

Obverse: Versioning for Objects (Draft)

Matthew Parkinson

March 4, 2005

Abstract

In this paper we address some of the the issues in versioning for a Java like language. In particular we address the issues of using multiple versions of a library at the same time. We present a series of solutions that statically, and link-time, ensure the absents of type errors.

1 Introduction

Versioning issues in Java arise due to two forms of distribution:

1. distributed programmers; and
2. distributed computation.

(1) refers to different programs, or components, having conflicting or incompatible library requirements. This problem is often called “DLL hell”, or in Java: “JAR Hell”. Consider deploying a web-server using the Java servlet engine, Tomcat. You install two servlets and they both depend on an XML library. Unfortunately they depend on different and incompatible versions of this library. You copy both versions into the relevant directory, but one of the servlets fails as only one version of the library can be loaded. Software is available to detect these issues early e.g. Krysalis-Version, but they do not allow multiple coexisting versions.

The problem worsens when we consider distributed computation, (2). We have multiple sites interacting. Ensuring all the sites have the same version of the code is often impractical. In this world we will have multiple coexisting versions, and we need to ensure the inter-operate correctly. Note: although we do not explicitly model this distributed world, our choice of semantics allows us to consider many of the issues that arise.

The key issue we identify in this paper is using multiple versions of the same class. Java does not directly support this. We can play tricks using class loaders, but this does not provide static guarantees. Moreover, the programmer can not specify one method that takes version 1 of a class *C*, and another method that takes version 2. Clearly we need add versions directly to the language.

In this paper we provide a small fragment of Java extended with version information called Obverse. We extend the syntax of Java with explicit version numbers, e.g.

```
x = new C<v>();
```

This means construct an object of class *C* and version *v*. These versions allow the programmer to specify which version of a class they require.

Figure 1 gives a road map to the development of Obverse. The first diagram, (a), expresses what currently exists in Java: each class only has a single version. Code that requires two versions of a particular class is not supported. We address this by allowing multiple versions to exist, illustrated in (b). Each version is really a different class with its own name, e.g. *C<1>* is distinct from *C<2>*. Programmers can specify the versions of a class their methods take as arguments and return. We prove the language is type sound.

So far the different versions are completely distinct, we extend the language to allow the programmer to specify forward and backward compatibility between versions, illustrated in (c). This compatibility allows us to use a different version than initially intended. Rather than providing explicit syntax and semantics for upgrading modules on the fly, we provide a non-deterministic semantics for constructing classes that provides a compatible version of the class. This non-determinism allows us to consider the possible upgrade schemes without additional semantic machinery.

The final extension to Obverse is to consider replacing a class’s parent with a new version, shown in (d). There are several alternatives to restrict this issue to be sound, and we discuss the relative merits of each.

We conclude with a discussion of a possible implementation, and related and future work.

2 Obverse

In this section we provide a small fragment of a Java like language with explicit version names called Obverse. This fragment is close to FJ [3], but uses im-

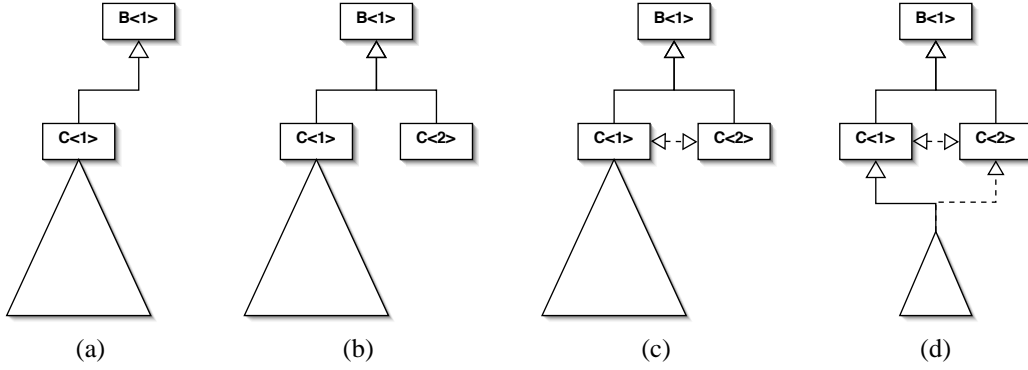


Figure 1: Adding a new version

perative state. We use this fragment to demonstrate a very simple versioning policy, and prove that this system is type-safe.

Obverse contains imperative fields and local variables, inheritance, and method overriding, but does not contain interfaces, field shadowing, method overloading or constructors. The syntax is given in Figure 2. We use the following meta-variables f , C , m , v , x to range over field names, class names, method names, versions and program variables respectively. Syntactically, a program is a sequence of class definitions followed by a sequence of statements. These statements correspond to the `main` method in Java. We use τ for types and it ranges over versioned class names. Throughout the definition we use the type, or versioned class name, where a standard class name would appear in Java. Classes are defined by a name and version, their parent's name and version, a sequence of field definitions and a sequence of method definitions. Field definitions are a type and a field name. Method definitions are given by a return type, a method name, a sequence of argument types and names, and a body, where a body is a sequence of local variable declarations, a sequence of statements and a return variable. Statements are either a variable assignment, a field lookup, a field write, method call, a conditional test, or an object construction. Note we do not allow complex expressions such as $x.f.f$; expression can only be a variable. This restriction seriously reduces the complexity of the semantics.

We will restrict our consideration to well-formed class definitions: (1) no cyclic inheritance; (2) each field and method only defined once in each class; (3) each class only defined once; and (4) no field shadowing. We do not present the obvious details of these restrictions.

The semantics of the language are fairly standard

```

p :=  $\overline{cl} \ \overline{s}$ 
cl := class  $\tau$  extends  $\tau\{\overline{\tau f}; \overline{md}\}$ 
 $\tau$  := C < v >
md :=  $\tau$  m( $\overline{\tau x}$ ) { $\overline{\tau x}; \overline{s}$  return x;}
s := x=x;
    | x=x.f;
    | x.f=x;
    | x=x.m( $\overline{x}$ );
    | if (x==x) { $\overline{s}$ }else{ $\overline{s}$ };
    | x=new  $\tau$  ();

```

Figure 2: Syntax of Obverse

and are presented in Figure 3. The restrictions to simple statements leads to a simple stack of statements being evaluated without out having to consider evaluation contexts or frame stacks. Local variables are handled with substituting fresh names.

The key to running multiple versions of code is ensuring that old versions do not get passed to code expecting new versions and new versions do not get pass to code expecting old versions. This is enforced by the nominal type system. We define the sub-typing relation as

$$\frac{\text{class } \tau \text{ extends } \tau' \{ \dots \}}{\tau \prec_1 \tau'} \quad \frac{\tau \prec_1 \tau'}{\tau \prec \tau'}$$

$$\frac{\tau \prec \tau' \quad \tau' \prec \tau''}{\tau \prec \tau''} \quad \frac{}{\tau \prec \tau}$$

The sub-typing relationship does not allow any mixing of different versions e.g.

```

class C<1> extends B<1> { ... }
class C<2> extends B<1> { ... }
...
C<1> c;
c = new C<2> ();
...

```

The call to `new` will not type-check as $C<2>$ is not a subtype of $C<1>$. We are really treating $C<1>$ and $C<2>$ as different classes, and as such they are not expected to be compatible in anyway. In the next section we will relax this restriction. Note: we do not assume any structure on version names. In examples we use integers for simplicity but any scheme of naming can be used.

Next we prove Obverse is type sound, and hence show adding new versions preserves the old code's types. First we prove a set of simple properties of the system.

Lemma 2.1 (Properties). *We have the following properties of the type system.*

1. $\Gamma; H \vdash L$ and $H \vdash o \prec \Gamma(x)$ then $\Gamma; H \vdash L[x \mapsto o]$
2. $\vdash H$, $H \vdash o_2 \prec \text{ftype}(H(o_1), f)$ and $f \in \text{fields}(H(o_1))$ then $\vdash H[o_1, f \mapsto o_2]$
3. $\tau' \prec \tau$ and $\text{ftype}(\tau, f) = \tau''$ then $\text{ftype}(\tau', f) = \tau''$
4. $\tau' \prec \tau$ and $\text{mtype}(\tau, f) = \bar{\tau}'' \rightarrow \tau''$ then $\text{mtype}(\tau', f) = \bar{\tau}'' \rightarrow \tau''$
5. $\Gamma; H \vdash L$ then $\Gamma; H[o_1, f \mapsto o_2] \vdash L$
6. $\Gamma \vdash \bar{s}$ then $\Gamma', \Gamma \vdash \bar{s}$, where $\text{dom}(\Gamma') \cap \text{dom}(\Gamma) = \emptyset$
7. $\Gamma; H \vdash L$ then $\Gamma', \Gamma; H \vdash L$, where $\text{dom}(\Gamma') \cap \text{dom}(\Gamma) = \emptyset$
8. $\Gamma \vdash \bar{s}$ then $\theta(\Gamma) \vdash \theta(\bar{s})$
9. $H \vdash o \prec \tau$ and $\tau \prec \tau'$ then $H \vdash o \prec \tau'$

We can use these properties to prove soundness of this system.

Lemma 2.2 (Type Preservation). *If a configuration is well-typed, $\Gamma \vdash (L, H, \bar{s})$ and it reduces, $(L, H, \bar{s}) \rightarrow (L', H', \bar{s}')$, then for some Γ' the new configuration is well-typed, $\Gamma' \vdash (L', H', \bar{s}')$.*

Proof. We will proceed by case analysis on the reduction relation, assuming the following

$$(A) \quad \vdash H \quad (B) \quad \Gamma; H \vdash L \quad (C) \quad \Gamma \vdash \bar{s}$$

We must the prove a Γ' exists satisfying the following

$$(1) \quad \vdash H' \quad (2) \quad \Gamma'; H' \vdash L' \quad (3) \quad \Gamma' \vdash \bar{s}'$$

Case: $x=x'$; Pick Γ' equal to Γ .

1. Holds from (A) as heap is unchanged.
2. (B) implies $H \vdash L(x') \prec \Gamma(x')$, and (C) implies $\Gamma(x') \prec \Gamma(x)$ by Prop 9 and 1 holds.
3. Holds from (C) as \bar{s}' is a subsequence of \bar{s} and $\Gamma = \Gamma'$.

Case: $x=x'.f$; Pick Γ' equal to Γ .

1. Holds from (A) as heap is unchanged.
2. (A) implies $H \vdash H(L(x'), f) \prec \text{ftype}(H(L(x')), f)$, and (C) implies $\text{ftype}(\Gamma(x'), f) \prec \Gamma(x)$, (B) implies $H(L(x')) \prec \Gamma(x')$ as $L(x')$ is not null. By Prop 3 and 9 we get $H \vdash H(L(x'), f) \prec \Gamma(x)$. Hence holds by Prop 1.
3. Holds from (C) as \bar{s}' is a subsequence of \bar{s} and $\Gamma = \Gamma'$.

Case: $x.f=x'$; Pick Γ' equal to Γ .

1. (C) implies $\Gamma(x) \prec \text{ftype}(\Gamma(x'), f)$, (B) implies $H \vdash L(x) \prec \Gamma(x)$, by Prop 9 $H \vdash L(x) \prec \text{ftype}(\Gamma(x'), f)$, hence by Prop 2 holds.
2. Follows from (B) and Prop 5.
3. Holds from (C) as \bar{s}' is a subsequence of \bar{s} and $\Gamma = \Gamma'$.

Case: $x=\text{new } \tau()$; Pick Γ' equal to Γ .

1. $H(o) = \tau \Rightarrow \text{dom}(H(o, _)) = \text{fields}(C)$ holds from definition of rule, and $\forall(o, f). \text{dom}(H).H(o, f) \in \text{dom}(H) \cup \{\text{null}\}$ holds as all fields are set to null.
2. (C) implies $\tau \prec \Gamma(x)$ from reduction rule $H'(o) = \tau$, hence holds by Prop 1.
3. Holds from (C) as \bar{s}' is a subsequence of \bar{s} and $\Gamma = \Gamma'$.

Case: $x=x'.m(\bar{x}'')$; Choose $\Gamma' = \Gamma, \bar{x}'_1 : \bar{\tau}', \bar{x}'_2 : \bar{\tau}''$.

1. Holds as heap is unchanged.
2. (B), Prop 7 and Prop 1: $\Gamma'; H \vdash L[\bar{x}'_1 \mapsto \text{null}]$ (B) implies $H(L(\bar{x}'')) \prec \Gamma(\bar{x}'')$, (C) implies $\Gamma(\bar{x}'') \prec \bar{\tau}'' = \Gamma(\bar{x}'_2)$, therefore $H \vdash L(\bar{x}'') \prec \Gamma(\bar{x}'_2)$. By repeated application of Prop 1: $\Gamma'; H \vdash L[\bar{x}'_1 \mapsto \text{null}, \bar{x}'_2 \mapsto L(\bar{x}'')] as required.$
3. Follows directly from Prop 6 and Prop 8.

□

Lemma 2.3 (Progress). *If $\Gamma \vdash \text{conf}$ and non-terminal, then $\exists \text{conf}'. \text{conf} \rightarrow \text{conf}'$.*

Proof. By induction on the typing relation. □

Class Definitions

$$\begin{aligned} \tau &\prec_1 \tau' \\ \text{fields}_1(\tau) &= \bar{f} \\ \text{ftype}_1(\tau, f) &= \tau_1 \\ \text{mtype}_1(\tau, m) &= \bar{\tau}_2 \rightarrow \tau_2 \\ \text{mbody}_1(\tau, m) &= \bar{x}, \bar{\tau}_3 \bar{x}' ; \bar{s} \text{ return } x ; \\ &\text{where class } \tau \text{ extends } \tau'' \{ \bar{\tau} \bar{f} ; \bar{\text{md}} \} \in \text{p} \\ &\text{and } \tau_1 f ; \in \bar{\tau} \bar{f} ; \\ &\text{and } \tau_2 m (\bar{\tau}_2 \bar{x}) \{ \bar{\tau}_3 \bar{x}' ; \bar{s} \text{ return } x ; \} \in \bar{\text{md}} \end{aligned}$$

$$\begin{aligned} \text{mbody}(\tau, m) &= \text{mbody}_1(\tau, m) \circ \text{mbody}(\tau', m) \\ \text{fields}(\tau) &= \text{fields}_1(\tau) \cup \text{fields}(\tau') \\ \text{mtype}(\tau, m) &= \text{mtype}_1(\tau, m) \circ \text{mtype}(\tau', m) \\ \text{ftype}(\tau, f) &= \text{ftype}_1(\tau, f) \circ \text{ftype}(\tau', f) \\ &\text{where } \tau \prec_1 \tau' \text{ and } x \circ y = \begin{cases} x & x \text{ defined} \\ y & \text{otherwise} \end{cases} \end{aligned}$$

State

$$\begin{aligned} L &: \text{Var} \rightarrow \text{Val} \\ H &: (\text{Oid} \times \text{Field} \rightarrow \text{Val}) \times (\text{Oid} \rightarrow \text{Class} \times \text{Version}) \end{aligned}$$

Semantics

$$\begin{aligned} (L, H, x=x' ; \bar{s}) &\rightarrow (L[x \mapsto L(x')], H, \bar{s}) \\ (L, H, x=x'.f ; \bar{s}) &\rightarrow (L[x \mapsto H(L(x'), f)], H, \bar{s}) \\ &\text{where } L(x') \neq \text{null} \\ (L, H, x.f=x' ; \bar{s}) &\rightarrow (L, H[L(x), f \mapsto L(x')], \bar{s}) \\ &\text{where } L(x) \neq \text{null} \\ (L, H, \text{if}(x==x') \{ \bar{s}_1 \} \text{else} \{ \bar{s}_2 \} \bar{s}) &\rightarrow (L, H, \bar{s}_1 \bar{s}) \\ &\text{if } L(x) = L(x') \\ (L, H, \text{if}(x==x') \{ \bar{s}_1 \} \text{else} \{ \bar{s}_2 \} \bar{s}) &\rightarrow (L, H, \bar{s}_2 \bar{s}) \\ &\text{if } L(x) \neq L(x') \\ (L, H, x=\text{new } \tau() ; \bar{s}) &\rightarrow (L[x \mapsto o], H', \bar{s}) \\ &\text{if } \text{fields}(\tau) = \bar{f}, o \notin \text{dom}(H) \\ &\text{and } H' = H[o \mapsto \tau][o, \bar{f} \mapsto \text{null}] \\ (L, H, x=x'.m(\bar{x}') ; \bar{s}) &\rightarrow \\ (L[\bar{x}'_1 \mapsto \text{null}, \bar{x}'_2 \mapsto L(\bar{x}'')], H, \theta(\bar{s}')x=\theta(x'''')) ; \bar{s} & \\ \text{where } \theta = [\bar{x}'_1, \bar{x}'_2 / \bar{x}'_1, \bar{x}'_2], H(x) = \tau, & \\ \bar{x}'_1 \text{ and } \bar{x}'_2 \text{ are fresh,} & \\ \text{and } \text{mbody}(\tau, m) = \bar{\tau}' \bar{x}'_1 ; \bar{s}' \text{ return } x'''' ; & \\ (L, H, x=x'.f ; \bar{s}) &\rightarrow (L, H, \mathbf{NPE}) \\ (L, H, x'.f=x ; \bar{s}) &\rightarrow (L, H, \mathbf{NPE}) \\ (L, H, x=x'.m(\bar{x}) ; \bar{s}) &\rightarrow (L, H, \mathbf{NPE}) \\ &\text{where } L(x') = \text{null} \end{aligned}$$

Subtype relation

$$\frac{\tau \prec_1 \tau'}{\tau \prec \tau'} \quad \frac{\tau \prec \tau' \quad \tau' \prec \tau''}{\tau \prec \tau''} \quad \frac{}{\tau \prec \tau}$$

Statement Typing

$$\begin{aligned} \frac{\Gamma(x') \prec \Gamma(x)}{\Gamma \vdash x=x' ;} \\ \frac{\text{ftype}(\Gamma(x'), f) \prec \Gamma(x)}{\Gamma \vdash x=x'.f ;} \\ \frac{\Gamma(x) \prec \text{ftype}(\Gamma(x'), f)}{\Gamma \vdash x'.f=x ;} \\ \frac{\tau \prec \Gamma(x)}{\Gamma \vdash x=\text{new } \tau() ;} \\ \frac{\Gamma \vdash \bar{s}_1 \quad \Gamma \vdash \bar{s}_2 \quad \Gamma(x) \prec \Gamma(x') \vee \Gamma(x') \prec \Gamma(x)}{\Gamma \vdash \text{if}(x==x') \{ \bar{s}_1 \} \text{else} \{ \bar{s}_2 \}} \\ \frac{\bar{\tau} \prec \Gamma(\bar{x}'') \quad \tau \prec \Gamma(x')}{\text{mtype}(\Gamma(x), m) = \bar{\tau} \rightarrow \tau} \\ \Gamma \vdash x'=x.m(\bar{x}'') ; \end{aligned}$$

Method Typing

$$\frac{\Gamma \vdash \bar{s} \quad \Gamma(x) \prec \tau \quad \Gamma = \{ \bar{x} \mapsto \bar{\tau}, \bar{x}' \mapsto \bar{\tau}', \text{this} \mapsto \tau' \}}{\vdash_{\tau'} \tau m(\bar{\tau} \bar{x}) \{ \bar{\tau}' \bar{x}' ; \bar{s} \text{ return } x ; \}}$$

Class Typing

$$\frac{\vdash_{\tau} \bar{\text{md}} \quad \text{ftype}_1(\tau, -) \cap \text{ftype}(\tau', -) = \emptyset \quad \text{mtype}(\tau', -) \subseteq \text{mtype}(\tau, -)}{\text{class } \tau \text{ extends } \tau' \{ \bar{\tau}' \bar{f} ; \bar{\text{md}} \}}$$

Configuration Typing

$$\begin{aligned} \frac{(o = \text{null}) \vee (H(o) \prec \tau)}{H \vdash o \prec \tau} \\ \frac{\forall s \in \bar{s}. \Gamma \vdash s}{\Gamma \vdash \bar{s}} \\ \frac{\forall x \in \text{dom}(L). H \vdash L(x) \prec \Gamma(x)}{\Gamma ; H \vdash L} \\ \frac{\forall (o, f \mapsto o') \in H. H \vdash o' \prec \text{ftype}(H(o), f) \quad \forall o \in \text{dom}(H). \text{dom}(H(o, -)) = \text{fields}(H(o))}{\vdash H} \\ \frac{\Gamma \vdash \bar{s} \quad \Gamma, H \vdash L \quad \vdash H}{\Gamma \vdash (L, H, \bar{s})} \\ \frac{}{\Gamma \vdash (L, H, \mathbf{NPE})} \end{aligned}$$

Figure 3: Semantics and typing of Obverse

3 Subversioning relation

In this section we extend Obverse to allow the mixing of compatible code: new versions can be used in the place of the old, if they are compatible. We introduce a subversion relation, \ll , that characterises one version can be used in place of another. We will extend the syntax of programs to contain definitions for compatibility.

$p := \bar{c}l \bar{c}r \bar{s}$
 $cr := \tau \text{ replaces } \tau'$

This assertion is used by the programmer to allow τ to be used in place of τ' .

Next we consider what type of constraints must be checked for this assertion. We must be careful about how the classes are changed. Consider the following change to the class C.

```
class C<1> extends Object {
  void foo(C<1> x) {}
}
class C<2> extends Object {
  int f;
  void foo(C<2> x) {
    ... x.f ....
  }
}
```

As we allow the versions to coexist the change is not subversion compatible. Clearly $C<1> \not\ll C<2>$ as $C<1>$ does not have the additional field. For the method argument to be valid we require it to be contravariantly typed: $C<1> \prec C<2>$. Hence, $C<2> \not\ll C<1>$. We would be okay if we could write

```
...
void foo(C<1> x) {...}
...
```

However, then we would not be able to access the extra field of the argument. **Note:** It is always valid to access the extra field through the `this` pointer: we know who we are, we just can't trust anyone else.

Subversion compatibility is really binary compatibility where the version is part of the name.

Let us formalise the conditions on fields and methods.

$$\frac{\forall f. ftype(\tau', f) = \tau'' \Rightarrow ftype(\tau, f) = \tau''}{\forall m. mtype(\tau', m) = \bar{\tau} \rightarrow \tau'' \Rightarrow mtype(\tau, m) = \bar{\tau} \rightarrow \tau''} \tau \text{ replaces } \tau'$$

For a replacement to be valid, the replacing class must have all the methods and all the fields of the class it is replacing, and they must have exactly the same type including all the version information.

We define the subversioning relation, and extend the subtype relation, as follows:

$$\frac{\tau \text{ replaces } \tau'}{\tau \ll \tau'} \quad \frac{}{\tau \ll \tau}$$

$$\frac{\tau \ll \tau' \quad \tau' \ll \tau''}{\tau \ll \tau''} \quad \frac{\tau \ll \tau'}{\tau \prec \tau'}$$

The subversioning relation is transitive and reflexive, and the subtype relation contains the subversion relation.

Due to the definitions of Obverse in the previous section we only alter the reduction rule for `new`: the rest of the semantics and type system are unchanged.

$$(L, H, x = \text{new } \tau() ; \bar{s}) \rightarrow (L[x \mapsto o], H', \bar{s})$$

if $\tau' \ll \tau$, $fields(\tau') = \bar{f}$, $o \notin dom(H)$
and $H' = H[o \mapsto \tau'] [o, \bar{f} \mapsto null]$

We alter the reduction rule so it can non-deterministically select any replacement for the construction of the class. This non-deterministic strategy allows us to simulate upgrades and distributed computation without need to explicitly model them. Upgrades would be one version being selected up to a certain time, and then the new version selected from then on. This is simply one reduction sequence in the non-deterministic choice of many. We are partially modelling distributed sites having different versions, as each construction can select a different class. As with multiple sites were they each select the version they have. This does not model the requirements for marshaling between different sites, we will discuss this in the Future work.

We prove soundness of this system in the same way as before. In fact, only the `new` case for the type preservation proof and the properties 3 and 4 in Lemma 2.1 need reproving. The properties follow from the additionally checking of the subversion relation, and the `new` case follows the same proof.

Theorem 3.1 (Subversioning is sound.)

4 Example: Refactoring

Now let us demonstrate that the subversioning relations allows us to express useful changes. Consider the following simple example of moving a method up, or down, the inheritance hierarchy.

```
class B<1> extends Object {}
class C<1> extends B<1> {
  Object foo() {...}
}
```

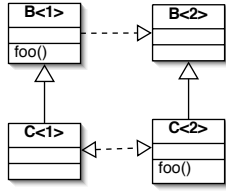


Figure 4: Refactoring

Initially the `foo` method is in the `C` class. Let us push the method up into its parent.

```
class B<2> extends Object {
  Object foo() {...}
}
class C<2> extends B<2> {
}
```

Now we make some statements about compatibility

```
C<2> replaces C<1>;
B<2> replaces B<1>;
```

These two `replaces` will pass the type-checking as `C<2>` has the same methods as `C<1>` and `B<2>` has more than `B<1>`. In fact we can make an additional assertion.

```
C<1> replaces C<2>;
```

This shows that pushing a method down the hierarchy is also a compatible change. We do not have `B<1>` replaces `B<2>` as `B<1>` does not have the `foo` method. We outline the sub-typing relation that exists in Figure 4. We use a dashed line for the replacement relation.

5 Inheritance

We have seen how to soundly replace classes with the new versions. However there is one form of replacement we haven't considered: if we replace a class, can we use it in its subclasses as a replacement parent. Consider

```
class C<1> extends B<1> {...}
class C<2> extends B<1> {...}
class D<1> extends C<1> {...}
...
C<2> replaces C<1>;
...
d = new D<1>();
d.foo();
```

Now we would like to be able to consider `D<1>` with the replaced parent `C<2>`. This kind of change is important for scalability as we don't want one change to require all the subclasses to be updated against this.

Changes of this form are both drastic and complicated. We run into two problems when considering this type of change.

- What is the dynamic type of the new classes?
- How do we ensure soundness?

The first problem with altering a class's parent is what is the new run-time name. In the run-time we have a single name `D<1>`, but `foo` could be inherited from either `C<1>` or `C<2>`. Normally dynamic dispatch would be decided by the run-time type. Hence we need our run-time types to describe these classes that don't correspond directly to the source code. We use σ to range over these extended types, and use the following syntax:

$$\sigma = (\tau, \sigma) \mid \tau$$

(τ, σ) means the class and version τ but its superclass is σ . So in the example, new `D<1>` could have two run-time types: `D<1>` and `(D<1>, C<2>)`. We must define the *mtype*, *ftype*, *fields* and *mbody* functions on these new types.

$$mtype((\tau, \sigma), m) \stackrel{\text{def}}{=} mtype_1(\tau, m) \odot mtype(\sigma, m)$$

$$mbody((\tau, \sigma), m) \stackrel{\text{def}}{=} mbody_1(\tau, m) \odot mbody(\sigma, m)$$

$$ftype((\tau, \sigma), f) \stackrel{\text{def}}{=} ftype_1(\tau, f) \odot ftype(\sigma, f)$$

$$fields((\tau, \sigma)) \stackrel{\text{def}}{=} fields_1(\tau) \cup fields(\sigma)$$

These functions follow the same definition as earlier, but use the replacement parent.

We have addressed the first problem by extending our type-system to have more type names. Ensuring soundness is however harder. There are two main issues

- replacement causing cyclic inheritance; and
- introducing fields and methods in both a new version and in a subtype.

The first problem is introducing a cyclic inheritance hierarchy. Figure 5 shows how this can occur. In this example we have allowed a class to replace its supertype's supertype. This is perfectly valid given the rules in the previous section. However, if we consider inheritance replacement then this causes issues. We can generate classes whose parent contains the same class.

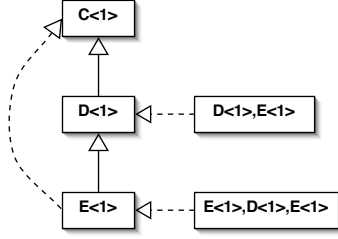


Figure 5: Cyclic inheritance

These leads to two issues: first, we can get infinite types; and second, what is the meaning of a class that inherits from itself. The first issue arises because the replacement relation is transitive. That means $E<1>, D<1>, E<1>$ is a valid replacement for $C<1>$, so $E<1>, D<1>, E<1>, D<1>, E<1>$, etc. are all generatable.

We cannot give these classes a sensible meaning, so we must rule them out. We define a type σ as well-formed if it does not contain a class twice:

$$\vdash (\tau_1, \dots, \tau_n) \mathbf{wf} \Leftrightarrow \text{parents}(\tau_n) \cap \{\tau_1, \dots, \tau_n\} = \emptyset$$

where if $\tau \prec_i \tau'$, then $\text{parents}(\tau) = \tau' \cup \text{parents}(\tau')$ and $\text{parents}(\text{Object} < v >) = \emptyset$.

We can illustrate the second problem with the following example:

```
class C<1> extends Object<1> {}
class C<2> extends Object<2> {
  C<2> m(C<2> o) {...}
}
class D<1> extends C<1> {
  D<1> m(D<1> o) {...}
}
C<2> replaces C<1>;
```

Both $D<1>$ and $C<2>$ introduce the same method, but at incompatible types. This shows that $(D<1>, C<2>) \not\prec C<2>$. This problem boils down to showing the following property

$$\vdash (\tau, \sigma) \mathbf{ok} \stackrel{\text{def}}{=} \text{ftype}_1(\tau, -) \cap \text{ftype}(\sigma, -) = \emptyset \wedge \text{mtype}(\sigma, -) \subseteq \text{mtype}((\tau, \sigma), -)$$

This is the same condition as given for Class Typing, but for the extended types.

We have three options to ensure our new classes satisfy this property:

Option 1 The first option is to restrict our attention to cases where the parent is a replacement in both directions, i.e. require

$C<2>$ replaces $C<1>$;
 $C<1>$ replaces $C<2>$;

This restriction boils down to only allowing replacements if classes have exactly the same fields and methods. This option allows us to add the following two rules to the subversioning relation:

$$\frac{\sigma \ll \tau \quad \tau \ll \sigma \quad \tau' \prec_1 \tau}{(\tau', \sigma) \ll \tau' \quad \tau' \ll (\tau', \sigma)}$$

This option prevents us from introducing new methods to the subclasses: an extension of the superclass is not an extension of the subclass. For simple changes such as minor bug fixes this would allow them to be applied to subtypes.

Option 2 The second option is to require a naming convention for method and fields. We enforce the method names specify their types, and field names specify their defining class. This rules out method name clashes: any clash is an overriding not an overloading, and field shadowing is impossible, as each class has its own namespace.

We extend the subversioning relation to allow all the updates as we have ruled out any problems.

$$\frac{\sigma \ll \tau \quad \tau' \prec_1 \tau}{(\tau', \sigma) \ll \tau'}$$

The enforced naming can restrict how we can modify a class, e.g. we can not allow fields to be moved up the hierarchy and remain in the naming convention.

This naming convention roughly corresponds to the same information that the compilation adds to the bytecode.

Option 3 Final option is to have link-time checks to see if the class that would be generated should. All the possible replacements are tested to see if they satisfy the requirement given earlier, e.g.

$$\frac{\sigma \ll \tau \quad \tau' \prec_1 \tau \quad \vdash (\tau, \sigma) \mathbf{ok}}{(\tau', \sigma) \ll \tau'}$$

This has the advantage of allowing lots of replacement classes to be used that would be ruled out by the previous two options, but it requires more checking and is potentially less predictable for the programmer.

6 Discussion

In this paper we have shown how to extend a Java like language with explicit version information, and how to allow compatible replacements. We then presented an extension to allow parents of classes to be replaced, and presented three options to ensure the replacements are sound.

Implementation The scheme outlined in §3 can be implemented as a bytecode preprocessor. We use additional interfaces to allow the subtype relation to be more flexible. Unfortunately there is a problem with field access: interfaces can not specify the mutable fields of a class. To circumvent this restriction we can replace all the field read and write instructions with getter and setter methods, and extend the classes, and interfaces, to contain these additional methods.

Figure 6 presents some useful stages in the processing. For each class, $C<1>$, we create an interface, $IC<1>$ that represents its signature, see (b). If a class $C<1>$ inherits from $B<1>$ then the relevant interfaces must have the same inheritance relation, see (c). Finally if a class $C<2>$ replaces $C<1>$ then $C<2>$'s interface must extend $C<1>$'s interface, $IC<1>$, see (d). This can introduce cycles in the interface inheritance. If a cycle occurs then the interfaces in the cycle should be replaced with a single interface. Then in every method signature, field and variable declaration, we replace the class names with the new interfaces names. We replace calls to `new` with a function that will select the correct version, i.e. the latest.

The scheme with inheritance, §5, can be implemented in the same way, but requires the generation of new classes to correspond to the classes with replaced parents.

Marshalling We have not modelled the communication between sites in this paper. The key issue is ensuring each site has a consistent view of the world, so they both interpret the class in the same way. Using hash-types of Leifer, Peskine, Sewell and Wansbrough [4] allows this global view to be constructed. Alternatively, we could relax this view by marshalling a constraint on the class hierarchy, e.g. required methods, classes and fields, and some version numbers, to allow code to rebind against slightly different versions.

Related and future work This work has considered multiple coexisting versions. Our motivation

was drawn from Sewell's work on providing versioning for a distributed ML like setting [6]. By studying an object-oriented setting we have found different problems. In particular, how replacement and inheritance can be combined. Sewell's work has been extended with other into the Acute language [7] which has a rich set of version constraints and policies. We have not consider any structure in our version names as this is orthogonal to the replacement mechanism we have described.

The .NET architecture has addressed some of the versioning issues by allowing assemblies to contain version information [5]. They allow multiple versions to be stored on a client and let the versioning policy select the correct version. However, it is unclear that this work can deal with the different versions interacting. It is about each application being happy, and each application can only require one version of the code. It is unclear how one could use several versions of the same library in one piece of code without having direct language support.

Zenger [8] takes a different approach to the versioning problem. He provides language primitives to allow code to be written in an extensible style. This removes many of the issues of versioning. The presentation given here requires less changes to the code.

Closely related to versioning is dynamic linking. Dynamic linking allows late updates to code to occur. Drossopoulou, Eisenbach, et al. have study dynamic linking in detail [2]. They provide a detail semantics of when linking errors will occur under changes of class. They play 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.

The version names in the code earlier are programmer supplied. In most cases these versions could be added by the compiler, if only one version of a library existed on the system. However inferring the version if there are multiple versions is harder. The system would require a metric for which version is considered better.

Directly compiling the version name into the code might be too restrictive. One might like to consider a polymorphic versioned bytecode. Polymorphic bytecode of Ancona, Damiani, Drossopoulou and Zucca [1] is an extension to Java bytecode that allows more flexible linking at run-time. They output constraints when they compile 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 use this work to provide

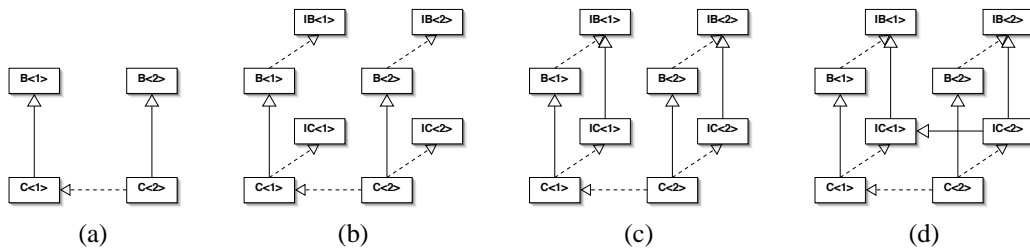


Figure 6: Adding interfaces for versioning

polymorphic version names. Rather than attempting to fill in all the version names statically, we simply construct constraints. These constraints 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.

Acknowledgements

We would like to thank Alan Lawrence, Gareth Stoye and the Acute team for their input. We acknowledge support from EC FET-GC project IST-2001-33234 PEPITO.

References

- [1] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for java-like languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, Long Beach, CA, USA, January 2005.
- [2] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In Pierpaolo Degano, editor, *12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *LNCS*, pages 38–53. Springer-Verlag, April 2003.
- [3] A. Igarashi, B.C. 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.
- [4] James Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proceedings of ICFP 2003: the 8th ACM SIGPLAN International Conference on Functional Programming (Uppsala)*, pages 87–98, August 2003.
- [5] S. Pratschner. Simplifying deployment and solving DLL hell with the .NET framework. msdn.microsoft.com/, 2001.
- [6] Peter Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (London)*, pages 236–247, January 2001.
- [7] Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. design rationale and language definition. Technical Report 605, University of Cambridge Computer Laboratory, October 2004. Also published as INRIA RR-5329. 193pp.
- [8] Matthias Zenger. *Programming Language Abstractions for Extensible Software Components*. PhD thesis, EPFL, Switzerland, 2004.