Source Compatibility for Java Packages

Yannick Welsch Arnd Poetzsch-Heffter
University of Kaiserslautern
{welsch,poetzsch}@cs.uni-kl.de

Abstract

Software libraries and platforms should often evolve in a way that existing client code is not affected. As a prerequisite, client code that compiles against the original version of the libraries should also compile against the modified version. In such a case, we call the new version source compatible with the old one. For languages with elaborate static encapsulation mechanisms like Java, source compatibility is a complex property and checking tools do not exist.

This paper defines source compatibility for packages of a formalized Java subset with all relevant access modifiers. As the definition quantifies over all possible client contexts, it cannot be used for automatic checking. We thus derive statically checkable conditions for compatibility that are proved necessary and sufficient. Such checkable conditions give interesting insight into the encapsulation of Java packages, allow to discuss language and program design aspects and provide the basis for package-local refactoring tools.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Theory

Keywords Java, Source Compatibility, Packages

1. Introduction

Application programming interfaces (APIs) play a central role in the maintenance and evolution of software. APIs are particularly important for the development and use of library components, applications, and device interfaces. An API is a more or less explicit contract between provided code and client code. Providers should realize the functionality provided by the API and clients should only access API parts of the provided code and not internal aspects of the implementation.

APIs evolve over time. Sometimes evolution steps do not preserve compatibility with API clients (called *breaking API changes* by [13]), but often libraries or components should be modified, extended, or refactored in such a way that client code is not affected. Software developers can use informal guidelines (e.g., [12]) and special tools (e.g., [18]) to check compatibility aspects. However, in order to fully automate compatibility checking, it needs to be put on a formal basis.

We focus on APIs which are realized through programming language support, namely Java packages. Interfaces and encapsulation are fundamental object-oriented concepts [37]. However, in most OO languages, package interfaces are only implicitly defined. A package interface provides public types to create objects, access public fields, and call methods on them (*caller interface*). Furthermore, a package interface provides the possibility to extend the functionality of the provided types via inheritance (*implementor interface*). Changes which are compatible with respect to the caller interface can be incompatible for the implementor interface. For example, narrowing the type of a method parameter breaks compatibility for callers whereas widening a parameter type breaks compatibility for the implementors (cf. [12]). To distinguish caller and implementor interface, OO languages support different access modifier (e.g., public and protected in Java).

A prerequisite for API compatibility is that all client code that compiles against the old API version also compiles against the new version. If two API versions satisfy this property, they are called *source compatible*. Checking source compatibility for packages is a difficult task with two central challenges:

Complexity: The complexity of package interfaces is often underestimated. The reason is the intricate interplay of mechanisms to express and restrict subtyping aspects, such as abstract and final types and methods, with the mechanisms to control encapsulation. For example, the information whether or not two non-public types are in a subtype relationship may affect source compatibility of Java packages.

Modularity: Source compatibility should be checked in a modular way, i.e., without knowing the client code. A checking technique is needed that can abstract from the infinite number of possible client contexts.

Interfaces are well-understood on the object or type level (programming in the small). Type systems allow compilers to check that type-related programming errors are avoided, e.g., suitable co-/contravariance typing and appropriate choice of access modifiers in overriding methods. However, as we will show, interface support on the package level (programming in the large) still needs improvement. We envision that future package constructs allow compilers to check for compatibility in the same way as today's compilers check for nominal or structural subtyping. Towards this goal, the paper makes the following technical contributions:

1. We define and discuss source compatibility for components, which are sets of packages.
2. We provide a formal definition of the context conditions and typing rules for a Java subset with all access modifiers and the modifiers *abstract* and *final*. This is the first detailed formalization of these language aspects.
3. We present *syntactic* conditions for checking source compatibility that avoid the quantification over all contexts.
4. We prove that the syntactic conditions are *necessary* and *sufficient*. In particular, we develop a new proof technique for *static context abstraction*, which identifies how the types of expressions in the context differ when compiled against one component or another. The abstraction also identifies the pivotal component classes that affect the context's typing.

Furthermore, we discuss possible language improvements to simplify source compatibility checking and present an investigation on how far such simplifications affect existing code. We also illustrate how source compatibility can support package-local refactoring and discuss language and program design aspects to improve future module systems.

Outline. The remainder of this paper is organized as follows. Sect. 2 informally introduces source compatibility. Sects. 3 and 5 formalize our Java subset in two steps. Sects. 4 and 6 derive the syntactic compatibility conditions for this language. Sect. 7 presents possible applications and discusses language and program design aspects when adapting the conditions to full Java. Sect. 8 presents related work and Sect. 9 concludes.

2. Source Compatibility

In the following, we first illustrate source compatibility by an example assuming that the reader has a basic understanding of Java's access rules (see Sects. 3 and 5 for a formal definition). Then, we define and discuss source compatibility and explain how to derive syntactic conditions that ensure compatibility.

2.1 Example

Let us consider the following implementation of a library called `util` providing simple collection facilities:

```
package util;
public interface List {...}
public class ArrayList implements List {...}
public class LinkedList implements List {...}
```

In future versions of the library, we want to factor out the commonalities of the two list implementations by introducing a common superclass `AbsList`. We adapt the type hierarchy accordingly:

```
abstract class AbsList implements List {...}
public class ArrayList extends AbsList {...}
public class LinkedList extends AbsList {...}
```

The question arises whether library users are affected by these changes. If client code compiles against the old version of the library, will it still compile against the new version? As the type hierarchy of the public types in the library is not modified, the changes preserve compatibility.

Let us make another change to the API of the library. We add a formely not existing method `addAll` to the `List` interface and implement the method accordingly in the subclasses. At first sight, possible clients seem to be unaffected by this change. As they have not used this `addAll` method before, they will still compile in the presence of this additional method. However, clients that have implemented the `List` interface to realize list implementations of their own will stop compiling as they do not provide an implementation of this additional method declared in the interface. As consequence, adding a method to a public interface breaks compatibility for implementors of this interface.

A developer of the presented library may have compatibility concerns for many more of the changes he/she intends to do. For example, if we assume that the `ArrayList` class had a method which was previously declared as protected, can we change the modifier to public in future versions of the library? Do private fields or methods have an influence on compatibility? Can we make abstract classes non-abstract? Similarly, can we make final classes non-final? Can we introduce new public types or fields? In this paper, we develop syntactic conditions to answer such questions automatically and correctly based on a formal model.

2.2 Terminology

We explained source compatibility before as the property between two component versions that allows all clients which compiled against the old component version to compile against the new one. However, when considering whether a client can compile against a component, one has to make the distinction between *observability* [23, §7.3 and §7.4.3]

(sometimes also called *visibility*[1]) and *accessibility* of types [7]. Observability is a property of the host platform. At compile time, observability of a type means that the compiler can locate the type definition. For most Java compilers (e.g. `javac`), observability can be influenced by setting the class- or source-path accordingly. Most module systems (e.g., [38]) on the JVM manage observability via class loaders (i.e., at runtime). In the context of this paper, we do not consider observability and focus on accessibility, which is a property based on the access modifiers of the language (e.g., private, protected, public). When we talk about a context compiling against a component, we assume that all the concerned packages and types are observable.

We use the following terminology throughout this paper. A *package* has a name and consists of a sequence of type declarations (cf. Sect. 3). We assume that packages are sealed (cf. [24], Sect. 2), meaning that once a package is defined no new class and interface definitions can be added to the package. A sequence of packages is called a *codebase*. A codebase X is called a *program component* or simply *component* (inspired by [26]) if and only if

- all names used in X are defined in X or are predefined (like class `Object`) and
- X is well-formed according to the context and typing rules of the language.

We write $\vdash X$ to express that the codebase X is a component. Adding packages to a codebase or joining codebases is denoted by juxtaposition of their names, e.g., the codebase KX is composed of codebase K and component X. With these notions, we define compatibility of a component Y with a component X as the property (similar to [26]) that every codebase which compiles against X must also compile against Y:

Definition 1 (Contextual compatibility)**.** A component Y is *(contextually) compatible* with a component X if and only if for any codebase K: $\vdash KX$ implies $\vdash KY$.

We call K a *(program) context*. It is important to note that the definition of contextual compatibility does not allow for automatic checking that a component Y is compatible with X, as it quantifies over an infinite set of contexts[2]. Compatibility is a preorder relation, that is, it is reflexive and transitive, but not antisymmetric. Conceptually, compatibility can be seen as a "structural subtype relation" on components (although the exact characterization of what a component signature or interface is will remain implicit in this presentation).

2.3 Discussion

We selected the above definition from a number of other candidates, mainly because it is simple and can handle the interesting practical scenarios. In particular, the compatibility definition allows us to compare single packages that do not import other packages. More importantly, it allows to check compatibility of components X and Y that share common library packages (e.g., `java.util`, etc.). What are other canditates to define compatibility? The first alternative is whether to define compatibility for packages or for components. As a package often imports other packages, two package versions might import different packages. If we define compatibility for packages and allow that packages import other packages, we have to be careful about the set of contexts. For example, consider two implementations Q_1 and Q_2 of a package with name p where Q_1 imports a package R_1 and Q_2 imports a package R_2 with a different name. Even if Q_1 and Q_2 are "intuitively" compatible, a context K which is well-formed with Q_1R_1 might not be well-formed with Q_2R_2 just because it has a conflict with R_2 (e.g., contains a package with the same name). Thus, one can only quantify over codebases K that are not in conflict with R_1 and R_2.

For some situations, the sketched problem can be handled by allowing the hiding of imported packages (as some module systems do). But, as illustrated later, there are other situations where even non-public types can affect the well-formedness of entities outside the package. That is why we defined compatibility for components X and Y. Compatibility of packages Q_1 and Q_2 of the example above is treated in our setting as compatibility of the components $X = Q_1R_1R_2$ and $Y = Q_2R_1R_2$.

Considering component compatibility, as we do, has the additional advantage that we can compare components where several packages have new versions. Furthermore, we can allow recursive package dependencies within the component. We also investigated more structured versions of the definition where compared components X and Y import from compatible components X' and Y'. Such a more structured definition would *not* lead to different results for the problem of this paper. However, it would be a step towards well-understood import interfaces and helpful to check compatibility of large components incrementally. We consider it as future work.

A restriction of our setting is that we only consider sealed packages. That is, we do not allow contexts to add new classes and interfaces to packages contained in the components X and Y that are compared for compatibility[3]. We have also investigated scenarios with *open* packages. But the generalization complicates the definitions and proofs in such a way that it deviates from the central ideas without adding substantial insight.

[1] Not to be confused with the meaning of visibility in the setting of declaration scopes for programming languages [23, §6.3.1]

[2] This is an even harder issue in the setting of behavioral equivalence.

[3] However, context codebases can still extend classes and interfaces from the components.

2.4 Towards Compatibility Conditions

In the next sections, we illustrate how to derive necessary and sufficient conditions for compatibility. Let us recapitulate the definition of (contextual) compatibility. A component Y is compatible with X, if for any (context) codebase K: $\vdash KX$ implies $\vdash KY$. The syntactic conditions we want to derive should directly relate the components X and Y without quantification over all context codebases K.

We introduce the syntactic compatibility conditions in two steps. The first step only considers those aspects of class and interface declarations that are not related to expresssions in method bodies. For these aspects, we introduce the conditions which guarantee that class and interface declarations in context codebases (K) remain well-formed (Sect. 4). In a second step, we study the conditions which guarantee that expressions (in method bodies of class declarations) in context codebases remain well-formed (Sect. 6).

As a prerequisite for a formal treatment of the syntactic conditions, we need a formalization of the language (called *PackageJava*) for which we consider compatibility. This formalization is also introduced in two steps, where at first we only present the context conditions (Sect. 3) and then later give the typing rules for expressions (Sect. 5).

3. *PackageJava* Formalization

In this section, we describe our Java subset formalized in the spirit of *ClassicJava* [20]. We support packages, interfaces, classes, inheritance and subtyping, and the usual Java modifiers (public, protected, private, final, abstract), but do not support method overloading and final fields. To our knowledge, this is the first formalization that captures the complex interplay of the accessibility modifiers with final and abstract classes and methods. The abstract syntax of our language is shown in Fig. 1. We use an overbar notation to denote syntactic sequence of arbitrary length and lift functions, predicates, and type judgements automatically to sequences when needed. To append elements to a sequence or to append two sequences, they are just juxtaposed. We also assume that there exists a function "size" returning the length of a sequence.

A codebase X consists of a sequence of packages containing classes and interfaces. At used occurrences, a type t is denoted by its fully qualified name $p.t$ where p is the name of the package in which t is declared[4]. Although expression typing will only be considered in Sect. 5, we already state the syntax here. Expressions can be variable names, the special **null** value, object creation expressions, casts, field access and writes, method calls and super calls. We assume every class to implicitly have a public constructor with no arguments and empty body.

[4] In our examples, we sometimes omit the package qualifier in locations where we refer to a type defined in the same package.

$$
\begin{array}{rcl}
K, X, Y & ::= & \overline{Q} \\
Q, R & ::= & \textbf{package}\ p\ \textbf{;}\ \overline{D} \\
D & ::= & \overline{N}\ \textbf{class}\ c\ \textbf{extends}\ p.c\ \textbf{implements}\ \overline{p.i}\ \{\ \overline{F}\ \overline{M}\ \} \\
 & | & [\textbf{public}]\ \textbf{interface}\ i\ \textbf{extends}\ \overline{p.i}\ \{\ \overline{M}\ \} \\
F & ::= & \overline{N}\ p.t\ f\ \textbf{;} \\
M & ::= & \overline{N}\ p.t\ m(\ \overline{p.t\ v}\ )\ \big(\ \textbf{;}\ |\ \{\ E\ \}\ \big) \\
N & ::= & \textbf{private}\ |\ \textbf{protected}\ |\ \textbf{public}\ |\ \textbf{abstract}\ |\ \textbf{final} \\
E & ::= & v\ |\ \textbf{null}\ |\ \textbf{new}\ p.c\ |\ (p.t)E\ |\ E.f\ |\ E.f = E \\
 & | & E.m(\ \overline{E}\ )\ |\ \textbf{super}.m(\ \overline{E}\ ) \\
t & ::= & c\ |\ i \\
c & \in & \text{class names, including } \mathit{Object} \\
i & \in & \text{interface names} \\
p, q, r & \in & \text{package names, including } \mathit{lang} \\
f & \in & \text{field names} \\
m & \in & \text{method names} \\
v & \in & \text{variable names, including } \textbf{this}
\end{array}
$$

Figure 1. Abstract syntax

Context conditions. In the following paragraphs, we introduce the notation used to describe the context conditions. It is important to note that most operators and relations have an additional parameter X, which represents the codebase X to be checked. In most type system formalizations, this parameter (representing the program) is left implicit. As we compare different codebases later in Sect. 4, it is useful to make explicit which codebase we are referring to.

We denote by $\mathit{PackageOnce}_X$ that package declarations in a codebase X may not share the same package name (i.e., packages are sealed). Class and interface names in each package must be unique ($\mathit{TypeOncePerPackage}_X$) and field and method names declared in each type must be unique ($\mathit{MemberOncePerType}_X$). We define $\mathcal{C}_X$ as the set of (fully qualified) class identifiers for which there is a declaration D in codebase X. Similarly, $\mathcal{I}_X$ represents the set of declared interfaces. We then define $\mathcal{C}_X^{\mathbf{o}} \stackrel{\text{def}}{=} \mathcal{C}_X \cup \{\mathit{lang.Object}\}$ and the set of types $\mathcal{T}_X \stackrel{\text{def}}{=} \mathcal{C}_X^{\mathbf{o}} \cup \mathcal{I}_X$.

The following symbols express relations between the types. For a codebase X, we define the direct subtype relation $<_X$ as the least relation with the following properties. The relation contains $p.c <_X q.c'$ if a class c in package p extends a class c' in package q. Similarly, it contains $p.i <_X q.i'$ if an interface i in package p extends the interface $q.i'$. If a class c in package p implements an interface $q.i$, it contains $p.c <_X q.i$. The direct subtype relation also contains $p.i <_X \mathit{lang.Object}$ for each interface type $p.i$ in $\mathcal{I}_X$. We write $\leq_X$ as the transitive and $\leq_X$ as the reflexive, transitive closure of $<_X$. In order to type the **null** expression, we add the null type $\bot$ and write $p.t_\bot$ for the fully qualified types including the null type, i.e., $p.t_\bot ::= p.t \mid \bot$. We add $\bot \leq_X p.t$ to the subtype relation $\leq_X$ for all $p.t \in \mathcal{T}_X$.

$$\lesssim_X \text{ is antisymmetric} \quad (C1_X)$$

$$\_.\_ <_X p.c \rightarrow \neg\text{final}_X(p.c) \quad (C2_X)$$

$$\neg(p.t <_X p.t) \quad (C3_X)$$

$$p.t <_X q.t' \rightarrow \text{acctype}_X(q.t', p) \quad (C4_X)$$

$$\neg(\text{final}_X(p.c) \wedge \text{abstract}_X(p.c)) \quad (C5_X)$$

$$\langle f, \_, \overline{N}\rangle \Subset_X p.c \rightarrow \neg(\text{final}(\overline{N}) \vee \text{abstract}(\overline{N})) \quad (C6_X)$$

$$\langle m, \_, \overline{N}, E_0\rangle \Subset_X p.t \rightarrow (\text{abstract}(\overline{N}) \leftrightarrow E_0 = \mathbf{;}) \quad (C7_X)$$

$$\langle m, \_, \overline{N}, \_\rangle \Subset_X p.i \rightarrow \text{abstract}(\overline{N}) \wedge \text{public}(\overline{N}) \quad (C8_X)$$

$$\langle m, \_, \overline{N}, \_\rangle \Subset_X p.t \wedge \text{abstract}(\overline{N}) \rightarrow \neg\text{private}(\overline{N}) \wedge \neg\text{final}(\overline{N}) \quad (C9_X)$$

$$\langle \_.\_.m, \_, \overline{N}, \_\rangle \in_X p.c \wedge \text{abstract}(\overline{N}) \rightarrow \text{abstract}_X(p.c) \quad (C10_X)$$

$$\text{ovr}_X(p.c, q.c', m, T) \rightarrow \text{ovrok}_X(p.c, q.c', m, T) \quad (C11_X)$$

$$\langle \_.\_.m, T, \overline{N}_1, \_\rangle \in_X p.t \wedge \langle \_.\_.m, T', \overline{N}_2, \_\rangle \in_X p.t \rightarrow T = T' \wedge \overline{N}_1 = \overline{N}_2 \quad (C12_X)$$

$$\langle \_.\_.m, T, \_, \_\rangle \in_X p.i \wedge q.c <_X p.i \rightarrow \langle \_.\_.m, T, \overline{N}, \_\rangle \in_X q.c \wedge \text{public}(\overline{N}) \quad (C13_X)$$

$$\langle \_.\_.m, \overline{p.t} \twoheadrightarrow p_0.t_0, \overline{N}, \_\rangle \in_X q.t' \wedge \text{public}_X(q.t') \wedge \text{public}(\overline{N}) \rightarrow \text{public}_X(p_0.t_0\ \overline{p.t}) \quad (C14_X)$$

$$\langle \_.\_.m, \overline{p.t} \twoheadrightarrow p_0.t_0, \overline{N}, \_\rangle \in_X q.c \wedge \text{public}_X(q.c) \wedge \text{protected}(\overline{N}) \wedge \neg\text{final}_X(p.c) \rightarrow \text{public}_X(p_0.t_0\ \overline{p.t}) \quad (C15_X)$$

Figure 2. Context conditions

To describe that types provide methods or fields, we introduce the $\Subset_X$ (direct membership) relation. We write $\langle f, q.t, \overline{N}\rangle \Subset_X p.c$ to describe that a field f of type $q.t$ with modifiers $\overline{N}$ is declared in a class c of package p. Similarly, we write $\langle m, T, \overline{N}, E_0\rangle \Subset_X p.t$ to describe that a method m of signature $T = (\overline{q.t} \twoheadrightarrow q_0.t_0)$, where $\overline{q.t}$ are the parameter types and $q_0.t_0$ is the return type, is declared in a type $p.t$ with modifiers $\overline{N}$. If m is a method with an expression body E, then $E_0 \stackrel{\text{def}}{=} E$, otherwise $E_0 \stackrel{\text{def}}{=} \mathbf{;}$.

The treatment of accessibility modifiers is a central aspect of the language formalization. We assume that at most one of the modifiers private, protected, and public appears in a modifier sequence $\overline{N}$ of a declaration and that classes are neither private not protected. We use the following helper predicates: $\text{public}_X(p.c)$ holds if and only if the class c is defined with the public modifier in package p; $\text{public}(\overline{N})$ holds if and only if $\overline{N}$ contains the modifier **public**. There are analogous predicates for the other modifiers. In particular, $\text{package}_X(...)$ and $\text{package}(...)$ are used for package-local accessibility. We assume that $\text{public}_X(\mathit{lang.Object})$ holds for any codebase X under consideration.

We define the context conditions for a codebase X in Fig. 2. Free logical variables in the conditions are universally quantified. In order to improve readability, we often use the place-holder "_" instead of a free logical variable that occurs only once in the formula. For each condition, we refer to the sections of the JLS (Java Language Specification [23]) which cover this topic.

Condition (C1) requires that there are no cycles in the type hierarchy (§8.1.4, §8.1.5, §9.1.3). Condition (C2) states that final classes cannot be extended (§8.1.4) and (C3) disallows a type to declare itself as its supertype (§8.1.4). In a codebase X, a type $q.t$ is called *accessible* in package p if and only if $q.t$ is public or part of the same package (§6.6.1):

$$\text{acctype}_X(q.t, p) \stackrel{\text{def}}{=} q.t \in \mathcal{T}_X \wedge (\text{public}_X(q.t) \vee p = q)$$

The condition (C4) states that types occurring in a supertype declaration must be accessible (§8.1.4, §8.1.5, §9.1.3). Conditions (C5) - (C9) state that abstract and final modifiers are only applicable at certain program locations. Classes can not be both abstract and final (§8.1.1.2). Fields can not be declared abstract. We also do not consider final fields in our formalization. Abstract methods must have no body (§8.4.3.1). Interface methods must be declared public and abstract, which is done implicitly in Java (§9.4). Abstract methods can neither be private nor final (§8.4.3.1).

Inheritance and overriding. As *PackageJava* supports inheritance and subtyping, we introduce the notions of (transitive) field and method membership, inheritance, and overriding. The membership symbol $\in_X$, defined in Fig. 3, captures the idea of considering all transitively inherited members (§6.4.3 and §6.4.4). A field or method is a member of a type if the type defines this field or method (**D-FIELD** and **D-METHOD**). Note that the membership relation $\in_X$, in contrast to the direct membership relation $\Subset_X$, additionally contains the definition site of the field or method, which is relevant for defining the accessibility of fields or methods.

Members with accessibility $\overline{N}$ defined in a package p (e.g., methods or fields) can only be inherited to a (direct subtype in) package q (written $\text{inherit}(p, q, \overline{N})$) if they are declared public or protected, or if they are declared with package accessibility and $p = q$ (**INHERITABLE**[5]). There is an important distinction between field and method inheritance. Rule **INH-FIELD** shows that declaring a field f in a class, independent of its type and modifiers, hides fields with same name f (of arbitrary type) declared in superclasses (§8.3).

The definition of method inheritance depends on the notion of method overriding. It is important to note that overriding, in contrast to inheritance (§8.4.8), is a relation which does not necessarily relate methods of direct sub-

[5] The JLS does not provide an explicit definition for this property.

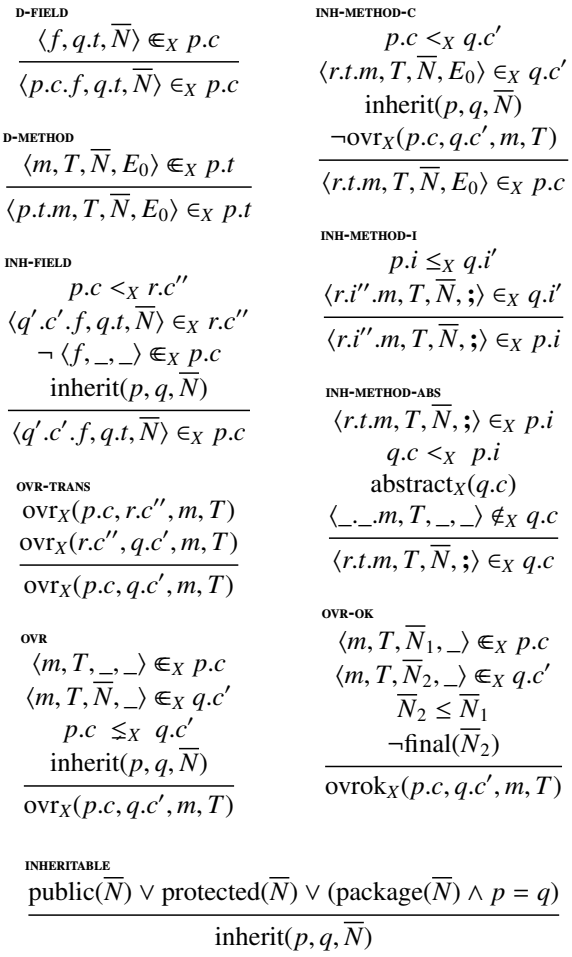

Figure 3. Definitions of field and method membership, inheritance and overriding

/supertypes[6] (§8.4.8.1). A method defined in a class $p.c$ overrides a method with same name and signature[7] in a class $q.c'$ if $p.c$ is a subtype of $q.c'$ and if the overridden method permits access (see rule **OVR**). The overriding property is closed under transitivity (see rule **OVR-TRANS**). A class $p.c$ inherits a method from its superclass (**INH-METHOD-C**) if there is no method in $p.c$ which overrides the method. An interface inherits all methods from its super-interfaces (**INH-METHOD-I**) as methods in interfaces are public and abstract. A special case exists where an abstract class can inherit a method from its super-interfaces, but only once for a certain method name and type signature (**INH-METHOD-ABS**).

[6] This leads to the interesting effect that you can override methods which you cannot inherit. Another interesting fact is that there exist overriding methods which can not access any of the methods they override using a super call.

[7] We do not consider covariance of return types in this paper.

Using these new definitions, we can now precisely describe context conditions with respect to our modifiers. Condition (C10) in Fig. 2 states that if a class contains abstract methods (including inherited ones), it must be declared as abstract[8] (§8.1.1.1). Condition (C11), using rule **OVR-OK** from Fig. 3, requires that an overriding definition uses weaker accessibility modifiers (§8.4.8.3). The relation $<$ is the least order on the following accessibility modifiers, where private $<$ (no modifier) $<$ protected $<$ public. The total order $\leq$ is the reflexive, transitive closure of $<$. While validating corner cases of our formalization, we were able to find a bug[9] with respect to correct method overriding of non-public methods in the Eclipse Java compiler.

Condition (C12) prohibits method overloading (a further restriction of our Java subset, as this would otherwise complicate the presentation) and condition (C13) requires correct implementation of methods (§9.1.3).

Context conditions similar to (C14) and (C15) do not exist in standard Java. These conditions restrict the amount of well-formed Java programs, as we require public (or protected) methods of public types to have only public parameter and return types. As we will see later, this simplifies the set of compatibility conditions a lot. According to our investigation, this additional restriction has no impact on existing practical Java programs. We discuss our findings in Sect. 7.1. These additional restrictions give us the guarantee that we can always create a class in another package which implements a given public interface (as we can implement all the methods), which does not hold in Java. We could formulate a similar simplifying restriction for field types. This was not done to illustrate the impact on the compatibility conditions (which would become simpler in the setting of that additional constraint as is also illustrated in Sect. 7.1).

Well-formedness rules. The well-formedness rules of Fig. 4 describe exactly whether a codebase is well-formed (note that we consider at first only codebases with **null**-valued method bodies). The following judgements are used.

- $\vdash X$ denotes that the codebase X is well-formed, i.e., X is a component.
- $X \vdash Q$ denotes that the package declaration Q is well-formed in codebase X.
- $X, p \vdash D$ denotes that the type declaration D (class or interface) of package p is well-formed in codebase X.
- $X, p.t \vdash M$ denotes that the method M declared in type $p.t$ is well-formed in codebase X.
- $X, p.c, \Gamma \vdash E : q.t_\perp$ denotes that expression E in class $p.c$ has type $q.t_\perp$ under local variable typing Γ (in codebase X).
- $X, p.c, \Gamma \vDash E : q.t$ denotes that expression E in class $p.c$ has type $q.t$ using subsumption (in codebase X). The judg-

[8] This is a slight simplification of the JLS, ignoring a corner case.

[9] Eclipse Bugzilla, Bug 271303 : `https://bugs.eclipse.org/271303`

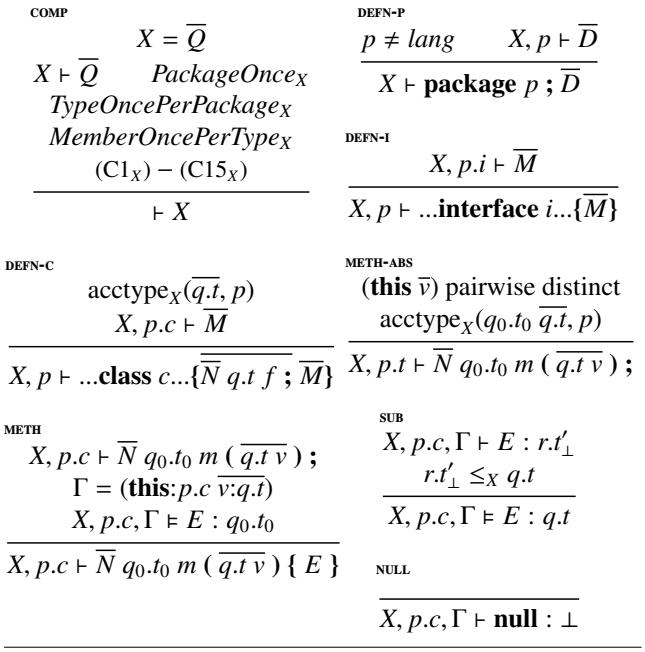

Figure 4. Well-formedness of codebases with null-valued method bodies

ment is defined only for types different to the null-type $\perp$. We have introduced this additional judgement in order to reduce the number of logical variables in the typing rules.

A codebase X is a component (see rule **COMP**) if it satisfies the context conditions and each package declaration in it is well-formed. A package declaration is well-formed if each type declaration in it is well-formed (and so on). Methods and field types must be accessible from the context where they are defined (**DEFN-C**, **METH-ABS** and **METH**). The **null** value (**NULL**) types to the null type $\perp$ but may type to any type in X under the subsumption judgment (**SUB**).

4. Compatibility Conditions

As a step towards the main theorem, this section investigates a restricted form of compatibility. Let us call a method body consisting of the expression **null** a ***null*-valued method body**.

Definition 2 (Weak compatibility)**.** Two components X and Y are called *weakly compatible*, if and only if for any codebase K in which all method bodies are **null**-valued: $\vdash KX$ implies $\vdash KY$.

As the number of contexts are smaller for weak compatibility, compatibility implies weak compatibility, but not the other way around. In this section, we derive syntactic conditions that allow for checking weak compatibility. The conditions are necessary, i.e., if two components are weakly compatible they satisfy the conditions. They are also sufficient, i.e., if two components satisfy the conditions they are weakly compatible.

In the following paragraphs, we present the syntactic conditions. At the end of this section, we explain how to prove that they are necessary and sufficient.

Package names. Let us start by considering the case where there exists a package name in Y which does not occur in X. A context K may then use the same package name and compile against X but not Y, as we require packages to be sealed (*PackageOnce*). We thus require that the set of package names occurring in Y is a subset of those occurring in X. If we had a more powerful module system that could hide some of the packages, we would only require that the exported package names occurring in Y also occur in X. We inspect similar restrictions for types.

Type names and hierarchy. The context conditions (C4) and the rules **DEFN-C** and **METH-ABS** require that the types appearing in supertype declarations, field definitions and method signatures are accessible (acctype(...)). Consider for example the component X, consisting of the utility package `util`, and the following codebase K which is a context for X:

```
package myutil;
class Stack extends util.LinkedList {
  void push(Object o) {...}
  void addAll(util.List l) {...}
}
```

If the resulting program is well-formed (i.e., $\vdash KX$), the types `LinkedList`, `List` and `Object` must be accessible and thus be public types in X. In order for the code to be also well-formed for future versions Y of X, `LinkedList`, `List` and `Object` must also be accessible (and thus be public) types in Y. Our syntactic condition requires that every type in X which is public must also exist in Y and be public there too.

$$\forall p.t \in \mathcal{T}_X : \text{public}_X(p.t) \rightarrow p.t \in \mathcal{T}_Y \wedge \text{public}_Y(p.t) \qquad (R1_{X,Y})$$

If the class `LinkedList` is not declared as final in X, it should not be declared as final in Y either, as otherwise KY may contain a class that subclasses a final class, which is not allowed by (C2). This leads to the condition that every public class which is not final in X must not be final in Y either.

$$\forall p.c \in C_X : \text{public}_X(p.c) \wedge \neg\text{final}_X(p.c) \rightarrow \neg\text{final}_Y(p.c) \qquad (R2_{X,Y})$$

Method overriding and implementation. The following conditions result from the presence of method overriding (see (C11)) and the correct implementation of interface methods (see (C13)) in our language. Let us consider a class `MyList` in our context K which implements the interface `List` of the component X. If `List` is accessible in X, it must also be accessible in Y according to our previous condition (R1). However, the interface `List` in Y may contain more

methods than in X (which, by (C8), must all be public and abstract). Then `MyList` may not implement the methods which additionally appear in Y (i.e., ($C13_{KY}$) does not hold).

We thus require that every public interface in X has at least[10] all the methods of the corresponding interface in Y (i.e., that no new methods occur in Y, as is also explained in [12] and in our introductory example).

$$\forall p.i \in \mathcal{I}_X : \text{public}_X(p.i) \wedge \langle \_.\_.m, T, \_, \_\rangle \in_Y p.i \rightarrow \langle \_.\_.m, T, \_, \_\rangle \in_X p.i \quad (\text{R3}_{X,Y})$$

A similar restriction also exists for classes of X and Y. There, additional access modifiers need to be considered, as methods in classes are not necessarily public and abstract. We consider public and protected methods, as only these are inherited across package boundaries (cf. **INHERITABLE**) Similar to before, assume that the class `LinkedList` in Y has a method `m` which is not in X. A context class which subclasses `LinkedList` (i.e., `List` must be public and not final) may already have a method with the same name `m` but with any kind of modifier. If Y now defines `m` as a public method, X must do so too, or else our context may not correctly override (i.e., use a weaker modifier) the method `m` from `LinkedList` when compiled against Y (i.e., ($C11_{KY}$) does not hold). If Y defines `m` as a final method, X must do so, too, otherwise we suddenly override a final method which is not allowed. If Y defines `m` as an abstract method, X must do so, too, or else a non-abstract context class might not implement all the abstract methods (i.e., ($C10_{KY}$) does not hold). The following condition captures all these cases.

$$\begin{aligned}\forall p.c \in C_X : \text{public}_X(p.c) \wedge \neg\text{final}_X(p.c) \\ \wedge \langle \_.\_.m, T, \overline{N}_2, \_\rangle \in_Y p.c \\ \wedge (\text{public}(\overline{N}_2) \vee \text{protected}(\overline{N}_2)) \\ \rightarrow \exists \overline{N}_1.\, \langle \_.\_.m, T, \overline{N}_1, \_\rangle \in_X p.c \wedge \overline{N}_2 \leq \overline{N}_1 \\ \wedge (\text{final}(\overline{N}_2) \rightarrow \text{final}(\overline{N}_1)) \\ \wedge (\text{abstract}(\overline{N}_2) \rightarrow \text{abstract}(\overline{N}_1)) \quad (\text{R4}_{X,Y})\end{aligned}$$

Now, we can define syntactic compatibility for contexts with **null**-valued method bodies:

Definition 3 (Weak syntactic compatibility)**.** Two components X and Y are *weakly syntactically compatible*, written $X \lhd Y$, if and only if all the package names in Y also exist in X and the syntactic conditions ($R1_{X,Y}$)-($R4_{X,Y}$) hold.

We can now state in the characterization lemma for weak compatibility. Note that the method bodies of the components under consideration (X and Y) are of no importance, as the typing of these method bodies is not influenced by K.

Lemma 1 (Weak compatibility)**.** *Two components X and Y are weakly compatible if and only if $X \lhd Y$.*

[10] The converse must not hold if we only look at correct overriding. However, it will be discussed in the setting of well-formed method call expressions in Sect. 6.

Proof. In the following, we explain the proof technique. A proof sketch covering all cases is given in Sects. A.3.1 and A.4.1 of the appendix. To prove that the syntactic conditions are necessary, we assume that they do not hold and then give a construction of a context K such that $\vdash KX$ but $\nvdash KY$ (proof by contrapositive). To prove that the syntactic conditions are sufficient, we use a direct proof. We illustrate both proof directions for the syntactic condition (R1).

$\Rightarrow$ We assume that the condition ($R1_{X,Y}$) does not hold for some $p.t$. We then construct a context codebase K the following way:

```
package k;
class C extends lang.Object { p.t f; }
```

The name k must neither occur in X nor Y. It still needs to be proven that this KX is well-formed ($\vdash KX$). This is the case as $p.t$ is public in X. KY is not well-formed ($\nvdash KY$) as $\text{acctype}_{KY}(p.t, k)$ does not hold.

$\Leftarrow$ We assume that $X \lhd Y$, i.e., especially the condition ($R1_{X,Y}$) holds. We want to show that for all K such KX is well-formed ($\vdash KX$) the codebase KY is also well-formed ($\vdash KY$). KY is well-formed if (by Def. of **COMP**) it satisfies the context conditions. We present the proof for one context condition, namely ($C4_{KY}$), which states that every declared supertype must exist and be accessible.
To show that ($C4_{KY}$), we assume that $p.t <_{KY} q.t'$ and then show that $\text{acctype}_{KY}(q.t', p)$ holds. The proof goes by case analysis on $p.t <_{KY} q.t'$. We only consider the case where $p.t \in \mathcal{T}_K$ and $q.t' \in \mathcal{T}_Y$: This means that K contains a type declaration which declares $q.t' \notin \mathcal{T}_K$ as supertype. As $\vdash KX$ holds and especially ($C4_{KX}$) holds, we know (by Def. of acctype) that $q.t' \in \mathcal{T}_X$ and $\text{public}_X(q.t')$. By our syntactic condition ($R1_{X,Y}$), we then know that $q.t' \in \mathcal{T}_Y$ and $\text{public}_Y(q.t')$. This also means that $q.t' \in \mathcal{T}_{KY}$ and $\text{public}_{KY}(q.t')$ and thus $\text{acctype}_{KY}(q.t', p)$. □

5. Expression Typing

In this section, we complete the static semantics of *Package-Java* by providing the typing rules. We start with an important definition.

Accessibility of members. Similar to accessibility of types, we now define accessibility of methods and fields. As access to protected members is a bit more complicated, we first illustrate it using the codebase X in Fig. 5. All four field access expressions happen in the class $\mathtt{p_2.C_2}$. While most people would agree that the access to a protected field can only occur in subclasses of the class where the field is defined (*), it is less known that the static type of the reference the field is accessed on also affects accessibility. In our example, (*) holds for all four accesses. They only differ in the static type of the reference (`v`) that the field is accessed on. The access in $\mathtt{m_1}$ is not valid, as the static type of the reference $\mathtt{p_1.C_1}$ is not a subtype of the accessing location $\mathtt{p_2.C_2}$ (i.e., $\mathtt{p_1.C_1}$

```
package p1;
public class C1 { protected Object f; }

package p2;
public class C2 extends p1.C1 {
  void m1(p1.C1 v) { v.f } //ERROR
  void m2(p2.C2 v) { v.f } //OK
  void m3(p3.C3 v) { v.f } //OK
  void m4(p2'.C2' v) { v.f } //ERROR
}

package p3;
public class C3 extends p2.C2 {}

package p2';
public class C2' extends p1.C1 {}
```

Figure 5. Example for protected access

$\not\leq_X$ p2.C2). Similarly, the access in $\mathtt{m}_4$ is not valid, as $\mathtt{p}_2'.\mathtt{C}_2'$ $\not\leq_X$ p2.C2. This allows a very peculiar access protection; the subclasses p2.C2 and $\mathtt{p}_2'.\mathtt{C}_2'$ are not allowed to access each other's f field.

We give a formal definition of member accessibility. The predicate $\text{accmember}_X(p.t, q.t', r.t'', \overline{N})$ holds if a member defined in type $p.t$ with access modifier $\overline{N}$ is accessible through a reference of type $q.t'$ from the type $r.t''$ (§6.6.1).

$$\frac{\text{private}(\overline{N}) \to p.t = r.t'' \qquad \text{package}(\overline{N}) \to p = r \qquad \text{protected}(\overline{N}) \to p = r \vee (r.t'' \leq_X p.t \wedge q.t' \leq_X r.t'')}{\text{accmember}_X(p.t, q.t', r.t'', \overline{N})}$$

Access to a private member is only possible from the same type. Similarly, access to a package member is only possible from within the same package. Accessibility of protected members is discussed more thoroughly in [9, 41]. In the wording of the JLS (§6.6.2), "a protected member ... of an object may be accessed from outside the package in which it is declared only by code that is responsible for the implementation of that object". Consequently, in our formalization, we require the type $r.t''$ to be involved in the implementation of $q.t'$. In our example, for the field accesses v.f in the different methods $\mathtt{m}_1$, $\mathtt{m}_2$, $\mathtt{m}_3$ and $\mathtt{m}_4$, $p.t$ (in the definition of the accmember predicate) corresponds to $\mathtt{p}_1.\mathtt{C}_1$, $r.t''$ corresponds to $\mathtt{p}_2.\mathtt{C}_2$ and, depending on the method, $q.t'$ corresponds to $\mathtt{p}_1.\mathtt{C}_1$, $\mathtt{p}_2.\mathtt{C}_2$, $\mathtt{p}_3.\mathtt{C}_3$ or $\mathtt{p}_2'.\mathtt{C}_2'$.

Typing rules. The typing rules for expressions are given in Fig. 6. The instance creation expression (**NEW**) allows only instances of non-abstract classes. The field access rule **GET** (§15.11.1) uses the previously defined field membership relation $\in_K$. For a field access $E.f$, the type of E must be accessible and the field must also be accessible. The field assignment rule **SET** is interesting because it does not require the involved type ($p_1.t_1$) to be accessible (§15.26.1). The

$$\textbf{NEW}\ \frac{\text{acctype}_X(q.c', p) \quad \neg\text{abstract}_X(q.c')}{X, p.c, \Gamma \vdash \textbf{new}\ q.c' : q.c'} \qquad \textbf{VAR}\ \frac{(v{:}q.t) \in \Gamma}{X, p.c, \Gamma \vdash v : q.t}$$

$$\textbf{GET}\ \frac{X, p.c, \Gamma \vdash E : q.c' \quad \text{acctype}_X(q.c', p) \quad \langle p_0.c_0.f, p_1.t_1, \overline{N}\rangle \in_X q.c' \quad \text{accmember}_X(p_0.c_0, q.c', p.c, \overline{N})}{X, p.c, \Gamma \vdash E.f : p_1.t_1}$$

$$\textbf{SET}\ \frac{X, p.c, \Gamma \vdash E.f : p_1.t_1 \quad X, p.c, \Gamma \vdash E' : p_1.t_1}{X, p.c, \Gamma \vdash E.f = E' : p_1.t_1}$$

$$\textbf{WCAST}\ \frac{\text{acctype}_X(q.t, p) \quad X, p.c, \Gamma \vdash E : q.t}{X, p.c, \Gamma \vdash (q.t)E : q.t}$$

$$\textbf{CALL}\ \frac{X, p.c, \Gamma \vdash E : q.t \quad \text{acctype}_X(q.t, p) \quad \langle r.t'.m, \overline{p.t} \twoheadrightarrow p_0.t_0, \overline{N}, \_\rangle \in_X q.t \quad \text{accmember}_X(r.t', q.t, p.c, \overline{N}) \quad X, p.c, \Gamma \vdash \overline{E : p.t}}{X, p.c, \Gamma \vdash E.m(\overline{E}) : p_0.t_0}$$

$$\textbf{CAST2}\ \frac{\text{acctype}_X(r.c'', p) \quad X, p.c, \Gamma \vdash E : q.c' \quad r.c'' \leq_X q.c'}{X, p.c, \Gamma \vdash (r.c'')E : r.c''}$$

$$\textbf{SUPER}\ \frac{p.c <_X q.c' \quad \langle r.t'.m, \overline{p.t} \twoheadrightarrow p_0.t_0, \overline{N}, \_\rangle \in_X q.c' \quad \neg\text{abstract}(\overline{N}) \quad \text{accmember}_X(r.t', q.c', p.c, \overline{N}) \quad X, p.c, \Gamma \vdash \overline{E : p.t}}{X, p.c, \Gamma \vdash \textbf{super}.m(\overline{E}) : p_0.t_0}$$

$$\textbf{CAST3}\ \frac{\text{acctype}_X(r.i, p) \quad X, p.c, \Gamma \vdash E : q.c' \quad \text{final}_X(q.c') \to q.c' \leq_X r.i}{X, p.c, \Gamma \vdash (r.i)E : r.i}$$

$$\textbf{CAST4}\ \frac{\text{acctype}_X(r.t, p) \quad X, p.c, \Gamma \vdash E : q.i \quad \text{final}_X(r.t) \to r.t \leq_X q.i}{X, p.c, \Gamma \vdash (r.t)E : r.t}$$

Figure 6. Typing rules for expressions

left-hand side field access and the right-hand side expressions can thus be both of non-public types. Consequently, two non-public types potentially being in a subtype relationship can have an effect on the typing of field assignment expressions in a context. The **CALL** and **SUPER** rules check if there exists an accessible and type-compatible method to be called (§15.12). As the target method of a super call is statically determined (static binding), this method must not be abstract (§15.12.3).

For casts, there exist four rules, one for widening casts (**WCAST**) and three which may lead to narrowing casts (**CAST***) or to an incomparable other static type. They show the more restricted casting in presence of the final modifier as the compiler can now statically prove some casts to always fail at runtime (§5.5).

6. Compatibility Conditions for Expressions

In this section, we look at program contexts with arbitrary method bodies and develop syntactic conditions for compatibility. Our main theorem shows that these conditions are nec-

Implementation Q

```
package p;

public abstract class C
         implements I {
  public I f;
  protected C g;
}

interface I {}
```

Implementation R

```
package p;

public class C {

  public D f;
  public C g;
}

final class D {}
```

Figure 7. Advanced example

essary and sufficient. The central challenge to achieve this goal is that there are infinitely many expressions in possible program contexts of a component X. How can we capture the constraints, that these expressions put on a component Y for compatibility, in a finite way? As the typing of method bodies does not depend on other method bodies, it will suffice to look at the typing of expressions in program contexts. We first explain the relevant aspects by an example. Then, we introduce (Sect. 6.2) and formalize our approach (Sect. 6.3). The last subsection presents the main theorem (Sect. 6.4).

6.1 Example

In order to show more complex aspects of compatibility, we introduce two different implementations (Q and R) of a package `p` in Fig. 7.

Let us now analyze whether R is compatible with Q. Both versions of class `C` define a field `f` of non-public type `I` and `D`, respectively. If we have a context which has a variable `v` of static type `C`, the cast expression `(Comparable)v.f` is well-formed for Q (as there might be an object of an interface type `Comparable` referenced by `f`) but not for R (see typing rule **CAST3**) as it can be statically ensured that the final class `D` can never have subclasses implementing the interface `Comparable`.

If we remove the final modifier on class `D`, does this ensure compatibility of R with Q? To answer this, let us consider the field assignment expression **this**`.f = `**this**`.g`. The expression is well-formed in a subclass of `C` when Q is used (as `C` is a subtype of `I`) but not when R is used (no subtyping between `C` and `D`). The same issue would still occur if both fields `f` and `g` had non-public types (in the given example, only `f` has a type (`I` or `D`) which is not public). Thus, it can have an effect on the well-formedness of entities outside of a package whether or not two non-public types are in a subtype relationship. If we have, for example, two (accessible) fields of non-public types which are in a subtype relationship in Q but not in R, we can easily construct an expression which types with Q but not R. The same is applicable if, instead of the field `f` which we assign to, there is a method with a non-public parameter type, or if, instead of the field we assign from, there is a method with a non-public return type.

Another difference between Q and R is the access modifier of the field `g`. Even if it is more accessible in R, can we conclude that the difference does not lead to incompatibility of Q and R? In order to answer this question, we must consider all possible contexts. To illustrate how to deal with this (possibly infinite) set of contexts, let us introduce an example context K (representing a class of contexts) in Fig. 8 for our two components Q and R of Fig. 7:

```
package k;
class MyClass extends p.C {
  Object m() { E }
}
```

Figure 8. Example context

There are now infinitely many choices to define E such that E is well-typed in KQ (i.e., $\vdash KQ$). Possible choices for E include **this**`.f`, **new** `MyClass()`, **new** `MyClass().g`, **this**`.g.f`, **this**`.f = `**this**`.g` and `(p.C)`**null**. We want to require that all of these expressions are also well-typed in KR. We need to find a way *a*) to characterize these infinitely many expressions in a finite way and *b*) use the finite characterization to check that the expressions remain well-typed for R.

6.2 General Approach

The first concern (*a*) is addressed by introducing an appropriate abstraction relation and the second one (*b*) by adapting the typing rules to abstract contexts.

Abstraction relation. We introduce an abstraction relation $\Theta_{X,Y}$, derived from X and Y, that ranges over the types of the components X and Y and characterizes for all possible expressions (in all possible contexts) how their types differ when compiled against X or Y. The abstraction relation $\Theta_{Q,R}$, based on the components Q and R from Fig. 7, relates among others the types `I` and `D` because there exists a context with an expression which types to `I` when compiled against Q and which types to `D` when compiled against R. Using expression **this**`.f` for E in the context K of Fig. 8 yields such a witness, as it types to `I` when compiled against Q and to `D` when compiled against R.

The abstraction relation $\Theta_{X,Y}$ is complete and precise in the following sense. If there is a context with an expression E which types to the type `T1` (of the component X) when compiled against X and which types to `T2` (of the component Y) when compiled against Y, then the types `T1` and `T2` are related by the abstraction $\Theta_{X,Y}$. Conversely, if two types are in the relation, then there always exists a context with an expression E which types to the first or second type depending against which component X or Y the context is compiled.

To check if the expression E in a class of the context is well-typed, it seems relevant to know which class of X (or Y) the context class is subclassing. Consider for example again our component Q from Fig. 7. If the field `f` is declared as protected, then the expression $E \equiv$ **this**`.f` is only well-typed if it is located in a class which is a subclass of `C`. The abstraction relation accounts for this by identifying the component class under which two types are related. For example, if the field `f` is declared as protected, then the types `I` and `D` are only related for expressions which occur in a subclass of `C`. Our abstraction relation $\Theta_{Q,R}$ contains entries of the form (`C` $\vdash$ `I`, `D`), meaning that there exist expressions in contexts which occur in a subclass of (the component class) `C` and which type to `I` when compiled against Q and which type to `D` when compiled against R.

Syntactic conditions. If we have two types `T1` and `T2` related by the abstraction relation representing a set of possible expressions E in the context, we can now formulate restrictions for all these expressions E in a simple way. For example, if a field access expression E`.f` is well-typed when using X, it should also be well-typed when using Y. However, we can not use the typing rules of our language directly to check these properties, as they rely on the whole program. The solution is to specialize the typing rules for our setting (e.g., we know that we check only expressions in the context and that the packages in the context and the component are distinct) in such a way that they only need the information provided by the abstraction Θ. Syntactic conditions can then be stated using the abstraction as follows: If field `f` is accessible as a member of `T1`, it should also be accessible as a member of `T2`.

6.3 Formalization

This section presents the formalization of the general approach described above. We inductively define the (finite) type relation $\Theta_{X,Y}$ for two components X and Y in Fig. 9. The ternary relation $\Theta_{X,Y}$ characterizes for all possible expressions (in all possible contexts) how their types (usually $p_1.t_1$ and $p_2.t_2$) differ when compiled against X or Y. It further identifies the pivotal component classes ($q_0.c_0$) that have an effect on the context's typing[11]. Before we explain the definition of Θ, we further illustrate the abstraction relation using the following lemma.

Lemma 2 (Static context abstraction)**.** *Consider two weakly compatible components X and Y. Then, for all $q_0.c_0 \in C_X^{\mathbf{o}}$ with $\text{public}_X(q_0.c_0) \wedge \neg\text{final}_X(q_0.c_0)$, and for all $p_1.t_1 \in \mathcal{T}_X$ and $p_2.t_2 \in \mathcal{T}_Y$, the following two statements are equivalent:*

1. *There exist*
   - *some context K with only **null**-valued method bodies such that $\vdash KX$,*
   - *some class $k.c \in C_K$ such that $q_0.c_0$ is the nearest superclass of $k.c$ not in C_K,*
   - *some expression E and*
   - *some $\Gamma = (\textbf{\textit{this}}{:}k.c\ \overline{v{:}q.t})$ with $\text{acctype}_{KX}(\overline{q.t}, k)$*

   such that
   - *$KX, k.c, \Gamma \vdash E : p_1.t_1$ and*
   - *$KY, k.c, \Gamma \vdash E : p_2.t_2$*
2. *$(q_0.c_0 \vdash p_1.t_1, p_2.t_2) \in \Theta_{X,Y}$*

Proof. Given in Sect. A.2 of the appendix. □

$$\textbf{T1}\ \frac{q_0.c_0 \in C_X^{\mathbf{o}} \quad \text{public}_X(q_0.c_0) \quad \neg\text{final}_X(q_0.c_0) \quad p.t \in \mathcal{T}_X \quad \text{public}_X(p.t)}{(q_0.c_0 \vdash p.t, p.t) \in \Theta_{X,Y}}$$

$$\textbf{T2}\ \frac{\begin{array}{c}(q_0.c_0 \vdash q_1.t_1', q_2.t_2') \in \Theta_{X,Y} \quad \text{public}_X(q_1.t_1') \quad \text{public}_Y(q_2.t_2') \\ \langle \_.\_.f, p_1.t_1, \overline{N}_1\rangle \in_X q_1.t_1' \quad \langle \_.\_.f, p_2.t_2, \overline{N}_2\rangle \in_Y q_2.t_2' \\ \text{public}(\overline{N}_1) \quad \text{public}(\overline{N}_2)\end{array}}{(q_0.c_0 \vdash p_1.t_1, p_2.t_2) \in \Theta_{X,Y}}$$

$$\textbf{T3}\ \frac{\begin{array}{c}q_0.c_0 \in C_X^{\mathbf{o}} \quad \text{public}_X(q_0.c_0) \quad \neg\text{final}_X(q_0.c_0) \\ \langle \_.\_.f, p_1.t_1, \overline{N}_1\rangle \in_X q_0.c_0 \quad \langle \_.\_.f, p_2.t_2, \overline{N}_2\rangle \in_Y q_0.c_0 \\ \text{public}(\overline{N}_1) \vee \text{protected}(\overline{N}_1) \quad \text{public}(\overline{N}_2) \vee \text{protected}(\overline{N}_2)\end{array}}{(q_0.c_0 \vdash p_1.t_1, p_2.t_2) \in \Theta_{X,Y}}$$

$$\textbf{T4}\ \frac{\begin{array}{c}(q_0.c_0 \vdash q_1.t_1', q_2.t_2') \in \Theta_{X,Y} \quad \text{public}_X(q_1.t_1') \quad \text{public}_Y(q_2.t_2') \\ \langle \_.\_.m, \overline{r_1.t_1''} \twoheadrightarrow p_1.t_1, \overline{N}_1, \_\rangle \in_X q_1.t_1' \\ \langle \_.\_.m, \overline{r_2.t_2''} \twoheadrightarrow p_2.t_2, \overline{N}_2, \_\rangle \in_Y q_2.t_2' \\ \text{size}(\overline{r_1.t_1''}) = \text{size}(\overline{r_2.t_2''}) \quad \text{public}(\overline{N}_1) \quad \text{public}(\overline{N}_2)\end{array}}{(q_0.c_0 \vdash p_1.t_1, p_2.t_2) \in \Theta_{X,Y}}$$

$$\textbf{T5}\ \frac{\begin{array}{c}q_0.c_0 \in C_X^{\mathbf{o}} \quad \text{public}_X(q_0.c_0) \quad \neg\text{final}_X(q_0.c_0) \\ \langle \_.\_.m, \overline{r_1.t_1''} \twoheadrightarrow p_1.t_1, \overline{N}_1, \_\rangle \in_X q_0.c_0 \\ \langle \_.\_.m, \overline{r_2.t_2''} \twoheadrightarrow p_2.t_2, \overline{N}_2, \_\rangle \in_Y q_0.c_0 \\ \text{size}(\overline{r_1.t_1''}) = \text{size}(\overline{r_2.t_2''}) \quad \text{public}(\overline{N}_1) \vee \text{protected}(\overline{N}_1) \\ \text{public}(\overline{N}_2) \vee \text{protected}(\overline{N}_2)\end{array}}{(q_0.c_0 \vdash p_1.t_1, p_2.t_2) \in \Theta_{X,Y}}$$

Figure 9. Inductive definition of Θ, representing corresponding occurrences of types in an expression of a context

The lemma asserts that the abstraction relation is complete and precise. In case of our example in Fig. 8 which extended the example from Fig. 7, the triple (`p.C` $\vdash$ `p.I`, `p.D`) resulting from the expressions **this**`.f`, **this**`.g.f`, and **this**`.f = ` **this**`.g` and the triple (`p.C` $\vdash$ `p.C`, `p.C`) resulting from the expressions **new** `MyClass().g` and `(p.C)`**null** should both be contained in the relation $\Theta_{Q,R}$ according to Lemma 2. The following rules **T1** (for (`p.C` $\vdash$ `p.C`, `p.C`)) and

[11] In a scenario of open packages, Θ would additionally contain an abstraction of the package name where the context expression is located.

T3 (for (`p.C` ⊢ `p.I`, `p.D`)) show that both triples are indeed in $\Theta_{Q,R}$.

The rules **T1**, **T3** and **T5** in Fig. 9 are the base constructors for Θ. Rule **T1** represents among others cast expressions $(p.t)E$, local variables v where $(v{:}p.t) \in \Gamma$, or instance creation expressions **new** $p.t$. Rules **T3** and **T5** represent field access expressions $E.f$ or method call expressions $E.m(...)$ where the type of E is a type of the context (e.g., `MyClass` in the previous example) and which is a subtype of $q_0.c_0$ (e.g., `p.C`). In this case, the field f to be accessed must be public or protected (see Def. of accmember). The step constructors **T2** and **T4** represent expressions (in the context) of the form $E.f$ or $E.m(...)$ where the type of E is a type defined in the component X and Y, respectively. In this case, the field f to be accessed must be public as the context can not be involved in the implementation of f (see again Def. of accmember) because X and Y are supertype-closed. All rules have in common that the class $q_0.c_0$ must be public and non-final, otherwise there exists no expression in the context which is located in a subclass of $q_0.c_0$.

Typing rule abstraction. Using our abstraction Θ of corresponding typed expressions in contexts, we can now formulate conditions on this abstraction which preserve well-typing. This is done by providing for each typing rule checked in the context K an equivalent syntactic check abstracting from K. As the typing rules for expressions (see Fig. 6) can only be applied to check well-formedness if we have a concrete context, we provide an abstraction of the typing rules for expressions in Fig. 10.

$$\phi_X^{\text{NEW}}(p.c) = \text{public}_X(p.c) \wedge \neg\text{abstract}_X(p.c)$$

$$\phi_X^{\text{GET}}(f, p.t) = \begin{array}{l}\text{public}_X(p.t) \wedge \langle \_.\_.f, \_.\_, \overline{N}\rangle \in_X p.t \\ \wedge\ \text{public}(\overline{N})\end{array}$$

$$\phi_X^{\text{CALL}}(m, p.t, \overline{r.t}) = \begin{array}{l}\text{public}_X(p.t)\ \wedge\ \text{public}_X(\overline{N}) \\ \quad \wedge\ \langle \_.\_.m, \overline{r.t} \twoheadrightarrow r.t, \overline{N}, \_\rangle \in_X p.t\end{array}$$

$$\phi_X^{\text{GETSUB}}(f, q_0.c_0) = \begin{array}{l}\langle \_.\_.f, \_.\_, \overline{N}\rangle \in_X q_0.c_0 \\ \wedge\ (\text{public}(\overline{N}) \vee \text{protected}(\overline{N}))\end{array}$$

$$\phi_X^{\text{CALLSUB}}(m, q_0.c_0, \overline{r.t}) = \begin{array}{l}\langle \_.\_.m, \overline{r.t} \twoheadrightarrow r.t, \overline{N}, \_\rangle \in_X q_0.c_0 \\ \wedge\ (\text{public}(\overline{N}) \vee \text{protected}(\overline{N}))\end{array}$$

$$\phi_X^{\text{WCAST}}(p.t, q.t') = q.t' \leq_X p.t$$

$$\phi_X^{\text{CAST2}}(p.t, q.t') = p.t \in C_X^{\mathbf{o}} \wedge q.t' \in C_X^{\mathbf{o}} \wedge p.t \leq_X q.t'$$

$$\phi_X^{\text{CAST3}}(p.t, q.t') = \begin{array}{l}p.t \in \mathcal{I}_X \wedge q.t' \in C_X^{\mathbf{o}} \\ \wedge\ (\text{final}_X(q.t') \rightarrow q.t' \leq_X p.t)\end{array}$$

$$\phi_X^{\text{CAST4}}(p.t, q.t') = q.t' \in \mathcal{I}_X \wedge (\text{final}_X(p.t) \rightarrow p.t \leq_X q.t')$$

$$\phi_X^{\text{CAST}}(p.t, q.t') = \begin{array}{l}\phi_X^{\text{WCAST}}(p.t, q.t') \vee \phi_X^{\text{CAST2}}(p.t, q.t') \\ \vee\ \phi_X^{\text{CAST4}}(p.t, q.t') \vee \phi_X^{\text{CAST3}}(p.t, q.t')\end{array}$$

Figure 10. Typing rule abstraction

The conditions (e.g., ϕ^{NEW}, etc.) look very similar to the premisses of the corresponding typing rules (e.g., **NEW**) in Fig. 6. The definition ϕ^{NEW} represents the well-formedness check on context expressions of the form **new** $p.c$ where $p.c$ is a type of the component. The premiss acctype(...) of the typing rule **NEW** is abstracted to public(...) as we know that the accessing package (part of the context) and the accessed package (part of the component) are different. The definition of ϕ^{GET} represents an abstraction of the premisses of the typing rule **GET**. It checks whether the field f is accessible in the context on a reference of type $p.t$ which is part of the component. The premiss acctype(...) of the typing rule **GET** is abstracted to public(...) as for ϕ^{NEW}. The premiss accmember(...) in **GET** is abstracted to public(...) in ϕ^{GET} for the following reasons: As the accessing package and the accessed package are different, the predicate package(...) never holds. It is also clear that private(...) can not hold. Furthermore, as our context can not be involved in the implementation (see Def. of accmember) of $p_1.t_1$ or $p_2.t_2$ because X and Y are components, e.g., supertype-closed, the predicate protected(...) will not hold either. The definition of ϕ^{CALL} follows the same reasoning as for ϕ^{GET}.

The definition ϕ^{GETSUB} checks whether access to a field f is possible on a reference of a type of the context (e.g., `MyClass`) which is a subtype of $q_0.c_0$. In this case, the abstraction of accmember(...) is not simply public(...) as protected members are also accessible from subclasses of $q_0.c_0$. The abstraction is again similar for ϕ^{CALLSUB}. The definition of ϕ^{CAST} checks whether a cast from an expression of type $q.t'$ (of the component) to the type $p.t$ is possible. This holds if it is either a valid widening or narrowing cast.

Syntactic conditions. The presented abstractions of the typing rules allow us now to define the following syntactic compatibility conditions.

$\boxed{\forall(\_.\_ \vdash p_1.t_1, p_2.t_2) \in \Theta_{X,Y}:}$

$$\phi_X^{\text{GET}}(f, p_1.t_1) \rightarrow \phi_Y^{\text{GET}}(f, p_2.t_2) \qquad \text{(R5}_{X,Y}\text{)}$$

$$\phi_X^{\text{CALL}}(m, p_1.t_1, \overline{r.t}) \rightarrow \phi_Y^{\text{CALL}}(m, p_2.t_2, \overline{r'.t'})\ \wedge\ \overline{r.t \leq_Y r'.t'} \qquad \text{(R6}_{X,Y}\text{)}$$

$$\forall p.t \in \mathcal{T}_X : \text{public}_X(p.t) \wedge \phi_X^{\text{CAST}}(p.t, p_1.t_1) \rightarrow \phi_Y^{\text{CAST}}(p.t, p_2.t_2) \qquad \text{(R7}_{X,Y}\text{)}$$

$$\forall(p.t, Q) \in \mathcal{T}_X^{\leq} : \phi_{XQ}^{\text{CAST}}(p.t, p_1.t_1) \rightarrow \phi_{YQ}^{\text{CAST}}(p.t, p_2.t_2) \qquad \text{(R8}_{X,Y}\text{)}$$

Rule (R5) states that every field access expression of the form $E.f$ which is well-typed when compiled against X must also be well-typed when compiled against Y (otherwise Y is not compatible with X). The rationale behind rule (R6) is a similar one. Here, we additionally require that the parameters which work for the method call when compiled against X also work for the method call when compiled against Y.

Rules (R7) and (R8) handle cast expressions and are a bit more complicated. Rule (R7) deals with casts of the form $(p.t)E$ where both $p.t$ and the type of E ($p_1.t_1$ or $p_2.t_2$) are types which belong to the components X and Y. Rule (R8) deals with casts where $p.t$ is a type which has been defined in the context. Remember that for our example at the beginning of this section, the expression `(Comparable)v.f` detected an incompatibility. As the *cast goal* `Comparable` might be a

type defined in the context, it is not sufficient to try and cast $p_1.t_1$ and $p_2.t_2$, respectively, to all possible (public) types of the component X. The definition $\mathcal{T}^{\leq}$ generates an interface (similar to `Comparable`) which detects the incompatibility. It is a finite equivalence class of cast goals in the context (the precise definition of $\mathcal{T}^{\leq}$ is given in Def. 5 of the appendix).

$$\forall p.c \in C_X : \phi_X^{\text{NEW}}(p.c) \rightarrow \phi_Y^{\text{NEW}}(p.c) \tag{R9$_{X,Y}$}$$

$$\forall q_0.c_0 \in C_X^{\mathbf{o}} : \text{public}_X(q_0.c_0) \wedge \neg\text{final}_X(q_0.c_0) \wedge \phi_X^{\text{GETSUB}}(f, q_0.c_0) \rightarrow \phi_Y^{\text{GETSUB}}(f, q_0.c_0) \tag{R10$_{X,Y}$}$$

$$\forall q_0.c_0 \in C_X^{\mathbf{o}} : \text{public}_X(q_0.c_0) \wedge \neg\text{final}_X(q_0.c_0) \wedge \phi_X^{\text{CALLSUB}}(m, q_0.c_0, \overline{r.t}) \rightarrow \phi_Y^{\text{CALLSUB}}(m, q_0.c_0, \overline{r'.t'}) \wedge \overline{r.t \leq_Y r'.t'} \tag{R11$_{X,Y}$}$$

$$(q_0.c_0 \vdash p_1.t_1, p_2.t_2) \in \Theta^{\mathbf{f}}_{X,Y} \wedge (q_0.c_0 \vdash p_1'.t_1', p_2'.t_2') \in \Theta_{X,Y} \wedge p_1'.t_1' \leq_X p_1.t_1 \rightarrow p_2'.t_2' \leq_Y p_2.t_2 \tag{R12$_{X,Y}$}$$

Rule (R9) checks that every instance creation expression which is well-typed when compiled against X is also well-typed when compiled against Y. It is sufficient to consider all the class types of the component (and not the context) as the context remains unchanged when compiled against Y instead of X. This means that a class of the context which is not abstract when compiled against X will not become abstract when compiled against Y. Rules (R10) and (R11) represent field access expressions $E.f$ and method call expressions $E.m(...)$ in the context where E is a type of the context (which is a subtype of $q_0.c_0$) and f or m has been defined in the component X or Y.

Rule (R12) is a bit more complicated. We define $\Theta^{\mathbf{f}}$ as a subset of Θ where we only keep the elements of Θ whose construction has ultimately been done by the rule **T1**, **T2** or **T3** (see Fig. 9). These are the types that might occur in a context by an expression of the form $E.f$. Rule (R12) then checks that, according to the premisses of the rule **SET**, that the right-hand side expression of the field assignment is a subtype of the left-hand side field type. For our example, this check fails, as $\Theta^{\mathbf{f}}_{Q,R}$ contains (`p.C` $\vdash$ `p.I`, `p.D`) and $\Theta_{Q,R}$ contains (`p.C` $\vdash$ `p.C`, `p.C`) and `p.C` $\leq_Q$ `p.I` but `p.C` $\not\leq_R$ `p.D`. The context expression **`this.f = this.g`** represents this situation.

The relation Θ provides just enough context information to check each of the premisses, which is one of the key points of the formalization.

6.4 Syntactic Compatibility & Main Theorem

We can finally give our definition of syntactic compatibility, which requires that all of the previous conditions hold.

Definition 4 (Syntactic compatibility)**.** A component Y is syntactically compatible with a component X if and only if $X \lhd Y$ (see Def. 3) and (R5$_{X,Y}$)-(R12$_{X,Y}$) hold.

The main theorem and central result of this paper states that our definition of syntactic compatibility is well-chosen, as it coincides with contextual compatibility (from Def. 1).

Main Theorem. *Two components are contextually compatible* **(1)** *if and only if they are syntactically compatible* **(2)***.*

Proof. Sketch:

- **(1) ⇒ (2):** We show that the syntactic compatibility conditions are necessary. *Proof by contrapositive*. We assume that Y is not syntactically compatible with X, e.g., there exists a rule (Rn) which does not hold. Then we give the construction of a context K such that $\vdash KX$ but $\nvdash KY$.
- **(2) ⇒ (1):** We show that the syntactic compatibility conditions are sufficient. *Direct Proof*. We assume Y is syntactically compatible with X and also assume that there exists a codebase K such that $\vdash KX$. Then we show that $\vdash KY$. The proof goes by case analysis over all possible context conditions / typing rules.

For a more detailed proof sketch, we refer to Sects. A.3.1 and A.4.1 of the appendix. We use our static context abstraction lemma (Lemma 2) extensively in these proofs. On one hand it allows us to construct a context violating compatibility if two components are not syntactically compatible. On the other hand, it can reduce all possible contexts to a finite abstraction. □

Our main theorem ensures that the syntactic check is equivalent to proving that two components are compatible for all possible contexts. The syntactic check is also computable, because the type relation Θ is finite, as there exists only a finite number of packages, classes, fields and methods in X and Y and all of the used helper functions and predicates are computable. We have validated the approach by creating a source compatibility checker [42] for our Java subset.

7. Applications and Future Work

In this section we discuss the impact of compatibility on language and program design by illustrating issues and solutions for a selected choice of language constructs. We also present possible applications for syntactic compatibility relations.

7.1 Language and Program Design

Simpler accessibility system. In Sect. 3 we restricted our language by additional accessibility conditions that go beyond those given in the JLS. Context conditions (C14) and (C15) require that public (or protected) methods of public types only have public parameter and return types. For example, while being a legal Java program, we reject the following code as the parameter type of the method `m` is not public.

```
package p;
public interface I { public abstract I m(J j); }
interface J {}
```

The C# language specification [19] defines similar restrictions. Sect. 10.5.4 on accessibility constraints presents

conditions that among others require parameter and return types to be at least as accessible as the method itself. These additional constraints lead to important properties; they ensure for example that public interfaces can always be implemented, which is not the case in Java. For example the interface I above cannot be implemented outside of p, although it is public[12].

The constraints further ensure that at each call site the method parameter and return types are types which are accessible to the calling context. We can thus provide an expression of exactly that type at the call site. This is also the reason why we can specify the restriction $\overline{r.t} \leq_Y \overline{r'.t'}$ in rule (R6) as we always know by (R1) that the types $\overline{r.t}$ exist in Y as they exist and are public in X. If (C14) and (C15) were not enforced, $\overline{r.t}$ might not be types that are accessible to the context. Thus, we would have to check for each possible element $(q_0.c_0 \vdash p_1.t_1, p_2.t_2)$ in $\Theta_{X,Y}$ that if $p_1.t_1$ is a subtype of one element of the type sequence $\overline{r.t}$, then $p_2.t_2$ is also a subtype of the corresponding element of $\overline{r'.t'}$ (where $\overline{r.t}$ and $\overline{r'.t'}$ would not necessarily have to be public types). This would have led to substantially more complexity of the proof and the checking.

To further substantiate our claim that the restrictions (C14) and (C15) are acceptable and reasonable, we investigated the impact of this simplification on real, industrial-strength Java libraries and programs. We developed a tool [42] that can analyze huge codebases for *counter examples* violating (C14) and (C15). To handle full Java code, the tool goes beyond the subset considered in this paper. In particular, it can handle nested classes with all access modifiers and covers the additional cases for access modifiers in methods' signatures as well. In our analysis, we mainly focused on libraries and frameworks, because they usually provide more interesting encapsulation aspects.

The results are shown in Table 1. As we did not eliminate duplicates which occur due to inheritance of methods, a realistic count would lead to even less occurrences. In summary, the number of occurrences (= violations) is very small (blank space indicates no occurrence), and most of them are in Eclipse packages containing the name "internal". These packages are, according to the Eclipse naming conventions [17], part of the platform implementation and not part of the exposed API. We found that most of the occurrences were design errors which can easily be fixed. In conclusion, the additional context conditions lead to simpler package interfaces and simplify compatibility checking without imposing restrictions for practical use of the language. By a similar restriction on fields, we could also simplify the syntactic rule (R12), which is expensive to check. The new simplified version would only need to check that the type hierarchy of the public types in X is preserved in Y.

[12] This problem is even more apparent when abstract methods in classes use nested types of restricted accessibility.

Ambiguous names. We considered the set of package, class and variable identifiers to be disjoint in our formalization, which is not the case in Java. For example, in Java, the name `a.b.c` might refer to the class `c` in package `a.b`, but could as well refer to the static member class `c` of the class `b` in package `a`. To deal with this issue, the JLS provides precedence rules for disambiguation (§6.3.2). For example, variables will be chosen in preference to types and types will be chosen in preference to packages. As consequence, even adding a private field can be an incompatible change. As an illustrative example, consider the following code (inspired by [6, Puzzle 68]) which consists of the package `p` (which represents the component to be evolved) and the package k which represents one possible context.

```
package p;
public class c1 {
  public static class c2 { public static Object c3; }
}
public class d { Object c3; }

package k;
class c { Object m(){ return p.c1.c2.c3; }}
```

If we add "**private static** `d c2;`" as a static field to class c_1, the field obscures the class c_2 and the context expression $p.c_1.c_2.c_3$ does not compile anymore against our new component as the private field c_2 is not accessible.

In our Java subset, we avoided issues of ambiguous type names, as we require all our type names to be fully qualified. For full Java, however, adding a public type to a component can result in abiguous type names in the context if the context uses *import* wildcards (see [11]). Naming conventions and programming guidelines in general [17, 32, 36] help to avoid the issues described above. In particular, one might want to restrict the set of contexts (e.g., contexts should not use *import* wildcards).

Adding a method. As we have shown in Sect. 4, adding a method declaration to an interface is already an API breaking change. In order to deal with this issue, Eclipse developers adopt the following convention described in [13]. They create a new interface which extends the old interface with the new method. However, this has the drawback that, in order to use the new interface, clients need to resort to casting in order to access the additional methods.

Another possible solution that we could imagine is to give programmers more control over the two different API's (client and implementor) provided by the interface. For example (as advocated for in [5]), programmers might declare interfaces which can be publicly used from a client point of view, yet only implemented within the same package[13].

[13] This restriction can currently be encoded in Java by adding a dummy method with a non-public parameter type to the interface.

Codebase (packages considered)	Total methods	Occurrences (number of methods)				
		Accessibility of method:				
		public			**protected**	
		Accessibility of parameter type:				
		protected	**package**	**private**	**package**	**private**
JRE rt.jar Version 1.6.0_16-b01 (com.sun.* | sun.*)	77508	5	47		162	
JRE rt.jar Version 1.6.0_16-b01 (rest)	47975				2	
Eclipse 3.5.1 (org.eclipse.*.internal*)	146397	44	79	17	51	2
Eclipse 3.5.1 (org.eclipse.* rest)	48604		5			
NetBeans 6.7.1 (org.netbeans.*)	164314		35		24	
ActiveMQ 5.3.0 (*.activemq*)	16183		4		3	
BCEL 5.2	2628					
JUnit 4.7	966		2			
Lucene 2.9.1	8966		2			

Table 1. Number of methods in public types which have the given characteristics

Checked exceptions. Proper handling of checked exceptions is enforced by the JLS (§11.2). If a method declares that it might throw a checked exception, then call sites have to provide an exception handling block to deal with the exceptions (or alternatively they can just declare throwing the exception themselves). It is thus obvious that, if we change a method in our component such that it can throw a checked exception by adding a throws clause, possible calling contexts will stop from compiling.

The more interesting question is whether removing the throws clause of a method is an incompatible change. For API consumers which extend the type where the method is declared, this may break the requirement that overriding methods must not be declared to throw more checked exceptions than the overridden ones (§8.4.6). For API users, which use the method at calling sites, removing the throws clause is also an incompatible change. This is due to the fact that if a client declares an exception handling block for a checked exception, then there must be a preceding call site of a method which declares throwing such an exception (§11.2.3). In order to provide better support for this last kind of API users, the JLS could instruct compilers to only issue warnings as this additional restriction does not have an influence on type safety.

Further topics. We have only considered a selected range of programming language constructs which influence compatibility. In our formalized subset, we have not considered constructors with reduced accessibility, static members, nested classes, nested packages, generics and many more Java features.

Future programming languages and module systems should consider such compatibility issues, e.g., a good definition of a module should lead to simple syntactic compatibility conditions. We have seen in this section, that for OO languages, sometimes more static restrictions than in Java (e.g., (C14) and (C15)), sometimes less restrictions (e.g., checked exceptions) would provide better support for compatibility. We argue that simpler compatibility checking should be a design goal for programming language design. The aim is to reduce the number of breaking changes by providing the right abstractions for API evolution. A prerequisite is that language designers become aware of compatibility issues when designing the abstractions and well-formedness rules of the language.

7.2 Static Compatibility Checking

The Eclipse Platform provides guidelines [12] and tools [18] to support (compatible) API evolution. The tools detect *binary* incompatibilities and usage of non-API code between plug-ins (where API code must be tagged as such). However they do not detect source incompatible changes as presented here.

Based on the conditions for syntactic compatibility given previously, we can statically check compatibility. The conditions are designed in such a way that the reader can better compare them to the typing rules and such that the proofs become readable. They are not meant to be used directly for automated checking. That is, the naive checking algorithm that first computes $\Theta_{X,Y}$ and then checks the rules one by one is not optimal. It will lead to checking many subconditions (e.g., that a type is public) several times, but it provides the specification for better algorithms. Lemma 2 and its proof in Sect. A.2 of the appendix help to generate counter-examples if compatibility does not hold.

Instead of defining the syntactic compatibility relation directly on package implementations, as it was done here, it is also possible to derive (syntactic) package signatures from the implementations and then define compatibility based on these signatures (like SML [34] signatures and signature subtyping). This is a two-step process which might lead to better module/package designs. Currently, the package signature is hidden in the definitions of compatibility. A possible application for this is modular typechecking at the

package level, e.g., the compiler may not need to know about non-public types to typecheck other packages. It would also be interesting to apply the proof approach (using Θ) to other programming languages and module systems.

7.3 Package-local Refactoring

Most semantic-preserving refactoring techniques assume that the complete program is available (the usual closed-world view). They allow to reason about semantic preservation of refactorings for one single context, the given program. This might fit well for developers of a single program, but not at all for library or component developers. While most of the existing work which tries to address this issue, e.g., [4, 8, 14, 22, 25], tracks modifications of the library and creates compatibility layers or adapts the clients, we consider a far simpler setting where no such tracking is needed.

Let us consider the *Rename Variable* refactoring as described in [40]. The refactoring should work in such a way that all bindings are preserved, i.e., all accesses to a declaration should be preserved by the renaming. In a complete program, all accesses to a declaration are known. In the setting of refactoring a library, however, this assumption does not hold. Our finite, precise and complete relation Θ additionally provides an abstraction for all such accesses from outside the package.

We see two ways to realize package-local refactoring. One way would be to do the refactoring and then check if the new library version is (syntactically) compatible with the old one. Another way would be to statically prove that a certain class of refactorings guarantee (syntactic) compatibility. One could also restrict the set of possible contexts by describing acceptable contexts syntactically in the signature (contract) of the package, which would be a prerequisite for true *modular* refactoring. We plan on further investigating both scenarios in the future.

8. Related Work

In this section, we present related work that has not been discussed so far. Dmitriev [15] investigates `make` technology for the Java language, in particular how a change to a class *may* affect other classes. Source incompatible changes (at the class, not package level) are listed in a semi-formal way, but neither proven necessary nor sufficient. To our knowledge there is no other work for object-oriented languages that makes source compatibility the focus of investigation. In the following, we relate our topic to work on behavioral equivalence of object-oriented components, binary compatibility, separate compilation, object-oriented module systems, language design aspects, and refactoring techniques.

Behavioral equivalence. Two classes, two packages, or generally two components are called behavioral equivalent if they have the same interface behavior. Source compatibility is a prerequisite for behavioral equivalence: If two components are not source compatible, there is a context that compiles with one, but not with the other component and thus the components are not equivalent. Koutavas and Wand [29] present proof techniques to show that two classes are equivalent, but source compatibility is almost trivial in the language subset they consider, without packages and with only very restricted use of access modifiers. Closely related to our work is the notion of compatibility of Jeffrey and Rathke [26] who develop a fully abstract trace semantics for a Java-like core language with a package construct. They develop a syntactic characterization of compatibility[14] ([26], Sect. 3) as a prerequisite to a fully abstract semantics of packages. However, the setting they consider has a non-standard package concept and only two different access modifiers.

Binary compatibility. Chapter 13 of the Java Language Specification [23] defines properties for binary compatibility: a set of changes that developers are permitted to make to their packages, classes, or interfaces. This set must guarantee that preexisting class files which linked with the previous (package, class or interface) versions still link with the current versions. As already mentioned in the JLS (§13.2), binary compatibility is different from source compatibility. For example (§13.4.6), introducing a new field, with the same name as an existing field in a subclass of the class containing the existing field declaration, does not break binary compatibility with preexisting binaries. However, at the source code level, this may lead to source incompatibility (typing error). A new declaration is added, changing the meaning of a name in an unchanged part of the source code, while the preexisting binary for that unchanged part of the source code retains the fully qualified, previous meaning of the name.

Binary compatibility gives weaker guarantees to clients as source compatibility. If we consider the case that a library developer has made binary compatible changes to his code, a client developer may not be able to recompile his ongoing project with the new version of the library (e.g., if he wants to make some fixes to his client code). Another important issue with compatibility as defined by the JLS is that only sufficient conditions[15] are given (e.g., §13.3). Furthermore, different encapsulation boundaries are considered (e.g., packages, classes and interfaces) which complicate the presentation even more.

Forman et al. [21] have investigated binary compatibility for IBM's System Object Model. They provide a set of transformations which should guarantee compatibility, but do not provide any proofs. Drossopoulou et al. [16] analyzed binary compatibility as it is defined in the JLS, show that some of the transformations described in the JLS do not guarantee successful linking, and prove their own binary compatibility criteria correct for a Java subset. However, they do not consider whether the criteria they give are necessary conditions for source compatibility.

[14] Their work provided one of the starting points of our studies.

[15] Not even this holds true, as shown in [16].

Separate compilation. Ancona and Zucca [2] give principal typings for a Java subset without access modifiers. Ancona et al. [3] also propose a compositional compilation scheme for open codebases, e.g., which do not contain all used types. Although this work has goals different to ours, one of the main common aspects is that they have to find a representation for all possible contexts.

Lagorio [30] investigates how to extract dependency information from Java sources to deal with dependencies as we have alluded to in the "Ambiguous names" paragraph of Sect. 7.1.

Module systems and language design. Many module systems [1, 10, 27, 31, 44, 45] have been proposed for Java. We focus on the (currently) most popular ones. The OSGi Alliance provides a module system [38] for Java which focuses on the run-time module environment. However, as the module system is not tightly integrated with the Java language, the compile-time module environment may differ from the run-time module environment. Project Jigsaw [39] aims at providing a simple, low-level module system to modularize the JDK. However, it will not be an official part of the Java SE 7 Platform Specification.

Most of the aforementioned module systems focus more on visibility issues than on accessibility (as explained in Sect. 2). The Java Specification Request (JSR) 294 [28] defines a standard for module accessibility but does not fix the module boundaries. This allows module systems (e.g., like [38]) to fix module boundaries on top of it.

The existing module systems do not really solve the question, what the API of a module is. Very often, this is defined as the aggregation of the API of a set of packages or types. However, it remains unclear what the actual API of a package or type is. With the presented compatibility conditions we aim to initiate further research on alternative definitions of modules and their interplay with compatibility.

The following work studies the Java accessibility modifiers. Müller and Poetzsch-Heffter [35] identify the changes that access modifiers in a program can have on the program semantics. Schirmer [41] gives a formalization of the access modifiers and shows interesting runtime properties with respect to access integrity. The Java subset we formalize further considers the modifiers *abstract* and *final*, which also have a considerable impact on compatibility.

A setting similar to ours is the *fragile base class* problem [33]. It studies the effects that changes in a base class can have on its unknown subclasses, although mostly at the semantic level.

Refactoring. There exists a lot of work in the refactoring community to deal with API evolution. Most solutions work by creating compatibility layers for the libraries or adapting the client programs (in binary and source setting), for example [4, 8, 14, 22, 25], but this addresses a different issue. The question, what the API of a library actually is, is left unanswered or only partially answered for a fixed set of clients.

Steimann and Thies [43] describe a semantic-preserving refactoring technique to refactor programs under constrained accessibility. However, as for most of the work on refactoring, they operate on programs where all accesses to program entities are known, i.e., the usual closed-world view. Our goal is to have a refactoring technique which is modular in the sense that the parts of the library, which form the API, are only modified in a compatible way.

9. Conclusions

Evolution of code becomes more and more important. Automatic checks of compatibility between two versions of an API can be of great help to validate evolution steps. We presented a formal foundation for a language-based approach to modular compatibility checking. In particular, we developed syntactic conditions that allow to test components written in *PackageJava*, a substantial Java subset, for compatibility. Based on the conditions, we implemented a source compatibility checker [42] for *PackageJava* to validate the approach.

A further technical contribution of this paper is an abstraction technique for expressions in program contexts. We illustrated this abstraction technique for *PackageJava* and used it to verify that the developed syntactic compatibility conditions are necessary and sufficient. More generally, the technique allows to analyze different language constructs and (static) encapsulation mechanisms with respect to source compatibility aspects. This is especially important for languages with complex encapsulation properties (such as most OO languages) and gives new insight into the extensibility of packages.

In Sect. 7, we discussed current and future applications of our techniques in the area of language and program design, compatibility checking, and refactoring. In summary, we presented source compatibility as an important notion that deserves more attention both in language and program design. Most notably, improvements in program design (e.g., guidelines) as well as language design (e.g., future module systems and languages) should allow a wider range of compatible changes and enable better tool support.

Acknowledgments

This work has been partially supported by the EU-project *FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods*. We thank Mathias Weber for developing the analysis tool that we used in Sect. 7.1. We thank Ilham Kurnia, Patrick Michel, Jan Schäfer, and anonymous reviewers for their constructive feedback on an earlier version of this paper.

References

[1] D. Ancona and E. Zucca. True modules for Java-like languages. In *ECOOP*, pages 354–380, 2001.

[2] D. Ancona and E. Zucca. Principal typings for Java-like languages. In *POPL*, pages 306–317, 2004.

[3] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *POPL*, pages 26–37, 2005.

[4] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA*, pages 265–279, 2005.

[5] M. Biberstein, V. C. Sreedhar, and A. Zaks. A case for sealing classes in Java, 2002. `http://www.haifa.il.ibm.com/info/ple/papers/class.pdf`.

[6] J. Bloch and N. Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley Professional, 2005.

[7] A. Buckley. JSR 294 and module systems. `http://blogs.sun.com/abuckley/en_US/entry/jsr_294_and_module_systems`.

[8] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM*, pages 359–369, 1996.

[9] A. Coglio. Checking access to protected members in the Java virtual machine. *Journal of Object Technology*, 2005.

[10] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. MJ: A rational module system for Java and its applications. In *OOPSLA*, pages 241–254, 2003.

[11] J. D. Darcy. Kinds of compatibility. `http://blogs.sun.com/darcy/entry/kinds_of_compatibility`.

[12] J. des Rivières. Evolving Java-based APIs. `http://wiki.eclipse.org/Evolving_Java-based_APIs`.

[13] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution*, pages 83–107, 2006.

[14] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: refactoring-aware binary adaptation of evolving libraries. In *ICSE*, pages 441–450, 2008.

[15] M. Dmitriev. Language-specific make technology for the Java programming language. In *OOPSLA*, pages 373–385, 2002.

[16] S. Drossopoulou, D. Wragg, and S. Eisenbach. What is Java binary compatibility? In *OOPSLA*, pages 341–358, 1998.

[17] Eclipse Naming Conventions. `http://wiki.eclipse.org/Naming_Conventions`.

[18] Eclipse PDE API Tools. `http://www.eclipse.org/pde/pde-api-tools/`.

[19] ECMA. C# Language Specification (Standard ECMA-334, 4th edition). `http://www.ecma-international.org/publications/standards/Ecma-334.htm`.

[20] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL*, pages 171–183, 1998.

[21] I. R. Forman, M. H. Conner, S. Danforth, and L. K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA*, pages 426–438, 1995.

[22] T. Freese. Towards refactoring support in API evolution and team development. In *ICSE*, pages 953–956, 2006.

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[24] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *OOPSLA*, pages 241–253, 2001.

[25] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *ICSE*, pages 274–283, 2005.

[26] A. Jeffrey and J. Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *ESOP*, pages 423–438, 2005.

[27] JSR 277: Java Module System. `http://jcp.org/en/jsr/detail?id=277`.

[28] JSR 294: Improved Modularity Support in the Java Programming Language. `http://jcp.org/en/jsr/detail?id=294`.

[29] V. Koutavas and M. Wand. Reasoning about class behavior. In *Informal Workshop Record of FOOL*, 2007.

[30] G. Lagorio. Capturing ghost dependencies in Java sources. *Journal of Object Technology*, pages 77–95, 2004.

[31] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *OOPSLA*, pages 211–222, 2001.

[32] S. Microsystems. Code conventions for the Java programming language. `http://java.sun.com/docs/codeconv/`.

[33] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *ECOOP*, pages 355–383, 1998.

[34] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[35] P. Müller and A. Poetzsch-Heffter. Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur. In *Java-Informations-Tage*, pages 1–10, 1998.

[36] NetBeans Code Conventions. `http://netbeans.org/community/guidelines/code-conventions.html`.

[37] O. Nierstrasz. A survey of object-oriented concepts. In *Object-Oriented Concepts, Databases, and Applications*, pages 3–21. 1989.

[38] OSGi Service Platform. `http://www.osgi.org/`.

[39] Project Jigsaw. `http://openjdk.java.net/projects/jigsaw/`.

[40] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for Java. In *OOPSLA*, pages 277–294, 2008.

[41] N. Schirmer. Analysing the Java package/access concepts in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 16(7):689–706, 2004.

[42] Source Compatibility for Java Packages Website. `http://softech.cs.uni-kl.de/~scomp`.

[43] F. Steimann and A. Thies. From public to private to absent: Refactoring Java programs under constrained accessibility. In *ECOOP*, pages 419–443, 2009.

[44] R. Strnisa, P. Sewell, and M. J. Parkinson. The Java module system: core design and semantic definition. In *OOPSLA*, pages 499–514, 2007.

[45] M. Zenger. KERIS: Evolving software with extensible modules. *Journal of Software Maintenance*, 17(5):333–362, 2005.

A. Appendix: Lemmata and Proof Sketches

A.1 Helper Lemmata and Definitions

Lemma 3. *Consider two components X and Y such that $X \lhd Y$. Then*

*for all contexts K with only **null**-valued method bodies such that $\vdash KX$,*
for all classes $k.c \in C_K$,
for all expressions E and
for all $\Gamma = (\textbf{\textit{this}}{:}k.c\ \overline{v{:}r.t'})$ with $\text{acctype}_{KX}(\overline{r.t'}, k)$:

if

$KX, k.c, \Gamma \vdash E : p_1.t_{1\perp}$ *and*
$KY, k.c, \Gamma \vdash E : p_2.t_{2\perp}$

then exactly one of the following holds:

(1) $p_1.t_{1\perp} = \perp \wedge p_2.t_{2\perp} = \perp$
(2) $p_1.t_{1\perp} \in (C_K \cup \mathcal{I}_K) \wedge p_1.t_{1\perp} = p_2.t_{2\perp}$
(3) $p_1.t_{1\perp} \in \mathcal{T}_X \wedge p_2.t_{2\perp} \in \mathcal{T}_Y$

Proof. Goes by induction on height of expression E. We define the (unordinary) height function h as follows: $h(E.f) \stackrel{\text{def}}{=} h(E.f = E') \stackrel{\text{def}}{=} h(E.m(\overline{E})) \stackrel{\text{def}}{=} 1 + h(E)$, otherwise $h(\text{E}) \stackrel{\text{def}}{=} 0$. The rationale behind this function is that to determine the type of expressions, only some of their subexpressions are relevant.

Induction basis: h(E) = 0

Case **NULL**: $p_1.t_{1\perp} = p_2.t_{2\perp} = \perp$.
Case **VAR**: Then $p_1.t_{1\perp} = p_2.t_{2\perp}$ as Γ is identical in both cases.
Case **NEW**: $E =$ **new** $q'.c'$ Again $p_1.t_{1\perp} = p_2.t_{2\perp} = q'.c'$.
Case **CAST**: again similar argument.
Case **SUPER**: $E =$ **super**$.m(\overline{E})$: Either the method m has been defined in K and trivially $p_1.t_{1\perp} = p_2.t_{2\perp}$. Otherwise the method must be defined both in X and Y and as these codebases are closed (i.e., do not use types of K), $p_1.t_{1\perp} \in \mathcal{T}_X \wedge p_2.t_{2\perp} \in \mathcal{T}_Y$.

Induction step:

Case **GET**: $E = E'.f$: By I.H. and Def. of **GET** we know that either (2) or (3) holds for E'. If (2) holds, then proceed as in **SUPER** case. If (3) holds, the field must be defined both in X and Y and as these codebases are closed $p_1.t_{1\perp} \in \mathcal{T}_X \wedge p_2.t_{2\perp} \in \mathcal{T}_Y$.
Case **SET** or **CALL**: similar to **GET**. □

Lemma 4. *Consider a codebase X, a class $p.c \in C_X$, an expression E and a local variable typing $\Gamma = (\textbf{\textit{this}}{:}p.c\ \overline{v{:}r.t'})$ such that $\text{acctype}_X(\overline{r.t'}, p)$. Then*

$X, p.c, \Gamma \vdash E : \_.\__\perp$ *iff* $X, p.c \vdash$ *lang.Object m(* $\overline{r.t'\ v}$ *) { E }*

Proof. Follows from definition of typing rules **METH** and **SUB** and property that *lang.Object* is a supertype of all other types. □

Lemma 5. *Consider a codebase X, a class $p.c \in C_X$ and a non-abstract method definition M with name m where the name m does not occur in X. Let X' be the codebase X, where we additionally add M as member to the definition of $p.c$ and $X, p.c \vdash M$. Then $\vdash X$ iff $\vdash X'$.*

Proof. By context conditions and rule **DEFN-C**. □

A.2 Proof Sketch of Lemma 2 from page 11

Proof. We show both directions of the lemma.

(1) ⇒ (2): Goes by induction on the height (from Lemma 3) of expression E. The reason why induction on E is a valid proof method is given in the subsection following this proof called "Additional explanations".

Induction basis: h(E) = 0

Case **VAR**: Then $p_1.t_1 = p_2.t_2$ as Γ is identical in both cases. From restrictions on Γ (i.e., $\text{acctype}_{KX}(p_1.t_1, k)$), it follows that $\text{public}_X(p_1.t_1)$ must hold as $p_1 \neq k$. We then see that by **T1** the triple $(q_0.c_0 \vdash p_1.t_1, p_1.t_1)$ is contained in $\Theta_{X,Y}$.
Case **NEW**: $E =$ **new** $q'.c'$ As $\text{acctype}_{KX}(q'.c', k)$ must hold (due to typing rule **NEW**), use the argument as before.
Case **CAST**: again similar argument.
Case **SUPER**: $E =$ **super**$.m(\overline{E})$: If the method m has been defined in a class of a package k' of K with return type $p_1.t_1$, then $p_1.t_1 = p_2.t_2$. As at the method definition site, $\text{acctype}_{KX}(p_1.t_1, k')$ must hold, and $k' \neq p_1$, $\text{public}_X(p_1.t_1)$ holds and then the claim follows directly from **T1**. Otherwise, m is a member of $q_0.c_0$ and is accessible from $k.c$, which is exactly the case if the method is protected or public and thus the claim follows from **T5**.
Case **NULL**: not applicable as $p_1.t_1 \neq \perp \wedge p_2.t_2 \neq \perp$.

Induction step:

Case **GET**: $E = E'.f$: By Lemma 3 (appendix) and **GET**, E' must be either of a type defined in K or otherwise the I.H. holds for E'.
In the former case, the field f has either been defined in K (with a type $p_1.t_1$), and thus according to **T1**, $(q_0.c_0 \vdash p_1.t_1, p_1.t_1) \in \Theta_{X,Y}$ holds, or the field has been inherited from a public non-final class $q_0'.c_0'$. If $q_0'.c_0' = q_0.c_0$, then, similar to the **SUPER** case above, the claim holds due to **T3**. Otherwise, the field must be public (as we have not accessed it from a subclass). By **T1**, we have $(q_0.c_0 \vdash q_0'.c_0', q_0'.c_0') \in \Theta_{X,Y}$ and by **T2**, $(q_0.c_0 \vdash p_1.t_1, p_2.t_2) \in \Theta_{X,Y}$.
In the latter case, the field has not been defined in K (as X and Y supertype-closed) but in X resp. Y. As $k.c$ can not be a supertype of the (static) type of E' (as X supertype-closed), the field is accessible iff it is public (see Def. of accmember) and then the claim follows from **T2**.

Case **SET** or **CALL**: similar to **GET**.

(2) $\Rightarrow$ (1): Goes by induction on (the inductive definition of) $\Theta_{X,Y}$. We always give an appropriate construction of K, $k.c$, E and Γ.

Induction basis:

Case **T1**: $(q_0.c_0 \vdash p.t, p.t) \in \Theta_{X,Y}$. Let K then be:

package k; **abstract class** c **extends** $q_0.c_0$ {}

As E we have $(p.t)$**null** and as Γ we have (**this**:$k.c$).
As $q_0.c_0$ is public and not final, we can subclass c. Nothing prevents $q_0.c_0$ from being abstract however, thus our class k.c must be abstract as well for $\vdash KX$ to hold. Our E then types correctly in both KX and KY as rule **WCAST** holds because $p.t$ is public.

Case **T3** or **T5**: $(q_0.c_0 \vdash p_1.t_1, p_2.t_2) \in \Theta_{X,Y}$. Let K be:

package k; **abstract class** c **extends** $q_0.c_0$ {}

As E we have **this**.f or **this**.$m(\overline{\textbf{null}})$ and as Γ we have (**this**:$k.c$).
It should be clear that accmember holds whenever the field f is public or protected, as the class c is a direct subclass of $q_0.c_0$ and $q_0 \neq k$ (see rule **INHERITED-FIELD**). For the method call it is similar.

Induction step:

Case **T2** or **T4**:
The induction hypothesis ($(q_0.c_0 \vdash r_1.t_1, r_2.t_2) \in \Theta_{X,Y}$) gives us a construction of K, $k.c$, e and Γ. We can then just replace E by $E.f$ or $E.m(\overline{\textbf{null}})$. This new expression then typechecks for the following reasons. If $r_1.t_1$ and $r_2.t_2$ are public, then they are accessible (acctype). If the field f or the method m are public, they are also accessible (accmember). □

Additional explanations

In order to prove the direction (1) $\Rightarrow$ (2) of Lemma 2, let us first abstract the lemma to a simpler form. We abstract the all-quantified variables $q_0.c_0$, $p_1.t_1$ and $p_2.t_2$ into an abstract variable x. We then rearrange the existentially quantified variables in (1) such that $\exists E$ is the outermost quantification of (1) and consider the remaining part of (1) as a fixed predicate over x and E (which we write $P[x, E]$). We also consider (2) as a predicate $Q[x]$ over x. Our abstract version of Lemma 2 then has the form $\forall x : ((\exists E : P[x, E]) \rightarrow Q[x])$. Lemma 6 then shows that we can do the proof by induction on E.

Lemma 6. *The following first order formulas* $\forall x : ((\exists y : P[x, y]) \rightarrow Q[x])$ *and* $\forall y : (\forall x : (P[x, y] \rightarrow Q[x]))$ *are equivalent.*

Definition 5 (Equivalence class for types in the context)**.** This defines the (finite) set $\mathcal{T}_X^{\leq}$ which represents an equivalence class for all possible types in the context of X.

$\forall p.c \in C_X^{\mathbf{o}}, \forall \overline{p.i} \in \mathcal{I}_X$ where $\overline{p.i}$ pairwise distinct:

$(q.i', Q) \in \mathcal{T}_X^{\leq}$ if
- $Q =$ **package** q; **public interface** i' **extends** $\overline{p.i}$ {}
- $\vdash XQ$

$(q.c', Q) \in \mathcal{T}_X^{\leq}$ *if*
- $Q =$ **package** q; **public abstract class** c' **extends** $p.c$ **implements** $\overline{p.i}$ {}
- $\vdash XQ$

$(q.c'', Q) \in \mathcal{T}_X^{\leq}$ if
- $Q =$ **package** q; **public final class** c'' **extends** $p.c$ **implements** $\overline{p.i}$ { $\overline{M}$ }
- $\overline{M}$ are the implementations (with **null**-valued method bodies) of all the (abstract) methods from super-types
- $\vdash XQ$

where q is a package name not occurring in X, and the names $q.i'$, $q.c'$ and $q.c''$ are uniquely determined by $\overline{p.i}$ and $p.c$. This ensures that for a finite X, $\mathcal{T}_X^{\leq}$ is also finite.

A.3 Proof Sketch that Contextual Compatibility implies Syntactic Compatibility

Proof. By contrapositive.

A.3.1 Declarations

Most parts of this proof are also explained in Sect. 4. Assume the following syntactic rule does not hold:

(R1): Context K then has a class which declares a field of the type $q.t$. The type $q.t$ is then accessible (acctype) in X but not Y which violates rule **DEFN-C**.

(R2): There exists a class $q.c$ which is final in Y but not X. Context K then has an (abstract) class which declares $q.c$ as its super-class. Thus ($C4_{KY}$) does not hold anymore.

(R3): We can create a subinterface in K of $p.i$ which declares m with some type which is not T which violates (C12).

(R4): We can create a subclass $q.c'$ in K of $p.c$ which declares m with some type which is not T. Then $q.c'$ typechecks with KX but not KY (C14). Furthermore, accessibility can not be increased in Y, otherwise we can declare a method in $q.c'$ which overrides the method in $p.c$, but with weaker accessibility modifiers (C11). Overriding is always possible because of our additional typing rules allowing only public types in public interfaces. Overriding of final methods is prohibited. Regarding the abstractness constraint, imagine a non-abstract subclass which does not override the (non-abstract) method but all other abstract methods. This leads to a problem (C10) as the non-abstract subclass then has an abstract method when typechecking against Y.

A.3.2 Expressions

Assume rule (Rn) does not hold:

(R5) - (R6): Assume (R5) does not hold. From Lemma 2 we can construct a context K and an expression E

such that E types to $p_1.t_1$ resp. $p_2.t_2$. We then construct the expression E'. For (R5), $E' = E.f$. For (R6), $E' = E.m(\overline{(r.t)\textbf{null}})$. Then prove that if ϕ^*_Y does not hold, the typing rule * does not hold either in KY and construct K' by Lemma 4 and Lemma 5 such that $\vdash K'X$ but $\nvdash K'Y$.

(R7): Construct context as previously described with the expression $(p.t)E$.

(R8): An appropriate context is provided by $\mathcal{T}^{\leq}$.

(R9): Construct context with the expression **new** $p.c$.

(R10): Construct context as for Lemma 2, case **T3**, with the expression **this**$.f$.

(R11): Construct context as for Lemma 2, case **T5**, with **this**$.m(\overline{\textbf{null}})$.

(R12): Construct expressions $E_1.f$ and E_2 as for Lemma 2 and create new expression $E' = E_1.f = E_2$. Then proceed as in the previous proof cases.

□

A.4 Proof Sketch that Syntactic Compatibility implies Contextual Compatibility

Proof. By direct proof. The proof goes case by case over all possible context conditions / typing rules. *Proof schema:* Assume typing condition (Cn_{KX}) holds. Then show that (Cn_{KY}) must hold too due to syntactic compatibility.

A.4.1 Declarations

(C1) and (C4) follow from (R1) and fact, that K can only add new subtypes.

(C2) follows from (R2).

(C3) and (C5)-(C9) are local properties.

(C10)-(C13) follow from (R3) and (R4).

(C14) and (C15) follow from (R1) and the fact that it must hold for KX and Y.

$\mathit{TypeOncePerPackage}_{KY}$ and $\mathit{MemberOncePerType}_{KY}$ hold trivially

$\mathit{PackageOnce}_{KY}$ holds as $\mathit{PackageOnce}_{KX}$ holds and Y can not contain more package names than X (Def. of $X \lhd Y$).

Again we do not consider method bodies at first. All package definitions are then well-formed, if their type definitions are well-formed. By looking at the typing rules **COMP**, **DEFN-P**, **DEFN-C**, **DEFN-I**, **METH-ABS** and **METH**, this is the case if $\text{acctype}_{KX}(\overline{q.t}, p) \rightarrow \text{acctype}_{KY}(\overline{q.t}, p)$ which follows from (R1). It remains to check the expressions in method bodies.

A.4.2 Expressions

We can check expressions case by case (and assume all other expressions to be **null**-valued). The following remains to be proven: For all codebases K with only **null**-valued method bodies, $k.c \in C_K$, $\Gamma = (\textbf{this}{:}k.c, \overline{v{:}q.t})$ such that $\text{acctype}_X(\overline{q.t}, k)$: $KX, k.c, \Gamma \vdash E : p_1.t_{1\perp} \rightarrow KY, k.c, \Gamma \vdash E : p_2.t_{2\perp}$. The proof goes by induction on the structure of E.

Induction basis:

Case **VAR**: v where $(v{:}q.t) \in \Gamma$: if $q.t \in \mathcal{T}_X$ then follows from (R1) else trivial.

Case **NEW**: $E = \textbf{new}\ q.c()$. If $q.c \in C_X$ then by (R9) else trivial.

Case **NULL**: trivial.

Induction step: We assume that $KX, k.c, \Gamma \vdash E : p_1.t_{1\perp}$ holds and prove that $KY, k.c, \Gamma \vdash E : p_2.t_{2\perp}$ holds too. As E is well-typed in KX, we know that for all sub-expressions E_0 of E, the following must hold: $KX, k.c, \Gamma \vdash E_0 : p'_1.t'_{1\perp}$. By I.H., $KY, k.c, \Gamma \vdash E_0 : p'_2.t'_{2\perp}$ follows. Lemma 3 then states that exactly one of the following holds:

(1) $p'_1.t'_{1\perp} = \perp \wedge p'_2.t'_{2\perp} = \perp$
(2) $p'_1.t'_{1\perp} \in (C_K \cup \mathcal{I}_K) \wedge p'_1.t'_{1\perp} = p'_2.t'_{2\perp}$
(3) $p'_1.t'_{1\perp} \in \mathcal{T}_X \wedge p'_2.t'_{2\perp} \in \mathcal{T}_Y$

Case **GET**: $E = E_0.f$. By rule **GET**, we know that $q'_1.t'_{1\perp} \neq \perp$. We thus distinguish two cases:

- Assume (2) holds. If the field f is defined in K, the claim follows trivially. Otherwise, it follows from (R10).
- Assume (3) holds. By Lemma 2, $(k.c \vdash p'_1.t'_1, p'_2.t'_2) \in \Theta_{X,Y}$. By (R5), the claim follows.

Case **SET**: $E = E_0.f = E_v$ similar to **GET**, but additionally by (R12).

Case **CALL**: $E = E_0.m(E_1, ...E_n)$ similar to **GET**, but (R11) and (R6) instead of (R10) and (R5).

Case **CAST**: $E = (p.t)E_0$. If $p.t \in \mathcal{T}_X$, we know that $\text{acctype}_{KX}(p.t, k)$ iff $\text{public}_X(p.t)$. By (R1), $p.t \in \mathcal{T}_Y$ and $\text{public}_Y(p.t)$. Thus $\text{acctype}_{KY}(p.t, k)$. If $p.t \notin \mathcal{T}_X$, the condition $\text{acctype}_{KX}(p.t, k) \rightarrow \text{acctype}_{KY}(p.t, k)$ follows trivially.
We distinguish our usual three cases:

- Assume (1) holds. As $\perp \leq_{KY} p.t$ holds by Def. of $\leq_{KY}$, trivial (**WCAST**).
- Assume (2) holds. If $p.t \in \mathcal{T}_X$, the claim follows from (R1), (R2) and (R12). Otherwise, trivial.
- Assume (3) holds. If $p.t \in \mathcal{T}_X$, then the claim follows from (R7). Otherwise, it follows from (R8).

Case **SUPER**: $E = \textbf{super}.m(E_1, ...E_n)$ similar to **CALL** but additionally because of (R4) (the abstract part).

We also have to check whether the types of method bodies are subtypes of the return type of the method signatures. This is covered by (R12). □