# Fine-Grained API Evolution for Method Deprecation and Anti-Deprecation

S. Alexander Spoon

LAMP, Station 14
Swiss Federal Institute of Technology in Lausanne (EPFL)
CH-1015 Lausanne
lex@lexspoon.org

## Abstract

API evolution is the process of migrating an inter-library interface from one version to another. Such a migration requires checking that all libraries which interact through the interface be updated. Libraries can be updated one by one if there is a transition period during which both updated and non-updated libraries can communicate through some transitional version of the interface. Static type checking can verify that all libraries have been updated, and thus that a transition period may end and the interface be moved forward safely. A fine-grained checker can do so for individual changes to an interface, thus allowing interface changes to be interleaved. Anti-deprecation is a novel type-checking feature that allows static checking for more interface evolutions periods than deprecation alone. Anti-deprecation, along with the more familiar deprecation, is formally studied as an extension to Featherweight Java. This study finds weaknesses in current popular deprecation checkers.

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

**General Terms** Design, Languages, Management, Standardization

**Keywords** Java, Scala, interfaces, components, API evolution, interface evolution

"In Java when you add a new method to an interface, you break all your clients.... Since changing interfaces breaks clients you should consider them as immutable once you've published them." –Erich Gamma [21]

"`NoSuchMethodError`" –Java VM, all too frequently

## 0. Fair notice

As fair notice, this article further develops work that was presented at an OOPSLA workshop, the workshop on Library-Centered Software Design (LCSD) [19]. The main new contribution of this article is the allowance for individual transitions to forward via the *wait set*. The earlier LCSD version allowed only all or no transitions in a program to move forward.



*FOOL* 2006 Nice, France



## 1. Overview

Libraries communicate with each other via application programming interfaces (API's), or *interfaces* for short. The key idea with interfaces is that so long as a set of libraries conform to their interfaces, those libraries will tend to function together when they are combined. This approach is a key part of standard discussions of software modularity [2].

This interfaces idea supports independent evolution of libraries, in that libraries can be updated so long as they continue to conform to their interfaces. However, this strategy does not address evolution of the interfaces themselves. Since in practice the first definition of an interface is often insufficient, practitioners need some approach for improving interfaces. This is the problem of *interface evolution*.

Interface evolution arises in practice for large projects with multiple independent development groups. The Eclipse project, for example, includes plugin code written by development groups all over the world. For such projects, substantial attention is put onto the problem of safely upgrading interfaces [5].

*Transition periods* provide a general mechanism for evolving the interfaces between independently maintained libraries. A transition period is a period of time during which both updated and non-updated libraries can successfully communicate through an evolving interface.

During a transition period, all libraries that conform to the original version of an interface must be allowed to continue to function. As the transition period progresses, more and more libraries should be updated for the forthcoming version of the interface, while continuing to work with the transitional version of the interface. A transition period can successfully terminate when all libraries communicating through the interface have been either updated or abandoned. At that time, the interface itself can be upgraded.

Static type checking can be used to verify that a transition period may be safely entered or left. At the beginning of a transition period, static checking can ensure that all libraries conforming to the current interface will continue to conform to the new, transitional interface. At the end of a transition period, static checking can ensure that all checked libraries are ready for individual changes to the in-

```
public interface ConnectionListener {
  public void connectionClosed();
  public void connectionClosedOnError(Exception e);
}

public interface ConnectionListener2
extends ConnectionListener {
  public void connectionAuthenticated();
}
```

**Figure 1.** Two interfaces from Eclipse. The second interface is the same as the first except that it requires one new method.

```
public interface ConnectionListener {
  public void connectionClosed();
  public void connectionClosedOnError(Exception e);


  forthcoming
  public void connectionAuthenticated();
}
```

**Figure 2.** With forthcoming methods, the new method could have been gradually phased into the original interface. Once all relevant code has been updated for the new method, the `forthcoming` keyword can be removed, resulting in a single, updated `ConnectionListener` interface.

terface to progress. The same checker can ideally be used for both purposes; it can detect which changes can progress in the normal course of type checking the program.

This article studies static type checking for *deprecation* and *anti-deprecation* of methods. Deprecation is widely used, while anti-deprecation appears to be novel for programming languages. After describing the features in general, the article defines them formally as an extension to Featherweight Java [11], and proves several core properties about the formalism. This systematic study not only defines the new feature, but unearths two places where current deprecation checkers could be improved.

## 2. Static transition checking

Static checking can help both entering and leaving transition periods. When entering a transition period, the checker can verify that clients will continue to compile and run, even if not all libraries using the interface are available. As the transition period moves forward, each library's developers can use the checker as they update their library to verify that their updates are sufficient for the next version of the interface. Once all libraries have been updated and checked, it is safe to move the interface forward.

Put another way, the entries and exits of transition periods are refactorings [14]. If the static checker is satisfied, then crossing these end points causes a change in program syntax but not in program behavior.

This approach allows libraries to be updated even when—especially when—the interface maintainer does not have control over all libraries which communicate through the interface. The conditions for entering a transition period are weak enough that interface maintainers can begin a transition period without waiting for the other library maintainers. Leaving the transition period has much stricter requirements. However, the requirements are such that each library can be updated individually. Once the (loose) organization of library maintainers has decided that sufficient checking has occurred, and if no errors are known to be present, the transition period can be left.

Note that some organizational process is required for transition periods to work. A typical process would involve a length of calender time, *e.g.* six months or one year. A transition would be introduced, library maintainers would be given the specified amount of time to update their libraries, and then the transition would be followed through. Library maintainers who do not update, end up with broken libraries. More sophisticated processes would include feedback from the library maintainers that could extend the transition period. Many specific arrangements are possible, and they are left as beyond the scope of the present article.

Transition checking can be integrated with a normal type checker. Each time the type checker encounters code that type checks now but which will have trouble after a planned interface change occurs, the checker emits a *transition warning* about why the change cannot yet be moved forward. *Fine-grained* transition checking additionally calculates a specific *wait set*. Each time the checker emits a warning, it adds one or more planned interface changes to the wait set. When the checker completes, any change not included in the wait set may safely progress.

## 3. Anti-deprecation

Deprecation allows a static checker to emit warnings whenever a caller tries to use a method that is expected to disappear in a future version of an interface. A complementary scenario is also important: sometimes a future version of an interface will require an additional method. An annotation for such future required methods could be called *anti-deprecation*.

The typical usage for anti-deprecation is shown in Figures 1 and 2. Figure 1 shows one of Eclipse's "I*2" interfaces, an interface that is an extension of an earlier interface. Experience with the framework showed that the earlier interface was too thin, but given the nature of Java interfaces, new methods could not be added to the existing, published interface. Thus, the Eclipse developers added a second interface which merely extends the first interface and adds one new method.

With forthcoming methods, the designers would have had the option to phase in the method to the existing interface, as shown in Figure 2. During a transitional period, the new method would be added but with the keyword `forthcoming`. As long as the keyword is there, the method cannot be called, but the type checker can use it to emit warnings. In particular, the type checker can emit a warning whenever it sees a class that implements the interface but does not implement the forthcoming method. Such a class can compile and run for now, but will have a problem when the forthcoming method is upgraded to a normal method.

The full checking rules for forthcoming methods are as follows. As described in the above example, if a forthcoming method is abstract, then any non-abstract class that implements the interface in question must either implement the method with the correct type signature or receive a warning. Additionally, a forthcoming method my override another forthcoming method, but it will receive a warning if it does. Finally, a forthcoming method cannot override a non-forthcoming method at all. Allowing this case adds

complexity without utility. If a method is already present, then what is the use of encouraging it further in a subclass?

The combination of deprecation and anti-deprecation allows for an additional class of changes that neither mechanism supports alone: arbitrary changes to a method's signature. For example, one might wish to change the set of exceptions thrown by a method, or change a method's return type, or change its public or private visibility.

Such changes can always be accomplished using four transition periods. The approach is analogous to swapping two variables by using an auxiliary method. The first period introduces a new, forthcoming version of the method with a different name than the original method. Since the method is new, it can be given any type signature at all. The second period removes the forthcoming keyword from the new method and deprecates the original method, thus inducing callers to use the new version of the method. The third period removes the deprecated original method and immediately replaces it with a forthcoming method with the new signature and the original name. The fourth period deprecates the temporary method name, thus inducing clients to change back to using the original method.

Alternatively, developers can choose a shorter two-phase sequence if they are content for the new method to have a different name from the original. They can simply stop after the first two transition phases.

These rules for `forthcoming` and `deprecated` might seem pessimistic. These rules are formed under the assumption that developers in other groups might both implement any interface and invoke the methods it advertises. If this assumption were changed to restrict what other developers can do, then some interface changes could be safely performed with fewer or even no transition periods.

For example, suppose that one party controls an interface along with all of its implementors. In that case, that party can add methods to the interface without needing a transition period. They can simply make a simultaneous release of the updated interface and the updated implementors of that interface. Likewise, if one party controls all callers to an interface, *e.g.* as with call backs, that party can remove methods from the interface without needing a transition period.

The present work addresses the less constrained scenario where outside developers can both implement an interface and call through it. The main reason for this choice is that it is the more general and difficult case. However, notice that even when outside developers are expected to be more constrained in their work, it is desirable to allow them the greater flexibility. At the least, it is useful for testing if programmers can implement their own mock objects to stand in place of the usual ones [12, 9].

## 4. Extending Featherweight Java

While `deprecated` and `forthcoming` are simple to describe, it proves tricky to develop the precise rules for checking them so that transition periods can be safely entered and left. In order to determine the precise checking rules, the bulk of this article focuses on a formal study of a small language including these keywords.

The keywords are added to Featherweight Java (FJ) [11], a language that has several appealing characteristics: it is tiny, making it amenable to formal study; it uses familiar syntax, so that the work is more approachable; and it captures two features at the heart of object-oriented languages, message sending and inheritance.

$$
\begin{array}{rcl}
L & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \{\ \bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\ \} \\
K & ::= & C(\bar{C}\ \bar{f})\ \{\ \texttt{super}(\bar{f});\ \texttt{this}.\bar{f} = \bar{f};\ \} \\
M & ::= & C\ m(\bar{C}\ \bar{f})\ MB \\
MB & ::= & \{\ \texttt{return}\ e;\ \} \mid \texttt{abstract} \\
X & ::= & \texttt{deprecated}\ m; \\
N & ::= & \texttt{forthcoming}\ M; \\
e & ::= & x \mid e.f \mid e.m(\bar{e}) \mid \texttt{new}\ C(\bar{e}) \mid (C)e
\end{array}
$$

**Figure 3.** Syntax of FJ-ADF

$$MS \ ::= \ C.m$$

**Figure 4.** Method specifications

In one way, though, the FJ language is a little too small for the present purpose: it does not include a notion of interfaces. Instead of adding a full interface concept, it suffices to add abstract methods. Abstract methods allow abstract classes, which for the present purposes serve as perfectly fine interfaces. The full extended language is called FJ-ADF because it is Featherweight Java with three new keywords: `abstract`, `deprecated`, and `forthcoming`.

A few notational issues bear comment. When a line of code is written down by itself as an assumption, the meaning is that that line of code appears somewhere in the program. A sequence is written $\bar{x}$, denoting the sequence $x_1, \ldots, x_n$, where $\#(\bar{x}) = n$. The empty sequence is $\bullet$ by itself, while a comma between two sequences denotes concatenation. Pairs of sequences are a shorthand for a sequence of pairs; for example, $\bar{C}\ \bar{x}$ means $C_1\ x_1\ \ldots\ C_n\ x_n$. The notation $x \in \bar{y}$ means that $x = y_i$ for some $i$. Negation, written $\neg P$, means that $P$ cannot be proven with the available inference rules.

### 4.1 Syntax and semantics

The syntax of FJ-ADF is given in Figure 3. There are a few differences from FJ:

- Methods can be abstract. Any class that defines or inherits an abstract method is considered abstract and cannot be instantiated with `new`.
- Methods can be forthcoming. A forthcoming method cannot be used now, but will be added to a future version of the class.
- Each class has a list of deprecated methods. Deprecated methods are going to be removed in a future version of the class.

An entire program is denoted $CT$ or $CT'$. Notationally, $CT$ is a table, and $CT(C)$ is the class in the program named $C$. Valid programs have several syntactic restrictions: the inheritance hierarchy is non-cyclical, all field names and parameter names are distinct, $\texttt{Object} \notin dom(CT)$, every class name appearing in the program is in the domain of $CT$.

The *fields* function, defined in Figure 5, computes the complete list of fields in a class.

Method specifications, defined in Figure 4, provide a way to designate individual methods: a class name plus a method name. The set $MS$ includes all possible method specifications.

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\texttt{class } C \texttt{ extends } D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad \mathit{fields}(D) = \bar{D}\ \bar{g}}{\mathit{fields}(C) = \bar{D}\ \bar{g}, \bar{C}\ \bar{f}}$$

**Figure 5.** Field lookup

$$\text{MB-Conc}\ \frac{\texttt{class } C \texttt{ extends } D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad B\ m(\bar{B}\ \bar{x})\ \{\ \texttt{return } e;\ \} \in \bar{M}}{\mathit{mbody}(m, C) = \bar{x}.e}$$

$$\text{MB-Abs}\ \frac{\texttt{class } C \texttt{ extends } D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad B\ m(\bar{B}\ \bar{x})\ \texttt{abstract} \in \bar{M}}{\mathit{mbody}(m, C) = \texttt{abstract}}$$

$$\text{MB-Inher}\ \frac{\texttt{class } C \texttt{ extends } D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad m \notin \bar{M} \qquad \mathit{mbody}(m, D) = MB}{\mathit{mbody}(m, C) = MB}$$

$$\frac{\mathit{mbody}(m, C) = \texttt{abstract}}{\mathit{abstract}(C)}$$

**Figure 6.** Method lookup

The *mbody* function, defined in Figure 6, is used during evaluation to find the method responding to a message-send expression. It is the same as in FJ except that there is a new clause to support abstract methods. Notice that *mbody* consistently ignores the deprecation ($\bar{X}$) and forthcoming ($\bar{N}$) portions of class definitions. Those portions are only used for type checking, not for execution.

The *abstract* function, also defined in Figure 6, checks whether a class defines or inherits an abstract method. Such classes are not allowed to be instantiated.

The semantics of the language are exactly the same as those of FJ. They are given in Figure 7.

### 4.2 The wait set, *Wait*

The formal definition of static transition checking presumes a *wait set*, denoted *Wait*, is provided before checking starts. This set specifies all method specifications that are assumed to be held back. The type checker assumes that all deprecated methods in *Wait* will not be removed, and that all forthcoming methods in *Wait* will not be promoted to normal methods. Methods not in the wait set, might or might not be changed according to their `deprecated` and `forthcoming` declarations, and the checker must verify that these changes are safe.

As one example, if there is an expression in the program that invokes a deprecated method, then that method must be an element of *Wait*. Otherwise, removing that method could cause type checking to fail or the program to stop running.

The type checking rules are presented as if the wait set is fixed and is supplied to the checker. It is also possible to infer a wait set in the course of type checking, as discussed below in Section 6.

On a technical note, the wait set is presented as a global variable because it does not vary during type checking. Introducing the extra variable to all type judgments would add clutter without clarification. As a result, the predicates *candep*, *postneeds*, and *postabs* are sensitive to the wait set even though it does not appear in their argument lists.

On another technical note, the complement of the wait set is written as $MS - Wait$. Since $MS$ is the set of all method specifications, $MS - Wait$ is the set of method specifications that are not held back.

$$\text{R-Field}\ \frac{\mathit{fields}(C) = \bar{C}\ \bar{f}}{(\texttt{new } C(\bar{e})).f_i \longrightarrow e_i}$$

$$\text{R-Invk}\ \frac{\mathit{mbody}(m, C) = \bar{x}.e_0}{(\texttt{new } C(\bar{e})).m(\bar{d}) \longrightarrow [\bar{d}/\bar{x}, \texttt{new } C(\bar{e})/\texttt{this}]e_0}$$

$$\text{R-Cast}\ \frac{C <: D}{(D)(\texttt{new } C(\bar{e})) \longrightarrow \texttt{new } C(\bar{e})}$$

$$\text{RC-Field}\ \frac{e \longrightarrow e'}{e.f \longrightarrow e'.f}$$

$$\text{RC-Invk-Recv}\ \frac{e \longrightarrow e'}{e.m(\bar{e}) \longrightarrow e'.m(\bar{e})}$$

$$\text{RC-Invk-Arg}\ \frac{e \longrightarrow e'}{e_0.m(\bar{d}, e, \bar{f}) \longrightarrow e_0.m(\bar{d}, e', \bar{f})}$$

$$\text{RC-New-Arg}\ \frac{e \longrightarrow e'}{\texttt{new } C(\bar{d}, e, \bar{f}) \longrightarrow \texttt{new } C(\bar{d}, e', \bar{f})}$$

$$\text{RC-Cast}\ \frac{e \longrightarrow e'}{(C)e \longrightarrow (C)e'}$$

**Figure 7.** Evaluation

### 4.3 Type checking

Subtyping for FJ-ADF is shown in Figure 9. As with FJ, subtyping corresponds exactly to the subclass hierarchy.

The *mtype* function, defined in Figure 8, looks up the type of the responding method when a message is sent to an instance of a particular class. As compared to FJ, FJ-ADF's *mtype* function has two new parameters: one determining which forthcoming methods to include, and one determining which deprecated methods to include. The choice of these parameters is typically influenced by *Wait*; deprecated methods are only considered when they are in *Wait*, and forthcoming methods are only considered when they are not in *Wait*.

Generally, a deprecated or forthcoming method is only visible to *mtype* if it appears in the *depr* or *forth* argument, respectively. As one exception, MT-Depover allows a deprecated method to be visible even when it is not in *depr*, so long as it overrides another method that is otherwise visible. Without this exception, *mtype* would suffer an inconsistency: specifying a more specific subclass for the first argument could cause *mtype* to become undefined. Lemma 2 would not hold.

The *mavail* function, also shown in Figure 8, claims that a method is available in a particular class. Its arguments are the same as for *mtype*. It is typically used in the negative, to claim that a method is not available at all.

Post-abstract classes are those that might become abstract after the program evolves forward. The *postabs* function, defined in Figure 10, gives a conservative notion of post-abstract classes. It is defined in terms of a more specific *postneeds* predicate which claims that a method might be ab-

$$\text{MT-Here}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad B\ m(\bar{B}\ \bar{x})\ MB \in \bar{M} \quad (C.m \in depr) \vee (m \notin \bar{X})}{mtype(m, C, forth, depr) = \bar{B} \to B}$$

$$\text{MT-Forth}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad B\ m(\bar{B}\ \bar{x})\ MB \in \bar{N} \quad C.m \in forth}{mtype(m, C, forth, depr) = \bar{B} \to B}$$

$$\text{MT-Inher}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad m \notin \bar{M} \quad m \notin \bar{N} \quad mtype(m, D, forth, depr) = \bar{B} \to B}{mtype(m, C, forth, depr) = \bar{B} \to B}$$

$$\text{MT-Depover}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad B\ m(\bar{B}\ \bar{x})\ MB \in \bar{M} \quad C.m \notin depr \quad m \in \bar{X} \quad mtype(m, D, forth, depr) = \bar{B} \to B}{mtype(m, C, forth, depr) = \bar{B} \to B}$$

$$\frac{mtype(m, C, forth, depr) = \bar{B} \to B}{mavail(m, C, forth, depr)}$$

**Figure 8.** Method type lookup

$$C <: C \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ E\ \{\ldots\} \quad E <: D}{C <: D}$$

**Figure 9.** Subtyping

$$\frac{postneeds(m, C)}{postabs(C)}$$

$$\text{PN-Abs}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad B\ m(\bar{B}\ \bar{x})\ \texttt{abstract} \in \bar{M}}{postneeds(m, C)}$$

$$\text{PN-Deprec}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad postneeds(m, D) \quad m \in \bar{X} \quad C.m \notin wait}{postneeds(m, C)}$$

$$\text{PN-Forth}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad B\ m(\bar{B}\ \bar{x})\ \texttt{abstract} \in \bar{N} \quad C.m \notin wait}{postneeds(m, C)}$$

$$\text{PN-Inher}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\} \quad postneeds(m, D) \quad m \notin \bar{M}}{postneeds(m, C)}$$

**Figure 10.** Post-abstract classes

$$\text{Dep-Wait}\ \frac{C.m \in Wait}{candep(m, C)}$$

$$\text{Dep-Unavail}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \quad \neg mavail(m, D, MS - Wait, Wait)}{candep(m, C)}$$

$$\text{Dep-Postneeds}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \quad postneeds(m, D)}{candep(m, C)}$$

**Figure 14.** Patterns of overriding can force a deprecated method to be in the wait set. Even though the method is not accessed directly in the program text, it might be accessed indirectly due to method dispatch.

stract in the class following either the removal of `deprecated` methods or the upgrading of `forthcoming` methods to normal methods or both, excluding methods in the wait set. A class that has any *postneeds* method is considered post-abstract.

The type checker of FJ needs to be updated in two ways for FJ-ADF. First, it needs to address the three new keywords. Second, it works with a *wait set*. An FJ-ADF typing judgment is written $\Gamma \vdash e : C$. As usual, $\Gamma$ is a static typing environment, $e$ is an expression, and $C$ is a type (*i.e.*, a class).

The typing rules for expressions are shown in Figure 11. Only two of these rules differ from FJ. First, the T-Invk judgment must specify the two extra parameters of *mtype*. The first additional argument is always $\emptyset$, because methods that are merely forthcoming are not allowed to be invoked, regardless of the wait set. The second additional argument is *Wait*, because it is precisely the deprecated methods in the wait set which may be invoked.

The other changed rule is T-New, which now disallows instantiating abstract classes. This rule means that an invariant during evaluation is that all instantiated objects are concrete, thus making it safe for T-Invk to consider abstract methods as potential callees. Post-abstract classes, a superset of the abstract classes (Lemma 6), are also disallowed; a class is not allowed to be instantiated if it might become abstract after forthcoming changes.

The rules for checking methods are given in Figure 12. The main change from FJ is that all forthcoming methods not held in the wait set must have types conforming to the methods above and below them. An additional change, included for simplicity, is that abstract and forthcoming methods are only allowed when they do not override a previously existing non-forthcoming method. There is no useful purpose to adding a forthcoming method underneath an existing method. Adding an abstract method under an existing method is sometimes useful, but the complication adds no insight for the present purposes.

Finally, the rule for typing a class is given in Figure 13. The main addition is that all deprecated methods must be allowed by *candep*, which is defined in Figure 14. The *candep* predicate forces extra deprecated methods into the wait set, beyond those already required by T-Invk and T-New, depending on patterns of inheritance and overriding. The predicate places no restriction on methods that are in the wait set (Dep-Wait), and it provides two cases where a method may be left out of the wait set.

The first allowed case is when a method is unavailable in all of the superclasses of a class (Dep-Unavail). Note that the *mavail* occurrence in the assumptions of Dep-Unavail is exactly the negation of that in T-Invk, and further that a failing *mavail* test remains false if you move up the class

$$\text{T-Var}\ \frac{x : C \in \Gamma}{\Gamma \vdash x : C}$$

$$\text{T-Field}\ \frac{\Gamma \vdash e_0 : C_0 \qquad \mathit{fields}(C_0) = \bar{C}\ \bar{f}}{\Gamma \vdash e_0.f_i : C_i}$$

$$\text{T-New}\ \frac{\mathit{fields}(C) = \bar{D}\ \bar{f} \qquad \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C} <: \bar{D} \qquad \neg \mathit{postabs}(C)}{\Gamma \vdash \texttt{new}\ C(\bar{e}) : C}$$

$$\text{T-Invk}\ \frac{\begin{array}{c}\Gamma \vdash e_0 : C_0 \\ \mathit{mtype}(m, C_0, \emptyset, \mathit{Wait}) = \bar{D} \to C \\ \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C} <: \bar{D}\end{array}}{\Gamma \vdash e_0.m(\bar{e}) : C}$$

$$\text{T-UCast}\ \frac{\Gamma \vdash e_0 : D \qquad D <: C}{\Gamma \vdash (C)e_0 : C}$$

$$\text{T-DCast}\ \frac{\Gamma \vdash e_0 : D \qquad C <: D \qquad C \neq D}{\Gamma \vdash (C)e_0 : C}$$

$$\text{T-SCast}\ \frac{\Gamma \vdash e_0 : D \qquad D \not<: C \qquad D \not<: C \qquad \textit{stupid warning}}{\Gamma \vdash (C)e_0 : C}$$

**Figure 11.** Typing of expressions

$$\text{T-Method-Fresh}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \\ \neg \mathit{mavail}(m, D, \mathit{MS} - \mathit{Wait}, \mathit{MS}) \\ \bar{x} : \bar{C}, \texttt{this} : C \vdash e_0 : E_0 \qquad E_0 <: C_0\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \{\ \texttt{return}\ e_0;\ \}\ \ \text{OK IN}\ C}$$

$$\text{T-Method-Over}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \\ \mathit{mtype}(m, D, \mathit{MS} - \mathit{Wait}, \mathit{MS}) = \bar{C} \to C_0 \\ \bar{x} : \bar{C}, \texttt{this} : C \vdash e_0 : E_0 \qquad E_0 <: C_0\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \{\ \texttt{return}\ e_0;\ \}\ \ \text{OK IN}\ C}$$

$$\text{T-Method-Abs}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \\ \neg \mathit{mavail}(m, D, \mathit{MS} - \mathit{Wait}, \mathit{MS})\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \ \texttt{abstract}\ \ \text{OK IN}\ C}$$

$$\text{T-Method-Forth}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \\ \neg \mathit{mavail}(m, D, \mathit{MS} - \mathit{Wait}, \mathit{MS}) \\ C_0\ m(\bar{C}\ \bar{x})\ \mathit{MB}\ \ \text{OK IN}\ C\end{array}}{\texttt{forthcoming}\ C_0\ m(\bar{C}\ \bar{x})\ \mathit{MB}\ \ \text{OK IN}\ C}$$

**Figure 12.** Typing of methods

$$\text{T-Class}\ \frac{\begin{array}{c}K = C(\bar{D}\ \bar{g},\ \bar{C}\ \bar{f})\ \{\ \texttt{super}(\bar{g});\ \texttt{this}.\bar{f} = \bar{f};\ \} \\ \mathit{fields}(D) = \bar{D}\ \bar{g} \\ \forall m \in \bar{X} : m \in \bar{M} \qquad \forall m \in \bar{N} : m \notin \bar{M} \\ \bar{M}\ \text{OK IN}\ C \\ \bar{N}\ \text{OK IN}\ C \\ \forall m \in \bar{X} : \mathit{candep}(m, C)\end{array}}{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \bar{C}\ \bar{f};\ K\ \bar{M}\ \bar{X}\ \bar{N}\ \}\ \ \text{OK}}$$

**Figure 13.** Typing of classes

hierarchy and perform the same *mavail* query in a superclass (Lemma 2).

The second case is that a class *postneeds* the specified method name. In such a case, even though *mavail* sees that method name, the method in that class will not be invoked. The reason is that the receiver class is post-abstract, and thus T-New will not allow the receiver class to be instantiated. It will only be allowed to implement a non-post-abstract subclass, and that subclass must by implement the method name with a method that is either in the wait set or is not deprecated. Since the method will not be invoked in this case, it is safe to deprecate and remove it.

## 5. Properties

Given the careful definition of FJ-ADF, we can now study some properties that it achieves. The properties are divided into two parts: typical type-soundness properties and properties to support statically checked interface evolution.

### 5.1 Type soundness

FJ-ADF is type sound in the usual sense: it enjoys both subject reduction and progress theorems.

**Theorem 1.** *(Subject Reduction). If* $\Gamma \vdash e : C$ *and* $e \longrightarrow e'$*, then* $\Gamma \vdash e' : C'$ *for some* $C' <: C$*.*

The proof structure is very close to that for subject reduction for FJ. The main differences are in the supporting lemmas.

The first lemma is that *mtype*'s last two arguments do not change the type the function calculates, but only whether the function is defined or not. Enlarging either one of the last two arguments can cause the function to become defined, but never to become undefined.

**Lemma 1.** *(Internal Consistency of mtype). Suppose that* $mtype(C, m, forth, depr) = \bar{C} \to C_0$*. Suppose that* $forth \subseteq forth'$ *and* $depr \subseteq depr'$*. Then,* $mtype(C, m, forth', depr') = \bar{C} \to C_0$*.*

*Proof.* Straightforward. Most of the justification rules of Figure 8 can still be used if the third and forth arguments of *mtype* are increased. The only exception is MT-Depover, which cannot be used if $C.m \notin depr$ but $C.m \in depr'$. In that case, however, the application of MT-Depover can be replaced by an application of MT-Here. □

The following lemma shows that, ignoring forthcoming methods, once *mtype* returns a result at one point in the class hierarchy, it returns the same result deeper in the hierarchy.

**Lemma 2.** *(Subclasses and mtype). Suppose CT is OK. If* $mtype(m, D, \emptyset, Wait) = \bar{C} \to C_0$*, then for all* $C <: D$*, also* $mtype(m, C, \emptyset, Wait) = \bar{C} \to C_0$*.*

*Proof.* Induct on the derivation that $C <: D$. $C = D$ is a trivial case, so suppose that $C$ `extends` $E$ and $E <: D$. By the inductive assumption, $mtype(m, E, \emptyset, Wait) = \bar{C} \to C_0$. This means that $mavail(m, E, \emptyset, Wait)$, and thus by Lemma 1 that both $mavail(m, E, \emptyset, MS)$ and also $mavail(m, E, MS - Wait, MS)$.

If $C$ does not include a method $m$ at all, including among its forthcoming methods, then MT-Inher gives the desired result. If $m$ is defined in $C$, then the method must have been justified as OK using T-Method-Over; the other T-Method-* rules cannot apply because their *mavail* requirements cannot be met. Given that T-Method-Over was used, all of the requirements are met to use either MT-Here or MT-Depover, depending on whether $m$ is deprecated in $C$, to show that $mtype(m, C, \emptyset, Wait) = \bar{C} \to C_0$. T-Method-Over requires that the new method has the same type signature as the inherited method, and thus that MT-Here or MT-Depover will give this inherited type. □

**Lemma 3.** *(Term Substitution Preserves Typing). If* $\Gamma, \bar{x} : \bar{B} \vdash e : D$*, and* $\Gamma \vdash \bar{d} : \bar{A}$ *where* $\bar{A} <: \bar{B}$*, then* $\Gamma \vdash [\bar{d}/\bar{x}]e : C$*, for some* $C <: D$*.*

*Proof.* Induct on the derivation of $e$'s type, and do a case analysis on the last typing rule used. The proof is then exactly the same as with FJ, except that the argument for case T-Invk must use the updated Lemma 2 given above. □

**Lemma 4.** *(Weakening). If* $\Gamma \vdash e : C$*, then* $\Gamma, x : B \vdash e : C$*.*

*Proof.* Straightforward induction. □

The next lemma is modified from that for FJ by adding two arguments to the use of *mtype*. The choice of parameters—$\emptyset$ and *Wait*—are those used by T-Invk.

**Lemma 5.** *Suppose that CT is OK,* $mbody(m, C_0) = \bar{x}.e$*, and* $mtype(m, C_0, \emptyset, Wait) = \bar{D} \to D$*. Then, there is a* $D_0$ *with* $C_0 <: D_0$*, and a* $C$ *with* $C <: D$*, such that* $\bar{x} : \bar{D}, \texttt{this} : D_0 \vdash e : C$*.*

*Proof.* Straightforward induction on the derivation that $mtype(m, C_0, \emptyset, Wait) = \bar{D} \to D$. □

*Proof.* (Subject Reduction). The proof directly follows that for FJ. Induct on the derivation that $e \longrightarrow e'$ and perform a case analysis on the final reduction rule used. □

**Theorem 2.** *(Progress). Suppose that CT holding MS is OK, and e is any well-typed expression.*

1. *If e includes* (`new` $C_0(\bar{e})$)$.f$ *as a subexpression, then* $fields(C_0) = \bar{C}\ \bar{f}$ *and* $f \in \bar{f}$ *for some* $\bar{C}$ *and* $\bar{f}$*.*
2. *If e includes* (`new` $C_0(\bar{e})$)$.m(\bar{d})$ *as a subexpression, then* $mbody(m, C_0) = \bar{x}.e_0$ *and* $\#(\bar{x}) = \#(\bar{d})$ *for some* $\bar{x}$ *and* $e_0$*.*

*Proof.* If e includes (`new` $C_0(\bar{e})$)$.f$ as a subexpression, then the desired result follows directly from $e$ being well-typed and the consequent requirements of rule T-Field. Consider, then, any subexpression (`new` $C_0(\bar{e})$)$.m(\bar{d})$. Since the expression is well-typed, $C_0$ must not be abstract. Furthermore, T-Invk requires that $mtype(m, C_0, \emptyset, Wait) = \bar{D} \to C$ for some $\bar{D}$ and $C$, and it also requires that $\#(\bar{d}) = \#(\bar{D})$. Examination of *mtype* shows then that some superclass of $C_0$ must define a non-forthcoming method named $m$ with $\#(\bar{D})$ parameters. Since $C_0$ is not abstract, that method must also not be abstract, and so it can only be concrete. Thus, $mbody(m, C_0) = \bar{x}.e_0$ for some $\bar{x}$ and $e_0$. □

As with FJ, several theorems follow immediately from Theorem 1 and Theorem 2. FJ-ADF is *type sound*, in that all terminating program executions either compute a value or get stuck at an incorrect cast. Furthermore, cast-free programs do not get stuck and thus always proceed to produce a value if they terminate. Since these theorems follow so directly, the precise definitions and theorem statements are omitted from this article.

### 5.2 Safe transitions

This section shows how to use the wait set of FJ-ADF to evolve interfaces safely. There are two properties given which show when it is safe to add a deprecated or forthcoming method, thus entering a transition period. Following, there are two theorems showing that, when a program strictly checks, it is safe to remove deprecated methods as well as to upgrade forthcoming methods to normal methods. Finally, there are four theorems showing that when the four described safe changes are made, the resulting programs not only type check but continue to behave identically.

Because these properties all involve two programs, there are two versions of each relation and function. To disambiguate between the two versions when it is not clear from context, the program can be used as a subscript. For example, $abstract_{CT}(C_0)$ means that $C_0$ is abstract in program $CT$, and $\Gamma \vdash_{CT'} x : e$ means that $x$ type checks in program $CT'$.

For simplicity, the wait set *Wait* is held fixed for the transition theorems. The introduction theorems could be tightened by only requiring $A.m$ to be in the wait set *after* the discussed introduction and not before. Additionally requiring $A.m$ to be in the wait set *before* the introduction simplifies the theory with no practical impact.

All of these properties discuss a single program being updated from one version to the next. However, the properties are carefully written to support updating single *classes* when that class is going to be used in many different programs.

For the two introduction theorems, beyond requiring transitional type checking, the properties only make requirements on the superclasses of the modified class. Thus, transitional changes can be introduced safely so long as the superclasses of the changed class are immediately available. Further, the requirements on superclasses are weak enough that the superclasses are allowed to be modified without invalidating the requirements of the theorem.

The two removal theorems, to contrast, require that all interesting programs be strictly checked before it is safe to perform the removal. This is potentially a lot of work, but, as discussed in Section 2, the programs do not need to be tested all at once. They can be tested one by one throughout the transition period, as each collaborating development group finds time.

**Theorem 3.** *(Deprecation Introduction). Let $CT$ be any class table that is OK, class $A$ be a class in $CT$, and $m$ be any non-encouraged, non-deprecated method defined by $A$. Suppose that $A.m \in Wait$. Define $CT'$ as the same class table as $CT$ except that $m$ is deprecated in class $A$. Given these assumptions, whenever $\Gamma \vdash_{CT} e : C$, it is also true that $\Gamma \vdash_{CT'} e : C$. Further, $CT'$ is OK.*

*Proof.* Examine the helper functions and relations. The subtype relation (<:) is the same for $CT$ and $CT'$. The helper functions *fields* and *mbody* are also the same. Further, *mtype* is equivalent in the two programs when its fourth argument includes $A.m$. Conversely, whenever $mtype_{CT'}$ is defined with a fourth argument of $\emptyset$, $mtype_{CT}$ is defined with the same arguments.

Since $A.m \in Wait$, *postneeds* is equivalent in $CT$ and $CT'$. The only non-trivial case is if $postneeds_{CT}(m, A)$, but in that case $A.m$ must be abstract, and thus also $postneeds_{CT'}(m, A, Wait)$. By extension, *postabs* is equivalent for $CT$ and $CT'$.

Given all these equivalences, all typing deductions (the T-* rules) that apply for $CT$ also apply for $CT'$. Thus, given a deduction that $\Gamma \vdash_{CT} e : C$, the same deduction can be used to show that $\Gamma \vdash_{CT'} e : C$. Likewise, $CT'$ must be OK, using the same deduction as used for $CT$. □

**Theorem 4.** *(Forthcoming Introduction). Let $CT$ be any class table that is OK, and let $A$ be a class in $CT$ which does not define or inherit any method named $m$,* i.e. *it is the case that $\neg mavail(m, A, MS, MS)$. Suppose $A.m \in Wait$. Suppose that the following method:*

$$B\ m(\bar{B}\ \bar{x})\ body$$

*is OK in $A$. Define $CT'$ to be the same class table as $CT$ except that $A$ has the following additional method definition:*

$$\texttt{forthcoming}\ B\ m(\bar{B}\ \bar{x})\ body$$

*Then, whenever $\Gamma \vdash_{CT} e : C$, it is also true that $\Gamma \vdash_{CT'} e : C$. Further, $CT'$ is OK.*

*Proof.* The same proof structure is used as for Theorem 3. First consider the helper functions and relations. Subtyping (<:) is the same, and *fields* is the same.

The *mtype* predicate is more complex to analyze. When its third argument does not include $A.m$, *e.g.* when the third argument is $\emptyset$ or $MS - Wait$, the predicate is equivalent in $CT$ and $CT'$, *i.e.* if it is defined in one program, it is also defined in the other and with the same value. This is easily seen because every derivation step used to calculate *mtype* in one program, can also be used in the other.

Similarly, since $A.m \in Wait$, *postabs* and *postneeds* are equivalent for $CT$ and $CT'$.

Given these equivalences, all of the inference rules can be used, and thus every type deduction used for $CT$ can also be used for $CT'$. Note that all uses of *mtype* or *mavail* during transitional checking specify a third argument that does not include anything in *Wait*. □

The next two lemmas show that *postabs* behaves as expected. These properties are used to prove the last two safe transition theorems.

**Lemma 6.** *If $abstract(C)$, then $postabs(C)$.*

*Proof.* A straightforward induction on the derivation that $mbody(m, C) = \texttt{abstract}$. Each derivation rule matches one of the available rules for deriving $postneeds(m, C)$. □

**Lemma 7.** *Suppose $CT$ type checks, and that $CT'$ is the same as $CT$ except that possibly some deprecated methods not in Wait have been removed, and possibly some forthcoming methods not in Wait have been promoted to normal methods. Then, whenever $postneeds_{CT'}(m, C)$, it must also be that $postneeds_{CT}(m, C)$.*

*Proof.* Straightforward induction over the derivation that $postneeds_{CT'}(m, C)$. □

**Corollary.** *(Accuracy of postabs). Suppose the conditions of Lemma 7 are met. Then, whenever $\neg postabs_{CT}(C)$, it must also be that $\neg postabs_{CT'}(C)$.*

*Proof.* Follows directly from Lemma 7. □

**Theorem 5.** *(Deprecation Removal). Let $CT$ be OK, let $A$ be a class in $CT$ which defines a method named $m$ that is deprecated, and suppose $A.m \notin Wait$. Define $CT'$ to be the same class table as $CT$ except that $m$ is removed from $A$. Then, whenever $\Gamma \vdash_{CT} e : C$, it is also true that $\Gamma \vdash_{CT'} e : C$. Furthermore, $CT'$ is OK.*

*Proof.* Again, start by examining the helper functions. The subtype relation (<:) is the same for $CT$ and $CT'$, as is the *fields* function.

Lemma 7 and its corollary apply, and so any class $\neg postabs$ in $CT$ is also $\neg postabs$ in $CT'$.

The behavior of *mtype* is more complex to analyze. Two cases include all usages of *mtype* during type checking. First, if $A.m \notin depr$, then $mtype(n, C, forth, depr)$ is equivalent for $CT$ and $CT'$. This can be seen by inducting over the calculation of *mtype*; the only non-trivial subcase is usage of MT-DEPOVER for $m$ in class $A$, but that subcase can be replaced in $CT'$ by a use of MT-INHER.

The second case is that of $mtype(n, C, MS - Wait, MS)$. Whenever *mtype* with these arguments is defined for $CT$, it is either defined as the same value for $CT'$, or it is undefined for $CT'$. Whenever it is defined for $CT'$, it is also defined for $CT$ and with the same value. The first property can be seen by inducting over the calculation of $mtype_{CT}$, where the only non-trivial case is the use of MT-HERE for class $A$ and method $m$. If *mtype* is defined at all in $CT'$ for these arguments, then its definition must be justified with MT-INHER. In that case, T-METHOD-OVER is the only possible way to type check $A.m$ in $CT$, in which case MT-INHER must give the same result in $CT'$ as the type MT-HERE gave in $CT$. The second property can be seen by inducting over the calculation of $mtype_{CT'}$, where analogously one use of MT-INHER must be replaced by a use of MT-HERE.

Whenever $postneeds_{CT}(n, C)$, either $postneeds_{CT'}(n, C)$, or else $\neg mavail_{CT'}(n, C, MS - Wait, Wait)$. This can be seen by inducting over the derivation that $postneeds_{CT}(n, C)$. The only non-trivial case is deriving $postneeds_{CT}(m, A)$. If PN-ABS was used for this derivation, then T-METHOD-ABS ensures that $\neg mavail_{CT'}(n, C, MS - Wait, Wait)$. If PN-DEPREC was used for this derivation, then PN-INHER can instead be used for $CT'$. Given these equivalences for *postneeds*, whenever $candep_{CT}(n, C)$, it is also true that $candep_{CT'}(n, C)$.

Finally, given all of the above equivalences, all typing derivations that were used in $CT$ can also be used for $CT'$. □

**Theorem 6.** *(Forthcoming Upgrade). Let $CT$ be a class table that is OK, and let $A$ be a class in $CT$ which has the following method definition:*

$$\texttt{forthcoming}\ B\ m(\bar{B}\ \bar{x})\ body$$

*Suppose that $A.m \notin Wait$. Define $CT'$ to be the same class table except that the above method definition is replaced by the same method without the* `forthcoming`*:*

$$B\ m(\bar{B}\ \bar{x})\ body$$

*Then, whenever $\Gamma \vdash_{CT} e : C$, $\Gamma \vdash_{CT'} e : C$. Furthermore, $CT'$ is OK.*

*Proof.* The *fields* function remains the same.

The *postneeds* predicate is also the same, and thus so is *postabs*. The same derivations can be used in $CT$ and $CT'$, except that for the specific case of $postneeds(m, A)$, we must exchange a use of PN-ABS with a use of PN-FORTH.

Whenever $mtype(n, C, \emptyset, Wait)$ is defined in $CT$, it is defined in $CT'$ with the same arguments. Whenever $mtype(n, C, MS - Wait, MS)$ is defined in either $CT$ or $CT'$, it is defined in the other program and with the same value.

The OK check for method $m$ of class $A$ must change from using T-METHOD-FORTH to T-METHOD-ABS. All other derivations can remain the same, given the above equivalences for the helper functions. □

Given the previous four theorems, it is straightforward to show that in addition to the updated $CT'$ being type safe, all type-safe expressions evaluate equivalently both in $CT$ and $CT'$.

**Theorem 7.** *Let $CT$ and $CT'$ be as in Theorem 3. If $e \longrightarrow_{CT} e'$ and $\Gamma \vdash e : C$, then $e \longrightarrow_{CT'} e'$.*

*Proof.* Since *fields* and *mbody* are the same for $CT$ and $CT'$, the same derivation used for $e \longrightarrow_{CT} e'$ can be used to show that $e \longrightarrow_{CT'} e'$. □

**Theorem 8.** *Let $CT$ and $CT'$ be as in Theorem 4. If $e \longrightarrow_{CT} e'$ and $\Gamma \vdash e : C$, then $e \longrightarrow_{CT'} e'$.*

*Proof.* The *fields* function is the same for $CT$ and $CT'$. The *mbody* function is the same except that with some arguments it is defined as `forthcoming` in $CT'$ whereas in $CT$ it is undefined. Since we assume that $e$ does move forward a step, the cases where *mbody* are undefined in $CT$ are irrelevant. Thus, *mbody* is the same for $CT$ and $CT'$ for all argument tuples that are relevant. Therefore, the same derivation used to show that $e \longrightarrow_{CT} e'$ can be used to show that $e \longrightarrow_{CT'} e'$. □

**Theorem 9.** *Let $CT$ and $CT'$ be as in Theorem 5. If $e \longrightarrow_{CT} e'$ and $\Gamma \vdash e : C$, then $e \longrightarrow_{CT'} e'$.*

*Proof.* Induct on the derivation that $e \longrightarrow_{CT} e'$. The only non-trivial case is if the rule R-INVK is used. Note that whenever $mtype_{CT}(m, C, \emptyset, Wait)$ is defined, then $mbody_{CT}(m, C) = mbody_{CT'}(m, C)$. This can be seen by inducting over the calculation of $mtype_{CT}(m, C, \emptyset, Wait)$. Given this, the case for R-INVK is also straightforward. □

**Theorem 10.** *Let $CT$ and $CT'$ be as in Theorem 6. If $e \longrightarrow_{CT} e'$ and $\Gamma \vdash e : C$, then $e \longrightarrow_{CT'} e'$.*

*Proof.* The same as for the previous theorem. □

## 6. Wait-set inference

The wait set does not have to be supplied to the type checker, even though the typing rules are described as if it were. It is possible for a type checker to begin with an empty wait set, and then to add methods to the wait set in the course of normal type checking.

The core property supporting this approach is as follows. Roughly, if the wait set is enlarged, then all previously verified type judgements remain true. Thus, it is sufficient to add items to the wait set as problems are determined. The one complication is that enlarging the wait set can cause uses of *candep* to no longer succeed. In particular, if a deprecated method is added to the wait set, then any deprecated methods overriding it must also be added. Thus, whenever the wait set is enlarged, it is necessary to additionally add any deprecated methods for which *candep* is no longer true.

**Lemma 8.** *Suppose $CT$ is OK and expression $e$ type checks assuming wait set $Wait$. Suppose that $Wait' \supseteq Wait$ is another wait set such that $candep(m, C)$ for all deprecated methods $C.m$ in $CT$. Then, $CT$ is also OK and $e$ also type checks with wait set $Wait'$.*

*Proof.* The type judgements can mostly be reused as they are under the larger wait set. By assumption, *candep* approves of all deprecated methods, and thus the non-trivial part of T-Class is satisfied. For the typing of methods (T-Method-*), most of the rules can be reused due to Lemma 1. The one exception is that T-Method-Over may not be usable with the larger wait set, but whenever it cannot, then it can be replaced by T-Method-Fresh. For the typing of expressions using T-Invk, Lemma 1 again ensures that enlarging the wait set will allow this expression to continue to type check. The rest of the typing rules for expressions, with the exception of T-New, refer to nothing that is sensitive to the wait set.

The remaining case is T-New, which requires that the instantiated class not be post-abstract. Let us consider the behavior of *postneeds* and *postabs* under the different wait sets. After examining the definition of *postneeds*, it is clear that if $postneeds(m, C)$ under wait set $Wait'$, then also $postneeds(m, C)$ under wait set $Wait$. In turn, whenever $C$ is *not* post-abstract under wait set $Wait$, it is also not post-abstract under wait set $Wait'$. Thus, any use of T-New under wait set $Wait$ also applies under wait set $Wait'$. □

Given these properties, a combined checker can both check types and infer a wait set. The algorithm is as follows. Start with an empty wait set. Add all forthcoming methods which, if not held, would cause one of the T-Method-* rules to fail. Specifically, add those forthcoming methods which are overridden either by another forthcoming method, an abstract method, or a method with a different type signature. Next, iteratively add all deprecated methods for which *candep* cannot be established otherwise.

Then check the expressions of the program. In addition to the usual type-checking checks, the checker might need to increase the wait set whenever T-Invk or T-New is used. If a message-send expression invokes a deprecated method, thus causing T-Invk to be unusable, then the method should be added to the wait set so that T-Invk can succeed. If a `new` expression instantiates a class that is post-abstract but not abstract, causing T-New to be unusable, then the checker should add to the wait set each inherited method of the instantiated class that is either: a forthcoming abstract method, or a deprecated concrete method which overrides an abstract method. Whenever the wait set is increased due to either T-Invk or T-New, the checker must also add any deprecated methods for which *candep* no longer holds true.

As future work, this algorithm can be refined in two major ways. First, it is unclear that this algorithm finds *minimal* wait sets, or indeed that there always exists a single minimal wait set at all with the current typing rules. Second, no limit is established on the methods for which *candep* can fail after a method is added to a wait set. As described, the entire class table must be scanned, but it appears sound to scan only those methods overriding the newly added method, and furthermore to stop scanning past any methods which are checked but do not need to be added to the wait set.

## 7. Weaknesses in current tools

Existing checking tools do not include enough checks to guarantee safe exits from transition periods. For a checker to make such guarantees, it must include a check for each place the FJ-ADF type system is sensitive to the wait set, and it must include FJ-ADF's checks on overriding between different kinds of methods. There are four such places in the type system. Three apply to deprecation-only checkers (with no anti-deprecation), but only one of those three is checked by Sun `javac` version 1.5.0_06 or the checker within Eclipse 3.2.

```
abstract class A
{
  abstract int foo(int x);
}

class B extends A
{
  /**
   * @deprecated
   */
  int foo(int x) {
    return x+1;
  }
}

class Client
{
  void run() {
    A a = new B();
  }
}
```

**Figure 15.** Removing a method can cause a class to become abstract. Instantiating such a class should cause a deprecation warning.

All code samples in this section mark deprecation in the comments, rather than with a `deprecated` keyword, in order to match the requirements of existing implementations.

**Method invocation** The T-Invk rule does not allow a message-send expression to invoke a method that is deprecated, unless that method is in *Wait*. Current checkers capture this familiar rule.

**Post-abstract methods** The T-New rule does not allow instantiating a post-abstract class, *i.e.* a class that might be abstract after a change not in *Wait* is moved forward. An example is given in Figure 15. Class `B` is not abstract currently, but it will become abstract once the deprecated method `foo` is removed. Thus, while `B` is not abstract currently, it will be after its deprecated method is removed. An ideal transition checker should issue a warning for code that instantiates `B`, because such code will no longer function if the deprecated method is removed. No warning is given, though, by `javac` or Eclipse.

**Deprecated methods and overriding** Overriding patterns can prevent the safe removal of a deprecated method, even when T-New and T-Invk identify no problems. For a deprecated method to be removable, it must additionally not be callable indirectly, via a non-removable method in a superclass.

An example problem appears in Figure 16. The code in class `C` type checks by considering method `A.frob`, but at run time it invokes `B.frob`. If method `B.frob` is removed, then the behavior of the program will change and `B`'s invariants might be broken. If this behavior change is truly acceptable, then `B.frob` should be removed instead of deprecated. Again, `javac` and Eclipse do not issue a warning for this code.

```
class A
{
  void frob() {
     System.out.println("frobbed!");
  }
}

class B extends A
{
  int accesses = 0;

  /**
   * @deprecated
   */
  void frob() {
    accesses += 1;
    super.frob();
  }
}

class Client
{
  void run() {
    A a = new B();
    a.frob();
  }
}
```

**Figure 16.** Deprecating a method that overrides a concrete method can result in invariants being broken.

**Forthcoming methods and overriding** Overriding patterns also put restrictions on which forthcoming methods may be upgraded to normal methods. The specific requirements are discussed informally in Section 3. Anti-deprecation is not supported at all in existing tools, and so no code examples are given.

## 8. Related work

There has been substantial work supporting interface evolutions that are refactorings [4, 1, 10, 15]. When such work applies, the benefit can be immense, because the transition period can be shortened or even eliminated. Nonetheless, many desirable interface changes are not refactorings at all. For example, not all uses of deprecated methods can be rewritten to use non-deprecated methods. Sometimes the basic functionality is being removed. For such changes, some kind of transition period is necessary, and checking tools can help entering and leaving those transition periods safely.

There has been work on language features to help manage or eliminate incompatibilities due to interface upgrades. The reuse contracts of Steyaert, *et al.*, allow detection of a variety of upgrade problems when given only the new version of an interface and not the old one [20]. The `override` keyword of C# and Scala prevents accidental override of newly added methods in a superclass [7, 22, 13, 16]. The present work focuses more on managing the transition periods than on detecting or ameliorating problems after an interface changes.

Interface definitions of various kinds have long supported recording deprecation. Two examples for programming languages are Java's deprecation annotations [3] and Eiffel's `obsolete` keyword [8]. The present work uses the same concept but adds anti-deprecation.

Dig and Johnson have quantitatively studied the kinds of interface changes that occurred during the lifetime four software systems [6]. The authors start with the developers' change logs and the version control systems for each software system, and then use these data sources to identify the relative frequency of several kinds of API changes. For example, they classify over 80% of the API changes as some kind of refactoring. Software science such as this provides invaluable input for those designing transition mechanisms that are to be useful in practice.

## 9. Future work

The present work is entirely theoretical. It remains future work to try the `forthcoming` annotations and the new checking rules in practice. Two platforms are promising for such a study: Eclipse and Scala Bazaars [17, 18]. Eclipse, as previously discussed, is a very widely used platform with many components developed independently. Scala Bazaars is a code-sharing network for Scala users. Users share Scala code compiled to Java bytecodes, and the compiled libraries all too frequently become incompatible due to seemingly trivial changes in the inter-library interfaces.

The theoretical work is also not complete. There are still interface evolutions that are impossible to check with FJ-ADF, and in the future more transitions will be investigated, *e.g.* changes in the classes that are inherited. Also, inference for the wait set is still poorly understood, as described in Section 6.

Finally, a number of techniques complement evolution checking. Detection remains important: how do developers become aware that they are making an interface change? Interfaces themselves can be more flexible. For example, there could be a construct, analogous to `instanceof`, for dynamically testing whether an object implements a `forthcoming` method. Organizational questions arise as well. For example, is it helpful in practice to record a default "interface evolution rate" at the package level, or should every change have its own rate recorded, or should the tools avoid this question entirely?

## 10. Conclusion

Interface evolution is a recurring practical problem. This article investigates one technique, static checking for deprecation and anti-deprecation, which can make interface evolution more graceful. Even these simple method-level evolutions exhibit some subtlety, and a formal study of them brings out weaknesses in existing implementations.

This work provides but one piece of a general understanding of interface evolution. Checking tools can potentially check more than the method additions and removals discussed in this article. Furthermore, checking itself is just one tool in the toolbox for developers to address interface evolution.

## 11. Acknowledgments

The anonymous reviewers read the original submission very closely. Their many suggestions have greatly improved the presentation.

The members of LAMP provide a wonderful environment to nurture research.

The LCSD 2006 workshop participants gave much encouragement and helpful feedback on an earlier version of this work.

## References


[1] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *Proc. of Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[2] Douglas Bell. *Software Engineering: A Programming Approach*, chapter 6: Modularity. Addison Wesley, 3rd edition, 2005.

[3] Gilad Bracha, James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 3rd edition, 2005.

[4] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *Proc. of International Conference on Software Maintenance (ICSM)*, 1996.

[5] Jim des Rivières. Evolving Java-based APIs. `http://www.eclipse.org/eclipse/development/java-api-evolution.html`.

[6] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proc. of International Conference on Software Maintenance (ICSM)*, September 2005.

[7] ECMA. *ECMA-334: C# Language Specification*. European Association for Standardizing Information and Communication Systems (ECMA), second edition, December 2002.

[8] ECMA. *ECMA-367: Eiffel: Analysis, Design and Programming Language*. European Association for Standardizing Information and Communication Systems (ECMA), 2nd edition, June 2006.

[9] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, not objects. In *Companion to the ACM conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, New York, NY, USA, 2004. ACM Press.

[10] Johannes Henkel and Amer Diwan. Catchup! Capturing and replaying refactorings to support API evolution. In *Proc. of International Conference on Software Engineering (ICSE)*, 2005.

[11] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. of Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999.

[12] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In *Proc. of eXtreme Programming and Flexible Processes in Software Engineering (XP)*, 2000.

[13] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, 2004.

[14] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[15] Jeff H. Perkins. Automatically generating refactorings to support API evolution. In *Proc. of Program Analysis for Software Tools and Engineering (PASTE)*, September 2005.

[16] Scala web site. `http://scala.epfl.ch`.

[17] Scala Bazaars web site. `http://www.lexspoon.org/sbaz`.

[18] Alexander Spoon. Package universes: Which components are real candidates? Technical Report LAMP-REPORT-2006-002, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[19] S. Alexander Spoon. Anti-deprecation: Towards complete static checking for API evolution. In *Proc. of the Workshop on Library-Centered Software Design (LCSD 2006)*, Portland, Oregon, October 2006. ACM.

[20] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proc. of Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1996.

[21] Bill Venners. A conversation with Erich Gamma, part III. `http://www.artima.com/lejava/articles/designprinciples.html`, June 2005.

[22] Visual C# web page. `http://msdn.microsoft.com/vcsharp/`.