Anti-Deprecation: Towards Complete Static Checking for API Evolution

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. Anti-deprecation is a novel type-checking feature that allows static checking for more interface evolutions periods. Anti-deprecation, along with the more familiar deprecation, is formally studied as an extension to Featherweight Java. This formal study unearths weaknesses in two widely used deprecation checkers.

"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

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-scale 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 to progress to the next version of the interface. The same checker can be used for both purposes if the checker has two strictness levels. The *strict* level is used to check the exit from transition periods, while the looser *transitional* level is used for all other type-checking purposes.

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

LCSD 2006 Portland, Oregon, USA

```
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);

  encouraged public void connectionAuthenticated();
}
```

Figure 2: With encouraged methods, the new method could have been gradually phased into the original interface.

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.

Not all libraries need to be available to those maintaining the interface. The conditions for entering a transition period are typically weak, thus giving interface maintainers broad liberty to start an interface transition. Leaving the transition period requires more work, but it does not need to be finished immediately. Every library whose components use the interface must be checked with the strict checker, but those checks can occur throughout the transition period. Once the (loose) organization of library maintainers have decided that sufficient checking has occurred, and if no errors are known to be present, the transition period can be left.

Organization processes for deciding that enough library assemblies have been checked that a transition period may be left are beyond the scope of this article. Presumably, however, some such agreement has been reached among the library developers. As one example arrangement, the maintainers of the interface might commit to a minimum length of evolution period. That length might be *e.g.* six months, a year, or five years. Anyone building assemblies that use that interface must periodically check their library, with a period no longer than the agreed length of evolution periods.

A static transition checker can be described as having two modes: *transitional* and *strict*. If a library passes the transitional checker, then the library can communicate with other libraries through the interface. If a library additionally passes the strict checker, then the library will also continue to work if the interface is updated. The strict checker takes into consideration extra annotations describing the desired interface changes, while the transitional checker mostly ignores such annotations.

Implementations can combine the two checking modes. All code must pass the transitional checker, while failure to additionally pass the strict checker causes interface-evolution warnings.

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 encouraged methods, the designers would have had the option to phase in the method to the existing interface, as shown in Figure 2.

A simple way to annotate anti-deprecation is to add an `encouraged` keyword to the language. Unlike other methods, a method marked as `encouraged` cannot be called. Its presence only serves to mark that a future version of the interface will include that same method as `abstract`.

During transitional checking, `encouraged` methods are, for the most part, treated as if they were not present at all. The only restriction is that encouraged methods cannot override other non-encouraged methods. Allowing such would be complicated and unhelpful—after all, if a method is already present due to inheritance, what use is it to encourage it further? The one exception, that encouraged methods can nonetheless override other encouraged methods, is necessary so that encouraged methods can be added *over* other encouraged methods. In strict checking, even this case is not allowed, and the encouraged method deeper in the hierarchy needs to be removed.

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

Figure 3: Syntax of FJ-ADE

During strict checking, `encouraged` methods add several requirements for programs to pass the checker. First, any method that overrides an `encouraged` method must have the required parameter types and return type. This requirement is present so that when an encouraged method is later promoted to a required method, all methods overriding it will have conforming types. Second, every subclass of a class with an encouraged method must either implement the method or be considered abstract and uninstantiable.

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 first period introduces a new 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 deprecates the original method, thus inducing callers to use the new version of the method. The third period replaces the deprecated original method with an encouraged method of the desired signature. 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 `encouraged` 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 `encouraged` 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.

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 purpose serve as perfectly fine interfaces. The full extended language is called FJ-ADE because it is Featherweight Java with three new keywords: `abstract`, `deprecated`, and `encouraged`.

The notation is generally that of FJ. 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 $\neg P$, is not boolean negation, but instead means that P cannot be proven with the available inference rules.

The syntax of FJ-ADE 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 encouraged. An encouraged method will be added to a future version of the class with the specified type signature.
- Each class has a list of deprecated methods. Deprecated methods are going to be removed in a future version of the class.

Subtyping for FJ-ADE is shown in Figure 7. As in FJ, it exactly follows the class hierarchy.

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

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

$$\text{T-New}\ \frac{\begin{array}{c}\mathit{fields}(C) = \bar{D}\ \bar{f} \qquad \mathit{str}; \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C} <: \bar{D} \\ \neg \mathit{abstract}(C) \qquad (\mathit{str} = \texttt{trans}) \vee (\neg \mathit{postabs}(C))\end{array}}{\mathit{str}; \Gamma \vdash \texttt{new}\ C(\bar{e}) : C}$$

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

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

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

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

Figure 4: Typing of expressions

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

$$\text{T-Method-Over}\ \frac{\begin{array}{c}\mathit{str}; \bar{x} : \bar{C}, \texttt{this} : C \vdash e_0 : E_0 \qquad E_0 <: C_0 \\ \texttt{class}\ C\ \mathit{extends}\ D\ \{\ldots\} \\ \mathit{mtype}(m, D, (\mathit{str} = \texttt{strict}), \texttt{true}) = \bar{D} \to D_0 \\ \bar{C} = \bar{D} \qquad\qquad C_0 = D_0\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \{\ \texttt{return}\ e_0;\ \}\ \ \mathit{str}{-}\text{OK IN}\ C}$$

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

$$\text{T-Method-Enc}\ \frac{\begin{array}{c}\texttt{class}\ C\ \mathit{extends}\ D\ \{\ldots\} \\ \neg \mathit{mavail}(m, D, (\mathit{str} = \texttt{strict}), \texttt{true})\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \ \texttt{encouraged}\ \ \mathit{str}{-}\text{OK IN}\ C}$$

Figure 5: Typing of methods

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

Figure 6: Typing of classes

$$C <: C$$

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

Figure 7: Subtyping

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

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

Figure 8: Field lookup

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

The *fields* function, defined in Figure 8, computes the complete list of fields in a class.

The *mtype* function, defined in Figure 9, looks up the type of a method assuming it is invoked on a particular class. As compared to FJ, FJ-ADE's *mtype* function has two new flag parameters: one determining whether to include methods that are merely encouraged, and one determining whether to include methods that have been deprecated. While FJ's *mtype* considers all methods equally, FJ-ADE's *mtype* optionally declines to consider deprecated or encouraged methods or both, according to the two flags. Such methods are significant or not in different contexts in the type checker, and thus *mtype* must have extra parameters.

One particular complication is the treatment of deprecated methods when the fourth flag is `false`. In that case, *mtype* is still defined for that method if the chain of methods it overrides includes a non-deprecated method, and if all methods in that chain up to the non-deprecated method have the same type signature. The addition of this case means that core properties about *mtype* remain simple. See Lemma 1 and Lemma 2.

The *mavail* relation, also shown in Figure 9, claims that a method is available in a class without being specific about the method's type. Its arguments are the same as for *mtype*.

The *mbody* function, defined in Figure 10, is used during evaluation to find the method responding to a message-send expression. It is the same as in FJ except that there are two new clauses to support abstract and encouraged methods.

The *abstract* function, also defined in Figure 10, checks whether a class defines or inherits an abstract method. Note that this definition ignores `encouraged` methods, because

$$\text{MT-Here}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{X}\ \bar{M}\} \\ B\ m(\bar{B}\ \bar{x})\ MB \in \bar{M} \\ \mathit{enc} \vee (MB \neq \texttt{encouraged}) \\ \mathit{dep} \vee (m \notin \bar{X})\end{array}}{\mathit{mtype}(m, C, \mathit{enc}, \mathit{dep}) = \bar{B} \to B}$$

$$\text{MT-Inher}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{X}\ \bar{M}\} \\ m \notin \bar{M} \\ \mathit{mtype}(m, D, \mathit{enc}, \mathit{dep}) = \bar{B} \to B\end{array}}{\mathit{mtype}(m, C, \mathit{enc}, \mathit{dep}) = \bar{B} \to B}$$

$$\text{MT-Depover}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{X}\ \bar{M}\} \\ B\ m(\bar{B}\ \bar{x})\ MB \in \bar{M} \qquad m \in \bar{X} \\ \mathit{mtype}(m, D, \mathit{enc}, \texttt{false}) = \bar{B} \to B\end{array}}{\mathit{mtype}(m, C, \mathit{enc}, \texttt{false}) = \bar{B} \to B}$$

$$\frac{\mathit{mtype}(m, C, \mathit{enc}, \mathit{dep}) = \bar{D} \to D}{\mathit{mavail}(m, C, \mathit{enc}, \mathit{dep})}$$

Figure 9: Method type lookup

those methods are not yet available.

Post-abstract classes are those that might become abstract after the program evolves forward. The *postabs* function, defined in Figure 11, gives a conservative notion of post-abstract classes. It is defined in terms of a *postneeds* function which claims, more specifically, that the class might lack a particular method following either the removal of `deprecated` methods or the upgrading of `encouraged` methods to `abstract` or both. A class that *postneeds* any method at all is considered post-abstract.

The type checker of FJ needs to be updated in two ways for FJ-ADE. First, it needs to address the three new keywords. Second, it needs to have both a strict and transitional mode. An FJ-ADE typing judgement is written $\mathit{str}; \Gamma \vdash e : C$. As usual, Γ is a static typing environment, e is an expression, and C is a type (*i.e.*, a class). The *str* flag specifies whether to use strict type checking ($\mathit{str} = \texttt{strict}$) or transitional type checking ($\mathit{str} = \texttt{trans}$).

The typing rules for expressions are shown in Figure 4. Only two rules differ from FJ. First, the T-Invk judgement must specify the two extra parameters of *mtype*. The first argument is always `false`, because methods that are present merely for encouragement are not allowed to be invoked, not even in transitional mode. In transitional mode, encouraged methods might not be implemented yet. In strict mode they must be available, but they are left unavailable so that the strict checker does not admit any programs the transitional checker rejects. The second argument is `true` in transitional mode and `false` otherwise, because deprecated methods can be used only during transitional checking.

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

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

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

$$\text{MB-Enc}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{K\ \bar{X}\ \bar{M}\} \quad B\ m(\bar{B}\ \bar{x})\ \texttt{encouraged}\in \bar{M}}{mbody(m,C) = \texttt{encouraged}}$$

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

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

Figure 10: Method lookup

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

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

$$\text{PN-Enc}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{K\ \bar{X}\ \bar{M}\} \quad B\ m(\bar{B}\ \bar{x})\ \texttt{encouraged}\in \bar{M}}{postneeds(m,C)}$$

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

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

Figure 11: Post-abstract classes

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \quad \neg mavail(m,D,\texttt{false},\texttt{false})}{candep(m,C)}$$

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

Figure 12: Deprecated methods can only override other deprecated methods and potentially abstract methods.

methods as potential callees. In strict mode, T-New also disallows instantiating post-abstract classes. Post-abstract classes are not abstract now, but might become so after forthcoming interface changes.

The rules for typing methods are given in Figure 5. The main change from FJ is that, under strict typing, any method overriding an encouraged method must have the same signature that was encouraged. An additional change is that abstract methods and encouraged methods may only override encouraged methods. In principle abstract methods could be allowed in more places, but the complication provides no insight for the present purposes.

Finally, the rule for typing a class is given in Figure 6. The only difference from FJ is that the list of deprecated methods must be checked. The precise rule is given by the *candep* relation shown in Figure 12. Deprecated methods may not override concrete methods; they may only override deprecated, abstract, and encouraged methods.

It is not useful to have a deprecated method to override a concrete, non-deprecated method. Code can type check against the superclass with no deprecation warning, because the superclass's implementation is not deprecated. At run time, such code might actually invoke the deprecated method. In such a case, removing the deprecated method will mean the program's behavior changes.

If this behavior change is acceptable, and the overriding method does not need to be called, then that method should simply be removed outright. If the change is not acceptable, then either the method should be kept indefinitely, or the superclass's method should be deprecated so that no clients can call it.

That concludes the typing rules. The semantics of FJ-ADE, which are exactly the same as those of FJ, are shown in Figure 13.

5. PROPERTIES

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

The proofs contain no surprises, so they and some lemmas are omitted. The full proofs appear in an extended technical report [18].

5.1 Type soundness

There are three type-soundness properties worth dwelling on. The first two show that FJ-ADE is type sound in the usual sense: it enjoys both subject reduction and progress theorems. The last property is that strict checking implies transitional checking.

Theorem 1. *(Subject Reduction). Suppose CT is str-OK. If* $str;\Gamma \vdash e : C$ *and* $e \longrightarrow e'$*, then* $str;\Gamma \vdash e' : C'$ *for some* $C' <: C$*.*

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

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

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

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

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

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

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

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

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

Figure 13: Evaluation

mas.

The first lemma is that *mtype*'s last two arguments do not affect the type the function calculates, but only whether the function is defined or not. Further, changing one argument or both from `false` to `true` can only cause the function to change from undefined to defined, never from defined to undefined. That is, the truer the third and fourth arguments, the more often *mtype* is defined.

Lemma 1. *(Internal Consistency of mtype). Suppose dep, dep′, enc, and enc′ are four booleans such that enc ⇒ enc′ and dep ⇒ dep′. If mtype*$(C, m, enc, dep) = \bar{C} \to C_0$*, then mtype*$(C, m, enc', dep') = \bar{C} \to C_0$.

The following lemma shows that, roughly, once *mtype* returns a result at one point in the class hierarchy, it returns the same result deeper in the hierarchy under that point. Note, though, that this is only true so long as `encouraged` methods are ignored; during transitional checking, methods are allowed to change the type signature when they override a method that is merely `encouraged`.

Lemma 2. *(Subclasses and mtype). Suppose CT is str-OK and that mtype*$(m, D, \texttt{false}, dep) = \bar{C} \to C_0$*. For all* $C <: D$*, also mtype*$(m, C, \texttt{false}, dep) = \bar{C} \to C_0$.

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

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

The next lemma is modified from that for FJ by adding two arguments to the use of *mtype*. The choice of parameters—`false` and $(str = \texttt{trans})$—are those used by T-Invk.

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

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

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

As with FJ, several theorems follow immediately from Theorem 1 and Theorem 2. FJ-ADE 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.

Finally, strict type checking abides by its name: strict type checking is strictly more strict than transitional type checking.

Theorem 3. *(Strict Checking). Suppose CT is strict-OK. If* $\texttt{strict}; \Gamma \vdash e : C$*, then* $\texttt{trans}; \Gamma \vdash e : C$*. Further, CT is also trans-OK.*

5.2 Safe transitions

This section shows how to use the strict and transitional modes of FJ-ADE to evolve interfaces safely. There are two properties given which show when it is safe to add a deprecated or encouraged 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 encouraged methods to abstract 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.

The notation needs extra precision, because these properties all involve two programs. There are two versions of each relation and function, one for each program under discussion. To disambiguate between the two versions when it is not clear from context, the program is used as a subscript. For example, $\mathit{abstract}_{CT}(C_0)$ means that C_0 is abstract in program CT, and $str; \Gamma \vdash_{CT'} x : e$ means that x type checks in program CT' with checking mode *str*.

All of these properties discuss a single program being updated from one version to the next. However, as discussed in Section 2, the properties are carefully written to support updating single *classes* when that class is going to be used in many different programs.

Specifically, the two introduction theorems, require only transitional type checking plus properties of 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 can themselves be modified according to the introduction theorems without invalidating the requirements of the introduction theorems.

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 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 4. *(Deprecation Introduction). Let CT be any class table that is trans-OK, class A be a class in CT, and m be a method of class A. Suppose that if m overrides a method, then that method is either encouraged or deprecated,* i.e. *if A* `extends` *B then* $\neg mavail(m, B, \texttt{false}, \texttt{false})$. *Define CT′ as the same class table as CT except that m is deprecated in class A. Given these assumptions, whenever* $\texttt{trans}; \Gamma \vdash_{CT} e : C$, *it is also true that* $\texttt{trans}; \Gamma \vdash_{CT'} e : C$. *Further, CT′ is trans-OK.*

THEOREM 5. *(Encouragement Introduction). Let CT be any class table that is trans-OK, and let A be a class in CT which does not define or inherit a non-encouraged method named m,* i.e. *it is the case that* $\neg mavail(m, A, \texttt{false}, \texttt{true})$. *Define CT′ to be the the same class table as CT except that A has the following additional method definition:*

$$B\ m(\bar{B}\ \bar{x})\ \texttt{encouraged}$$

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

THEOREM 6. *(Deprecation Removal). Let CT be any class table that is strict-OK, and let A be a class in CT which defines a method named m that is deprecated. Define CT′ to be the same class table as CT except that m is removed from A. Then, whenever* $\texttt{strict}; \Gamma \vdash_{CT} e : C$, *it is also true that* $\texttt{strict}; \Gamma \vdash_{CT'} e : C$. *Furthermore, CT′ is strict-OK.*

THEOREM 7. *(Encouragement Upgrade). Let CT be a class table that is strict-OK, and let A be a class in CT which has the following method definition:*

$$B\ m(\bar{B}\ \bar{x})\ \texttt{encouraged}$$

Define CT′ to be the same class table except that the above method definition is replaced by this one:

$$B\ m(\bar{B}\ \bar{x})\ \texttt{abstract}$$

```
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 14: Removing a method can cause a class to become abstract. Instantiating such a class should cause a deprecation warning.

Then, whenever $\texttt{strict}; \Gamma \vdash_{CT} e : C$, $\texttt{strict}; \Gamma \vdash_{CT'} e : C$. *Furthermore, CT′ is strict-OK.*

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

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

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

THEOREM 11. *Let CT and CT′ be as in Theorem 7. If* $e \longrightarrow_{CT} e'$ *and* $\texttt{strict}; \Gamma \vdash e : C$, *then* $e \longrightarrow_{CT'} e'$.

6. WEAKNESSES IN CURRENT TOOLS

Today's practical deprecation checkers do not flag all code that can fail if a deprecated method is removed. Instead, they detect only direct accesses to deprecated features. This section examines four general categories of checks that a full transition checker should include. Along the way, this section examines the level of support of each category in Sun `javac` version 1.5.0_06 and Eclipse 3.2.

Method invocation

In strict mode, the T-INVK rule does not allow a message-send expression to invoke a method that is deprecated. Current checkers capture this familiar rule.

Post-abstract methods

In strict checking mode, the T-NEW rule does not allow instantiating a post-abstract class, *i.e.* a class that might be abstract after a transition phase is left. An example is given in Figure 14. Class `B` is not abstract currently, but it will be come abstract once the deprecated method `foo` is

```
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 15: Deprecating a method that overrides a concrete method can result in invariants being broken.

removed. Thus, while `B` is not abstract currently, it will be after its deprecated method is removed. A proper 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

Not all overrides of abstract, encouraged, and deprecated methods are allowed. Deprecated methods should only override other deprecated methods, abstract methods, and encouraged methods.

An example problem appears in Figure 15. The code in class `C` type checks by considering method `A.foo`, but at run time it invokes `B.foo`. If method `B.foo` 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.foo` should be removed instead of deprecated. Again, `javac` and Eclipse do not issue a warning for this code.

Encouraged methods and overriding

The requirements on encouraged methods are discussed in Section 3. Anti-deprecation is not supported at all in existing tools.

7. 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.

8. FUTURE WORK

The present work is entirely theoretical. It remains future work to try the `encouraged` annotations and the new checking rules in practice. Two platforms are promising for such a study: Eclipse and Scala Bazaars [17, 19]. 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. First, there are still interface evolutions that are impossible to check with FJ-ADE. For example, the checker does not support changes in constructor signatures nor changes in the classes that are inherited. It remains future work to investigate transition checking rules that are more general.

Additionally, this theory's checker produces a coarse result: either all changes may proceed, or none. Future work will check each change individually instead of having a bulk `strict` versus `trans` checking mode. One formalism that looks promising is to replace the checking mode with a *hold* set, where the *hold* set includes the set of changes which may not yet progress. If checking succeeds with a method left out of *hold*, then that one method may be updated even if the others are not ready. Given such a mechanism, new transition periods can begin while old ones are in the middle, without adding an additional obstacle to the old transition

period.

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: *e.g.* there could be a construct, analogous to `instanceof`, for dynamically testing whether an object implements an `encouraged` 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?

9. 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 the formal study brings out weaknesses in existing tools.

This work is only a beginning, though. Checking tools can potentially check more than method additions and removals. Furthermore, checking tools themselves are just one tool in the toolbox for developers to address interface evolution.

10. ACKNOWLEDGMENTS

Thank you to the anonymous reviewers. Your careful reading and feedback have made this article much more readable and relevant.

11. 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. Anti-deprecation: Towards complete static checking for api evolution (extended version). Technical Report LAMP-REPORT-2006-004, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

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

[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/`.