UpgradeJ: Incremental Typechecking for Class Upgrades*

Gavin Bierman
Microsoft Research Cambridge

Matthew Parkinson
University of Cambridge

James Noble
Victoria University of Wellington

September 29, 2015

Abstract

One of the problems facing developers is the constant evolution of components that are used to build applications. This evolution is typical of any multi-person or multi-site software project. How can we program in this environment? More precisely, how can language design address such evolution? In this paper we attack two significant issues that arise from constant component evolution: we propose language-level extensions that permit multiple, co-existing versions of classes and the ability to dynamically upgrade from one version of a class to another, whilst still maintaining type safety guarantees and requiring only lightweight extensions to the runtime infrastructure. We show how our extensions, whilst intuitive, provide a great deal of power by giving a number of examples. Given the subtlety of the problem, we formalize a core fragment of our language and prove a number of important safety properties.

1 Introduction

What's our big ticket item? Upgrades, people, upgrades. That's how we make the dough. ... Why be you, when you can be new?

Ratchet, in a speech to the Bigweld company board
Taken from Robots Movie, Twentieth Century Fox Animation.

Modern programming languages typically provide support for separate compilation and dynamic linking of components. This allows for code to be developed at multiple sites and shared across multiple applications, supporting code evolution and reuse. Programmers can build applications from these components, utilizing the runtime infrastructure to dynamically link in the components as required.

Experience has shown that this style of software construction is extremely fragile: because both context code and components evolve independently, there are few guarantees a program will actually "run anywhere"—or even typecheck—when linked dynamically against the motley collections of components found in most installed systems. There are many instances of this problem—commonly known as "DLL hell" or more recently "JAR hell"—servlet engines that depend on different, incompatible versions of XML libraries; web tools that rely on rendering engines from specific versions of open-source web browsers, so upgrading the browsers breaks the associated tools; language runtimes that depend on exact versions of ActiveX code support and so on.

*This is an extended and revised version of a paper that appears in ECOOP 2008

A number of solutions to this problem have been proposed, ranging from third-party tools, particular programming patterns, centralized management systems (e.g. RPM [4]), dynamic, reflective package infrastructures (e.g. OSGi [27]), to runtime architectural support (e.g. .NET and JVM). Most of these solutions are external to the application itself, and place a burden on the runtime infrastructure. Rather than solving the problem of evolving and incompatible programs and components, they just move it sideways, into tools, middleware, or external policies that allow flexible bindings but make few guarantees about the compatibility between a program and the components to which it may be bound.

In this paper, we aim to tackle the problem of program and component upgrading and evolution head-on, giving control to the programmer. Rather than having implicit rules about how programs can be bound, we make component versions explicit: every class and type in the program has a version number. We provide language support for upgrading classes in a variety of ways, and provide an assymetrical, incremental (but not iterative) type system that checks upgrades for consistency with the currently-running program. This enables us to be explicit about component compatibility; to give guarantees about which changes to classes are at least type safe (and which are not); and so to write code that is robust against multiple upgrades of the same component.

Having decided on language support for upgrading, an immediate question is at what level of granularity do we provide such support? Unfortunately, many issues concerning programming in the large are still being resolved for Java-like languages, e.g. witness the ongoing discussions on providing modules for Java [36]. In this paper we address upgrading in the small rather than in the large.

In any case, we argue that upgrading of classes is the *essence of the problem*—even if language support is eventually provided at some higher level, matters will still boil down to class definitions in Java-like languages. As we shall see, this is a highly non-trivial problem. The issues of correctness are subtle enough that we believe that a precise approach is essential, and required prior to any implementation or software engineering issues.

The conceptual contribution of this paper is embodied in the design of UpgradeJ, a Java-like language with support for type-safe dynamic class upgrading. We extend classes to have explicit version numbers, e.g.

```
class Button[1] extends Widget[1] {
  Font[1] font = new Font[1=]();
  Colour[2] colour = new Colour[3+]();
}
```

and types declare the versions of classes they will accept (the `font` field stores objects compatible with `Font` version `1`, while the `colour` field stores `Colour` instances compatible with version `2`). Then, `new` expressions also include version numbers with the class names, but in addition they include information about instances' upgradeability. Hence the new `Font` object instance will remain fixed at version `1`, whilst the new `Colour` object will be version `3` but may be upgraded later. (The exact behaviour of these annotations will be explained in more detail in §2 and formalized precisely in §4.)

Programmers can also request instances of the most evolved version of a particular class. For example, given:

```
Colour[3] latest = new Colour[3++]();
```

the instance actually stored in `latest` will be the most evolved version of `Colour` version `3` at object creation time. Moreover, it may be subsequently upgraded.

UpgradeJ then allows classes to be updated with newer versions dynamically. There are a number of ways that this could be supported; but for simplicity we model upgrading with

upgrade statements of the form: `upgrade`. When an upgrade statement is executed the program will be upgraded if any suitable upgrades are available.

Not all upgrades make sense, or can be supported trivially. The technical contribution of the paper is exactly how we enforce the safe upgrading of classes to be *incremental*—so that any class declaration is only ever typechecked once—whilst ensuring that an upgrade can never break the type safety of a running program.

Compared to some previous work, the focus of UpgradeJ is on what we call *class upgrading*: adding in new classes to a running system, and performing minor or major upgrades of existing classes. Unlike many other approaches, UpgradeJ does *not* perform any kind of object or instance upgrading. In other words we never alter a runtime object, just perhaps its behaviour. As a result, we expect that the features of UpgradeJ should be able to be implemented efficiently: each class or method definition is checked only once when first presented to the system; and UpgradeJ never requires any (expensive) traversals, inspections or bulk modifications of the heap. Indeed, our aim in this paper is to explore the design space of upgrading mechanisms that are strictly less powerful than object updates, although we argue in §5.4.4 that object updating could be implemented in UpgradeJ by combining class upgrades with a couple of reflective primitives. For similar reasons of practicality, we do not consider any kind of functional correctness between upgrades: we work only with types, and not with behavioural specifications.

The rest of the paper is organized as follows. We give an extensive, examples-driven introduction to the support for class upgrades in UpgradeJ in §2, beginning with support for class versions, then describing three different kinds of upgrades: *new class upgrades* that introduce new subclasses; *revision upgrades* that change the code of existing classes; and *evolution upgrades* that extend existing classes (but do not change existing instances). In §3 we consider a more realistic example and show how UpgradeJ can be used to dynamically upgrade a long-running server application. In §4 we give a precise definition of a featherweight version of UpgradeJ, FUJ, and define formally its type system and operational semantics. We can prove that FUJ is type-sound. The paper continues in §5 with a discussion of possible extensions to UpgradeJ that are not currently incorporated into the formalism.

2 An introduction to UpgradeJ

2.1 Explicit versions and new class upgrades

UpgradeJ extends Java syntactically by requiring all class names (other than `Object` by convention) to be annotated by a version number in square brackets after the class name.[1] For example:

```
class Button[1] extends Object {
 Object press() { ... }
}
class AnimatedButton[1] extends Button[1] {
 Object fancyPress() { ... this.press(); ... }
}
```

UpgradeJ programs can include *upgrade statements*, written `upgrade`. When an upgrade statement is executed, the program waits to receive an upgrade (this could be via a command prompt or from a file). The upgrade is typechecked and if correct is applied to the program. Having explicit upgrade statements allows programmers to control the timing of upgrades to the application. UpgradeJ supports three forms of upgrade that we shall now discuss in turn.

[1] One can imagine tool support that would alleviate the burden of writing version numbers. This is discussed further in §5.

The simplest form of upgrade supported by UpgradeJ is called a *new class upgrade*. It allows new class definitions to be added (at runtime) to the class table. For clarity, a new class upgrade is written as a class definition prefixed with the keyword `new`. To differentiate upgrades from standard code in this paper, we present them in a shaded box. For example, in the example above the class `AnimatedButton[1]` could have been defined via a new class upgrade as follows:

```
new class AnimatedButton[1] extends Button[1] {
 Object fancyPress() { ... this.press(); ... }
}
```

UpgradeJ will typecheck the upgrade in the context of the current program state: if the tests pass, then the current program is upgraded to include the new definitions. An important design feature of UpgradeJ is that typechecking of upgrades is *incremental*, that is, only the new definitions in the upgrade are typechecked. Old definitions are never re-checked: the typechecker will check the correctness of each class definition only once (either when supplied as part of the initial program, or when it arrives as an upgrade).

At this point there is no way an UpgradeJ program can *use* any classes introduced by a new class upgrade: references from old classes to new classes will fail because the old classes will have been typechecked before the upgrades arrive: we call this the "*no time travel*" principle. As we shall see later, new class upgrades are still very useful as they allow new code to be installed; other upgrade forms will allow this code to be put to work.

2.2 Revision upgrades

Returning to our simple button example, let's imagine that `Button` has also a method `bgColour` which returns the colour of their background. For example, the first version of `Button` was clearly written around 1990:

```
class Button[1] extends Object {
 Object press(){ ... }
 Colour bgColour() { return new BeigeColour[1](); }
}
```

By the mid-90s, these buttons have begun to look dated. In UpgradeJ we can use a *revision upgrade* to provide a revision of an existing class to fix this problem. The revision upgrade is written as follows:[2]

```
new class Button[2] extends Object revises Button[1]{
 Object press(){...}
 Colour bgColour() { return new GreyColour[1](); }
}
```

To allow upgrades to affect running programs, we provide new forms of instantiation. As in Java, objects are created by calling `new`, however in UpgradeJ programmers must supply both a version number for the class *and* an annotation of either '`+`' to denote an upgradeable instance or '`=`' to denote a non-upgradeable (or exact) instance. For example, `new Button[2=]()` creates a new instance of `Button[2]`, the `=` ensures that the object will have the *exact* version `2` (in other words, if the `Button` class is subsequently upgraded this instance is insensitive to those upgrades). By contrast, upgradeable objects take advantage of all revisions as soon as they are supplied: after a revision upgrade, any methods sent to an upgradeable object will execute the revised method definitions.

[2] Actually, this is sugar for two primitive UpgradeJ upgrades: first, a new class upgrade introducing the class `Button[2]`, and second, a *revises statement* `Button[2] revises Button[1]`. Our formalization in §4 uses these primitive forms.

For example, we can create two instances of `Button[1]`, one exact and one upgradeable, both of which will have a beige background. Then, we can execute an `upgrade` statement (whose effect is to revise `Button[1]` to `Button[2]` as above), and ask each button for its `bgColour`. The exact button object will still return a `BeigeColour` instance, while the upgradeable button will return `GreyColour`.

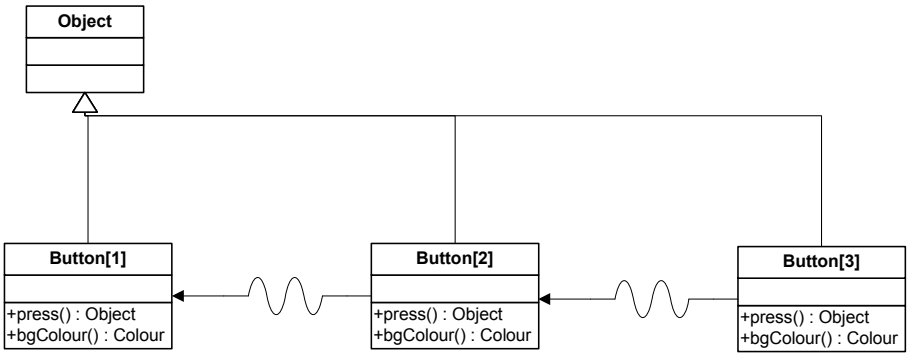
```
Button[1] x = new Button[1=](); // exact
Button[1] u = new Button[1+](); // upgradeable
x.bgColour(); // returns BeigeColour
u.bgColour(); // returns BeigeColour

upgrade; // Button[2] revises Button[1]

x.bgColour(); // returns BeigeColour
u.bgColour(); // returns GreyColour
```

One point to note here is that the *types* of the variables storing the buttons are the same — both are just `Button[1]`. This is because every class introduced as a revision upgrade, just as every class introduced as a new class upgrade, is a *subtype* of the class being upgraded. A type like `Button[1]` will accept any `Button[1]` (as per usual); any subclass of `Button[1]` (defined either in the initial program or supplied via a new class upgrade); and any other upgrade of `Button[1]`.[3] We discuss supporting exact annotations on types in §2.6.

As fashions change, we can upgrade again:

```
new class Button[3] extends Object revises Button[2]{
 Object press(){ ... }
 Colour bgColour() { return new TransparentAquaColour[1](); }
}
```

Multiple upgrades can be hard to follow, so we draw class diagrams showing version numbers explicitly, and revision relationships with a wavy arrow. The three versions of the `Button` class that we have defined so far are shown as follows:



To support the dynamic behaviour of upgradeable objects, however, UpgradeJ must place some restrictions on the bodies of revision upgrades: the classes must have the same name, the upgrade cannot revise a class that has already been revised, and (most importantly) the resulting revised class must have *exactly the same fields and method signature as the class it is revising*, and implement every interface. By the method signature of a class, we mean all the methods and their types that are understood by objects of that class, including inherited methods. Hence, the methods themselves need not reside in the same class; this allows for *refactoring* by upgrades (see §2.5).

So, for example,

[3] UpgradeJ supports explicit syntax for this. In fact, `Button[1]` is shorthand for `Button[1+]`.

```
new class Button[4] extends Object revises Button[2] { ... }
```

is an invalid upgrade if `Button[2]` has already been revised to `Button[3]`; and

```
new class Button[5] extends Object revises Button[3] {
 ...
 Integer transparency;
 Integer setTransparency(Integer t){...}
 ...
}
```

is invalid because it includes a new field and a new method to the `Button` class.

The restrictions on version numbers and names are primarily there to make the type names consistent. The linear ordering on revisions (only the latest revision can itself be revised) is important to support upgradeable objects: there is a simple, nonbranching sequence of revisions, the *latest* revision of a class is always obvious, and so it's clear which methods an upgradeable object should run.

The restriction that the resulting revised class must have exactly the same fields and method signature means that revised classes can change method bodies, and omit or override methods declared concretely in ancestor classes. This restriction is necessary to support the incremental nature of UpgradeJ, and to avoid any heap inspection. A revision cannot add (or remove) fields from an object, because that would require the heap representation of every upgradeable object to be changed. Methods cannot be added into a class because they cannot be checked incrementally. We do not expect these restrictions to be too arduous in practice because they reflect the intent of revision upgrades: to *revise* an existing class, not to introduce new functionality.

2.3 Evolution upgrades

New class upgrades allow new fields and methods to be defined, but require a new class to be created: existing instances cannot take advantage of the upgrade. On the other hand, revision upgrades take immediate effect across all upgradeable instances of the class, but cannot add fields and methods. The final type of upgrade supported by UpgradeJ is the *evolution upgrade* that is, in some sense, a combination of the other two upgrade forms.

Evolution upgrades may add new methods and fields, but do not update existing objects. Rather, evolution upgrades are supported by another form of `new`, written `new C[v++]()` that creates an upgradeable object of the *latest evolution* of a class—in effect, doing a dynamic dispatch from a class to its most recent evolution upgrade.

Returning to our simple button example, we can add "2007" design and animation features to the button class with an evolution upgrade:[4]

```
new class Button[6] extends Object evolves Button[3]{
 Integer animationRate;
 void tick() {this.redraw(); }
 Colour bgColour() { return new VistaBlackColour[1](); }
 ...
}
```

Writing `new Button[1++]` will create a new instance of the *latest revision of the latest evolution upgrade* of the `Button[1]` class.

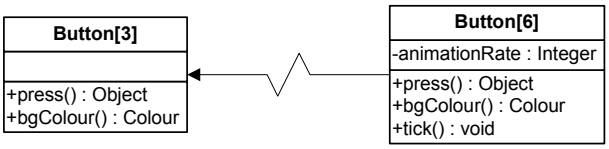
[4] Again, we use some syntactic sugar: this evolution upgrade can be decomposed into a new class upgrade and an *evolves statement* (in this case `Button[6] evolves Button[3]`).

```
Button[1] e = new Button[1++]();
e.bgColour(); // returns TransparentAquaColour

upgrade; // Button[6] evolves Button[3]
e.bgColour(); // returns TransparentAquaColour

e = new Button[1++](); // latest creation
                       // now Button[6] is the latest kind of button
e.bgColour(); // returns VistaBlackColour
```

Note that this example demonstrates that, unlike revision upgrades, evolution upgrades do not upgrade the behaviour of existing instances. As with other upgrades, the types of the variables do not need to change; every upgrade is still a subtype of its target class; a variable at version n will be compatible with every subsequent version of that class.

There are restrictions on evolution upgrades. Whereas revision upgrades must preserve the same fields and method signatures of the revised class, evolution upgrades can extend both. Thus the new version of the class must include the fields and method signatures of the old version, but it can add new fields and new methods.

We also introduce a diagrammatic form for evolution upgrades. We introduce an evolution relationship between classes which is denoted using a "sawtooth" arrow (this is intended to symbolize the breaking change possible with an evolution upgrade). For example:



2.4 Revision, Evolution, and Inheritance

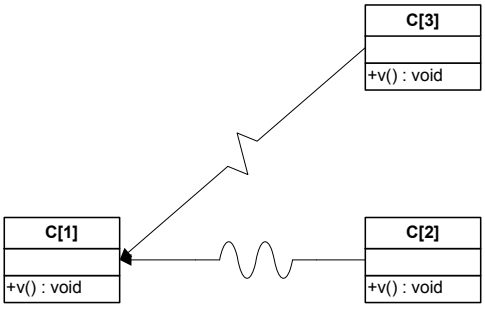
UpgradeJ has three different relationships between classes: the traditional inheritance relationship (that can be extended with new class upgrades), plus the revision and evolution relationships introduced to support upgrades. How do these relationships interact?

First, UpgradeJ permits a single class to have both revision and evolution upgrades. For example, consider the following definitions, where `C[1]` is revised by `C[2]` and, in addition, evolved into `C[3]`:

```
class C[1] {
  void v() { print "one"; }
}
```

```
new class C[2] revises C[1] {
  void v() { print "two"; }
}
new class C[3] evolves C[1] {
  void v() { print "three"; }
}
```

giving the following class structure:



There are three forms of object creation in UpgradeJ: (1) exact creation giving a fixed object; (2) creating an upgradeable object (that follows the `revises` relationship); and (3) creating an upgradeable object of the latest version (that follows both the `evolves` relationship and then the `revises` relationship):

```
new C[1=]().v();  // outputs "one"
new C[1+]().v();  // outputs "two"
new C[1++]().v(); // outputs "three"
```

Second, inheritance (the `extends` relationship) interacts quite straightforwardly with upgrades. Message sends to upgradeable objects always take account of revision upgrades (while sends to exact objects always ignore them) so upgradeable objects also see revisions to their superclasses:

```
new class D[2] extends C[1] {}
```

```
new D[2=]().v(); // outputs "one"
new D[2+]().v(); // outputs "two"
```

while new class and evolution upgrades will only affect message sending if instances of their classes are involved directly.

2.5 Refactoring

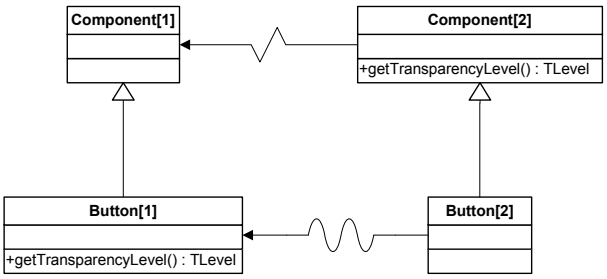
As revision upgrades are required to preserve only the fields and method *signatures* of the classes they revise, we can move methods around the hierarchy using a combination of revision and evolution or new class upgrades. The key here is that provided a revised class has the same signatures and fields as the target class it is revising, the two classes need have no other relationship. Given a couple of simple classes:

```
class Component[1] { }
class Button[1] extends Component[1] {
 TLevel getTransparencyLevel(){...}
}
```

We can evolve the `Component` superclass to define a `getTransparencyLevel()` method, and then *revise* the `Button` subclass to inherit that method from the new superclass, and so *removing the method from the subclass.*

```
new class Component[2] evolves Component[1] {
   TLevel getTransparencyLevel(){...}
}

new class Button[2] extends Component[2] revises Button[1] { }
```

The resulting structure is shown below: note that, because we use a revision upgrade upon the Buttons, this change will apply dynamically to every upgradeable instance of `Button[1]`.



2.6 Exact version types

Sometimes programmers will need to restrict their code to a particular version of a class. For example, for historical or aesthetic reasons, a programmer might wish to write a `BeigeWindow` class that only uses the first beige version for its Buttons:

```
class BeigeWindow[1] extends Window {
  Button[1=] okButton, cancelButton;
  ...
}
```

To prevent the fields `okButton` and `cancelButton` from receiving more recent versions, we use *exact version types* declared with an "`=`" modifier.

An exact version type is compatible with only one (or a list of) explicit version(s): versions of the type outside that list are not subtypes of the exact version type. (The relationship between exact version types and version types is the same as that between exact types and subtypes in object-oriented languages [8]). An exact version type is assignable only from another exact type or an exact object creation. Hence, `okButton = new Button[2]();` would fail to typecheck, as `Button[2]` is *not* a subtype of the exact version type `Button[1=]`, but only of the non-exact type `Button[1]`.

In fact, our UpgradeJ language design supports a list of versions in exact types, and also allows that list to be open ended. For example, if it was known that the `Button[3]` upgrade introduced a bug that was subsequently removed by the `Button[5]` upgrade, a variable `safeButton` could be declared as: `Button[1=,2=,5] safeButton;` permitting exact `Button` objects of versions `1` and `2`, and any exact or upgradeable `Button`s of version `5` or above to be stored in the variable but not the buggy `Button[3]` or `Button[4]`.

Exact types, even more than exact objects, reduce the flexibility of software which uses them: we expect that they would be used sparingly, primarily to avoid bugs in particular versions of components. This is why the default for type declarations is that the types are upgradeable, and why exact types require the "`=`" annotation. Nevertheless, we expect there will be situations where programmers will demand that only a particular version of a component is used to build a system, and exact version types provide this guarantee.

2.7 Summary

UpgradeJ introduces a number of novel features to Java-like programming languages: explicit versions of classes, fixed version and upgradeable version objects, an upgrade statement, new class, revision, and evolution upgrades, and exact version types.

The following table summarizes the relationships between the main features of UpgradeJ: the kinds of upgrades versus the kinds of objects and constructor calls.

		Upgrade Type		
		New Class	Revision	Evolution
Class Definitions:	Redefine existing method bodies	N/A	yes	yes
	Add new fields	yes	no	yes
	Add new methods	yes	no	yes
Creation:	Exact `new C[1=]`	no	no	no
	Upgradeable `new C[1+]`	no	yes	no
	Latest `new C[1++]`	no	yes	yes
Method invocation:	Exact `T[1=,2=]`	no	no	no
	Upgradeable `T[1]`	no	yes	no

Revision and evolution upgrades may redefine methods (give new method bodies such that the resulting flattened class signature does not change), while only new class and evolution upgrades may declare new fields or methods. Creating an exact object "`new C[1=]`" sees no upgrades, while creating an upgradeable object "`new C[1+]`" sees revision upgrades and using latest creation "`new C[1++]`" creates an instance of the most recent revision of the most recent extension. Methods sent to exact objects again see no revisions, while methods sent to upgradeable objects see revision updates.

Finally (not shown in the table), exact version types are subtypes only of the exact versions given in the type, while all subsequent versions of a type (both exact and not-exact) are subtypes of earlier non-exact versions of that type.

3 Example: Upgrading a server application

In this section we present a fairly realistic example of the upgrading of a long-lived server application. This example first appeared in a functional programming setting in work on dynamic software updating by Bierman et al. [5] (although updates in that setting required a run-time typecheck of the entire program state). To make the example simpler, we ignore the issues of concurrency and assume a single-threaded, event-based software architecture. In order to save space, we also only give the essential code fragments to illustrate our point, rather than giving a full program.

3.1 Initial system

The code for our server is given below. The key class is `Server` which contains a private field, `myQ`, containing a queue of events, e.g. HTTP requests from clients or responses from handlers. (We do not give details of the `Queue` class for lack of space.) New events are created by the `NewEvent` method, which either enqueues the event and returns, or blocks if the queue is empty and no new events have occured. Once an event occurs, then it is removed from the head of the queue using the `remove` method. All events extend the `Event` class, which specifies a `handle` method. We assume for now just two events; `get_Event` and `upgrade_Event` (the programmer has forgotten about put events; this will be added later). The `upgrade_Event` simply executes an upgrade statement and leaves the event queue untouched. This enables the server to be upgraded. Note that after this upgrade has taken place, the next statement is the recursive call to `loop`, i.e. no remaining computation exists at this point.

```
class Event[1]{
 void handle (Queue[1] q);
}
class get_Event[1] extends Event[1]{
 void handle (Queue[1] q){ ... }
}
```

```
class upgrade_Event[1] extends Event[1]{
 void handle (Queue[1] q){ upgrade; }
}
class Server[1]{
 Queue[1] myQ = new Queue[1]();
 void newEvent(){...}
 void loop(){
  newEvent();                          // Enqueues new event
  Event[1] e = (Event[1])myQ.remove(); // remove head of myQ
  e.handle(myQ);
  loop();
 }
}
```

The code for the `main` method of our application then simply creates an *upgradeable* instance of the `Server` class and invokes its `loop` method, as follows.

```
Server[1] s = new Server[1+](); // Upgradeable object!
s.loop();                       // Do the work
```

3.2 First upgrade: Handling put events

As mentioned earlier, the programmer has forgotten about put events. These are easy to add to the system *dynamically* using new class and revision upgrades. First, we use a new class upgrade (by sending an upgrade event to the server) to add the new class `put_Event`.

```
new class put_Event[1] extends Event[1]{
     void handle (Queue[1] q){ ... }
}
```

We will also need to change the code of the `newEvent` method, as it will need to create instances of the `put_Event` class. As the signature of this method will be unchanged, this can be captured by a revision upgrade. We add the new revised class `Server[2]` which is identical to `Server[1]` save for the new code in the `newEvent` method.

```
new class Server[2] revises Server[1] {
     Queue[1] myQ = new Queue[1]();
     void newEvent(){ ... } //NEW CODE!
     void loop(){
      newEvent(); //Enqueues new event
      Event[1] e = (Event[1])myQ.remove();
                        // remove head of myQ
      e.handle(myQ);
      loop();
     }
}
```

Now the original instance of `Server[1]` will invoke the new code for the `newEvent` method the next time it enters `loop`.

3.3 Second upgrade: Adding a log

Now we consider a much more disruptive upgrade to our system: adding a log to the server and requiring that all events update the log when they are handled. First we need to change the `Event` class as follows (the classes `get_Event`, `put_Event` and `upgrade_Event` must also be changed accordingly).

```
new class Event[2] evolves Event[1] {
     void handle (Queue[1] q, Log[1] l);
}
```

Note that this is an *evolution* upgrade: the signature of the events has changed. We also need an evolution upgrade to the `Server` class, as follows.

```
new class Server[3] evolves Server[2] {
     Queue[1] myQ = new Queue[1]();
     Log[1]  myLog = new Log[1]();
     void newEvent(){ ... }
     void handOver(Queue[1] q){ myQ=q; loop(); }
     void logv1Event(Event[1] e){ ... }
     void loop(){
      newEvent(); //Enqueues new event
      Object e = myQ.remove();
      if (e instanceof Event[2])
        (Event[2])e.handle(myQ,myLog);
      else {
        e.handle(myQ);
        logv1Event(e);
      }
      loop();
     }
}
```

This new version of the `Server` class has a new field, `myLog` to contain the system log. It also contains a new method, `logv1Event` to enable the logging of a `Event[1]` object. The body of the `loop` method is similar except that we now need to inspect each event to see if it can log itself or not. The `handOver` method will be more apparent after the next revision upgrade. (Note here the use of co-existing revision and evolution upgrades.)

```
new class Server[2.1] revises Server[2]{
     Queue[1] myQ = new Queue[1]();
     void newEvent(){ ... }
     void loop(){
      Server[3] s = new Server[3+]();
      s.handOver(myQ);
     }
}
```

This *revision* upgrade of `Server[2]` changes only the body of the `loop` method. Recall that after the upgrade event the next call is to the latest revision of the `loop` method. Hence our original (version `1`) instance of `Server` will invoke the `loop` method defined in version `2.1`. This `loop` method now simply creates a fresh `Server[3]` object and invokes its `handOver` method. The `handOver` method accepts the state from the old `Server` object and executes the `loop` method of the `Server[3]` object. Hence we have elegantly transitioned from an old to a new version of the server *at runtime*, whilst both maintaining the state and guaranteeing type safety!

4 Formalizing UpgradeJ

Featherweight UpgradeJ (FUJ) is to UpgradeJ what other core calculi such as FJ [19] and MJ [6] are to Java. It is a small, but expressive subset of the language that is used to verify formal properties of the language. FUJ is slightly unusual in that it has an extremely compact form,

which facilitates a very simple operational semantics, however, it is as expressive as more familiar core calculi. It is important to note that FUJ programs are syntactically correct UpgradeJ programs.

4.1 Syntax

The syntax of FUJ class definitions, types, field and method definitions, and statements is defined as follows.

`T, S, U ::=`	Type
`C[v1=,...,vn=]`	Version list type ($n \geq 1$)
`C[v+]`	Version range type
`K, J, I ::= C[v=]`	Exact version type
`R ::=`	Runtime type
`C[v=]`	Exact type
`C[v+]`	Upgradeable type
`L ::= class I extends J{ T̄ f̄; M̄ }`	Class definitions
`M ::= S m(T̄ x̄){ B return y; }`	Method definition
`B ::= T̄ x̄; s̄`	Method block
`t, s ::=`	Statement
`x = y;`	Assignment
`x = y.f;`	Field access
`x.f = y;`	Field update
`x = y.m(z̄);`	Method invocation
`x = (T)y`	Cast
`if(x == y){s̄} else {t̄}`	Equality test
`if(x instanceof I){s̄} else {t̄}`	Instance test
`x = new C[v=]();`	Object creation
`x = new C[v+]();`	Object creation
`x = new C[v++]();`	Object creation
`x = new Object();`	`Object` creation
`upgrade;`	Upgrading
`Z ::=`	Upgrade definitions
`new L`	New class upgrade
`C[v=] revises C[w=]`	Revision upgrade
`C[v=] evolves C[w=]`	Evolution upgrade

In the syntax rules we assume a number of metavariables: `f` ranges over field names, `C` over class names, `m` over method names, `v`, `w` over versions,[5] and `x`, `y`, `z` over program variables. We assume that the set of program variables includes a designated variable `this`, which cannot be used as an argument to a method. We follow FJ and use an 'overbar' notation to denote sequences.

FUJ types are ranged over by `S, T, U` and can be either an exact version type, of the form `C[v=]`, or a version list type, written `C[v1=,...,vn=]`, or a version range type, written `C[v+]`. To simplify some definitions we use the metavariables `I` and `J` to range over exact version types. As with FJ, for simplicity we do not include any primitive types in FUJ. In FUJ there is a special

[5] Purely for presentational simplicity, versions are restricted to be integers.

exact version class `Object[1=]` which we abbreviate to `Object`. We do *not* allow this class to be revised or evolved, so it remains the root of the inheritance hierarchy.

A FUJ class definition, `L`, contains a collection of field and method definitions. For simplicity, in this paper we shall not consider constructor methods; they do not complicate the treatment of versioning and we simply model that fields are initialised to `null`. A field is defined by a type and a name. A method definition, `M`, is defined by a return type, a method name, an ordered list of arguments—where an argument is a variable name and a type—a method block, `B`, and a return statement.

The real economy of FUJ is that we do not have any syntactic forms for expressions (or even promotable expressions [6]), and that the forms for statements are syntactically restricted. All expression forms appear only on the right-hand side of assignments. Moreover expressions only ever involve variables. In this respect, our form for statements is reminiscent of the A-normal form for λ-terms [17]. A statement, `s`, is either an assignment, a field access, a field update, a method invocation, a cast, an instance conditional, an object creation, or an upgrade statement. In spite of the heavy syntactic restrictions, we have not lost any expressivity; it is quite simple to translate FJ or MJ programs into FUJ. Another advantage of our approach is that we have no need for the 'stupid' rules of FJ.

In FUJ we assume a rather large amount of syntactic regularity to make the definitions compact. All class definitions must (1) include a supertype; (2) start with all the declarations of the variables local to the method (hence a method block is a sequence of local variable declarations, followed by a sequence of statements); (3) have a `return` statement at the end of every method; and (4) write out field accesses explicitly, even when the receiver is `this`.

A FUJ upgrade is either a new class upgrade (which consists of a class definition prefixed with a `new` modifier) or a revision upgrade (which is of the form `I revises J`) or an evolution upgrade (which is of the form `I evolves J`).

4.2 Class Tables and Subclassing

Following FJ, we take an FUJ program to be a pair $(CT, \mathtt{B})$ of a class table CT and a method block `B`. This method block corresponds to the `main` method. As we are interested in upgrading the class table it cannot be assumed to be fixed and implicit as in FJ.

A FUJ class table, CT, is a triple $\langle \mathcal{C}, \textbf{revises}, \textbf{evolves} \rangle$. The first component is a map from exact version types to class definitions. The second and third are relations between exact version types. We use some shorthand and write $CT \vdash \mathtt{I} \triangleq \mathtt{\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}}$ and $CT \vdash \mathtt{I}$ **extends** `J` where $\mathcal{C}(\mathtt{I}) = \mathtt{class\ I\ extends\ J\{}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{\}}$. We also write $\mathtt{I} \in dom(CT)$ to mean $\mathtt{I} \in dom(\mathcal{C})$. We write $CT \vdash \mathtt{I}$ **revises** `J` when $(\mathtt{I}, \mathtt{J}) \in$ **revises**, and similarly $CT \vdash \mathtt{I}$ **evolves** `J` when $(\mathtt{I}, \mathtt{J}) \in$ **evolves**. The **revises** and **evolves** relations are initially empty and are incremented by the action of upgrade definitions.

By looking at a class table, we can read off a subtype relation between types. We write $CT \vdash \mathtt{S} <: \mathtt{T}$ when `S` is a subtype of `T` given the class table CT. This relation is slightly more complicated than for FJ because we have three relations between types (extends, revises and evolves) and also support version list and version range types. The rules for forming valid subtyping judgements are defined as follows.

$$\frac{\texttt{v=} \in \overline{\texttt{w}}}{CT \vdash \texttt{C[v=]} <: \texttt{C}[\overline{\texttt{w}}]}\ \text{[ST-In]} \qquad \frac{CT \vdash \texttt{C[v=]}\ \textbf{revises}\ \texttt{C[w=]}}{CT \vdash \texttt{C[v=]} <: \texttt{C[w+]}}\ \text{[ST-RevRng]}$$

$$\frac{CT \vdash \texttt{C[v=]}\ \textbf{evolves}\ \texttt{C[w=]}}{CT \vdash \texttt{C[v=]} <: \texttt{C[w+]}}\ \text{[ST-EvRng]} \qquad \frac{CT \vdash \texttt{S} <: \texttt{T} \quad CT \vdash \texttt{T} <: \texttt{U}}{CT \vdash \texttt{S} <: \texttt{U}}\ \text{[ST-Trans]}$$

$$\frac{CT \vdash \texttt{C[v=]}\ \textbf{revises}\ \texttt{C[w=]}}{CT \vdash \texttt{C[v+]} <: \texttt{C[w+]}}\ \text{[ST-RngRng1]} \qquad \frac{CT \vdash \texttt{C[v=]}\ \textbf{evolves}\ \texttt{C[w=]}}{CT \vdash \texttt{C[v+]} <: \texttt{C[w+]}}\ \text{[ST-RngRng2]}$$

$$\frac{CT \vdash \texttt{C[v=]}\ \textbf{extends}\ \texttt{I}}{CT \vdash \texttt{C[v+]} <: \texttt{I}}\ \text{[ST-Rng]} \qquad \frac{CT \vdash \texttt{C[v=]}\ \textbf{extends}\ \texttt{I}}{CT \vdash \texttt{C[v=]} <: \texttt{I}}\ \text{[ST-Ex]}$$

$$\frac{CT \vdash \texttt{C}[\texttt{v}_1\texttt{=}] <: \texttt{T} \cdots CT \vdash \texttt{C}[\texttt{v}_n\texttt{=}] <: \texttt{T}}{CT \vdash \texttt{C}[\texttt{v}_1\texttt{=}, \ldots, \texttt{v}_n\texttt{=}] <: \texttt{T}}\ \text{[ST-List]}$$

4.3 Correctness conditions

Unlike in normal fragments of Java where correctness conditions on the class table are so routine that they are traditionally omitted [19], they are *essential* in formalizing UpgradeJ. In some senses they are the very essence of UpgradeJ as the class table can be changed at runtime and all upgrade checks are made with reference to the class table. In other words an upgrade should not be able to compromise type safety.

The first correctness condition we impose is a well-formedness property on the three relations in the class table.

Definition 1. $\vdash CT$ *wfr iff*

1. $\forall \texttt{S}, \texttt{T}$. *If* $CT \vdash \texttt{S} <: \texttt{T}$ *and* $CT \vdash \texttt{T} <: \texttt{S}$ *then* $\texttt{T} = \texttt{S}$,
2. $\forall \texttt{K}, \texttt{I}, \texttt{J}$. *If* $CT \vdash \texttt{K}$ **extends** $\texttt{I}$ *and* $CT \vdash \texttt{K}$ **extends** $\texttt{J}$ *then* $\texttt{I} = \texttt{J}$, $\texttt{I} \in dom(CT)$ *and* $\texttt{K} \in dom(CT)$,
3. $\forall \texttt{K}, \texttt{I}, \texttt{J}$. *If* $CT \vdash \texttt{I}$ **revises** $\texttt{K}$ *and* $CT \vdash \texttt{J}$ **revises** $\texttt{K}$ *then* $\texttt{I} = \texttt{J}$, $\texttt{I} \in dom(CT)$ *and* $\texttt{K} \in dom(CT)$, *and*
4. $\forall \texttt{K}, \texttt{I}, \texttt{J}$. *If* $CT \vdash \texttt{I}$ **evolves** $\texttt{K}$ *and* $CT \vdash \texttt{J}$ **evolves** $\texttt{K}$ *then* $\texttt{I} = \texttt{J}$, $\texttt{I} \in dom(CT)$ *and* $\texttt{K} \in dom(CT)$.

Condition (1) ensures that the subtyping relation induced by the class table does not include cycles. Condition (2) reflects the fact that UpgradeJ supports only single inheritance. Analogously, UpgradeJ only supports single revision (3) and single evolution (4). Note that this does not preclude a class being both revised *and* evolved.

The next two correctness conditions we impose are on the **revises** and **evolves** relations.

Definition 2.

1. $CT \vdash \texttt{J}$ **revises** $\texttt{I}$ ok *iff* $\mathit{fields}(CT, \texttt{I}) = \mathit{fields}(CT, \texttt{J})$ *and* $\mathit{methSig}(CT, \texttt{I}) = \mathit{methSig}(CT, \texttt{J})$
2. $CT \vdash \texttt{J}$ **evolves** $\texttt{I}$ ok *iff* $\mathit{fields}(CT, \texttt{I}) \subseteq \mathit{fields}(CT, \texttt{J})$ *and* $\mathit{methSig}(CT, \texttt{I}) \subseteq \mathit{methSig}(CT, \texttt{J})$

(The auxiliary functions *fields* and *methSig* are defined in Figure 1.)

These correctness conditions for the upgrade relations formalize the discussions of §2. Thus a class J revises a class I if (i) the fields are identical, and (ii) the *method signatures* are identical. Notice that this does *not* force class J itself to have the same methods as class I; just that they support the same methods (possibly inherited from other classes). This allows us to support the refactoring upgrades described in §2.

The correctness rule for an evolution upgrade is similar but more permissive, as it allows the evolved class to have more fields and a larger method signature.

We can now give the overall correctness condition for a class table.

Definition 3. $\vdash CT = \langle \mathcal{C}, \textbf{revises}, \textbf{evolves} \rangle$ ok *iff*

1. $\vdash CT$ *wfr,*
2. $\forall \mathtt{I} \in dom(\mathcal{C}).CT \vdash \mathtt{I}$ ok*,*
3. $\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \textbf{revises} \implies CT \vdash \mathtt{I}$ **revises** J ok*, and*
4. $\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \textbf{evolves} \implies CT \vdash \mathtt{I}$ **evolves** J ok*.*

Informally, a class table is correct if (1) the class table relations are well-formed, (2) every class definition in the class table is correct (the formal definition of this is given later in this section), and (3-4) the revises and evolves relations are correct (in the sense of Definition 2).

4.4 Typing Rules

The typing rules for statements are given below where a typing environment Γ is a finite map from variables to types.

$$\frac{CT \vdash \mathtt{S} <: \mathtt{T}}{CT; \Gamma, \mathtt{x}: \mathtt{T}, \mathtt{y}: \mathtt{S} \vdash \mathtt{x = y;\ ok}}\ \text{[T-Assign]} \qquad \frac{CT \vdash \mathit{ftype}(CT, \mathtt{S}, \mathtt{f}) <: \mathtt{T}}{CT; \Gamma, \mathtt{x}: \mathtt{T}, \mathtt{y}: \mathtt{S} \vdash \mathtt{x = y.f;\ ok}}\ \text{[T-FAccess]}$$

$$\frac{CT \vdash \mathtt{S} <: \mathit{ftype}(CT, \mathtt{T}, \mathtt{f})}{CT; \Gamma, \mathtt{x}: \mathtt{S}, \mathtt{y}: \mathtt{T} \vdash \mathtt{y.f = x;\ ok}}\ \text{[T-FAssign]} \qquad \frac{}{CT; \Gamma, \mathtt{x}: \mathtt{Object} \vdash \mathtt{x = new\ Object();\ ok}}\ \text{[T-New1]}$$

$$\frac{CT \vdash \mathtt{C[v]} <: \mathtt{T}}{CT; \Gamma, \mathtt{x}: \mathtt{T} \vdash \mathtt{x = new\ C[v=]();\ ok}}\ \text{[T-New2]} \qquad \frac{CT \vdash \mathtt{C[v+]} <: \mathtt{T}}{CT; \Gamma, \mathtt{x}: \mathtt{T} \vdash \mathtt{x = new\ C[v+]();\ ok}}\ \text{[T-New3]}$$

$$\frac{CT \vdash \mathtt{C[v+]} <: \mathtt{T}}{CT; \Gamma, \mathtt{x}: \mathtt{T} \vdash \mathtt{x = new\ C[v++]();\ ok}}\ \text{[T-New4]} \qquad \frac{CT \vdash \mathtt{S} <: \mathtt{T} \quad CT \vdash \mathtt{S} <: \mathtt{R}}{CT; \Gamma, \mathtt{x}: \mathtt{T}, \mathtt{y}: \mathtt{R} \vdash \mathtt{x = (S)y;\ ok}}\ \text{[T-DCast]}$$

$$\frac{CT \vdash \mathtt{R} <: \mathtt{S} \quad CT \vdash \mathtt{S} <: \mathtt{T}}{CT; \Gamma, \mathtt{x}: \mathtt{T}, \mathtt{y}: \mathtt{R} \vdash \mathtt{x = (S)y;\ ok}}\ \text{[T-UCast]} \qquad \frac{}{CT; \Gamma \vdash \mathtt{upgrade;\ ok}}\ \text{[T-Upgrade]}$$

$$\frac{CT; \Gamma \vdash \overline{\mathtt{s}}\ \mathtt{ok} \quad CT; \Gamma \vdash \overline{\mathtt{t}}\ \mathtt{ok} \quad CT \vdash \mathtt{S} <: \mathtt{T}}{CT; \Gamma, \mathtt{x}: \mathtt{S}, \mathtt{y}: \mathtt{T} \vdash \mathtt{if(x == y)\{}\overline{\mathtt{s}}\mathtt{\}\ else\ \{}\overline{\mathtt{t}}\mathtt{\}\ ok}}\ \text{[T-If1]}$$

$$\frac{CT; \Gamma \vdash \overline{\mathtt{s}}\ \mathtt{ok} \quad CT; \Gamma \vdash \overline{\mathtt{t}}\ \mathtt{ok} \quad CT \vdash \mathtt{T} <: \mathtt{S}}{CT; \Gamma, \mathtt{x}: \mathtt{S}, \mathtt{y}: \mathtt{T} \vdash \mathtt{if(x == y)\{}\overline{\mathtt{s}}\mathtt{\}\ else\ \{}\overline{\mathtt{t}}\mathtt{\}\ ok}}\ \text{[T-If2]}$$

$$\frac{CT; \Gamma \vdash \overline{\mathtt{s}}\ \mathtt{ok} \quad CT; \Gamma \vdash \overline{\mathtt{t}}\ \mathtt{ok} \quad CT \vdash \mathtt{S} <: \mathtt{T}}{CT; \Gamma, \mathtt{x}: \mathtt{S} \vdash \mathtt{if(x\ instanceof\ T)\{}\overline{\mathtt{s}}\mathtt{\}\ else\ \{}\overline{\mathtt{t}}\mathtt{\}\ ok}}\ \text{[T-IfInst1]}$$

$$\frac{CT; \Gamma \vdash \overline{\mathtt{s}}\ \mathtt{ok} \quad CT; \Gamma \vdash \overline{\mathtt{t}}\ \mathtt{ok} \quad CT \vdash \mathtt{T} <: \mathtt{S}}{CT; \Gamma, \mathtt{x}: \mathtt{S} \vdash \mathtt{if(x\ instanceof\ T)\{}\overline{\mathtt{s}}\mathtt{\}\ else\ \{}\overline{\mathtt{t}}\mathtt{\}\ ok}}\ \text{[T-IfInst2]}$$

Fields:

$$\begin{array}{rcll} \mathit{fields}(CT, \texttt{Object}) & \stackrel{\text{def}}{=} & \emptyset & \\ \mathit{fields}(CT, \texttt{I}) & \stackrel{\text{def}}{=} & \{\overline{\texttt{T}}\ \overline{\texttt{f}}\} \cup \mathit{fields}(CT, \texttt{J}) & \text{where } CT \vdash \texttt{I} \triangleq \texttt{\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\}} \\ & & & \quad \text{and } CT \vdash \texttt{I}\ \textbf{extends}\ \texttt{J} \end{array}$$

Field lookup:

$$\begin{array}{rcll} \mathit{ftype}(CT, \texttt{I}, \texttt{f}) & \stackrel{\text{def}}{=} & \texttt{T} & \text{where } CT \vdash \texttt{I} \triangleq \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{g}}\texttt{;T f;}\overline{\texttt{U}}\ \overline{\texttt{h}}\texttt{; }\overline{\texttt{M}}\texttt{\}} \\ \mathit{ftype}(CT, \texttt{I}, \texttt{f}) & \stackrel{\text{def}}{=} & \mathit{ftype}(CT, \texttt{J}, \texttt{f}) & \text{where } CT \vdash \texttt{I} \triangleq \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{g}}\texttt{; }\overline{\texttt{M}}\texttt{\}}, \texttt{f} \notin \overline{\texttt{g}}, \\ & & & \quad \text{and } CT \vdash \texttt{I}\ \textbf{extends}\ \texttt{J} \\ \mathit{ftype}(CT, \texttt{C[v+]}, \texttt{f}) & \stackrel{\text{def}}{=} & \mathit{ftype}(CT, \texttt{C[v]}, \texttt{f}) & \\ \mathit{ftype}(CT, \texttt{C[v}_1, \ldots \texttt{v}_n\texttt{]}, \texttt{f}) & \stackrel{\text{def}}{=} & \mathit{ftype}(CT, \texttt{C[v}_1\texttt{]}, \texttt{f}) & \end{array}$$

Method type lookup:

$$\begin{array}{rcll} \mathit{mtype}(CT, \texttt{I}, \texttt{m}) & \stackrel{\text{def}}{=} & \overline{\texttt{T}} \to \texttt{S} & \text{where } CT \vdash \texttt{I} \triangleq \texttt{\{}\overline{\texttt{U}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\}}, \\ & & & \quad \text{and } \texttt{S m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ B return y; \}} \in \overline{\texttt{M}} \\ \mathit{mtype}(CT, \texttt{I}, \texttt{m}) & \stackrel{\text{def}}{=} & \mathit{mtype}(CT, \texttt{J}, \texttt{m}) & \text{where } CT \vdash \texttt{I} \triangleq \texttt{\{}\overline{\texttt{U}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\}}, \texttt{m} \notin \overline{\texttt{M}}, \\ & & & \quad \text{and } CT \vdash \texttt{I}\ \textbf{extends}\ \texttt{J} \\ \mathit{mtype}(CT, \texttt{C[v+]}, \texttt{m}) & \stackrel{\text{def}}{=} & \mathit{mtype}(CT, \texttt{C[v]}, \texttt{m}) & \\ \mathit{mtype}(CT, \texttt{C[v}_1, \ldots \texttt{v}_n\texttt{]}, \texttt{m}) & \stackrel{\text{def}}{=} & \mathit{mtype}(CT, \texttt{C[v}_1\texttt{]}, \texttt{m}) & \end{array}$$

Method body lookup:

$$\begin{array}{rcll} \mathit{mbody}(CT, \texttt{C[v=]}, \texttt{m}) & \stackrel{\text{def}}{=} & \overline{\texttt{x}}\texttt{.B return y;} & \text{where } CT \vdash \texttt{C[v=]} \triangleq \texttt{\{}\overline{\texttt{U}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\}}, \\ & & & \quad \text{and } \texttt{S m(}\overline{\texttt{S}}\ \overline{\texttt{x}}\texttt{)\{ B return y; \}} \in \overline{\texttt{M}} \\ & \stackrel{\text{def}}{=} & \mathit{mbody}(CT, \texttt{I}, \texttt{m}) & \text{where } CT \vdash \texttt{C[v=]} \triangleq \texttt{\{}\overline{\texttt{U}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\}}, \texttt{m} \notin \overline{\texttt{M}}, \\ & & & \quad CT \vdash \texttt{C[v=]}\ \textbf{extends}\ \texttt{I} \\ \mathit{mbody}(CT, \texttt{C[v+]}, \texttt{m}) & \stackrel{\text{def}}{=} & \mathit{mbody}(CT, \texttt{I+}, \texttt{m}) & \text{where } CT \vdash \texttt{I}\ \textbf{revises}\ \texttt{C[v=]} \\ & \stackrel{\text{def}}{=} & \overline{\texttt{x}}\texttt{.B return y;} & \text{where } \forall \texttt{I}.\neg(CT \vdash \texttt{I}\ \textbf{revises}\ \texttt{C[v=]}), \\ & & & \quad \text{and } CT \vdash \texttt{C[v=]} \triangleq \texttt{\{}\overline{\texttt{U}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\}}, \\ & & & \quad \text{and } \texttt{S m(}\overline{\texttt{S}}\ \overline{\texttt{x}}\texttt{)\{ B return y; \}} \in \overline{\texttt{M}}. \\ & \stackrel{\text{def}}{=} & \mathit{mbody}(CT, \texttt{J+}, \texttt{m}) & \text{where } \forall \texttt{I}.\neg(CT \vdash \texttt{I}\ \textbf{revises}\ \texttt{C[v=]}) \\ & & & \quad \text{and } CT \vdash \texttt{C[v=]} \triangleq \texttt{\{}\overline{\texttt{U}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\}}, \\ & & & \quad \text{and } \texttt{m} \notin \overline{\texttt{M}}, \\ & & & \quad \text{and } CT \vdash \texttt{C[v=]}\ \textbf{extends}\ \texttt{J} \end{array}$$

Method signature:

$$\begin{array}{rcll} \mathit{methSig}(CT, \texttt{Object}) & \stackrel{\text{def}}{=} & \emptyset & \\ \mathit{methSig}(CT, \texttt{I}) & \stackrel{\text{def}}{=} & \{\texttt{m}\colon \mathit{mtype}(CT, \texttt{I}, \texttt{m})\}^{\texttt{m} \in \overline{\texttt{M}}} \cup \mathit{methSig}(CT, \texttt{J}) & \\ & & \text{where } CT \vdash \texttt{I} \triangleq \texttt{\{}\overline{\texttt{U}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\}}, \text{ and } CT \vdash \texttt{I}\ \textbf{extends}\ \texttt{J} & \end{array}$$

Latest version:

$$\begin{array}{rcll} \mathit{latest}(CT, \texttt{J}) & \stackrel{\text{def}}{=} & \mathit{latest}(CT, \texttt{I}) & \text{if } CT \vdash \texttt{I}\ \textbf{evolves}\ \texttt{J} \\ & \stackrel{\text{def}}{=} & \mathit{latest}(CT, \texttt{I}) & \text{if } CT \vdash \texttt{I}\ \textbf{revises}\ \texttt{J}, \text{ and } \forall \texttt{K}.\neg(CT \vdash \texttt{K}\ \textbf{evolves}\ \texttt{J}) \\ & \stackrel{\text{def}}{=} & \texttt{J} & \text{otherwise} \end{array}$$

Figure 1: Auxiliary functions

$$\frac{mtype(CT, \mathtt{S}, \mathtt{m}) = \overline{\mathtt{T}_1} \to \mathtt{U} \quad CT \vdash \overline{\mathtt{T}_0} <: \overline{\mathtt{T}_1} \quad CT \vdash \mathtt{U} <: \mathtt{V}}{CT; \Gamma, \mathtt{x}: \mathtt{V}, \mathtt{y}: \mathtt{S}, \overline{\mathtt{z}}: \overline{\mathtt{T}_0} \vdash \mathtt{x = y.m(\overline{z}); \ ok}} \text{[T-Invoke]}$$

These rules are pretty routine. The remaining typing rules for statement sequences, method blocks, method definitions, and class definitions are similarly straightforward and are as follows.

Statement sequence typing:

$$\frac{CT; \Gamma \vdash \mathtt{s}_1 \ \mathsf{ok} \ \cdots \ CT; \Gamma \vdash \mathtt{s}_n \ \mathsf{ok}}{CT; \Gamma \vdash \mathtt{s}_1 \cdots \mathtt{s}_n \ \mathsf{ok}}$$

Method block typing:

$$\frac{CT; \Gamma, \overline{\mathtt{x}}: \overline{\mathtt{T}} \vdash \overline{\mathtt{s}} \ \mathsf{ok}}{CT; \Gamma \vdash \overline{\mathtt{T}} \ \overline{\mathtt{x}}; \overline{\mathtt{s}} \ \mathsf{ok}}$$

Method definition typing:

$$\frac{\begin{array}{c}\Gamma \stackrel{\text{def}}{=} \overline{\mathtt{x}}: \overline{\mathtt{T}}, \mathtt{this}: \mathtt{I} \quad CT; \Gamma \vdash \mathtt{B} \ \mathsf{ok} \quad CT \vdash \Gamma(\mathtt{y}) <: \mathtt{S} \quad CT \vdash \mathtt{I} \ \textbf{extends} \ \mathtt{J} \\ \text{If } mtype(CT, \mathtt{J}, \mathtt{m}) = \overline{\mathtt{T}_1} \to \mathtt{S}_1 \text{ then } \overline{\mathtt{T}_1} = \overline{\mathtt{T}} \text{ and } \mathtt{S}_1 = \mathtt{S}\end{array}}{CT \vdash \mathtt{S \ m(\overline{T} \ \overline{x})\{ \ B \ return \ y; \ \}} \ \mathsf{in} \ \mathtt{I} \ \mathsf{ok}}$$

Class definition typing:

$$\frac{}{CT \vdash \mathtt{Object} \ \mathsf{ok}}$$

$$\frac{CT \vdash \mathtt{I} \triangleq \mathtt{\{\overline{T} \ f; \ \overline{M}\}} \quad CT \vdash \mathtt{I} \ \textbf{extends} \ \mathtt{J} \quad CT \vdash \mathtt{J} \triangleq \mathtt{\{\overline{S} \ \overline{g}; \ \overline{N}\}} \quad \mathtt{\overline{f} \cap \overline{g} = \emptyset} \quad CT \vdash \mathtt{\overline{M}} \ \mathsf{in} \ \mathtt{I} \ \mathsf{ok}}{CT \vdash \mathtt{I} \ \mathsf{ok}}$$

We can now prove some lemmas that will be useful when we come to reason about the operational semantics in the following section.

Lemma 4 (Class table weakening)**.** *If $CT \subseteq CT'$, then*

1. *if $CT \vdash \mathtt{I}$ **extends** $\mathtt{J}$ then $CT' \vdash \mathtt{I}$ **extends** $\mathtt{J}$,*
2. *if $CT \vdash \mathtt{I}$ **revises** $\mathtt{J}$ then $CT' \vdash \mathtt{I}$ **revises** $\mathtt{J}$,*
3. *if $CT \vdash \mathtt{I}$ **evolves** $\mathtt{J}$ then $CT' \vdash \mathtt{I}$ **evolves** $\mathtt{J}$,*
4. *if $CT \vdash \mathtt{T} <: \mathtt{S}$ then $CT' \vdash \mathtt{T} <: \mathtt{S}$,*
5. *if $ftype(CT, \mathtt{S}, \mathtt{f}) = \mathtt{U}$ then $ftype(CT', \mathtt{S}, \mathtt{f}) = \mathtt{U}$,*
6. *if $fields(CT, \mathtt{S}) = \overline{\mathtt{T}} \ \overline{\mathtt{f}}$ then $fields(CT', \mathtt{S}) = \overline{\mathtt{T}} \ \overline{\mathtt{f}}$,*
7. *if $mtype(CT, \mathtt{S}, \mathtt{m}) = \overline{\mathtt{T}_1} \to \mathtt{U}$ then $mtype(CT', \mathtt{S}, \mathtt{m}) = \overline{\mathtt{T}_1} \to \mathtt{U}$,*
8. *if $methSig(CT, \mathtt{S}) = X$ then $methSig(CT', \mathtt{S}) = X$,*
9. *if $CT \vdash \mathtt{I}$ **revises** $\mathtt{J}$ ok then $CT' \vdash \mathtt{I}$ **revises** $\mathtt{J}$ ok,*
10. *if $CT \vdash \mathtt{I}$ **evolves** $\mathtt{J}$ ok then $CT' \vdash \mathtt{I}$ **evolves** $\mathtt{J}$ ok,*
11. *if $CT \vdash \mathtt{s}$ ok then $CT' \vdash \mathtt{s}$ ok,*
12. *if $CT \vdash \overline{\mathtt{s}}$ ok then $CT' \vdash \overline{\mathtt{s}}$ ok, and*

13. if $CT \vdash \texttt{I ok}$ *then* $CT' \vdash \texttt{I ok}$.

Proof. Parts (1), (2) and (3) hold trivially. Part (4) follows by rule induction on the subtyping relation. Parts (5), (6), (7) and (8) follow directly as the class definitions in CT are unchanged in CT'. Parts (9 and 10) then follow directly. Parts (11) and (12) are proved simultaneously by induction on the size of the terms $\texttt{s}$ and $\overline{\texttt{s}}$, using parts (4), (5) and (7). Finally, part (13) holds trivially. □

Lemma 5 (Auxilliary function properties)**.**

1. *If* $mtype(CT, \texttt{S}, \texttt{m}) = \overline{\texttt{T}_1} \to \texttt{U}$ *then* $mbody(CT, \texttt{S}, \texttt{m})$ *is defined.*
2. $ftype(CT, \texttt{S}, \texttt{f}) = \texttt{U}$ *iff* $\texttt{U f} \in fields(CT, \texttt{S})$.

Lemma 6 (Subtyping properties)**.** *If* $CT \vdash \texttt{T} <: \texttt{S}$ *then*

1. *if* $ftype(CT, \texttt{S}, \texttt{f}) = \texttt{U}$ *then* $ftype(CT, \texttt{T}, \texttt{f}) = \texttt{U}$,
2. $fields(CT, \texttt{S}) \subseteq fields(CT, \texttt{T})$, *and*
3. *if* $mtype(CT, \texttt{S}, \texttt{m}) = \overline{\texttt{T}_1} \to \texttt{U}$ *then* $mtype(CT, \texttt{T}, \texttt{m}) = \overline{\texttt{T}_1} \to \texttt{U}$.

4.5 Operational semantics

We define the operational semantics of FUJ in terms of labelled transitions between configurations (where l ranges over the labels). A configuration is a four-tuple, written $(CT, S, H, \overline{\texttt{s}})$, where CT is a class table, S is a stack which is a function from program variables to values, H is a heap which is a function from object identifiers to heap objects, and $\overline{\texttt{s}}$ is a sequence of statements that represents the code that is being executed. Because of the restricted syntactic form of FUJ we do not need the evaluation contexts of FJ [19] or the frame stacks and scopes of MJ [6]. The operational semantics are given in Figure 2.

The transition rules are fairly routine; the ones of interest are those dealing with object creation, method invocation and upgrades. The rule for creating a non-upgradeable object creates an object with a runtime type `C[v=]` and the rule for creating a revision upgradeable instance produces an object with a runtime type `C[v+]`. The rule dealing with creating a evolution upgradeable instance (`new C[v++]`) is a little more subtle. First we use the auxiliary function *latest* to discover the latest version of type `C[v=]`, which is, say, `I`. We then create an upgradeable instance of type `I`. We write this type `I+`, where `(C[v=])+` is defined to be `C[v+]`.

The rule for method invocation uses the auxiliary function *mbody* to return the body of method `m` for an object whose runtime type is `R`. The definition of *mbody* is given in Figure 1. Its behaviour is dependent on the runtime type of the object. If it is an exact type, then *mbody* behaves as it does for FJ. If the runtime type is `C[v+]`, then we look to see if the class has been revised. If there has been a revision, then we recursively search the revision. If there have been no revisions to the class and the method is implemented in class `C[v=]` then we use this implementation. If class `C[v=]` does not implement the method and there has not been a revision then we recursively search the superclass of `C[v=]`.

We have also included the transition rules that deal with erroneous situations, e.g. null pointer invocation. Rather than introduce exceptions we follow MJ [6] and define a number of "stuck states".

Now we consider the upgrade transition rules. We label the transition with the upgrade definition in the familiar way [34]. Each of the transition rules for upgrades must extend the CT while ensuring that the subtype relation is a partial order (Definition 1.1). Each transition rule builds on the following lemma to ensure this.

[R-Ass] $(CT, S, H, \texttt{x = y;}\overline{\texttt{s}}) \longrightarrow (CT, S[\texttt{x} \mapsto S(\texttt{y})], H, \overline{\texttt{s}})$

[R-FAcc] $(CT, S, H, \texttt{x = y.f;}\overline{\texttt{s}}) \longrightarrow (CT, S[\texttt{x} \mapsto F(\texttt{f})], H, \overline{\texttt{s}})$ where $S(\texttt{y}) = o$ and $H(o) = \langle \_, F \rangle$

[R-FAss] $(CT, S, H, \texttt{x.f = y;}\overline{\texttt{s}}) \longrightarrow (CT, S, H[o \mapsto \langle \texttt{R}, F' \rangle], \overline{\texttt{s}})$ where $S(\texttt{x}) = o$ and $H(o) = \langle \texttt{R}, F \rangle$ and $F' \stackrel{\text{def}}{=} F[\texttt{f} \mapsto S(\texttt{y})]$

[R-Cast] $(CT, S, H, \texttt{x = (T)y;}\overline{\texttt{s}}) \longrightarrow (CT, S[\texttt{x} \mapsto o], H, \overline{\texttt{s}})$ where $S(\texttt{y}) = o$, $H(o) = \langle \texttt{R}, F \rangle$, and $CT \vdash \texttt{R} <: \texttt{T}$.

$(CT, S, H, \texttt{if(x == y)\{}\overline{\texttt{s}}\texttt{\} else \{}\overline{\texttt{t}}\texttt{\}}\,\overline{\texttt{u}})$

[R-If1] $\longrightarrow (CT, S, H, \overline{\texttt{s}}\,\overline{\texttt{u}})$ if $S(\texttt{x}) = S(\texttt{y})$

[R-If2] $\longrightarrow (CT, S, H, \overline{\texttt{t}}\,\overline{\texttt{u}})$ otherwise

$(CT, S, H, \texttt{if(x instanceof T)\{}\overline{\texttt{s}}\texttt{\} else \{}\overline{\texttt{t}}\texttt{\}}\,\overline{\texttt{u}})$

[R-Ifl1] $\longrightarrow (CT, S, H, \overline{\texttt{s}}\,\overline{\texttt{u}})$ if $S(\texttt{x}) = o$, $H(o) = \langle \texttt{R}, F \rangle$, and $CT \vdash \texttt{R} <: \texttt{T}$

[R-Ifl2] $\longrightarrow (CT, S, H, \overline{\texttt{t}}\,\overline{\texttt{u}})$ otherwise

[R-New1] $(CT, S, H, \texttt{x = new Object();}\overline{\texttt{s}}) \longrightarrow (CT, S[\texttt{x} \mapsto o], H', \overline{\texttt{s}})$ where $\mathit{fields}(CT, \texttt{Object}) = \overline{T}\ \overline{\texttt{f}}$, $o \notin \mathit{dom}(H)$, and $H' \stackrel{\text{def}}{=} H[o \mapsto \langle \texttt{Object}, \{\overline{\texttt{f}} \mapsto \texttt{null}\} \rangle]$

[R-New2] $(CT, S, H, \texttt{x = new C[v=]();}\overline{\texttt{s}}) \longrightarrow (CT, S[\texttt{x} \mapsto o], H', \overline{\texttt{s}})$ where $\mathit{fields}(CT, \texttt{C[v=]}) = \overline{T}\ \overline{\texttt{f}}$, $o \notin \mathit{dom}(H)$, and $H' = H[o \mapsto \langle \texttt{C[v=]}, \{\overline{\texttt{f}} \mapsto \texttt{null}\} \rangle]$

[R-New3] $(CT, S, H, \texttt{x = new C[v+]();}\overline{\texttt{s}}) \longrightarrow (CT, S[\texttt{x} \mapsto o], H', \overline{\texttt{s}})$ where $\mathit{fields}(CT, \texttt{C[v=]}) = \overline{T}\ \overline{\texttt{f}}$, $o \notin \mathit{dom}(H)$, and $H' = H[o \mapsto \langle \texttt{C[v+]}, \{\overline{\texttt{f}} \mapsto \texttt{null}\} \rangle]$

[R-New4] $(CT, S, H, \texttt{x = new C[v++]();}\overline{\texttt{s}}) \longrightarrow (CT, S[\texttt{x} \mapsto o], H', \overline{\texttt{s}})$ where $\mathit{latest}(CT, \texttt{C[v=]}) = \texttt{I}$, $\mathit{fields}(CT, \texttt{I}) = \overline{T}\ \overline{\texttt{f}}$, $o \notin \mathit{dom}(H)$, and $H' \stackrel{\text{def}}{=} H[o \mapsto \langle \texttt{I+}, \{\overline{\texttt{f}} \mapsto \texttt{null}\} \rangle]$

[R-MCall] $(CT, S, H, \texttt{x}_0\texttt{ = y}_0\texttt{.m(}\overline{\texttt{z}_0}\texttt{);}\overline{\texttt{s}}) \longrightarrow (CT, S', H, (\texttt{B}\sigma)\ \texttt{x}_0\texttt{ = (y}\sigma\texttt{);}\overline{\texttt{s}})$ where $S(\texttt{y}_0) = o$, $H(o) = \langle \texttt{R}, F \rangle$, $\mathit{mbody}(CT, \texttt{R}, \texttt{m}) = \overline{\texttt{x}}\texttt{.B return y;}$, $(\texttt{y}_1, \overline{\texttt{z}_1}) \cap \mathit{dom}(S) = \emptyset$, $\sigma = [\texttt{this}, \overline{\texttt{x}} := \texttt{y}_1, \overline{\texttt{z}_1}]$, and $S' = S[\texttt{y}_1, \overline{\texttt{z}_1} \mapsto S(\texttt{y}_0), S(\overline{\texttt{z}_0})]$

[R-UExt] $(CT, S, H, \texttt{upgrade;}\overline{\texttt{s}}) \xrightarrow{\texttt{new L}} (CT', S, H, \overline{\texttt{s}})$ where $L = \texttt{class I extends J\{}\ \overline{\texttt{U}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\ \texttt{\}}$ $\texttt{I} \notin \mathit{dom}(CT)$, $CT' \stackrel{\text{def}}{=} CT \uplus \texttt{L}$ and $CT' \vdash \texttt{I ok}$

[R-URev] $(CT, S, H, \texttt{upgrade;}\overline{\texttt{s}}) \xrightarrow{\texttt{I revises J}} (CT', S, H, \overline{\texttt{s}})$ where $\neg(CT \vdash \texttt{I} <: \texttt{J})$, $\neg(CT \vdash \texttt{J} <: \texttt{I})$ $CT \vdash \texttt{I}$ **revises** $\texttt{J}$ ok, $\neg\exists \texttt{K}(CT \vdash \texttt{K}$ **revises** $\texttt{J})$ and $CT' \stackrel{\text{def}}{=} CT \uplus (\texttt{I}$ **revises** $\texttt{J})$

[R-UEvo] $(CT, S, H, \texttt{upgrade;}\overline{\texttt{s}}) \xrightarrow{\texttt{I evolves J}} (CT', S, H, \overline{\texttt{s}})$ where $\neg(CT \vdash \texttt{I} <: \texttt{J})$, $\neg(CT \vdash \texttt{J} <: \texttt{I})$ $CT \vdash \texttt{I}$ **evolves** $\texttt{J}$ ok, $\neg\exists \texttt{K}(CT \vdash \texttt{K}$ **evolves** $\texttt{J})$ and $CT' \stackrel{\text{def}}{=} CT \uplus (\texttt{I}$ **evolves** $\texttt{J})$

[R-FExp] $(CT, S, H, \texttt{x = y.f;}\overline{\texttt{s}})$
[R-FExp2] $(CT, S, H, \texttt{y.f = x;}\overline{\texttt{s}})$
[R-MExp] $(CT, S, H, \texttt{x = y.m(}\overline{\texttt{z}}\texttt{);}\overline{\texttt{s}})$
$\longrightarrow (CT, S, H, \mathbf{NPE})$ where $S(\texttt{y}) = \texttt{null}$

[R-CExp] $(CT, S, H, \texttt{x = (T)y;}\overline{\texttt{s}}) \longrightarrow (CT, S, H, \mathbf{CCE})$ where $S(\texttt{y}) = o$, $H(o) = \langle \texttt{R}, F \rangle$ and $CT \nvdash \texttt{R} <: \texttt{T}$

Figure 2: Operational semantics of FUJ

Lemma 7. *If R is a partial order, $\neg(xRy)$ and $\neg(yRx)$, then $(R \cup \{(x,y)\})^*$ is also a partial order.*

We consider the three 'upgrade' transition rules in turn.

Semantics of new class upgrades First we check that the new class has not already been defined. If it hasn't then we first add the definition to the class table (we use the shorthand $CT \uplus \mathtt{L}$ to mean that the map from class names to definitions is updated) and then check that the class definition is type correct. (It must be added first to allow for recursive uses of the class in its definition.)

The transition rule embodies the following property that follows from the definition of the typing rules.

Lemma 8. *If $\vdash CT$ ok, $\mathtt{I} \notin dom(CT)$, $CT' \stackrel{\text{def}}{=} CT \uplus$ `class I extends J{ U f; M }` and $CT' \vdash \mathtt{I}$ ok, then $\vdash CT'$ ok.*

Proof. Let $(\mathcal{C}, \textbf{revises}, \textbf{evolves}) = CT$.
Therefore $CT' = (\mathcal{C} \uplus \{\mathtt{I} \mapsto \mathtt{class\ I\ extends\ J\{\ \overline{U}\ \overline{f};\ \overline{M}\ \}}\}, \textbf{revises}, \textbf{evolves})$.
From $\vdash CT$ ok

$$\vdash CT \text{ wfr} \tag{A.1}$$
$$\forall \mathtt{I} \in dom(\mathcal{C}).CT \vdash \mathtt{I} \text{ ok} \tag{A.2}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \textbf{revises} \implies CT \vdash \mathtt{I}\ \textbf{revises}\ \mathtt{J} \text{ ok} \tag{A.3}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \textbf{evolves} \implies CT \vdash \mathtt{I}\ \textbf{evolves}\ \mathtt{J} \text{ ok} \tag{A.4}$$

Prove:

$$\vdash CT' \text{ wfr} \tag{P.5}$$
$$\forall \mathtt{K} \in dom(\mathcal{C} \uplus \{\mathtt{I} \mapsto \mathtt{class\ I\ extends\ J\{\ \overline{U}\ \overline{f};\ \overline{M}\ \}}\}).CT' \vdash \mathtt{K} \text{ ok} \tag{P.6}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \textbf{revises} \implies CT' \vdash \mathtt{I}\ \textbf{revises}\ \mathtt{J} \text{ ok} \tag{P.7}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \textbf{evolves} \implies CT' \vdash \mathtt{I}\ \textbf{evolves}\ \mathtt{J} \text{ ok} \tag{P.8}$$

By Lemma 4, we can see A.3 implies P.7, and A.4 implies P.8. We know the new class `I` is unrelated to any other classes in the hierarchy, so by Lemma 7 we can see A.1 implies P.5. Finally, P.6 follows from A.2 and the assumption $CT' \vdash \mathtt{I}$ ok.

□

Semantics of revision upgrades First we need to check that the revision upgrade will not introduce any cycles in the inheritance graph. Assuming that it does not we then check that the revision upgrade is type correct. Finally we extend the class table with this revision (we use the shorthand $CT \uplus (\mathtt{J}\ \mathtt{revises}\ \mathtt{I})$ to mean that the class table's **revises** relation is extended with the pair $(\mathtt{J}, \mathtt{I})$.)

This transition rule embodies the following property that follows from the definition of the typing rules.

Lemma 9. *If $\vdash CT$ ok, $\neg(CT \vdash \mathtt{I} <: \mathtt{J})$, $\neg(CT \vdash \mathtt{J} <: \mathtt{I})$, $\neg\exists \mathtt{K}(CT \vdash \mathtt{K}\ \textbf{revises}\ \mathtt{J})$, $CT \vdash \mathtt{I}\ \textbf{revises}\ \mathtt{J}$ ok and $CT' \stackrel{\text{def}}{=} CT \uplus (\mathtt{I}\ \textbf{revises}\ \mathtt{J})$, then $\vdash CT'$ ok.*

Proof. Let $(\mathcal{C}, \mathbf{revises}, \mathbf{evolves}) = CT$.
Therefore $CT' = (\mathcal{C}, \mathbf{revises} \uplus \{(\mathtt{I}, \mathtt{J})\}, \mathbf{evolves})$.
From $\vdash CT$ ok

$$\vdash CT \text{ wfr} \tag{A.9}$$
$$\forall \mathtt{I} \in dom(\mathcal{C}).CT \vdash \mathtt{I} \text{ ok} \tag{A.10}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \mathbf{revises} \implies CT \vdash \mathtt{I} \mathbf{\ revises\ } \mathtt{J} \text{ ok} \tag{A.11}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \mathbf{evolves} \implies CT \vdash \mathtt{I} \mathbf{\ evolves\ } \mathtt{J} \text{ ok} \tag{A.12}$$

Prove:

$$\vdash CT' \text{ wfr} \tag{P.13}$$
$$\forall \mathtt{I} \in dom(\mathcal{C}).CT' \vdash \mathtt{I} \text{ ok} \tag{P.14}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in (\mathbf{revises} \uplus \{(\mathtt{I}, \mathtt{J})\}) \implies CT' \vdash \mathtt{I} \mathbf{\ revises\ } \mathtt{J} \text{ ok} \tag{P.15}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \mathbf{evolves} \implies CT' \vdash \mathtt{I} \mathbf{\ evolves\ } \mathtt{J} \text{ ok} \tag{P.16}$$

By Lemma 4, we can see A.10 implies P.14, and A.12 implies P.16. We know that J is not revised by any other class in the hierarchy, so by Lemma 7 we can see A.9 implies P.13. Finally, P.15 follows from Lemma 4, A.11 and the assumption $CT \vdash \mathtt{I}$ **revises** J ok. □

Semantics of evolution upgrades This transition is similar to that for revision upgrades except that it involves the **evolves** relation. It embodies the following property.

Lemma 10. *If* $\vdash CT$ ok, $\neg(CT \vdash \mathtt{I} <: \mathtt{J})$, $\neg(CT \vdash \mathtt{J} <: \mathtt{I})$, $\neg\exists \mathtt{K}(CT \vdash \mathtt{K}$ **evolves** $\mathtt{J})$, $CT \vdash \mathtt{I}$ **evolves** J ok *and* $CT' \stackrel{\text{def}}{=} CT \uplus (\mathtt{I}$ **evolves** $\mathtt{J})$, *then* $\vdash CT'$ ok

Proof. Let $(\mathcal{C}, \mathbf{revises}, \mathbf{evolves}) = CT$.
Therefore $CT' = (\mathcal{C}, \mathbf{revises}, \mathbf{evolves} \uplus \{(\mathtt{I}, \mathtt{J})\})$.
From $\vdash CT$ ok

$$\vdash CT \text{ wfr} \tag{A.17}$$
$$\forall \mathtt{I} \in dom(\mathcal{C}).CT \vdash \mathtt{I} \text{ ok} \tag{A.18}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \mathbf{revises} \implies CT \vdash \mathtt{I} \mathbf{\ revises\ } \mathtt{J} \text{ ok} \tag{A.19}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \mathbf{evolves} \implies CT \vdash \mathtt{I} \mathbf{\ evolves\ } \mathtt{J} \text{ ok} \tag{A.20}$$

Prove:

$$\vdash CT' \text{ wfr} \tag{P.21}$$
$$\forall \mathtt{I} \in dom(\mathcal{C}).CT' \vdash \mathtt{I} \text{ ok} \tag{P.22}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in \mathbf{revises} \implies CT' \vdash \mathtt{I} \mathbf{\ revises\ } \mathtt{J} \text{ ok} \tag{P.23}$$
$$\forall \mathtt{I}, \mathtt{J}.(\mathtt{I}, \mathtt{J}) \in (\mathbf{evolves} \uplus \{(\mathtt{I}, \mathtt{J})\}) \implies CT' \vdash \mathtt{I} \mathbf{\ evolves\ } \mathtt{J} \text{ ok} \tag{P.24}$$

By Lemma 4, we can see A.18 implies P.22, and A.19 implies P.23. We know that J is not evolved by any other class in the hierarchy, so by Lemma 7 we can see A.17 implies P.21. Finally, P.24 follows from Lemma 4, A.20 and the assumption $CT \vdash \mathtt{I}$ **evolves** J ok. □

4.6 Type soundness

One advantage of our formal approach is that we are able to prove important safety properties of our system. The most fundamental property is *type soundness*: this means that the upgrades permitted by the FUJ transition rules do not compromise the underlying language-based security system of Java-like languages.

First we need to extend the notion of typing to FUJ configurations as follows.

$$\frac{}{CT; H \vdash \texttt{null} <: \texttt{T}} \qquad \frac{H(o) = \langle \texttt{R}, F \rangle \quad CT \vdash \texttt{R} <: \texttt{T}}{CT; H \vdash o <: \texttt{T}}$$

$$\frac{\forall \texttt{x} \in dom(\Gamma).\, CT; H \vdash S(\texttt{x}) <: \Gamma(\texttt{x})}{CT; \Gamma; H \vdash S \ \mathsf{ok}}$$

$$\frac{\begin{array}{ll}\forall \texttt{R}, F, o. & o \in dom(H) \wedge H(o) = \langle \texttt{R}, F \rangle \implies \\ & \forall \texttt{f} \in dom(F).\, CT; H \vdash F(\texttt{f}) <: \mathit{ftype}(CT, \texttt{R=}, \texttt{f}) \\ & \wedge\ dom(F) = \mathit{fields}(CT, \texttt{R=})\end{array}}{CT \vdash H \ \mathsf{ok}}$$

$$\frac{CT; \Gamma \vdash \overline{\texttt{s}} \ \mathsf{ok} \quad CT, \Gamma; H \vdash S \ \mathsf{ok} \quad CT \vdash H \ \mathsf{ok}}{\Gamma \vdash (CT, S, H, \overline{\texttt{s}}) \ \mathsf{ok}}$$

$$\frac{}{\Gamma \vdash (CT, S, H, \mathbf{NPE}) \ \mathsf{ok}} \qquad \frac{}{\Gamma \vdash (CT, S, H, \mathbf{CCE}) \ \mathsf{ok}}$$

As is familiar with type soundness proofs [39] we need also to prove various weakening lemmas; the most interesting of which is the following.

Lemma 11 (Class table weakening (part 2))**.** *If $CT \subseteq CT'$, then*

1. *if $CT, \Gamma, H \vdash S$ ok, then $CT', \Gamma, H \vdash S$ ok.*
2. *if $CT \vdash H$ ok, then $CT' \vdash H$ ok.*
3. *if $\Gamma \vdash (CT, S, H, \overline{\texttt{s}})$ ok, then $\Gamma \vdash (CT', S, H, \overline{\texttt{s}})$ ok.*

Proof. Follows directly from Lemma 4. □

We can then prove that the transition rules preserve type correctness as follows.

Lemma 12 (Type preservation)**.** *If $\Gamma \vdash (CT, S, H, \overline{\texttt{s}})$ ok, $\vdash CT$ ok, and $(CT, S, H, \overline{\texttt{s}}) \xrightarrow{l} (CT', S', H', \overline{\texttt{s}}')$, then there exists Γ' such that $\Gamma' \vdash (CT', S', H', \overline{\texttt{s}}')$ ok and $\vdash CT'$ ok.*

Proof.

$$\vdash CT \ \mathsf{ok} \tag{A.25}$$
$$CT; \Gamma \vdash \overline{\texttt{s}} \ \mathsf{ok} \tag{A.26}$$
$$CT, \Gamma; H \vdash S \ \mathsf{ok} \tag{A.27}$$
$$CT \vdash H \ \mathsf{ok} \tag{A.28}$$
$$(CT, S, H, \overline{\texttt{s}}) \xrightarrow{l} (CT', S', H', \overline{\texttt{s}}') \tag{A.29}$$

and we must prove there exists a Γ' such that

$$\vdash CT \ \mathsf{ok} \tag{P.30}$$
$$CT'; \Gamma' \vdash \overline{\texttt{s}}' \ \mathsf{ok} \tag{P.31}$$
$$CT', \Gamma'; H' \vdash S' \ \mathsf{ok} \tag{P.32}$$
$$CT' \vdash H' \ \mathsf{ok} \tag{P.33}$$

We proceed by case analysis on the reduction step (For compactness, we write $H(S(\mathtt{y}), \mathtt{f})$ to mean v where $S(\mathtt{y}) = o$ and $H(o) = \langle \mathtt{R}, F \rangle$ and $F(\mathtt{f}) = v$; and we write $H(S(\mathtt{y}))$ to mean $\mathtt{R}$ where $S(\mathtt{y}) = o$ and $H(o) = \langle \mathtt{R}, F \rangle$.)

Case 1 ([R-Ass]). Therefore, $CT = CT'$ and $H = H'$ and $\overline{\mathtt{s}} = \mathtt{x = y;}\overline{\mathtt{s}}'$ and $S' = S[\mathtt{x} \mapsto S(\mathtt{y})]$. We choose $\Gamma = \Gamma'$. The only non-trivial proof obligation is (P.32). By typing of statement, we know: $\Gamma(\mathtt{x}) = \tau$, $\Gamma(\mathtt{y}) = \tau'$ and $\tau' \prec \tau$. By typing of stack, we know: $CT; H \vdash S(\mathtt{y}) <: \tau'$. Hence, $CT; H \vdash S(\mathtt{y}) <: \tau$, and hence $CT; H \vdash S'(\mathtt{x}) <: \tau$ as required.

Case 2 ([R-FAcc]). Therefore, $CT = CT'$ and $H = H'$ and $\overline{\mathtt{s}} = \mathtt{x = y.f;}\overline{\mathtt{s}}'$ and $S' = S[\mathtt{x} \mapsto H(S(\mathtt{y}), \mathtt{f})]$. We choose $\Gamma = \Gamma'$. The only non-trivial proof obligation is (P.32). By typing of statement, we know: $\Gamma(\mathtt{x}) = \tau$, $\mathit{ftype}(\Gamma(\mathtt{y}), \mathtt{f}) = \tau'$ and $\tau' \prec \tau$. By typing of stack, we know: $CT; H \vdash S(\mathtt{y}) <: \Gamma(\mathtt{y})$, and hence $H(S(\mathtt{y})) \prec \Gamma(\mathtt{y})$. By typing of heap, we know: $CT; H \vdash H(S(\mathtt{y}), \mathtt{f}) <: \mathit{ftype}(H(S(\mathtt{y})), \mathtt{f})$. By definition of *ftype* we know $CT; H \vdash H(S(\mathtt{y}), \mathtt{f}) <: \mathit{ftype}(\Gamma(\mathtt{y}), \mathtt{f})$. By transitivity we know $CT; H \vdash H(S(\mathtt{y}), \mathtt{f}) <: \mathit{ftype}(\Gamma(\mathtt{x}),$ as required.

Case 3 ([R-Cast]). Therefore, $CT = CT'$ and $H = H'$ and $\overline{\mathtt{s}} = \mathtt{x = (C)y;}\overline{\mathtt{s}}'$ and $S' = S[\mathtt{x} \mapsto o]$ and $H(S(\mathtt{y})) = \langle R, F \rangle$. We choose $\Gamma = \Gamma'$. The only non-trivial proof obligation is (P.32). By typing of statement, we know: $\Gamma(\mathtt{x}) = \tau$, $\Gamma(\mathtt{y}) = \tau'$ and $\tau' \prec \tau$. By typing of stack, we know: $CT; H \vdash S(\mathtt{y}) <: \tau'$ and $CT \vdash R <: \tau'$. Hence, $CT; H \vdash S(\mathtt{y}) <: \tau$, and hence $CT; H \vdash o <: \tau$ as required.

Case 4 ([R-If*]). All follow trivially as no state is modified.

Case 5 ([R-New1-3]). We must show that both the stack and the heap are still well-formed. The stack follows trivially from the typing of the judgement. The object added to the heap is trivially well-formed as all its fields are set to `null`.

Case 6 ([R-New4]). This follows from the following property: If $\mathit{latest}(CT, \mathtt{R}) = \mathtt{I}$, then $CT \vdash \mathtt{I} <: \mathtt{R}$. This ensures that the stack is well-formed after the update. The heap is trivially well-formed as the new objects fields are all `null`.

Case 7 ([R-MCall]). Executing statement $\mathtt{x}_o = \mathtt{y}_o.\mathtt{m}(\overline{\mathtt{z}_o})\mathtt{;}$. We know $\Gamma(\mathtt{y}_o) = \tau$, $CT; \Gamma \vdash \overline{\mathtt{z}_o} <: \overline{\tau''}$, $CT; \Gamma \vdash \mathtt{x}_o <: \tau'$, $\mathit{mtype}(\tau, \mathtt{m}) = \overline{\tau''} \to \tau'$, $\mathit{mbody}(\tau_1, \mathtt{m}) = \overline{\mathtt{x}}.\mathtt{B}$ `return y;`, $H(S(\mathtt{y}_0) = \tau_1$ and $\sigma = [\mathtt{this}, \overline{\mathtt{x}} := \mathtt{y}_1, \overline{\mathtt{z}_1}]$. Choose $\Gamma' = \Gamma, \overline{\mathtt{z}_1} : \overline{\tau''}, \mathtt{y}_1 : \tau_1$. We know that $\mathit{mtype}(\tau_1, \mathtt{m}) = \mathit{mtype}(\tau, \mathtt{m})$ as $CT \vdash \tau_1 <: \tau$. The typing judgements are preserved by permutation and extension we know: $CT; \Gamma' \vdash \sigma\mathtt{B}$ ok and $CT \vdash \Gamma'(\sigma\mathtt{y}) <: \tau''$. Hence, the new code is well-typed. The new stack is well-typed as $CT \vdash S(\overline{\mathtt{z}_o}) <: \overline{\tau''}$ by transitivity of subtype relation, and hence $CT \vdash S'(\overline{\mathtt{z}_1}) <: \Gamma'(\overline{\mathtt{z}_1})$ as required.

Case 8 ([R-UExt]). This follows directly from Lemma 8.

Case 9 ([R-URev]). This follows directly from Lemma 9.

Case 10 ([R-UEvo]). This follows directly from Lemma 10.

□

Finally we can prove that an well-typed configuration is either a value, stuck or can make a transition.

Lemma 13 (Progress)**.** *If* $\Gamma \vdash (CT, S, H, \overline{\mathtt{s}})$ ok *then either* $\overline{\mathtt{s}} \equiv \epsilon$ *(ϵ denotes an empty sequence), or* $\overline{\mathtt{s}} \equiv \boldsymbol{NPE}$*, or* $\overline{\mathtt{s}} \equiv \boldsymbol{CCE}$ *or* $\exists l.\ (CT, S, H, \overline{\mathtt{s}}) \xrightarrow{l} (CT', S', H', \overline{\mathtt{s}}')$.

Proof. Consider only case where $\overline{\mathtt{s}}$ is a non-empty sequence of statements as other cases are trivial, an proceed by rule induction on typing derivation.

Hence assume:

$$CT; \Gamma \vdash \overline{\mathtt{s}}\ \mathsf{ok} \tag{A.34}$$

$$CT, \Gamma; H \vdash S\ \mathsf{ok} \tag{A.35}$$

$$CT \vdash H\ \mathsf{ok} \tag{A.36}$$

and prove:

$$\exists l.\ (CT, S, H, \overline{\mathtt{s}}) \xrightarrow{l} (CT', S', H', \overline{\mathtt{s}}') \tag{P.37}$$

We know $\overline{\mathtt{s}} = \mathtt{s}_1 \ldots \mathtt{s}_n$ and hence by (A.34):

$$CT; \Gamma \vdash \mathtt{s}_i\ \mathsf{ok} \tag{A.38}$$

Case split on possible typing of $\mathtt{s}_1$.

Case 1 (T-Assign). By (A.38) we know $\mathtt{y} \in \mathit{dom}(\Gamma)$, and hence by (A.38) that $S(\mathtt{y})$ is defined. Therefore it reduces.

Case 2 (T-FAccess). By (A.38) we know $\mathtt{y} : \mathtt{T} \in \Gamma$ and $\mathit{ftype}(CT, \mathtt{T}, \mathtt{f})$ is defined, and hence by (A.35) that $S(\mathtt{y}) = o$ and that $H(o) = \langle \mathtt{R}, F \rangle$ and $CT \vdash \mathtt{R} <: \mathtt{T}$. We know $\_\mathtt{f} \in \mathit{fields}(CT, \mathtt{T})$, and by lemma 6, we know $\_\mathtt{f} \in \mathit{fields}(CT, \mathtt{R})$. Hence, by (A.36) we know $F(\mathtt{f})$ is defined, and hence it can reduce.

Case 3 (T-FAssign). By (A.38) we know $\{\mathtt{x}, \mathtt{y}\} \subseteq \mathit{dom}(\Gamma)$, and hence by (A.35) that $S(\mathtt{x})$ and $S(\mathtt{y})$ are defined, and hence it can reduce.

Case 4 (T-New1). This can always reduce.

Case 5 (T-New2). This can always reduce provided class exists.

Case 6 (T-New3). This can always reduce provided class exists.

Case 7 (T-New4). This can always reduce provided class exists.

Case 8 (T-DCast). We know that $s_1 = \mathtt{x = (S)y;}$. By (A.38) we know $\mathtt{y} : \mathtt{T} \in \Gamma$, and hence by (A.35) that $S(\mathtt{y}) = o$ and that $H(o) = \langle \mathtt{R}, F \rangle$ and $CT \vdash \mathtt{R} <: \mathtt{T}$. Therefore can reduce by one of the cast reduction rules.

Case 9 (T-UCast). We know that $s_1 = \mathtt{x = (S)y;}$. By (A.38) we know $\mathtt{y} : \mathtt{T} \in \Gamma$, and $CT \vdash \mathtt{T} <: \mathtt{S}$ and hence by (A.35) that $S(\mathtt{y}) = o$ and that $H(o) = \langle \mathtt{R}, F \rangle$ and $CT \vdash \mathtt{R} <: \mathtt{T}$. Therefore $CT \vdash \mathtt{R} <: \mathtt{S}$, and hence can reduce by the non-exceptional reduction rule.

Case 10 (T-Upgrade). Can always reduce as always possible to introduce a new class.

Case 11 (T-If1). By (A.38) we know $\{\mathtt{x}, \mathtt{y}\} \subseteq \mathit{dom}(\Gamma)$, and hence by (A.35) that $S(\mathtt{x})$ and $S(\mathtt{y})$ are defined, and hence it can reduce.

Case 12 (T-If2). By (A.38) we know $\{\mathtt{x}, \mathtt{y}\} \subseteq \mathit{dom}(\Gamma)$, and hence by (A.35) that $S(\mathtt{x})$ and $S(\mathtt{y})$ are defined, and hence it can reduce.

Case 13 (T-IfInst1). By (A.38) we know $\mathtt{x} \in \mathit{dom}(\Gamma)$, and hence by (A.35) that $S(\mathtt{y}) = o$ and that $H(o) = \langle \mathtt{R}, F \rangle$. Therefore, it can reduce.

Case 14 (T-IfInst2). By (A.38) we know $\mathtt{x} \in \mathit{dom}(\Gamma)$, and hence by (A.35) that $S(\mathtt{y}) = o$ and that $H(o) = \langle \mathtt{R}, F \rangle$. Therefore, it can reduce.

Case 15 (T-Invoke). By (A.38) we know $\mathtt{y} : \mathtt{T} \in \Gamma$ and $\mathit{mtype}(CT, \mathtt{T}, \mathtt{m})$ is defined, and hence by (A.35) that $S(\mathtt{y}) = o$ and that $H(o) = \langle \mathtt{R}, F \rangle$ and $CT \vdash \mathtt{R} <: \mathtt{T}$. Hence by lemma 6 $\mathit{mtype}(CT, \mathtt{R}, \mathtt{m})$ is also defined, and hence by lemma 5 $\mathit{mbody}(CT, \mathtt{R}, \mathtt{m})$ is also defined, and it can reduce.

□

5 Discussion and Future Work

In this section we discuss several implications and potential enhancements of UpgradeJ.

5.1 Syntax and Environment Support

UpgradeJ relies on explicit version numbers being an integral part of a program's source. This is essential to our language-based approach: because the safety of UpgradeJ depends on the explicit versions, the versions must be in the source code. This is a key distinction between UpgradeJ and many other programming models, where versioning information is moved out of the program into configurations, policies, or XML files: the safety of the system then depends on the contents of those files, not just the program code.

Many programmers will understandably baulk at writing version numbers explicitly for every type. An obvious and clearly safe solution is simply allow programmers to write non-versioned types (e.g. just `Foo`) and by default require that the compiler interpret them conservatively: in definitions, as the earliest upgradable version (`Foo[1]`) and in `new` statements as latest creation (`Foo[1++]`). This will certainly make the program as flexible as possible, but has the great disadvantage that it will fail once programmers need a function added in a later evolution upgrade. For this reason, one might prefer some kind of IDE support: eliding version numbers where possible (unless the programmer specifically requests to see them); using standard local type inference to determine version numbers within method bodies; and perhaps inferring the earliest possible version based on the methods actually used on the type.

The most practical solution we can envision is that version numbers would be included in the manifest of class dependencies e.g. in Java-like `import` statements, e.g.

```
import com.ms.upgradej.* [3.0++];
import com.ms.vista.* [2.2+];
import com.ms.sqldb.* [2.1=];
```

which would import the latest versions of `upgradej` support above 3.0; revisions only to `com.ms.vista` above 2.2; and exactly version 2.1 of classes in the `sqldb` package. The semantics would be similar to the way import declarations disambiguate unqualified class names in Java: the versions given in the import declarations are applied to class or type names without version information in the program. For type safety, these definitions must be physically included within the program's source code: changing those definitions would require the program be re-typechecked. The point of UpgradeJ is that programs can be written flexibly *without* the need to change version numbers or manifests.

A related issue is programmers' assurance that the version numbers are correct on the libraries and external components against which their code is checked during development, and then linked at runtime. This is a standard problem in component-based systems, and the usual solutions (encoding version numbers and fingerprints with cryptographic signatures) would presumably be effective here.

5.2 Packaging Upgrades

In the formalism, and in our examples so far, we have restricted our upgrades to affecting a single class, to preserve clarity. However, it would be extremely convenient if multiple upgrades could be applied at a single upgrade point. This would enable mutually recursive classes to be provided via an upgrade, for example:

```
new class TabbedWindow[1] extends Component[5] {
 Vector[1]<TabbedPane[1]> panes;
 ...
```

```
}
new class TabbedPane[1] extends Component[5] {
 TabbedWindow[1] parent;
 ...
}
```

Note that in general such upgrades are *not* possible in a single upgrade system because of the incremental nature of the type system. Hence a move to upgrade groups or packages actually provides a richer upgrade capability.

Explicit upgrade groups or packages offer another interesting direction for future work, by opening the possibility of making versions relative to packages. Then a type such as `Button[MyPackage]` would denote the Button class as defined by the current package. The resulting type system would be related to static class family systems such as Concord or J& [20, 26], with `MyPackage` analogous to Concord's `MyGroup` or (more distantly) LOOJ's `ThisType` [8]. Given explicit packages, it is tempting to suggest that the syntactic defaults on versions should also be `MyPackage`, but this is unclear as such a design may unnecessarily restrict the flexibility of the resulting code.

UpgradeJ could also be made more flexible (or at least, the upgrades smaller) if the full text of the class or interface to be upgraded did not have to be supplied with every upgrade. Certainly, supplying the whole text has the advantages of clarity and straightforwardness, both in presenting and defining the system, and for programmers using it, and, for revision upgrades in particular, has the advantage of making the intent of the upgrade clear: to provide a revised definition of a class. On the other hand, allowing partial specifications of upgrades, supplying only the changes to the class to be upgraded, could not only reduce the size of individual upgrades, but could also make the system much more flexibile.

This additional flexibility could come from two areas. First, partial upgrades would not have to be tied to upgrading a particular version of a class: it may be possible to make such upgrades parametric in the target class version to which they should be applied. Thus if one upgrade affects one method of a class, and a second upgrade affects another method of the same class, either or both upgrades could be applied to the class in any order. In UpgradeJ, each upgrade specifies the precise version of the class it upgrades, and is accepted only if it is the first upgrade of its type (revision or evolution) to be applied to that class.

Second, partial upgrades could be parametric in the *class* to which they applied, not just the version of the class—so an upgrade adding logging or tracing (as in §6) could be applied to many different classes. This suggests that upgrading is in some ways similar to aspect-oriented programming [21, 22, 23], at least with dynamic weaving [28, 18]. Certainly there are some philosophical similarities: our "no time travel" property (old code does cannot see new types) is related to aspect-orientation's "obliviousness" [16] (especially syntactic obliviousness), and if the necessity for incremental software upgrades is a "concern" in the aspect-oriented sense, then supporting class upgrades is one way of separating that concern from the larger structure of a system. Many of the mechanisms supported by aspect-oriented languages—revising or replacing methods, adding interfaces, extending existing definitions, and so on—are similar to the mechanisms in UpgradeJ. Aspect-oriented systems are much more powerful than UpgradeJ, however, supporting many more different kinds of "upgrades" than we support here: adding fields to objects, adding methods to interfaces, replacing or wrapping methods calls, for example. Aspect-orientation also requires "quantification", so that a single aspect defintion (say adding tracing to a method) can be applied to all methods that meet some criteria in a single action—UpgradeJ has no equivalent mechanism. Thus, while there are some similarities, the approaches are quite different: UpgradeJ aims to provide tailored support for a single task (class upgrading) with a relatively simple but expressive mechanism (version numbers, upgrades, etc.) in an

incrementally type-safe framework, while most research in aspect-orientation seems to aim at a wide range of applications with necessarily complex mechanisms that sacrifice safety for power.

Finally, several of these points draw attention to the key assumption underlying UpgradeJ, and its model of version numbers in particular: that there is a single, globally coherent upgrade hierarchy. While this assumption may hold true within a single development project, or even within the framework libraries released by a single vendor, it may be too restrictive in the general case. A more complex versioning scheme, allowing local variations within an overall hierarchy (some kind of "version namespace") could be one way to address this problem, as could a system versioning packages rather than classes, but it is not clear at this stage that the resulting complexity would provide any significant benefit. Most software projects (even distributed open-source projects) do have a single point at which releases (and thus version numbers) can be allocated and validated: rather than solving all the problems of distributed development, we only hope to provide a language within which those problems can be made explicit.

5.3 Implementation

We do not yet have an implementation of UpgradeJ, although we are planning to produce a prototype based on Java. We have designed a series of annotations on classes and types (`@version()` to specify an upgradeable version, `@exact()` for an exact version, and `@latest()` for latest version creation) and plan to produce a basic pluggable type checker to implement the type system [2]. Then, we expect that typechecked UpgradeJ programs will be translated and executed on a JVM using HotSwap[6] to implement the upgrading. As part of this process, however, we use the annotations on classes and types to drive bytecode rewritings to create several JVML classes and interfaces for each UpgradeJ class, and use name mangling to encode versions into JVML class and type names.

For each UpgradeJ class we create two JVML classes, one for exact instances of the class, and one for variable instances — this means we do not need any extra per-object storage to distinguish between exact and upgradable objects. New class and evolution upgrades are implemented by using HotSwap to bring in new classes, while revision upgrades additionally overwrite the upgradeable versions of the classes that are being revised. The duplicate hierarchies means we get the effect of the two behaviours of the *mbody* lookup functions without having to change the standard JVM lookup. Methods can be removed where necessary by replacing them with calls to `super`; exact and upgradeable objects are created by instantiating the appropriate class; and latest creation will require a reflexive call implementing the dynamic lookup for the most recent upgrade.

Finally, to translate exact and upgradeable types, we also produce two JVML interfaces for each UpgradeJ class, one for each unitary exact type, and one for each upgradeable type: variables are declared as the appropriate interface, and each JVML class we produce implements the interfaces appropriate to its type; we also produce a single JVML interface to represent exact version set types. This means that most of the UpgradeJ runtime type structure is also encoded in the JVML types, but where necessary (for exact version set types of more than one version) we use bytecode rewriting and casts to encode the precise test.

[6] `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/`

5.4 Other extensions

5.4.1 Privacy

Private fields and methods grant much more flexibility to evolution upgrades whilst maintaining type safety. An evolution upgrade can completely redefine any private methods and fields, adding or deleting them, irrespective of the private fields in previous versions of the class. For example, we can upgrade a `Point` class to switch from `int` to `long` fields:

```
class Point[1] { private int x,y; ... }
class Point[2] evolves Point[1] { private long x,y; ... }
```

5.4.2 Reflection

The version numbers in the core design of UpgradeJ are not first class: they are required to form an increasing sequence but are otherwise uninterpreted. UpgradeJ implementations could certainly allow reflexive access to version numbers (e.g. via an `getVersion` method on `java.lang.Class`) and then allow reflexive object creation methods (`findClass` or `newInstance`) to accept version numbers. Reifying version numbers further, for example allowing an `int` or some special first-class `version` type as version numbers directly would require dependent typing, so we do not propose them in a general-purpose system.

5.4.3 Final versions

In UpgradeJ we allow any class to be upgraded. By analogy with Java's `final` modifier, we could imagine a `finalversion` modifier which prevents a class being upgraded. Such a modifier would decrease the flexibility of the system, and in a system with exact version types it's hard to see the real benefit of a supplier-side `finalversion` declaration, other than reducing the need for programmers to use exact types.

Alternatively, we could imagine an `upgradeable` modifier that permits a class to be upgraded: without such a modifier, classes could not be upgraded. The advantage of a class granting explicit permission to accept upgrades is that class upgrading does require some infrastructure (although much less than object updating): if classes are non-upgradeable by default this overhead can be eliminated except where required.

5.4.4 Object Updates

By design UpgradeJ provides *class upgrading* rather than *object updating*: UpgradeJ does not require any heap inspection for upgrading. Given class upgrading, however, it is interesting to consider how little additional support is required to provide object updates. Runtime support for a heap lookup primitive (`FIND`) and updating individual objects (value assignments "`:=`", or Smalltalk's "one way become") are sufficient for programmers to implement object updates in a library:

```
while ((Button[2] b = FIND Button[2]) != null) {
  b := Button[4](b.x, b.y);}
```

This code example searches for instances of `Button[2]` (assuming `FIND` returns a random instance of that class) and replaces them with new `Button[4]` instances. To preserve type safety, the r-value must be a subtype of the l-value (as usual in assignment), and the assignment needs to check that the l-value is quiescent (that is, check the stack so that the target object is nowhere bound to "`this`"). The return value could tested to check the success of the update, but in this case, if an object is not updated it will presumably be returned sometime later from `FIND`.

6 Related Work

There is a wealth of references in the area of dynamic software updating. The recent article by Stoyle et al. [35] offers an extensive overview of the area. Rather than repeat these references we shall simply provide the immediate surrounding context for UpgradeJ.

UpgradeJ supports multiple co-existing versions; an idea from our earlier work on updating ML-like modules [5]. By moving to an object-oriented setting we have found different problems, in particular, how upgrading and inheritance can be combined; how classes can be upgraded without heap inspection; and how the latest version of a class can be created.

The .NET architecture addresses versioning issues by allowing assemblies to contain version information [29, 9]. It allows multiple versions to be stored on a client and lets the versioning policy select the correct version. It is unclear, however, that this can deal with the different versions interacting, as it appears that each application can only require one version of the code. The more recent OSGi framework [27] provides stronger support for multiple versioning and updating, allowing bundles to be loaded, updated, and unloaded dynamically, and supporting multiple versions of classes within the same VM. Like .Net, however, OSGi does not have a formal model of version type safety: we hope that FUJ could in the future provide the basis for such a model.

Closely related to versioning is dynamic linking. Dynamic linking also allows late updates to code to occur. Drossopoulou et al. have studied dynamic linking in detail [14, 12, 15]. They provide details of when linking errors will occur under changes of class definitions, paying close attention to when different phases of the compilation occur, such as field layout. In this paper we have remained at a level close to the source code to avoid the problems they highlight. To avoid directly compiling versions into the code, one might like to consider a versioned variant of polymorphic bytecode [1], which is an extension to Java bytecode that allows more flexible linking at run-time. Constraints are output when compiling a class, so at link-time these constraints can be resolved, and linking can occur against different classes safely. We believe it should be possible to adapt this approach to handle versioning by producing constraints that specify that the version must have certain methods and fields. When it comes to link-time the class loader can attempt to solve the constraints. This would allow a great flexibility in which versions can be used.

UpgradeJ's revision upgrades have some similarity to various forms of object reclassification; for example, Kea [24], Predicate Classes [10] and Fickle [13]. Compared with UpgradeJ, these systems are much more flexible: classes can move around the hierarchy (implicitly based on values of instance variables in Kea and Predicate classes, or via an explicit reclassification operation in Fickle), and can gain or lose fields depending on that classification. In contrast, UpgradeJ supports revision upgrades taking objects to higher versions without affecting memory layout, and new class and evolution upgrades that can introduce new fields but do not affect existing classes. All UpgradeJ upgrades are "one way" operations: our "no time travel" principle means that upgraded objects can never be downgraded to previous versions.

Object-level updating has also been studied in depth. Techniques that search-and-replace objects on the heap via user-supplied update functions are well known, but generally rely on dynamic checks; CLOS, for example, directly supports class redefinition and allows programmers to update individual instances in various ways [33]. Some recent research has investigated how objects can be updated in a typesafe manner. For example, Boyapati et al. describe how ownership types can assist in updating aggregate objects in object-oriented databases [7].

More prosaically, the idea of incrementally defining and updating the classes rather than the objects is also not new. The earliest Smalltalk systems were in practice maintained by passing around "goodies"— patches that could affect multiple classes [30]. Modular Smalltalk propsed

an explicit class extension construct to support this [38]. More recently, systems like Changeboxes [25] have supported dynamic extensions to systems, with relatively flexible mechanisms for describing potential changes and runtime support for multiple coexisting versions. All these systems are checked dynamically, of course, whereas UpgradeJ is checked statically.

UpgradeJ's dynamic lookup over the `revises` and `extends` relationships has some commonality with the two-dimensional inheritance hierarchies found e.g. in NewtonScript [32]. The key difference here is that NewtonScript's secondary hierarchy follows interface widget's composition structure, while our secondary hierarchy follows dynamically upgraded versions of classes.

Open Classes [11] and Expanders [37] also allow new methods and fields to be added to pre-existing classes. Both these systems have restrictions to ensure unambiguous typesafe module composition which prevent replacing existing methods. In contrast, we can revise any method, and avoid ambiguity via incremental typechecking. Moreover, UpgradeJ allows classes to be upgraded at runtime.

Zenger [40] takes a different approach to the versioning problem. He proposes an extension of Java with an extensible module system, which allows modules to be upgraded. The main advantage of our work is that it does not require such a big leap from the original programming language.

A number of functional languages provide varying support for versioning and upgrading. Most notably, Erlang [3] is an untyped, first-order language that supports concurrency and module-level upgrading, but not multiple versions of the same module. Acute [31] is an extension of OCaml that has a rich set of version constraints and policies intended for distributed programming. It is interesting future work to see if similar support is possible in the UpgradeJ setting.

7 Conclusions

Programs, especialy long running, widely distributed programs, are no longer monolithic. Programs need to be upgraded with new features, new classes, and new methods even while they continue running. Previous work has focused on how to translate objects in the heap, in a type-safe and version-consistent way. This paper takes a different approach: in order to have a lightweight mechanism no heap update is applied, and assumptions on versions are made explicit. UpgradeJ supports *class upgrades* directly—adding new classes, revising existing classes, and evolving classes to incompatible versions—and typechecking is purely incremental. We hope UpgradeJ will provide a useful conceptual model of the core problems of software upgrading, and that it may inspire future language designs.

Acknowledgements We are grateful to Mike Hicks, Gareth Stoyle and Rok Strniša for useful discussions, and to Sophia Drossopoulou and the ECOOP referees for comments on earlier drafts. James Noble was supported in part by the EPSRC EP/D061644/1, the Royal Society of New Zealand Marsden Fund, and by Microsoft Research. Matthew Parkinson was supported by an EPSRC/RAEng research fellowship.

References

[1] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *Proceedings of POPL*, 2005.

[2] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proceedings of OOPSLA*, 2006.

[3] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent programming in Erlang*. Prentice Hall, 1996.

[4] E. Bailey. *Maximum RPM*. Sams, 1997.

[5] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proceedings of USE*, 2003.

[6] G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2004.

[7] C. Boyapati, B. Liskov, and L. Shrira. Lazy modular upgrades in persistent object stores. In *Proceedings of OOPSLA*, 2003.

[8] K. B. Bruce and J. N. Foster. LOOJ: Weaving LOOM into Java. In *Proceedings of ECOOP*, 2004.

[9] A. Buckley. A model of dynamic binding in .NET. In *Proceedings of FTfJP*, 2005.

[10] C. Chambers. Predicate classes. In *Proceedings of ECOOP*, 1993.

[11] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of OOPSLA*, 2000.

[12] S. Drossopoulou. Towards an abstract model of Java dynamic linking, loading and verification. In *Proceedings of TIC*, 2000.

[13] S. Drossopoulou, F. Damiani, M. Dezani, and P. Giannini. Fickle$_{II}$ more object reclassification. *ACM Transactions on Programming Languages and Systems*, 24(2), 2002.

[14] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus—towards a model of separate compilation, linking and binary compatibility. In *Proceedings of LICS*, 1999.

[15] S. Drossopoulou, G. Lagorio, and S. Eisenbach. Flexible models for dynamic linking. In *Proceedings of ESOP*, 2003.

[16] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[17] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of PLDI*, 1993.

[18] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of AOSD*, 2004.

[19] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[20] P. Jolly, S. Drossopoulou, C. Anderson, and K. Ostermann. Simple dependent types: Concord. In *6th ECOOP Workshop on Formal Techniques for Java-like Languages*, 2004.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10), Oct. 2001.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *Proceedings of ECOOP*, 1997.

[23] S. McDirmid and W. Hsieh. Aspect-oriented programming in Jiazzi. In *Proceedings of AOSD*, 2003.

[24] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In *Proceedings of ECOOP*, 1991.

[25] O. Nierstrasz, M. Denker, T. Gîrba, and A. Lienhard. Analyzing, capturing and taming software change. In *Proceedings of the Workshop on Revival of Dynamic Languages*, 2006.

[26] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested intersection for scalable software composition. In *Proceedings of OOPSLA*, 2006.

[27] OSGi Alliance. About the OSGi service platform. Downloaded from `osgi.org`, Nov. 2005.

[28] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect oriented programming. In *Proceedings of AOSD*, 2002.

[29] S. Pratschner. Simplifying deployment and solving DLL hell with the .NET framework. `http://msdn.microsoft.com`, 2001.

[30] S. Putz. Managing the evolution of Smalltalk-80 systems. In *Smalltalk-80: Bits of History, Words of Advice*. AW, 1984.

[31] P. Sewell, J. Leifer, K. Wansbrough, M. Allen-Williams, F. Zappa Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. Design rationale and language definition. Technical Report 605, University of Cambridge Computer Laboratory, Oct. 2004.

[32] W. R. Smith. Using a prototype-based language for user interface: The Newton project's experience. In *Proceedings of OOPSLA*, 1995.

[33] G. Steele. *Common Lisp the Language*. Digital Press, 1990.

[34] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of POPL*, 2005.

[35] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 29(4), 2007.

[36] R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. In *Proceedings of OOPSLA*, 2007.

[37] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *Proceedings of OOPSLA*, 2006.

[38] A. Wirfs-Brock and B. Wilkerson. An overview of Modular Smalltalk. In *Proceedings of OOPSLA*, 1988.

[39] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[40] M. Zenger. *Programming Language Abstractions for Extensible Software Components*. PhD thesis, EPFL, Switzerland, 2004.