A Type System for Checking Specialization of Packages in Object-Oriented Programming[*]

Ferruccio Damiani
Dipartimento di Informatica
Università di Torino
damiani@di.unito.it

Arnd Poetzsch-Heffter
University of Kaiserslautern,
Germany
poetzsch@cs.uni-kl.de

Yannick Welsch
University of Kaiserslautern,
Germany
welsch@cs.uni-kl.de

ABSTRACT

Large object-oriented software systems are usually structured using modules or packages to enable large-scale development using clean interfaces that promote encapsulation and information hiding. However, in most OO languages, package interfaces (or signatures) are only *implicitly* defined.

In this paper, we propose explicit package signatures that allow for modularly type-checking packages. We show how the signatures can be derived from packages and define a checkable specialization relation for package signatures. As main contribution, we show that if the package signatures of a new component version C_{new} specialize the signatures of the old version C_{old}, then C_{new} type-checks in *all* contexts in which C_{old} type-checks. That is, we extend checking of interface types to the level of packages.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*

Keywords

Java, Modularity, Package signature, Package specialization, Type-checking

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* in [6]), 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., [5]) and special tools (e.g., [7]) to check compatibility aspects. To be able to give precise guarantees, the checks need to be formally based on the language definition.

Our focus is on object-oriented APIs. In particular, we are interested in larger APIs consisting of several, often mutually recursive types together with their methods. We assume that the types of an API are contained in a package. For the technical developments in this paper, we build on Java packages. However, our results are transferable to other OO languages.

Interfaces and encapsulation are fundamental object-oriented concepts [17]. However, in most OO languages, package interfaces (or signatures) are only *implicitly* defined. A package interface provides public types to create objects, access public fields, and call methods on the objects (*caller interface*). Furthermore, a package interface provides the possibility to extend the functionality of the provided types via inheritance (*implementor interface*). Changes of a package 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. [5]). To distinguish caller and implementor interface, OO languages support different access modifiers (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, we say that the new version is a *(syntactic) specialization* of the old version, i.e., we use a syntactic notion of specialization. Nevertheless, checking this property for packages is a non-trivial 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 with the mechanisms to control encapsulation.
- ***Modularity:*** Specialization 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/type level (programming in the small). Type systems allow compilers to check that type-related programming errors are avoided, e.g., co-/contra-variance typing and appropriate choice of access modifiers in overriding methods. However, as we will show, interface support on the

[*]The authors of this paper are listed in alphabetical order. Work partially supported by the EU-project *FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models*, by the German-Italian University Centre (Vigoni program), and by MIUR (PRIN 2008 DISCO).

SAC'12 March 25-29, 2012, Riva del Garda, Italy.
Copyright 2011 ACM 978-1-4503-0857-1/12/03 ...$10.00.

package level (programming in the large) still needs improvement for OO languages. Future package constructs should enable compilers to check for package specialization in the same way as today's compilers check for nominal or structural subtyping.

Example.

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

```
package util;
public class ArrayList {...}
public class LinkedList {...}
```

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

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

If client code compiles against the old version of the library, can we guarantee that it still compiles against the new version? The challenge is to check specialization for packages for all possible client contexts.

Component specialization.

To further clarify of our terminology, we have to say what precisely a component or API is and what we mean by "some code compiles against some component". For the latter aspect, one has to make the distinction between *observability* [9, §7.3 and §7.4.3] (sometimes also called *visibility*[1]) and *accessibility* of types [2]. 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., [18]) 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.

Our central notion concerning code is that of a codebase. A *codebase* is a sequence of packages where a *package* has a name and consists of a sequence of nominal type declarations (cf. Sect. 2). We assume that packages are sealed (cf. [10], Sect. 2), meaning that once a package is defined no new class definitions can be added to the package. Adding packages to a codebase or joining codebases is denoted by juxtaposition, e.g., the codebase $\overline{\mathsf{PD}}\,\overline{\mathsf{PD}}'$ is composed of codebase $\overline{\mathsf{PD}}$ and codebase $\overline{\mathsf{PD}}'$.

A codebase $\overline{\mathsf{PD}}$ might contain usages of packages, classes, or methods that are not defined in $\overline{\mathsf{PD}}$. That is, in general, a codebase $\overline{\mathsf{PD}}$ has an import and export interface where the export interface contains the elements defined in $\overline{\mathsf{PD}}$. To gradually approach our goals, we first concentrate on definition-complete codebases, i.e., codebases containing the definitions of all used names. In Section 4.2, we lift this restriction.

Definition 1 (Component). *We say that a codebase* $\overline{\mathsf{PD}}$ *is a* component[2] *to mean that: (i) all names used in* $\overline{\mathsf{PD}}$ *are defined in* $\overline{\mathsf{PD}}$ *or are predefined (like class* Object*); and (ii)* $\overline{\mathsf{PD}}$ *is well typed according to the typing rules of the language. We write* $\overline{\mathsf{PD}}$ OK *to express that the codebase* $\overline{\mathsf{PD}}$ *is a component.*

Now, we can define specialization as a relation on two components (in [12], this relation is called compatibility[3]).

Definition 2 (Component specialization). *We say that* the component $\overline{\mathsf{PD}}''$ specializes a component $\overline{\mathsf{PD}}'$ *(written* $\overline{\mathsf{PD}}'' \leq \overline{\mathsf{PD}}'$*, for short) to mean that, for any codebase* $\overline{\mathsf{PD}}$, $\overline{\mathsf{PD}}\,\overline{\mathsf{PD}}'$ OK *implies* $\overline{\mathsf{PD}}\,\overline{\mathsf{PD}}''$ OK.

We call $\overline{\mathsf{PD}}$ a *(program) context*. It is important to note that the definition of specialization does not allow for automatic checking that a component $\overline{\mathsf{PD}}'$ specializes a component $\overline{\mathsf{PD}}''$, as it quantifies over an infinite set of contexts.[4] Specialization is a preorder relation, that is, it is reflexive and transitive, but not antisymmetric.

Goals and Outline.

The central goal is an effective technique for checking that a component specializes another one. After the introduction of our core language (Sect. 2), we formally define interfaces for packages, so-called *(package) signatures*, and a checkable specialization relation on such signatures (Sect. 3). In Sect. 4, we show that a component $\overline{\mathsf{PD}}''$ specializes a component $\overline{\mathsf{PD}}'$ if and only if the corresponding signature $\overline{\mathsf{PS}}''$ of $\overline{\mathsf{PD}}''$ specializes $\overline{\mathsf{PS}}'$ of $\overline{\mathsf{PD}}'$ (and this is checkable!). Then, we generalize this result to codebases, i.e., give up our assumption concerning definition-completeness. We conclude by discussing related and future work. For missing details in the formalization and proof sketches, we refer to the technical report [4].

2. A CALCULUS FOR JAVA PACKAGES

In this section we introduce the syntax and the typing of Imperative Featherweight Java with Packages (IFJP for short), a minimal core calculus for Java packages which builds on an imperative variant of FJ [11]. In IFJP fields are private, methods are non-private (that is, either public, or protected or with package accessibility), and each class has an implicit constructor that (like the implicit default constructor in Java) initializes all the fields to **null**. Restricting to private fields, non-private methods, and default constructors simplifies the formalization of Java packages without dropping interesting features.

2.1 Syntax

The syntax IFJP is given in Fig. 1. We use similar notations as FJ [11]. For instance: "$\overline{\mathsf{e}}$" denotes the possibly empty sequence "$\mathsf{e}_1, \ldots, \mathsf{e}_n$" and "$\overline{\mathsf{P.C\ f}}$;" stands for "$\mathsf{P}_1.\mathsf{C}_1\ \mathsf{f}_1; \ldots \mathsf{P}_n.\mathsf{C}_n\ \mathsf{f}_n$;". The empty sequence is denoted by "•" and the length of a sequence $\overline{\mathsf{e}}$ is denoted by $|\overline{\mathsf{e}}|$. We write $\ddot{\mathsf{e}}$ to denote a sequence that either is empty or consists of the single element e. Sequences of named elements (package definitions, class definitions, field definitions, etc.) are assumed to not contain elements with the same name. A codebase is a sequence of package definitions.

[1] Not to be confused with the meaning of visibility in the setting of declaration scopes for programming languages [9, §6.3.1].

[2] Inspired by [12].

[3] We do not use this term, because many readers consider "compatibility" not suitable to name a relation that may be not symmetric.

[4] An even harder problem is to check for behavioral specialization/-compatiblility; cf. [22].

$$\begin{array}{lll}
\mathsf{PD} & ::= \textbf{package}\ \mathsf{P};\ \overline{\mathsf{CD}} & \text{package definitions}\\
\mathsf{CD} & ::= \mathsf{CQ}\ \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{P.C}\ \{\ \overline{\mathsf{FD}};\ \overline{\mathsf{MD}}\ \} & \text{class definitions}\\
\mathsf{CQ} & ::= \textbf{public} \mid \bullet & \text{class qualifiers}\\
\mathsf{FD} & ::= \textbf{private}\ \mathsf{P.C\ f} & \text{field definitions}\\
\mathsf{MD} & ::= \mathsf{MH}\ \{\ \textbf{return}\ \mathsf{e};\ \} & \text{method definitions}\\
\mathsf{MH} & ::= \mathsf{MQ\ P.C\ m}\ (\overline{\mathsf{P.C\ x}}) & \text{method headers}\\
\mathsf{MQ} & ::= \textbf{public} \mid \textbf{protected} \mid \bullet & \text{method qualifiers}\\
\mathsf{e} & ::= \mathsf{x} \mid \textbf{null} \mid \mathsf{e.f} \mid \mathsf{e.f} = \mathsf{e} \mid \mathsf{e.m}(\overline{\mathsf{e}}) \mid & \text{expressions}\\
 & \quad \textbf{new}\ \mathsf{P.C()} \mid \mathsf{(P.C)e} &
\end{array}$$

Figure 1: IFJP **Syntax where P, C, f, m and x denote package, class, field, method and variable names, respectively; this is a variable name**

2.1.1 *Conventions on sequences of named elements*

Given a sequence of named elements $\overline{\mathsf{NE}}$, we write *names*($\overline{\mathsf{NE}}$) to denote the names of the elements of $\overline{\mathsf{NE}}$. The subsequence of the elements of $\overline{\mathsf{NE}}$ with names $\overline{\mathsf{n}}$ is denoted by *choose*($\overline{\mathsf{n}}, \overline{\mathsf{NE}}$). Following [11], we use a set-based notation for operators over sequences of named elements. For instance, $\mathsf{MD} = \mathsf{MQ\ I\ m}\,(\overline{\mathsf{I\,x}})\{\textbf{return}\ \mathsf{e};\} \in \overline{\mathsf{MD}}$ means that the method definition MD occurs in $\overline{\mathsf{MD}}$. In the union, intersection and difference of sequences, denoted by $\overline{\mathsf{NE}} \cup \overline{\mathsf{NE}}'$, $\overline{\mathsf{NE}} \cap \overline{\mathsf{NE}}'$ and $\overline{\mathsf{NE}} \setminus \overline{\mathsf{NE}}'$, respectively, it is assumed that if $\mathsf{n} \in$ *names*($\overline{\mathsf{NE}}$) and $\mathsf{n} \in$ *names*($\overline{\mathsf{NE}}'$), then *choose*($\mathsf{n}, \overline{\mathsf{NE}}$) = *choose*($\mathsf{n}, \overline{\mathsf{NE}}'$). In the disjoint union of sequences, denoted by their juxtaposition $\overline{\mathsf{NE}}\ \overline{\mathsf{NE}}'$, it is assumed that *names*($\overline{\mathsf{NE}}$) ∩ *names*($\overline{\mathsf{NE}}$) = ∅. Given a sequence of $n \geq 1$ named elements $\overline{\mathsf{NE}} = \mathsf{NE}_1 \cdots \mathsf{NE}_n$ we write $\overline{\mathsf{NE}}_{\setminus i}$ as short for $\overline{\mathsf{NE}} \setminus \mathsf{NE}_i$, where $i \in 1..n$.

2.1.2 *Sanity Conditions for Packages and Codebases*

We write *classes*($\overline{\mathsf{PD}}$) and *pubClasses*($\overline{\mathsf{PD}}$) to denote the set of (fully qualified) class identifiers for which there is a declaration or a public declaration in the codebase $\overline{\mathsf{PD}}$, respectively. We write $\mathit{pubClasses}^{\top}(\overline{\mathsf{PD}})$ to denote $\mathit{pubClasses}(\overline{\mathsf{PD}}) \cup \{\mathsf{lang.Object}\}$. We write $<:_{\overline{\mathsf{PD}}}$ to denote the immediate subclass relation given by the **extends** clauses of the classes declared in $\overline{\mathsf{PD}}$. We write $\mathsf{MQ} \leq \mathsf{MQ}'$ to mean that method qualifier MQ is less restrictive than or the same as method qualifier MQ′, that is, ≤ is the transitive and reflexive closure of the linear order: **public**, **protected**, •.

Codebases $\overline{\mathsf{PD}}$ have to satisfy the following sanity conditions:

1. If a class name P.C occurs within a class definition that is inside a package declaration with name P, then this class C must also be defined in this package declaration.

2. There are no cycles in the hierarchy defined by the subtyping relation $<:_{\mathsf{PD}}$.

3. In case of overriding, if a method with the same name is declared in a superclass, then this superclass method (*i*) must have the same parameters and return types;[5] and (*ii*) must have a more restrictive or the same qualifier.

4. The header of public and protected methods (defined or inherited) in public classes must only contain public types.

The fourth condition guarantees that we can always create a class in another package which extends a given public class (as we can implement all the methods), which does not hold in Java. The C# language specification [8] defines similar restrictions. Sect. 10.5.4 of [8], 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. The constraints further ensure that at each call site the method parameter and return types are types which are accessible to the calling context.

In the following, we assume that the codebases we consider always satisfy these sanity conditions.

2.2 Standard Type-Checking

In this section, we introduce a type system for IFJP components which follows the Java type system. The typing judgement for packages, $\overline{\mathsf{PD}} \vdash \mathsf{PD}$ (to be read "package PD is well typed with respect to the codebase $\overline{\mathsf{PD}}$"), is such that $\overline{\mathsf{PD}}$ must contain all the packages transitively used by PD.

Thus, the meaning of the judgment $\overline{\mathsf{PD}}$ OK ("$\overline{\mathsf{PD}}$ is a component"), introduced in Def. 1, can be formalized by the rule:

$$\textsc{std-component}\quad \frac{\overline{\mathsf{PD}} = \mathsf{PD}_1 \cdots \mathsf{PD}_n \qquad \forall i \in 1..n.\quad \overline{\mathsf{PD}}_{\setminus i} \vdash \mathsf{PD}_i}{\overline{\mathsf{PD}}\ \mathrm{OK}}$$

Note that, although the package definition PD_i does not occur in the left side of the typing judgement $\overline{\mathsf{PD}}_{\setminus i} \vdash \mathsf{PD}_i$, the associated typing rules ensure that all the definitions from PD_i are accessible while type-checking elements from PD_i. Every IFJP component is literally a well-typed Java program.

3. PACKAGE SIGNATURES

Package signatures are a formal representation of the relevant interface information of packages. In particular, they only contain information about public or protected program elements. Package-local methods and types are discarded. We introduce the syntax of package signatures, show how they can be used for type checking of components, and define a specialization relation for them.

3.1 Syntax of Signatures

The syntax of *package signatures* is presented in Fig. 2. A *codebase signature* is a sequence of package signatures.

$$\begin{array}{lll}
\mathsf{PS} & ::= \textbf{package}\ \mathsf{P};\ \overline{\mathsf{CS}} & \text{package signatures}\\
\mathsf{CS} & ::= \textbf{public class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{P.C}\ \{\ \overline{\mathsf{NS}};\ \} & \text{(non-local) class signatures}\\
\mathsf{NS} & ::= \mathsf{NQ\ P.C\ m}\ (\overline{\mathsf{P.C}}) & \text{(non-local) method signatures}\\
\mathsf{NQ} & ::= \textbf{public} \mid \textbf{protected} & \text{(non-local) method qualifiers}
\end{array}$$

Figure 2: IFJP**: Signatures**

The signature *psig*(PD) of a package PD can be automatically extracted from PD. Package signatures are sufficient for type checking of code that uses PD and can be used for automatic checking of specialization (see Sect. 4). This even works for codebases with mutually dependent packages. Given a codebase $\overline{\mathsf{PD}} = \mathsf{PD}_1 \cdots \mathsf{PD}_n$, we will write *psig*($\overline{\mathsf{PD}}$) to denote the codebase signature *psig*(PD_1) ⋯ *psig*(PD_n). The signature extraction function *psig* is defined in Fig. 3 and works on a per-package basis. It uses the auxiliary function *csig* to extract the signature from classes which in turn uses the auxiliary functions *pubSupClass* and *nonLocalMethods* such that:

- $\mathit{pubSupClass}_{\mathsf{PD}}(\mathsf{P}'.\mathsf{C}')$, denotes the smallest public superclass of the class P′.C′, which can be found by exploiting the immediate subclass relation $<:_{\mathsf{PD}}$. Namely: $\mathit{pubSupClass}_{\mathsf{PD}}(\mathsf{P}'.\mathsf{C}')$ is P′.C′ if either $\mathsf{P}' \neq$ *names*(PD)

[5] For simplicity, as in FJ [11], method overloading and covariant subtyping of the return type in method overriding are not considered.

or $\mathsf{P'.C'} \in \mathit{pubClasses}^{\top}(\mathsf{PD})$; and $\mathit{pubSupClass}_{\mathsf{PD}}(\mathsf{P'.C'})$ is $\mathit{pubSupClass}_{\mathsf{PD}}(\mathsf{P''.C''})$, where $\mathsf{P'.C'} <:_{\mathsf{PD}} \mathsf{P''.C''}$, otherwise.

- $\mathit{nonLocalMethods}_{\mathsf{PD}}(\mathsf{P.C})$, where $\mathsf{P} = \mathit{names}(\mathsf{PD})$, denotes the signatures of the public or protected methods that the class $\mathsf{P.C}$ either defines or inherits from its superclasses that are both local to the package PD and strict subclasses of $\mathit{pubSupClass}_{\mathsf{PD}}(\mathsf{P.C})$.

As an example for signature extraction, consider the following package definition PD:

```
package p;
public class A {
  public void m() { ... } }
class B extends p.A {
  void n() { ... }
  public void o() { ... }
}
public class C extends p.B {
  public void m() { ... }
}
public class D extends q.E {
  protected void p() { ... }
}
```

Extraction yields the following package signature $\mathit{psig}(\mathsf{PD})$:

```
package p;
public class A {
  public void m();
}
public class C extends p.A {
  public void o();
}
public class D extends q.E {
  protected void p();
}
```

The package-local class B does not appear in the signature. Using *pubSupClass*, the superclass of C becomes A in the signature. The superclass of D remains E as it is from another package (and must thus be a public class). The method m does not appear in C, as this information is already in the signature of A (see *nonLocalMethods*).

$$\frac{\begin{array}{c}\mathsf{P_0.C_0} = \mathit{pubSupClass}_{\mathsf{PD}}(\mathsf{P'.C'}) \\ \mathsf{P} = \mathit{names}(\mathsf{PD}) \qquad \overline{\mathsf{NS}} = \mathit{nonLocalMethods}_{\mathsf{PD}}(\mathsf{P.C}) \\ \dddot{\mathsf{CS}} = \begin{cases} \textbf{public class } \mathsf{C} \textbf{ extends } \mathsf{P_0.C_0}\ \{\ \overline{\mathsf{NS}}\ \} & \text{if } \mathsf{CQ} = \textbf{public} \\ \bullet & \text{otherwise} \end{cases}\end{array}}{\mathit{csig}_{\mathsf{PD}}(\mathsf{CQ}\ \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{P'.C'}\ \{\ \overline{\mathsf{FD}};\ \overline{\mathsf{MD}}\ \}) = \dddot{\mathsf{CS}}}$$

$$\frac{\begin{array}{c}\mathsf{PD}\ =\ \textbf{package}\ \mathsf{P};\ \mathsf{CD}_1 \cdots \mathsf{CD}_n \\ \mathit{csig}_{\mathsf{PD}}(\mathsf{CD}_1) = \dddot{\mathsf{CS}}_1\ \cdots\ \mathit{csig}_{\mathsf{PD}}(\mathsf{CD}_n) = \dddot{\mathsf{CS}}_n\end{array}}{\mathit{psig}(\mathsf{PD}) = \textbf{package}\ \mathsf{P};\ \dddot{\mathsf{CS}}_1 \cdots \dddot{\mathsf{CS}}_n}$$

Figure 3: IFJP**: Signature extraction functions *csig* and *psig***

Similarly as for codebases, we write $\mathit{pubClasses}(\overline{\mathsf{PS}})$ to denote the set of (fully qualified) class identifiers for which there is a declaration in the codebase signature $\overline{\mathsf{PS}}$. We write $\mathit{pubClasses}^{\top}(\overline{\mathsf{PS}})$ to denote $\mathit{pubClasses}(\overline{\mathsf{PS}}) \cup \{\mathsf{lang.Object}\}$. We write $<:_{\overline{\mathsf{PS}}}$ to denote the immediate subclass relation given by the **extends** clauses of the classes declared in $\overline{\mathsf{PS}}$.

We assume similar sanity conditions for codebase signatures than for codebases.

3.2 Signature-Based Type-Checking

As a first steps towards our main goal, we show that for type-checking a package PD it is sufficient to know the signatures of the packages used by PD. We call this *signature-based type-checking.* The corresponding typing judgement is $\overline{\mathsf{PS}} \vdash \mathsf{PD}$, to be read "package PD is well typed with respect to the codebase signature $\overline{\mathsf{PS}}$". The typing rules for this judgement, which mimic the standard typing rules, can be defined in a straightforward way.

We say that $\overline{\mathsf{PD}}$ is a component according to signature-base type-checking if $\overline{\mathsf{PD}}$ SIGOK can be derived by the following rule:

$$\textbf{SIG-COMPONENT}\quad \frac{\overline{\mathsf{PD}} = \mathsf{PD}_1 \cdots \mathsf{PD}_n \qquad \mathit{psig}(\overline{\mathsf{PD}}) = \overline{\mathsf{PS}} \qquad \forall i \in 1..n.\quad \overline{\mathsf{PS}}_{\setminus i} \vdash \mathsf{PD}_i}{\overline{\mathsf{PD}}\ \textsc{SigOK}}$$

The following theorem states that signature-based type-checking is equivalent to standard type-checking (cf. Sect. 2.2).

THEOREM 1 (CORRECTNESS AND COMPLETENESS OF SIGOK). $\overline{\mathsf{PD}}$ OK *if and only if* $\overline{\mathsf{PD}}$ SIGOK.

We now introduce a conservative extension of the signature-based type system for IFJP components by replacing the typing judgement for components, $\overline{\mathsf{PD}}$ SIGOK, with a typing judgement for codebases, $\overline{\mathsf{PS}} \vdash \overline{\mathsf{PD}}$ SIGOK ("codebase $\overline{\mathsf{PD}}$ is well-typed modulo codebase signature $\overline{\mathsf{PS}}$"). This allows us to modularly type-check codebases w.r.t. to the signature of other codebases.

The meaning of the typing judgment for codebases is formalized by the following rule:

$$\textbf{SIG-CODEBASE}\quad \frac{\overline{\mathsf{PD}} = \mathsf{PD}_1 \cdots \mathsf{PD}_n \qquad \mathit{psig}(\overline{\mathsf{PD}}) = \overline{\mathsf{PS}} \qquad \forall i \in 1..n.\quad \overline{\mathsf{PS}}_{\setminus i}\,\overline{\mathsf{PS}}' \vdash \mathsf{PD}_i}{\overline{\mathsf{PS}}' \vdash \overline{\mathsf{PD}}\ \textsc{SigOK}}$$

The following theorem states that signature-based typing for codebases is a conservative extension of signature-based typing for components. The proof is straightforward, by inspection of rules **SIG-CODEBASE** and **SIG-COMPONENT**.

THEOREM 2 (CONSERVATIVE EXTENSION OF SIGOK). $\bullet \vdash \overline{\mathsf{PD}}$ SIGOK *if and only if* $\overline{\mathsf{PD}}$ SIGOK.

After Theorem 2, we can safely write $\overline{\mathsf{PD}}$ SIGOK as short of $\bullet \vdash \overline{\mathsf{PD}}$ SIGOK.

The following proposition states that signature-based type-checking is indeed modular. Namely, if all the packages of a codebase $\overline{\mathsf{PD}}$ type-check modulo a codebase signature $\overline{\mathsf{PS}}'$, then for every component $\overline{\mathsf{PD}}'$ that has the signature $\overline{\mathsf{PS}}'$ it is guaranteed that $\overline{\mathsf{PD}}'\overline{\mathsf{PD}}$ is a component.

PROPOSITION 1 (MODULARITY). *Let* $\overline{\mathsf{PS}}' \vdash \overline{\mathsf{PD}}$ SIGOK. *If* $\overline{\mathsf{PD}}'$ SIGOK *and* $\mathit{psig}(\overline{\mathsf{PD}}') = \overline{\mathsf{PS}}'$, *then* $\overline{\mathsf{PD}}'\overline{\mathsf{PD}}$ SIGOK.

3.3 Signature Specialization

The following definition introduces a computable relation between codebase signatures. It provides an *effective* means to check component specialization.

DEFINITION 3 (SIGNATURE SPECIALIZATION). *We say that* the codebase signature $\overline{\mathsf{PS}}$ specializes the codebase signature $\overline{\mathsf{PS}}'$ *to mean that the judgement* $\overline{\mathsf{PS}} \leq \overline{\mathsf{PS}}'$ *can be derived by the rules in Fig. 4.*

As an example for the signature specialization, let us consider two package signatures such that the second one specializes the first one:

```
// first package signature
package p;
public class C { public void m(); }
public class D extends p.C {}

// second package signature
package p;
public class C { public void m(); }
public class E extends p.C {}
public class D extends p.E {}
public class F extends p.C { public void n(); }
```

We can see that classes (i.e., E and F) can be added as long as the subtype hierarchy of the existing types is not touched.

SPC-CLASS-SIG

$$\frac{\mathit{methods}_{\overline{\mathsf{PS}}}(\mathsf{P.C}) = \mathit{methods}_{\overline{\mathsf{PS}}'}(\mathsf{P.C})}{\textbf{public class } \mathsf{C}\ \cdots \le^{\mathsf{P}}_{(\overline{\mathsf{PS}},\overline{\mathsf{PS}}')} \textbf{ public class } \mathsf{C}\ \cdots}$$

SPC-PACKAGE-SIG

$$\frac{\forall\, i \in 1..n. \quad \mathsf{CS}_i \le^{\mathsf{P}}_{(\overline{\mathsf{PS}},\overline{\mathsf{PS}}')} \mathsf{CS}'_i}{\textbf{package } \mathsf{P};\ \mathsf{CS}_1 \cdots \mathsf{CS}_n\ \overline{\mathsf{CS}} \le_{(\overline{\mathsf{PS}},\overline{\mathsf{PS}}')} \textbf{package } \mathsf{P};\ \mathsf{CS}'_1 \cdots \mathsf{CS}'_n}$$

SPC-CODEBASE-SIG

$$\frac{\le:_{\overline{\mathsf{PS}}'} \subseteq \le:_{\overline{\mathsf{PS}}} \qquad \overline{\mathsf{PS}} = \mathsf{PS}_1 \cdots \mathsf{PS}_n \qquad \overline{\mathsf{PS}}' = \mathsf{PS}'_1 \cdots \mathsf{PS}'_n \qquad \forall\, i \in 1..n. \quad \mathsf{PS}_i \le_{(\overline{\mathsf{PS}},\overline{\mathsf{PS}}')} \mathsf{PS}'_i}{\overline{\mathsf{PS}} \le \overline{\mathsf{PS}}'}$$

Figure 4: IFJP**: Rules for signature specialization**

The following proposition states a key property of the codebase signature specialization relation, namely that typeability is preserved if the context is specialized.

PROPOSITION 2 (PREMISE SPECIALIZATION). *Let* $\overline{\mathsf{PS}}' \vdash \overline{\mathsf{PD}}$ SIGOK. *If* $\overline{\mathsf{PS}}'' \le \overline{\mathsf{PS}}'$ *and there exists* $\overline{\mathsf{PD}}''$ *such that* $\overline{\mathsf{PD}}''$ SIGOK *and* $\mathit{psig}(\overline{\mathsf{PD}}'') = \overline{\mathsf{PS}}''$, *then* $\overline{\mathsf{PS}}'' \vdash \overline{\mathsf{PD}}$ SIGOK.

Note that premise specialization enhances the modularity of signature-based type-checking by making it possible to replace the requirement $\mathit{psig}(\overline{\mathsf{PD}}') = \overline{\mathsf{PS}}'$ (in the statement of Proposition 1) with the more liberal requirement $\mathit{psig}(\overline{\mathsf{PD}}') \le \overline{\mathsf{PS}}'$.

4. CHECKING SPECIALIZATION

This section presents the main results of the paper. Theorem 3 states that component $\overline{\mathsf{PD}}'$ specializes $\overline{\mathsf{PD}}''$ if $\mathit{psig}(\overline{\mathsf{PD}}')$ specializes $\mathit{psig}(\overline{\mathsf{PD}}'')$. Theorem 4 generalizes the result to codebases, i.e., to sequences of packages $\overline{\mathsf{PD}}$ that might import definitions from outside $\overline{\mathsf{PD}}$.

4.1 Checking Component Specialization

The following theorem states reduces checking of component specialization (cf. Def. 2), which is *not effectively* computable, to signature specialization. The latter is *effectively checkable* using the rules in Fig. 4.

THEOREM 3 (COMPONENT SPECIALIZATION CHECKING). *Let* $\overline{\mathsf{PD}}'$ SIGOK, $\overline{\mathsf{PD}}''$ SIGOK, $\mathit{psig}(\overline{\mathsf{PD}}') = \overline{\mathsf{PS}}'$ *and* $\mathit{psig}(\overline{\mathsf{PD}}'') = \overline{\mathsf{PS}}''$. *Then:* $\overline{\mathsf{PS}}'' \le \overline{\mathsf{PS}}'$ *implies* $\overline{\mathsf{PD}}'' \le \overline{\mathsf{PD}}'$.

4.2 Checking Codebase Specialization

In this section we generalize the component specialization relation (Def. 2) and the component specialization checking result (Theorem 3) to codebases.

For example, we may want to evolve a codebase that uses library code, e.g. ArrayList from the java.util package. We may thus rely on the codebase signature $\overline{\mathsf{PS}}$ that exactly contains the package java.util with the ArrayList class. Codebase specialization then means that if we link our codebase to an implementation that has a signature that syntactically specializes $\overline{\mathsf{PS}}$ (i.e., an implementation of the java.util package, which may contain many more classes than ArrayList), then every codebase that type-checks with our old codebase and the library also checks with our new codebase and the library.

DEFINITION 4 (CODEBASE SPECIALIZATION). *We say that* the codebase $\overline{\mathsf{PD}}''$ specializes the codebase $\overline{\mathsf{PD}}'$ modulo the codebase signature $\overline{\mathsf{PS}}$ *(written* $\overline{\mathsf{PD}}'' \le_{\overline{\mathsf{PS}}} \overline{\mathsf{PD}}'$*, for short) to mean that: for every codebase* $\overline{\mathsf{PD}}$ *such that* $\mathit{psig}(\overline{\mathsf{PD}}) \le \overline{\mathsf{PS}}$, $\overline{\mathsf{PD}}\,\overline{\mathsf{PD}}'$ OK *and* $\overline{\mathsf{PD}}\,\overline{\mathsf{PD}}''$ OK, *it holds that* $\overline{\mathsf{PD}}\,\overline{\mathsf{PD}}'' \le \overline{\mathsf{PD}}\,\overline{\mathsf{PD}}'$.

Note that component specialization (Def. 2) is codebase specialization modulo the empty codebase signature.

The following theorem states that signature-based type-checking provides a(n effective) mean to check codebase specialization modulo a codebase signature (cf. Def. 4).

THEOREM 4 (CODEBASE SPECIALIZATION CHECKING). *Let* $\overline{\mathsf{PS}} \vdash \overline{\mathsf{PD}}'$ SIGOK , $\overline{\mathsf{PS}} \vdash \overline{\mathsf{PD}}''$ SIGOK , $\mathit{psig}(\overline{\mathsf{PD}}') = \overline{\mathsf{PS}}'$ *and* $\mathit{psig}(\overline{\mathsf{PD}}'') = \overline{\mathsf{PS}}''$. *Then:* $\overline{\mathsf{PS}}'' \le \overline{\mathsf{PS}}'$ *implies* $\overline{\mathsf{PD}}'' \le_{\overline{\mathsf{PS}}} \overline{\mathsf{PD}}'$.

4.3 Discussion

The notion of specialization investigated above is canonical in that it is based on all possible contexts. For practical purposes, however, less restrictive definitions can be helpful. For example, according to the notion of component specialization (Def. 2) a specialized component may not introduce new packages. The following definition shows how this condition can be relaxed by restricting the set of possible contexts.

DEFINITION 5 (COMP. SPEC. UP TO FRESHNESS). *We say that* the component $\overline{\mathsf{PD}}'$ specializes a component $\overline{\mathsf{PD}}''$ up to freshness of the package names $\overline{\mathsf{P}}$ *(written* $\overline{\mathsf{PD}}' \le^{\overline{\mathsf{P}}} \overline{\mathsf{PD}}''$*, for short) to mean that,*

1. $\mathit{names}(\overline{\mathsf{PD}}'') \cap \overline{\mathsf{P}} = \bullet$ *and* $\mathit{names}(\overline{\mathsf{PD}}') \setminus \mathit{names}(\overline{\mathsf{PD}}'') \subseteq \overline{\mathsf{P}}$, *and*
2. *for any codebase* $\overline{\mathsf{PD}}$ *not containing any the names in* $\overline{\mathsf{P}}$, $\overline{\mathsf{PD}}\,\overline{\mathsf{PD}}''$ OK *implies* $\overline{\mathsf{PD}}\,\overline{\mathsf{PD}}'$ OK.

Note that Def. 2 is a special version of Def. 5 with $\overline{\mathsf{P}} = \emptyset$. Similarly, we can define a generalized notion of signature specialization:

DEFINITION 6 (SIGN. SPEC. UP TO FRESHNESS). *We say that* the codebase signature $\overline{\mathsf{PS}}'$ specializes the codebase signature $\overline{\mathsf{PS}}''$ up to freshness of the package names $\overline{\mathsf{P}}$ *to mean that the judgement* $\overline{\mathsf{PS}} \le^{\overline{\mathsf{P}}} \overline{\mathsf{PS}}'$ *can be derived by the rule in Fig. 5. The relation* $\le$*, used in the premise of rule* **SPC-CODEBASE-SIG-UP-TO***, is the signature specialization relation introduced in Def. 3.*

The notion of component specialization up to freshness can be generalized to codebases.

Other enhancements of the specialization relations are possible, and may be especially needed if the source language is more complex. For example, it might be desirable to have specialization relations that allow for adding new public methods to existing classes.

$$\frac{\textit{names}(\overline{\mathsf{PS}}') \cap \overline{\mathsf{P}} = \bullet \qquad \textit{names}(\overline{\mathsf{PS}}'') \subseteq \overline{\mathsf{P}} \qquad \overline{\mathsf{PS}} \leq \overline{\mathsf{PS}}'}{\overline{\mathsf{PS}}''\overline{\mathsf{PS}} \leq^{\overline{\mathsf{P}}} \overline{\mathsf{PS}}'}\ \text{SPC-CODEBASE-SIG-UP-TO}$$

Figure 5: IFJP**: Rule for sign. specialization up to freshness**

5. CONCLUSIONS

We presented an effective technique for checking that a component specializes another one. Then, we generalized this result to codebases, i.e., gave up our assumption concerning definition-completeness. Finally, we outlined more flexible notions of specialization. In the last section, we discuss related and future work.

Related work.

Many module systems [21, 13, 1, 15, 3, 23] have been proposed for Java. Our discussion here focusses on the (currently) most popular ones. The OSGi Alliance provides a module system [18] 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 [19] aims at providing a simple, low-level module system to modularize the JDK. However, it is not an official part of the Java SE 7 Platform.

Most of the aforementioned module systems focus more on observability issues than on accessibility (as explained at the end of Sect. 1). The Java Specification Request (JSR) 294 [14] defines a standard for module accessibility but does not fix the module boundaries. This allows module systems (e.g., like [18]) 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 notion of signature-based type-checking we aim to initiate further research on alternative definitions of modules and their interplay with specialization.

Some authors have studied Java accessibility modifiers. Müller and Poetzsch-Heffter [16] identify the changes that access modifiers in a program can have on the program semantics. Schirmer [20] gives a formalization of the access modifiers and shows interesting runtime properties with respect to access integrity.

Future work.

In future work we would like to extend signature-based type-checking to larger subsets of Java and to identify features that are not compatible with the approach. We also plan to develop a prototype tool. We plan to analyze case studies to understand whether the use of these features can be avoided or limited or whether they could be replaced by other features that are compatible with signature-based type-checking. Our goal is to verify whether signature-based type-checking could be considered a feasible property for real languages and could be considered as a design principle that should be taken into account when adding new features to an existing language or designing a new programming language.

6. REFERENCES

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

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

[3] 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.

[4] F. Damiani, A. Poetzsch-Heffter, and Y. Welsch. A type system for checking specialization of packages in object-oriented programming. Technical Report 386/11, University of Kaiserslautern, October 2011. `https://softech.cs.uni-kl.de/Homepage/PublikationsDetail?id=174`.

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

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

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

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

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

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

[11] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

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

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

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

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

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

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

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

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

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

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

[22] Y. Welsch and A. Poetzsch-Heffter. Full abstraction at package boundaries of object-oriented languages. In *SBMF 2011*, LNCS, pages 28–43. Springer, 2011.

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