings they are usually obtained by making the type more general (typically by introducing polymorphism or intersection types), in Java-like languages the opposite happens, that is, principal typings can be obtained by making type environments less restrictive.

In the notion of type system introduced in this paper, we have considered fragments as pairs $(\mathtt{s},\mathtt{b})$. As already mentioned, an interesting alternative would be to consider the binary as part of the type. In this case, in order to get the principal typings property, we should introduce polymorphic types.

For instance, consider again the example in the Introduction:

```
class C extends Parent {
  ...
  Type1 m(Type2 x){ return new Used().g(x);}
}
```

If we do not care about which bytecode will be generated, then class `C` can be correctly linked with any class `Used` having a method $\alpha\ \mathtt{g}(\beta)$, for any types α,β s.t. $\alpha \leq \mathtt{Type1}$ and $\mathtt{Type2} \leq \beta$. Clearly, all these classes cannot be captured by a unique type environment Γ in our current type system. In order to do that, we should introduce type variables in the type environments, analogously to the approach followed in [13]; so we could model the requirements above as follows:

$$\alpha \leq \mathtt{Type1}, \mathtt{Type2} \leq \beta, \mathtt{Used.g(Type2)} \stackrel{\mathtt{m\text{-}res}}{\rightarrow} (\alpha,\beta)$$

However, in this way the compiler cannot generate bytecode for `C`, since method descriptors cannot contain type variables; as a consequence, either JVM should be modified, or we should introduce a sort of pre-bytecode that may contain type variables that must be instantiated during static inter-checking in order to produce valid bytecode (similar solutions can be found in literature [13, 12]).

Finally, another interesting topic for further investigation is the relation between the notions of binary compatibility [6] and inter-checking.

6 Acknowledgements

We would like to thank Eugenio Moggi for his helpful suggestions on the relation between Well's and our notion of principality.

7 References

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

[2] D. Ancona and G. Lagorio. Stronger typings for separate compilation of Java-like languages (Extended Abstract). In *5th Intl. Workshop on Formal Techniques for Java Programs 2003*, July 2003.

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

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

[5] F. Damiani. Rank 2 intersection types for local definitions and conditional expressions. *ACM Transactions On Programming Languages and Systems*, 25(4):401–451, 2003.

[6] S. Drossopoulou, D. Wragg, and S. Eisenbach. What *is* Java binary compatibility? In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*, volume 33(10) of *SIGPLAN Notices*, pages 341–358, October 1998.

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

[8] 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.

[9] 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.

[10] 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.

[11] G. Lagorio. Another step towards a smart compilation manager for Java. In *ACM Symp. on Applied Computing (SAC 2004), Special Track on Object-Oriented Programming Languages and Systems*, 2004. To appear.

[12] S. McDirmid, M.Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*. ACM Press, October 2001.

[13] Z. Shao and A.W. Appel. Smartest recompilation. In *ACM Symp. on Principles of Programming Languages 1993*, pages 439–450. ACM Press, 1993.

[14] 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.

A Proofs

Lemma 15

1. By induction on the derivation of $\Gamma_1 \vdash \Gamma_2$.
2. By rules (*conc*) and (*singleton*).
3. By induction on the derivation of $\Gamma_2 \vdash \Gamma_3$.
4. By rule (non standard) and (3). □

Lemma 16

$\Rightarrow$ If Γ is consistent, then we have applied rule (fragment), hence $\vdash \Gamma, \mathit{env}(\mathtt{s} : \tau)\diamond$ holds for some $\mathtt{s}$, τ. Then by lemma 15 (2) and (4) we get that $\vdash \Gamma\diamond$ holds.

$\Leftarrow$ If $\vdash \Gamma\diamond$ holds, then we can apply rule (fragment) with $n = 0$.

Theorem 18 (Sketch) The fact that Γ_2 is consistent follows from lemma 15 (4) and lemma 16. Then, we have to prove that, for all $\mathtt{s}$, τ, $\mathtt{b}$, if $\Gamma_2 \vdash \mathtt{s}:\tau \rightsquigarrow \mathtt{b}$ holds, then $\Gamma_1 \vdash \mathtt{s}:\tau \rightsquigarrow \mathtt{b}$ holds as well. This can be shown by induction on the derivation of $\Gamma_2 \vdash \mathtt{s}:\tau \rightsquigarrow \mathtt{b}$, by extending the claim to all other kinds of compilation judgments and by using the transitivity of entailment. □

Theorem 19 From Theorem 11 and Theorem 18. □

Theorem 20 (Sketch) We have to prove that,

> (1) for all typable $(\mathtt{s},\mathtt{b})$, there exists $\Gamma^{(\mathtt{s},\mathtt{b})}$, $\tau^{(\mathtt{s},\mathtt{b})}$ s.t. $\Gamma^{(\mathtt{s},\mathtt{b})} \vdash \mathtt{s}:\tau^{(\mathtt{s},\mathtt{b})} \rightsquigarrow \mathtt{b}$ holds and $\Gamma \vdash \Gamma^{(\mathtt{s},\mathtt{b})}$, $\tau = \tau^{(\mathtt{s},\mathtt{b})}$ for all Γ s.t. $\Gamma \vdash \mathtt{s}:\tau \rightsquigarrow \mathtt{b}$ holds.

Since $(\mathtt{s},\mathtt{b})$ is typable, at least one compilation judgment $\Gamma \vdash \mathtt{s}:\tau \rightsquigarrow \mathtt{b}$ holds. The proof is by induction on the derivation of this judgment, by extending the claim to all other kinds of compilation judgments.

Typings for expressions are the most interesting case. Then, property (1) becomes,

> (2) for all Π, and typable $(\mathtt{E}^s,\mathtt{E}^b)$ w.r.t. Π, there exists $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}$, $\mathtt{c}^{(\mathtt{E}^s,\mathtt{E}^b)}$ s.t. $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}; \Pi \vdash \mathtt{E}^s:\mathtt{c}^{(\mathtt{E}^s,\mathtt{E}^b)} \rightsquigarrow \mathtt{E}^b$ holds and $\Gamma \vdash \Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}$, $\mathtt{c} = \mathtt{c}^{(\mathtt{E}^s,\mathtt{E}^b)}$ for all Γ s.t. $\Gamma; \Pi \vdash \mathtt{E}^s:\mathtt{c} \rightsquigarrow \mathtt{E}^b$ holds,

where $(\mathtt{E}^s,\mathtt{E}^b)$ is typable w.r.t. Π if there exist Γ, $\mathtt{c}$ s.t. $\Gamma; \Pi \vdash \mathtt{E}^s:\mathtt{c} \rightsquigarrow \mathtt{E}^b$ holds.

We just outline the proof for the rule (field access). Then we know that

H1 $\Gamma; \Pi \vdash \mathtt{E}^s:\mathtt{c} \rightsquigarrow \mathtt{E}^b$ holds;

H2 $\Gamma \vdash \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ holds;

H3 (from (H1) by inductive hypothesis) there exists $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}$ s.t. property (2) holds.

Let us take $\Gamma^{(\mathtt{E}^s.\mathtt{f},\mathtt{E}^b \ll \mathtt{c}.\mathtt{f}\ \mathtt{c}' \gg)} = \Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$. We have to prove that

T1 $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'; \Pi \vdash \mathtt{E}^s.\mathtt{f}:\mathtt{c}' \rightsquigarrow \mathtt{E}^b \ll \mathtt{c}.\mathtt{f}\ \mathtt{c}' \gg$ holds;

T2 $\Gamma' \vdash \Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ holds for all Γ' s.t. $\Gamma; \Pi \vdash \mathtt{E}^s.\mathtt{f}:\mathtt{c} \rightsquigarrow \mathtt{E}^b \ll \mathtt{c}.\mathtt{f}\ \mathtt{c}' \gg$ holds.

First of all we prove that $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ is consistent. Since Γ is consistent by hypothesis, $\Gamma \vdash \Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}$ holds by (H3), $\Gamma \vdash \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ holds by (H2), by rule (conc) we can conclude that $\Gamma \vdash \Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ holds, and this implies that $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ is consistent by lemma 15 and lemma 16.

We prove now (T1). We can instantiate rule (field access) by premises:

- $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'; \Pi \vdash \mathtt{E}^s:\mathtt{c} \rightsquigarrow \mathtt{E}^b$ (this follows from the facts that $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ is consistent, hence $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}' \vdash \Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}$ holds by lemma 15 (2) and by soundness of environment entailment (Theorem 18)).
- $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}' \vdash \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ (this follows from the fact that $\Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ is consistent by rule (singleton).

Finally, (T2) follows from the fact that, if $\Gamma'; \Pi \vdash \mathtt{E}^s:\mathtt{c} \rightsquigarrow \mathtt{E}^b$ holds, then we must have instantiated rule (field access), hence we can apply to Γ' the previous reasoning and get that $\Gamma' \vdash \Gamma^{(\mathtt{E}^s,\mathtt{E}^b)}, \mathtt{c}.\mathtt{f} \overset{\mathtt{f\text{-}res}}{\rightarrow} \mathtt{c}'$ holds.

Theorem 21 From Theorem 13 and Theorem 20. □

Theorem 22 From Theorem 18 and Theorem 20. □

Theorem 23 (Sketch) First of all note that $\Gamma \leq \Gamma'$ implies that Γ' and Γ' are consistent (hence $\vdash \Gamma'\diamond$ holds).

The proof is by induction on the length of Γ'.

The case $\Gamma' = \Lambda$ is trivial by rule (empty).

Assume $\Gamma \leq \Gamma', \gamma$. Then $\Gamma \leq \Gamma'$ and $\Gamma \leq \gamma$ by transitivity, since $\Gamma', \gamma \leq \Gamma'$ and $\Gamma', \gamma \leq \gamma$ hold since Γ', γ is consistent from lemma 15 (2) and soundness of environment entailment.

Then, by inductive hypothesis $\Gamma \vdash \Gamma'$ holds and we can instantiate rule (conc) provided that we prove that $\Gamma \leq \gamma$ implies $\Gamma \vdash \gamma$ for each type assumption γ. This can be proved by case analysis.