Super and Inner — Together at Last!

David S. Goldberg
School of Computing
University of Utah
Salt Lake City, UT 84112
goldberg@cs.utah.edu

Robert Bruce Findler
Dept. of Computer Science
University of Chicago
Chicago, IL 60637
robby@cs.uchicago.edu

Matthew Flatt
School of Computing
University of Utah
Salt Lake City, UT 84112
mflatt@cs.utah.edu

Abstract

In an object-oriented language, a derived class may declare a method with the same signature as a method in the base class. The meaning of the re-declaration depends on the language. Most commonly, the new declaration *overrides* the base declaration, perhaps completely replacing it, or perhaps using **super** to invoke the old implementation. Another possibility is that the base class always controls the method implementation, and the new declaration merely *augments* the method in the case that the base method calls **inner**. Each possibility has advantages and disadvantages. In this paper, we explain why programmers need both kinds of method redeclaration, and we present a language that integrates them. We also present a formal semantics for the new language, and we describe an implementation for MzScheme.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*inheritance*

General Terms

Languages

Keywords

super, inner, inheritance, override, augment

1 Introduction

In a Java-like language, each method is overrideable by default, so a subclass can replace the functionality of a method with arbitrarily different functionality. A **super** form (or its equivalent) allows a subclass implementor to reuse a superclass's method, instead of replacing the method entirely. The choice, in any case, belongs to the *subclass* implementor. Correspondingly, as illustrated in Figure 1, method dispatch for an object begins at the bottom of the class hierarchy. Java-style overriding encourages the reuse of class implementations, since subclass implementors are relatively unconstrained in re-shaping the subclass.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

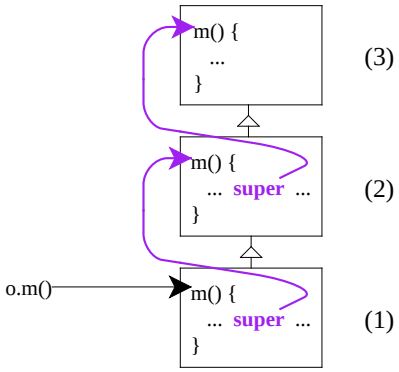

Figure 1. Java-style Method Overriding

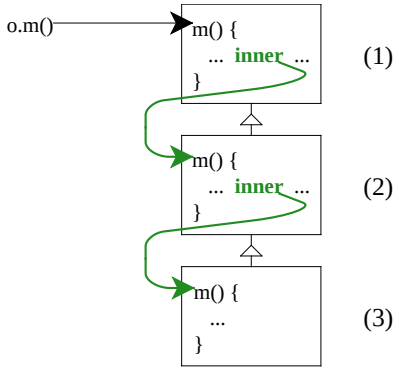

Figure 2. Beta-style Method Extension

In a Beta-like language, a method may be augmented, but the method cannot be replaced arbitrarily. A class enables method augmentation by calling **inner**, but it may perform work before and after the **inner** call, and it may skip the **inner** call altogether. The choice, in any case, belongs to the *superclass* implementor. Correspondingly, as illustrated in Figure 2, method dispatch for an object begins at the top of the class hierarchy. Controlled method extension encourages (though does not guarantee) subclasses that are behavioral subtypes [1, 21, 22] of the base class, since subclass implementors are relatively constrained.

Although programmers can simulate each form of method extension using the other, simulation patterns are clumsy. The patterns require that a programmer invent extra methods and give them distinct names, and the protocol for using and overriding or augmenting methods becomes a part of the documentation, rather than the

declared structure of the code. Furthermore, changing an existing method from one form of extension to the other requires modifications to existing code.

Some researchers, including Cook [7] and Clark [6], have observed the dual roles of **super** and **inner**, and they have developed unified object models with constructs that encompass both. We have taken a more direct approach, adding Beta-style methods and **inner** to an existing Java-style language.

Implementing our combination of **super** and **inner** requires only modest changes to a typical compiler and run-time system. In particular, the compilation of method and **super** dispatching is unchanged, and the implementation of **inner** is an adaptation of method dispatch (using an auxiliary dispatch table). Furthermore, our system does not constrain a method permanently to either Java-style or Beta-style refinement. That is, a derived class may use a different style of method overriding from its super class.

Since Beta-style method overriding is designed to help enforce invariants in the code, it trumps Java-style method overriding in our design. That is, a Java-style method extension only replaces the behavior of the method up to the nearest Beta method. In contrast, a Beta-style method controls the behavior of all of its subclasses. Consider the chain of method extensions in Figure 3. Three subchains of Java-style method extensions appear as three distinct sets of upward arrows in the figure. Each **beta** method, meanwhile, introduces a point of control over all later subclasses. This control appears in the figure as long-jumping arrows that delineate the three sets. The number to the right of each method shows its position in the overall order of execution.

We have implemented this combination of **super** and **inner** in MzScheme [12], and our design was motivated by problems building the DrScheme programming environment [11] using MzScheme's object system. In general, we find that most uses of the object system favor flexible reuse over behavioral control, which supports our decision to start with a Java-style object system. We have noted many exceptions, however, where our code is made more complex or less reliable by the possibility of unconstrained method overriding. We believe that our code will become cleaner and more reliable by using both kinds of methods, and the early results are promising.

Section 2 presents a programming task that can be implemented with only Java-style methods or only Beta-style methods, but it is best implemented with a combination. Section 3 describes in detail our method-dispatch algorithm to support both kinds of methods in a single class derivation. Section 4 defines a formal model of our language. Section 5 describes our implementation in MzScheme and initial experience.

2 The Case for Combining Super and Inner

Consider building a library of GUI widgets, including basic windows, bordered panels, and clickable buttons. All widgets correspond to areas on the screen, and they all react in various ways to mouse and keyboard actions. Furthermore, as the widget set grows, new widgets tend to resemble existing widgets, but with extra behavior. For all of these reasons, a class-based, object-oriented language is an excellent choice for implementing the widget library.

One possible class hierarchy for widgets is shown in Figure 4:

- The generic *Window* class includes a *paint* method to draw the content of the window. Subclasses of *Window* refine the *paint* method to draw specific kinds of widgets.

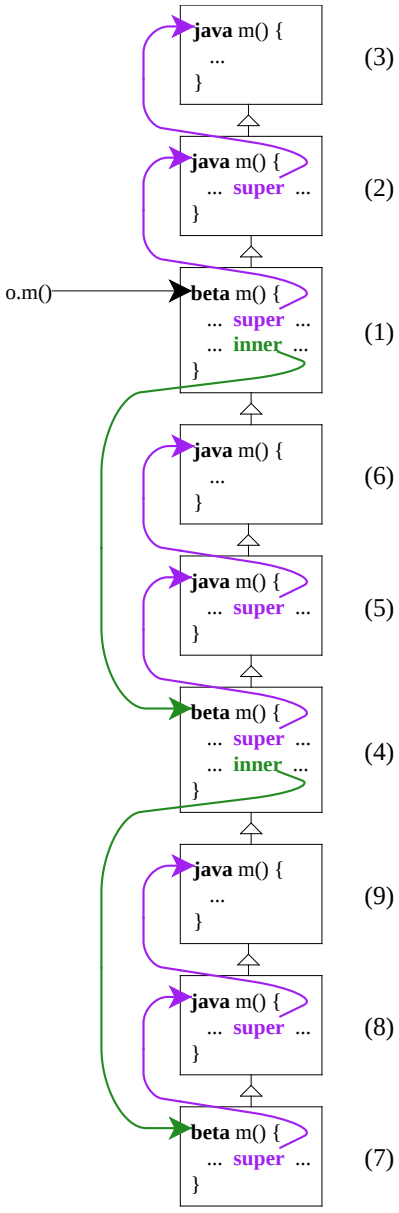

Figure 3. BETAJAVA Method Refinement

- Many widgets require a border, so *BorderWindow* refines *Window*'s *paint* to draw a border around the window. Subclasses of *BorderWindow* are expected to refine *paint* further to draw inside the border, but they are not expected to replace *paint* entirely, which would omit the border.
- The *Button* class of *BorderWindow* implements a clickable widget. It adds the *onClick* method, which is called when the user clicks inside the widget's border. The *Button* class also refines *paint* to provide a default button look, but subclasses may define an entirely different look for the button, as long as the border is intact.
- The *ImageButton* class refines the *paint* method of *Button* to draw a specific image for the button, supplanting the default button look (except for the border).

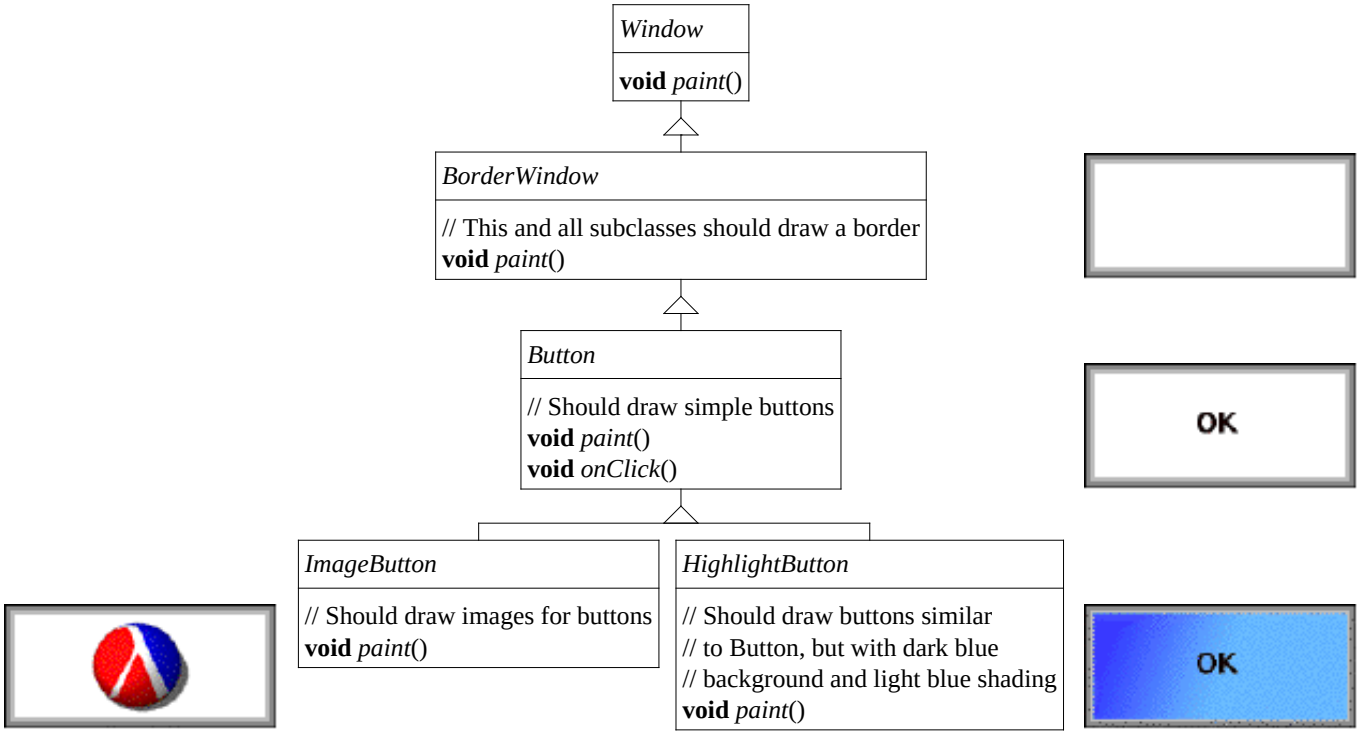

Figure 4. Class hierarchy for GUI classes

- The *HighlightButton* class, in contrast, builds on the default button look, but adds a dark blue background behind the label and a translucent texture over the label.

Implementing this class hierarchy in either Java (with **super**) or Beta (with **inner**) is straightforward, and yet the results are not entirely satisfactory. We consider each possibility in the following two sub-sections, and then show how **super** and **inner** together provide a more satisfactory implementation of the hierarchy.

2.1 Widgets in Java

To implement Figure 4 in Java, we start with a *Window* class whose *paint* method merely erases the background by painting it white, and a *BorderWindow* class that refines the *paint* method of *Window* to draw a border around the window:

```
class Window { ...
    void paint() {
        ...; // paint the background
    }
}

class BorderWindow extends Window { ...
    void paint() {
        super.paint(); // paint background
        ...; // draw a border
    }
}
```

The *paint* method in *BorderWindow* overrides the method in *Window*, which means that when *paint* is called on an instance of *BorderWindow*, control jumps to the *paint* method in *BorderWindow*, as opposed to *Window*. The **super** call in *BorderWindow* then explicitly dispatches to *paint* in *Window* to paint the background.

The next class is *Button*, which further refines *paint*:

```
class Button extends BorderWindow { ...
    void paint() {
        super.paint(); // paint background, border
        ...; // draw a button label
    }
}
```

Here, again, the new *paint* in *Button* uses **super** to paint at first like *BorderWindow*. In the *BorderWindow* case, however, this **super** call was optional; we could instead have chosen to paint *BorderWindow* backgrounds differently. A subclass of *BorderWindow* is always supposed to call *BorderWindow*'s *paint* to ensure that the window has a border. This constraint is merely part of the documentation for *BorderWindow*; it cannot be enforced by Java.

This problem becomes somewhat worse as we move to the *ImageButton* class:

```
class ImageButton extends Button { ...
    void paint() {
        super.paint(); // paints background, border — and label!
        ...; // draw image, somehow blotting out the label
    }
}
```

Since *ImageButton* is a subclass of *BorderWindow*, it is supposed to call **super** to ensure that a border is drawn. But calling **super** also draws a button label, and *ImageButton* intends to replace the label with an image.

C++ [25] offers a solution to the immediate problem in *ImageButton*. Instead of using **super**.*paint*(), *ImageButton* could refer directly to *BorderWindow::paint*(). The other problem remains, i.e., nothing forces *paint* in *ImageButton* to call *BorderWindow::paint*().

A more general solution is to simulate Beta-style methods in Java. A programmer can designate *paint* as final in *BorderWindow* and have it call a new method, *paintInside*. Subclass authors cannot override *paint*, ensuring that the border is always drawn, but they can override *paintInside* to redefine the interior painting. Unfortunately, this solution forces programmers to deal with different names for the same functionality in different parts of the class hierarchy: subclasses of *Window* are expected to override the *paint* method to add functionality, but subclasses of *BorderWindow* are expected to override the *paintInside* method. Besides increasing the burden on the programmer, these different names limit the ways in which mixins [3, 5, 14] can be applied, since mixin composition typically relies on matching method names. Furthermore, if *paint* is split into a **final** *paint* and a *paintInside* method after many classes have been derived, then names must be changed throughout the hierarchy below *BorderWindow* to accommodate the split.

Assuming that we split *paintInside* from *paint*, we can finish our widget set in Java as follows:

```
class ImageButton extends Button { ...
    void paintInside() {
        ...; // draw image (don't call super for label)
    }
}

class HighlightButton extends Button { ...
    void paintInside() {
        ...; // replace the background with dark blue
        super.paintInside(); // draws the label
        ...; // draw light blue shading on top of the label
    }
}
```

The *HighlightButton* class calls **super** to paint the default button label, but this class exploits its control over the timing of the **super** call. In particular, it draws the new background, then calls **super** to draw text on the new background. Reversing the order clearly would not work.

This last example, in particular, illustrates the overall philosophy of class extension in Java-like languages: *subclass implementors know better*. Methods are overrideable by default, so they can be replaced completely in a subclass, which tends to maximize the reuse of a class hierarchy. The only way that a superclass can insist on specific behavior, preventing subclasses from refining it, is to declare a method **final**. The idiom of a **final** method that calls a regular method (like our *paint*/*paintInside* example) appears commonly used in C++ and Java programs, inspiring design patterns such as the Template Method [15]. As we demonstrate in the next section, Beta-style method extension can more directly express a programmer's intent in such cases.

2.2 Widgets in Beta

In Beta, the *pattern* is the sole abstraction mechanism, and patterns are used to express types, classes, and methods. We are mainly interested in patterns as a class mechanism, so for our examples, we use a Java-like syntax with a Beta-like semantics.

The essential difference of Beta is the absence of **super** and the presence of **inner**. In our first two classes, *Window* explicitly allows subclasses to add functionality to *paint* by using **inner**:

```
class Window { ...
    void paint() {
        ...; // paint the background
        inner.paint();
    }
}

class BorderWindow extends Window { ...
    void paint() {
        ...; // draw a border
    }
}
```

When the *paint* method is called on an instance of *BorderWindow*, control jumps to the implementation of *paint* in *Window*. At the point where *Window*'s *paint* uses **inner**, control jumps to *paint* in *BorderWindow*. If an instance of *Window* is created, the **inner** in *paint* has no effect.

This implementation of *Window* is not quite the same as our implementation in Java, because the Beta *Window* always paints the background, but the painting was optional in the Java *Window* class. To fix this, we can simulate Java-style methods in Beta, just as we could simulate Beta-style methods in Java. In this particular case, we create a *paintBackground* method and only call it directly in *Window*'s *paint* when **inner** would do nothing. To accommodate such code, we introduce a new variant of **inner** that has an **else** statement, which is executed only if the **inner** has no target.

```
class Window { ...
    void paintBackground() {
        ...; // paint the background
    }
    void paint() {
        inner.paint() else paintBackground();
    }
}

class BorderWindow extends Window { ...
    void paint() {
      paintBackground();
        ...; // draw a border
      inner.paint();
    }
}
```

When *paint* is called for an instance of *Window*, the **inner** call has no target (i.e., no subclass), so *paintBackground*() is executed. When *paint* is called for an instance of *BorderWindow*, **inner** in *Window* jumps to *paint* in *BorderWindow*, which elects to paint the background by calling *paintBackground*().

The new *BorderWindow* also contains its own **inner** call in *paint*, so that the content of the window can be painted by subclasses. Thus, *Button* is implemented as follows:

```
class Button extends BorderWindow { ...
    void paintLabel() {
        ...; // draw a button label
    }
    void paint() {
        inner.paint() else paintLabel();
    }
}
```

Since the *BorderWindow* class does not give *Button* the option to skip border painting, the implementor of *Button* cannot accidentally omit the border by forgetting to call **super**.*paint*().

Meanwhile, *Button* uses the same **inner** programming pattern as *Window* to make label painting optional in subclasses. Much like introducing *paintInside* in Java to give the superclass control, introducing *paintLabel* in Beta gives subclasses control. Also, as in Java, this programming pattern proliferates method names, so it is similarly unfriendly to programmers and mixins. More significantly, this programming pattern must be used whenever a subclass should be able to completely replace the functionality of a method, and our experience suggests that such methods are the rule rather than the exception.

With *paintLabel* split from *paint*, we can finish our widget set in Beta as follows:

```
class ImageButton extends Button { ...
    void paint() {
        ...; // draw image (don't call paintLabel)
    }
}

class HighlightButton extends Button { ...
    void paint() {
        ...; // replace the background with dark blue
        paintLabel(); // draws the label
        ...; // draw light blue shading on top of the label
    }
}
```

The *HighlightButton* class calls *paintLabel* to paint the default button label, again exploiting its control over the timing of the *paintLabel* call. As written, these methods do not allow refinement in further subclasses, which would require the introduction of more *paintLabel*-like methods.

The necessity of methods like *paintLabel* highlights the overall philosophy of class extension in Beta-like languages: *superclass implementors know better*. Methods are not overrideable by default, so they cannot be replaced completely by subclasses, which tends to maximize the reliability of a class hierarchy. The only way that a superclass can release its control over behavior is to use **inner** with the default work in a new method. As we demonstrated in the previous section, Java-style method extension more directly express a programmer's intent in such cases.

2.3 Widgets in a Beta/Java Combination

In a sufficiently large program, the Java philosophy is right at times, and the Beta philosophy is right at other times; sometimes the subclass implementor knows better, and sometimes it is the superclass implementor. By including both Java-style and Beta-style method refinement in a programming language, we can support different philosophies for different parts of the program. Indeed, these philosophies can be mixed at a fine granularity by allowing a programmer to annotate individual method implementations as **java** or **beta**.[1] The resulting system is consistent and, we believe, conceptually simple.

Since the initial *Window* class was more cleanly implemented in Java, we begin our widget implementation with a **java** implementation of *paint*:

[1] We expect that any realistic language will have better keywords than **java** and **beta**, of course.

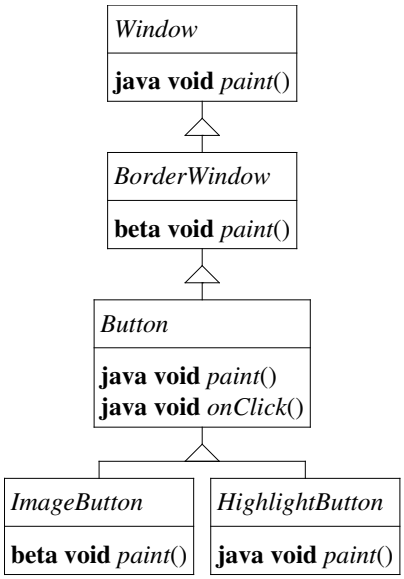

Figure 5. Hierarchy with both Java- and Beta-style methods

```
class Window { ...
    java void paint() {
        ...; // paint the background
    }
}
```

The **java** annotation indicates that a subclass can override this method. If the overriding method wants to use the original method, it can use **super**.

The *BorderWindow* class overrides *paint*, but its own implementation should never be overridden completely in later subclasses. That is, Beta more neatly implements the *paint* method for *BorderWindow*, so we annotate the implementation with **beta** and use **inner** in the body:

```
class BorderWindow extends Window { ...
    beta void paint() {
        super.paint(); // paint background
        ...; // draw a border
        inner.paint();
    }
}
```

The **beta** annotation indicates that this implementation of *paint* cannot be overridden. Operationally, when the *paint* method is called on an instance of *BorderWindow* or any subclass of *BorderWindow*, control jumps to *BorderWindow*'s implementation. This implementation, in turn, invokes the implementation in *Window*, then paints the border, and then allows control to jump to a subclass implementation.

The *Button* class, for one, accepts that control:

```
class Button extends BorderWindow { ...
    java void paint() {
        ...; // draw a button label
    }
}
```

Thus, when *paint* is called on an instance of *Button*, control initially jumps to *paint* in *BorderWindow*, but ultimately it arrives at *paint* in *Button*. The method in *Button* is declared **java**, however, so that a subclass can completely replace the button part of the method.

The *ImageButton* class completely replaces *paint* in *Button*, forcing any subclasses to draw an image. *HighlightButton* uses **super** to extend button painting, rather than replacing it entirely:

```
class ImageButton extends Button { ...
    beta void paint() {
        ...; // draw image (don't call super for label)
        inner.paint();
    }
}

class HighlightButton extends Button { ...
    java void paint() {
        ...; // replace the background with dark blue
        super.paint(); // draws the label
        ...; // draw light blue shading on top of the label
    }
}
```

When *paint* is called on an *ImageButton* instance, control jumps to *BorderWindow*, then to *Window*, then back through *BorderWindow* to *ImageButton*. When *paint* is called on a *HighlightButton* instance, control jumps to *BorderWindow*, then to *Window*, then back through *BorderWindow* to *HighlightButton*, then temporarily to *Button*, and finally back to *HighlightButton*.

At every point in this class derivation, a programmer specifies exactly the intent for refinement in subclasses. While the overall flow of control through methods can be complex, it is locally apparent what results will be achieved. In our example, it is clear that a *BorderWindow* subclass always uses *BorderWindow*'s *paint*, whereas a *Button* subclass has the option to replace *paint*.

The widget example shows how **beta** may be used any number of times. The implementor of *ImageButton* decided that drawing the images is mandatory, implementing this intent by annotating the method with **beta**. Because of this annotation, when *paint* is called for a subclass of *ImageButton*, control first jumps to *BorderWindow* (as required by the *BorderWindow* implementor), but always to *ImageButton* before any subclass of *ImageButton*.

As shown earlier, it is possible to simulate one form of method extension in a language that has the other form. The simulation is awkward compared to directly expressing the intended mode of method refinement. Furthermore, after a method has been written without the simulation pattern, converting it to use a simulation pattern requires extensive modification to descendant classes (since they must use the new method name introduced by the simulation). In contrast, changing a **beta** annotation to **java** (or vice-versa) requires modifying only classes that directly refine the changed method, and not the decedents of those classes.

With **java** and **beta** annotations, **final** is no longer necessary; a **final** method is simply a **beta** method that contains no calls to **inner**. In the same way that a Java compiler rejects overriding of a **final**, a compiler could statically reject declaration of a method in a subclass when a superclass has previously declared the method **beta** with no **inner** call.

3 From Java to a Beta/Java Combination

Syntactically, the difference between Java and our extension is the addition of **java** and **beta** keywords for methods in classes, plus the addition of an **inner** expression form:

Expression = **inner** . *Identifier* (*Expression*, ... *Expression*)
else *Statement*

An **inner** expression can appear only if the enclosing class contains a **beta** declaration for *Identifier*, or if such a declaration is inherited and the enclosing class contains no **java** declaration of *Identifier*. (Using **inner**.*m* outside of method **beta** *m* would be unusual, much like using **super**.*m* outside of a method *m*.) If a method has neither a **beta** nor **java** annotation, **java** is assumed.

3.1 Method Dispatch

Dynamically, the difference between Java and our extension to Java is in method dispatch, including support for **inner**. If a program contains no **beta** annotations (and therefore no **inner** expressions), then method dispatch proceeds exactly as in Java:

- A method call (of the form *expr*.*methodName*) uses the class of the method's object to determine the target method implementation. The target is the implementation in the superclass that is closest to the instantiated class.
- Each **super** call is resolved statically to a method implementation in the closest superclass.

If every method of a program is annotated **beta**, then method dispatching proceeds as in Beta:

- A method call uses the first implementation of a method in the class derivation, starting from the root class. This target can be resolved statically, assuming that the object expression's type is a class (as opposed to an interface).
- An **inner** call, in contrast, must use the class of *this* to find the target method. The target is the implementation in the subclass closest to the class that contains the **inner** call. If no target exists, then the default expression is evaluated.

Figure 6 shows one chain in our example GUI widget class hierarchy. The extra classes *ImagePopup* and *GrayImagePopup* illustrate further uses of **beta** and **java**. Arrows on the left side of the figure show how **inner** calls for *paint* jump from one class to another (the numbers will be explained in section 3.2), ending at an arrowhead, and arrows on the right show how **super** *paint* calls jump:

- A **super** call (arrow on the right) behaves exactly as in Java, always jumping to a statically determined implementation in a superclass. We disallow **super** calls to **beta** declarations, because they are not useful in our experience, and because they tend to produce infinite loops that are difficult to debug.
- An **inner** call (arrow on the left) is slightly different than in Beta, because the target is not always in the closest subclass. Instead, the target is the closest subclass that declares the method **beta**, or the *farthest* subclass if no subclass contains a **beta** declaration of the method.
- A method call in a mixed environment behaves much like an **inner** call, where the target of the initial call is the first **beta** implementation of the method if one exists, and the last **java** implementation otherwise.

External dispatch and **inner** go to the highest **beta** implementation of a method, because the programmer's intent in using **beta** is to

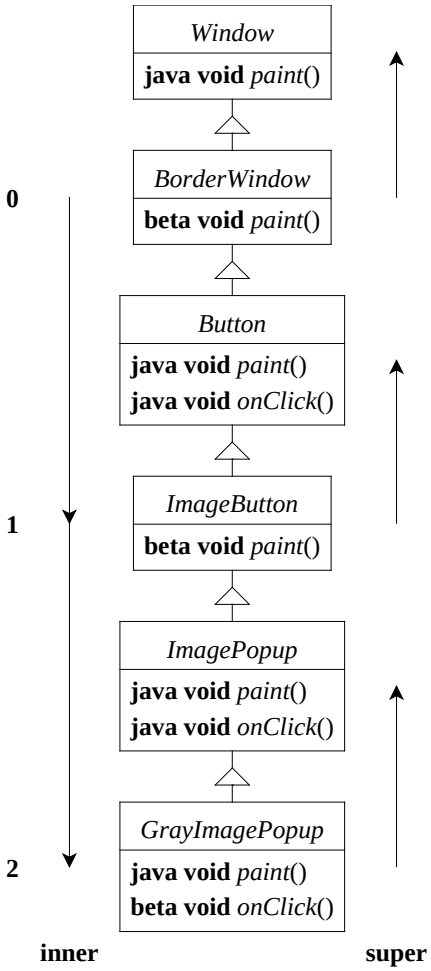

Figure 6. Method dispatch example, with arrows for inner and super calls inside the *paint* method

ensure that the code in that method will get called, no matter how subclasses refine the method. Ultimately, declaring a method **beta** should trump any future attempts to override it, because the enforced behavior may be necessary for the program to behave correctly.

An **inner** or external dispatch skips **java** methods, because the programmer's intent when using **java** is to allow overriding. The skipped **java** implementations are used only if the **inner** target chooses to call **super**.

3.2 Compiling Method Dispatch

Typically, dynamic method dispatch in Java uses a virtual method table, where a target method implementation is obtained by extracting it from a particular slot in the table. This strategy still works with **beta** methods, and the only change is in the construction of the table. Instead of installing the last **java** implementation of a method into the table, the first **beta** method (if any) should be installed. The relevant **beta** method resides in a superclass, so incremental compilation of classes in a hierarchy is the same as in Java.

An **inner** call also needs a dynamic dispatch table, but the **inner** table is slightly more complex. The target of an **inner** call is determined by both the class of the object on which the **inner** is called and the class declaration in which the **inner** call appears. Thus, the **inner** dispatch table is not simply linear in the number of **beta** methods. In fact, for each method, the table contains an array of target methods. An **inner** call can be mapped statically to an index for the method's array, where the index is the total number of **beta** declarations of the method in the **inner** call's class and its superclasses. Meanwhile, index 0 corresponds to the target for external method calls.

	0	1	2
paint	*BorderWindow*'s	*ImageButton*'s	*ImagePopup*'s
onClick	0		
	ImagePopup's		

Figure 7. Dispatch table for *ImageButton* in Figure 6

	0	1	2
paint	*BorderWindow*'s	*ImageButton*'s	*GrayImagePopup*'s
onClick	0	1	
	GrayImagePopup's	`null`	

Figure 8. Dispatch table for *GrayImageButton* in Figure 6

For example, when the *paint* method is called for an instance of *GrayImagePopup*, the numbers in Figure 6 correspond to the indices. An external call always starts with index 0, at *BorderWindow*. An **inner** call in *BorderWindow* jumps to the method at index 1, because the *BorderWindow* introduces the first **beta** declaration of *paint*. An **inner** call in *ImageButton* jumps to the method at index 2, because *ImageButton* introduces the second **beta** declaration of *paint*.

In general, for a particular method, class, and **inner** array index, the dispatch table contains one of three values:

- It contains null if no further refinements of the method are declared below **inner** calls that use the index.
- It contains the first **beta** declaration of the method below the **inner** call, if any such declaration exists.
- It contains the last **java** declaration of the method below the **inner** call, if any such delectation exists, and if no **beta** declaration is available.

Figure 7 shows the complete method dispatch table for *ImagePopup*, and Figure 8 shows the dispatch table for *GrayImagePopup*. In both cases, index 0 contains *BorderWindow*'s method for *paint*, and index 1 contains *ImageButton*'s method. At index 2, the *ImagePopup* table contains the implementation from *ImagePopup*, but it is replaced by *GrayImagePopup*'s implementation in the table for *GrayImagePopup*. Similarly, index 0 for *onClick* contains *ImagePopup*'s implementation in *ImagePopup*'s table, but it is overridden with *GrayImagePopup*'s implementation in *GrayImagePopup*'s table. Finally, index 1 for *onClick* in *GrayImagePopup* contains null, because no method refines the **beta** declaration of *onClick* in *GrayImagePopup*.

To explain table construction another way, a **beta** method occupies an index permanently, in all subclasses, and increases the size of the method's array, whereas a **java** method occupies an index only until it is replaced by a subclass implementation. If no methods are declared **beta** (either in the whole program or for a particular method), then this algorithm degenerates to the usual Java-style algorithm (either for the whole table or for an individual row). For an instance of *ImagePopup*, all declarations of *onClick* are **java**, so the *onClick* row in Figure 7 has a single slot, just as in a Java dispatch table.

In the case of *paint* for *ImagePopup*, every **inner** call has a target method, since the last *paint* method in the chain is declared **java**. The last *onClick* method of *GrayImagePopup*, however, is declared **beta**. If the *GrayImagePopup* contains an **inner** call for *onClick*, there is no target method, which means that the **inner**'s **else** expression is used at run time. This lack of a target is reflected by a null pointer for index 1 in the *onClick* row of the dispatch table. Thus, an **inner** call at run time first checks whether the relevant table slot is null; if so, it uses the **else** expression, otherwise it jumps to the table-indicated method.

A Java-style dispatch table always has size $O(m)$ for m distinct methods in the class, but the size of a dispatch table with **inner** depends on both the number of methods in the class and the number of **beta** implementations of the method. A two-dimensional array for the **inner** table would thus require $O(m \times n)$ space for m methods and a maximum **beta** depth of n. Our implementation uses an array of arrays, instead (as suggested by Figure 7 and Figure 8), so that the size is $O(m+p)$ for p total **beta** declarations, which tends to be much smaller than $O(m \times n)$.

3.3 Interfaces

The **beta** and **java** keywords apply only to method implementations in a class. Because interfaces reflect subtyping and not behavior, these keywords are not needed in an interface declaration. A method call through an interface behaves the same as an external method call using the object's type. In compilation terms, interface dispatch needs only the implementation that is stored in a virtual method table, so interface-based method calls are effectively unchanged compared to Java.

3.4 Differences from Beta

Technically, even for a program that contains only **beta** methods, our language differs from Beta in two respects that are unrelated to method dispatch:

- Our **inner** form contains explicit arguments, instead of implicitly using the enclosing method's arguments (or, more precisely, the current values of the argument variables, in the case that the variables have been assigned). This form of **inner** call more closely parallels **super**, allows the values passed to **inner** to be changed non-imperatively, and allows an **inner** call for a particular method to appear in any other method (again, like **super**).
- Our **inner** form includes a default expression to evaluate when no subclass implementation is available, whereas Beta defaults to a null operation. We include a default expression to make the language more value oriented.

$$
\begin{array}{rcl}
P & = & \mathit{defn} \ldots \mathit{defn}\ e \\
\\
\mathit{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c\ \{\ \mathit{meth} \ldots \mathit{meth}\ \} \\
\mathit{meth} & = & \mathit{kind}\ t\ m(t\ \mathit{var}, \ldots t\ \mathit{var})\{\ e\ \} \\
\mathit{kind} & = & \textbf{beta} \mid \textbf{java} \\
\\
c & = & \text{a class name or } \textbf{Object} \\
\mathit{md} & = & \text{a method name} \\
\mathit{var} & = & \text{a variable name or } \textbf{this} \\
t & = & c \\
\\
e & = & \mathit{var} \\
 & \mid & \textbf{null} \\
 & \mid & \textbf{new}\ c \\
 & \mid & e.\mathit{md}(e, \ldots e) \\
 & \mid & \textbf{super}\ \mathit{var}{:}c.\mathit{md}(e, \ldots e) \\
 & \mid & \textbf{inner}\ \mathit{var}{:}c.\mathit{md}(e, \ldots e)\ \textbf{else}\ e \\
\\
\check{e} & = & e \mid \bot \\
\check{c} & = & c \mid \bot \\
\hat{c} & = & c \mid \top
\end{array}
$$

Figure 9. Syntax of BETAJAVA

4 BetaJava Model

To demonstrate type soundness of our combination of Beta-style and Java-style methods, we define a complete formal model for BETAJAVA in the style of CLASSICJAVA [14].

The model simplifies Java considerably, eliminating constructs that are irrelevant to method dispatch. For example, the model does not include local variables, **if** statements, or exceptions. Unlike CLASSICJAVA, the BETAJAVA model further omits fields, but we have preserved enough CLASSICJAVA structure in our BETAJAVA model to ensure that fields could be added back to the model, exactly as they appear in CLASSICJAVA. We also omit interfaces from BETAJAVA, because **beta** and **java** play no role in interface declarations.

Figure 9 contains the syntax of BETAJAVA programs in our model. A program P consists of a sequence of class declarations followed by a single expression. The expression plays the role of `main` to start the program. Each class declaration contains a sequence of method declarations, and each method is annotated with either **beta** or **java**. A method body consists of a single expression, which is either a variable (i.e., a reference to a method argument or **this**), the **null** keyword, an object creation **new** c, or a method call. Each method call has one of three shapes:

- A method call of the form $e.\mathit{md}(e_1, \ldots e_n)$ is a normal call to the method *md* in the object produced by e.
- A method call **super** $\mathit{var}{:}c.\mathit{md}(e_1, \ldots e_n)$ must appear only within a method (as enforced by the type system). The *var* part of the call is intended to be **this**, which is implicit in Java. An explicit target simplifies our evaluation rules, but **this** could be inserted automatically by an elaboration step, as in CLASSICJAVA. Similarly, the class c named in a **super** call must be the name of the containing class's immediate superclass. Again, c could be inserted by elaboration, and our type system ensures that the correct c is named.
- A method call **inner** $\mathit{var}{:}c.\mathit{md}(e_1, \ldots e_n)$ **else** e must appear only within a method (again, as enforced by the type system). As with **super** calls, *var* is intended to be **this**, and a class

c is named for the convenience of our evaluation rules. For **inner**, the given *c* must be the class containing the **inner** call, as opposed its superclass, and the type system ensures this correlation. The extra **else** *e* at the end of an **inner** call provides the expression to evaluate if, at run time, no extending method is provided by *var* (in a subclass of *c*).

The non-terminals $\check{e}$, $\check{c}$, and $\hat{c}$ in Figure 9 are for auxiliary relations in the evaluation rules, and they are not part of the concrete syntax.

4.1 BetaJava Type Checking

The type-checking rules for BETAJAVA closely resemble those of CLASSICJAVA, building on a number of simple predicates and relations that are defined in Figure 10. For example, the CLASSESONCE(P) predicate checks that each class name is defined only once. The $\prec_P$ relation associates each class in the program P with the class that it extends, and $\Subset_P$ captures the method declarations of P.

The $\leq_P$ relation fills out the subclass relationship as the transitive closure of $\prec_P$. (The extension to $\top$ and $\bot$ is used in the evaluation rules.) Two additional predicates check global properties: CLASSESDEFINED(P) ensures that the class hierarchy of P forms a tree, and EXTENSIONSCONSISTENT(P) ensures that every declaration of a method in a class derivation uses the same signature.

Finally, the $\in_P$ relation combines the methods that a class declares with the methods that it inherits from its superclasses. The $\in_P$ relation can be interpreted as a function from classes to method-tuple sets, or as a function from class–method combinations to method implementations. Specifically, for a class *c* and method name *md*, $\in_P$ locates the most specific implementation of *md*, which is the one declared closest to *c* in the class hierarchy. This relation is the same as in CLASSICJAVA, except that the method implementation's *kind* is included on the left-hand side of the relation.

Complete type-checking rules for BETAJAVA appear in Figure 11. The rules include the following judgments:

$\vdash_{\mathsf{p}} P$	program P is well-typed
$P \vdash_{\mathsf{d}} \mathit{defn}$	class *defn* in program P has well-typed methods
$P, c \vdash_{\mathsf{m}} \mathit{meth}$	method *meth* in class *c* has a well-typed body
$P, \Gamma \vdash_{\mathsf{e}} e : t$	expression *e* has type *t* in environment Γ
$P, \Gamma \vdash_{\mathsf{s}} e : t$	the type of *e* is a subtype of *t* in environment Γ

To summarize the type rules, a program is well-typed if its class definitions are well-typed and its final expression is well-typed in an empty environment. A class definition is well-typed when its methods are well-typed. A method is well-typed when its body is well-typed in an environment that includes the method's arguments. For expressions, a **null** or **new** *c* expression is always well-typed, and *var* is well-typed if it is bound in the environment.

A call to a method *md* is well-typed if the $\in_P$ relation finds a consistent declaration of *md* in a particular class. In the case of a normal method call, the class is determined by the type of the target object. In the case of a **super** call, the class is named in the call, and it must be the superclass of the type of **this** (where the type of **this** effectively names the class where the **super** call appears). In the case of an **inner** call, the class *c* is named in the call, *c* must be the type of **this**, and the most-specific *md* for *c* must have kind **beta**.

4.2 BetaJava Evaluation

The relations $\in_P^{\mathsf{j}}$ and $\in_P^{\mathsf{b}}$ capture the essence of **beta**-sensitive method dispatch (see Figure 12). The $\in_P^{\mathsf{j}}$ and $\in_P^{\mathsf{b}}$ relations find a **java** or **beta** method only between classes c' and c'' in a chain of class extensions. These relations also accept a default expression e' to use if no method can be found. (These pieces are assembled as $[c', c''](e')$ to the right of $\in_P^{\mathsf{j}}$ or $\in_P^{\mathsf{b}}$.)

The two relations implement a two-phase search for a method. The $\in_P^{\mathsf{b}}$ relation first attempts to find a **beta** method, and if the search fails, it delegates to $\in_P^{\mathsf{j}}$ to find a **java** method. The $\in_P^{\mathsf{b}}$ relation uses $\max_{\leq_P}$ to find the highest **beta** method in a class derivation (i.e., closest to the root class), while $\in_P^{\mathsf{j}}$ uses $\min_{\leq_P}$ to find the lowest method (i.e., closest to the instantiated class).

Since it implements the Java-like part of method search, $\in_P^{\mathsf{j}}$ relation is similar to $\in_P$, except that it takes into account an upper bound c'' and a default expression. The upper bound corresponds to a class with an **inner** call, where a legal target method must appear in a subclass. If no method is found and the default expression is used, then arbitrary "method" arguments are selected by the relation, with the constraint that the argument variables do not appear in the expression.

The $\in_P^{\mathsf{b}}$ relation searches primarily for a **beta** method below the upper bound c''. If no **beta** method is found, $\in_P^{\mathsf{b}}$ uses $\in_P^{\mathsf{j}}$ to search for a **java** method, instead. Meanwhile, the default expression passed to $\in_P^{\mathsf{j}}$ corresponds to the default expression for an **inner** call, in case neither kind of method is found.

Using these two relations, the operational semantics for BETAJAVA is defined as a contextual rewriting system on pairs of expressions and stores. As in CLASSICJAVA, a store Σ is a mapping from generated *var*s to class-tagged records. Since the BETAJAVA model does not include fields or field assignments, the store is technically unnecessary, but we preserve it for consistency with CLASSICJAVA.

The complete evaluation rules are in Figure 13. Normal method calls use $\in_P^{\mathsf{b}}$ with **Object** as the upper bound, which finds either the first **beta** method or the last **java** method. An **inner** call also uses $\in_P^{\mathsf{b}}$, but with the class named in the call as an exclusive upper bound for finding a method. A **super** call merely uses $\in_P$, as in CLASSICJAVA, reflecting that **super** dispatch behaves as in Java.

The static and dynamic nature of method calls is apparent in the model's relations. For example, the use of $\in_P$ for **super** calls relies on no dynamic information, so it can be computed statically. Similarly, the result of $\in_P^{\mathsf{b}}$ for a method call can be precomputed if the type of the object expression includes a **beta** method; at run-time, the class will be a subtype of the static type, but the subtype cannot override the **beta** method. In contrast, the result for $\in_P^{\mathsf{b}}$ in an **inner** call cannot be pre-computed from just the object expression's type.

4.3 BetaJava Soundness

For a well-typed program, evaluation can either produce a value, loop indefinitely, or get stuck attempting to call a method of **null**. The last possibility would correspond to a run-time error in a Java implementation. These type rules preclude a "method not understood" run-time error, however, which is the essence of soundness for an object-oriented language.

$\textsc{ClassesOnce}(P)$ **iff** (**class** c $\cdots$ **class** c' is in P) implies $c \neq c'$

$\textsc{ObjectBuiltin}(P)$ **iff** **class Object** is not in P

$\textsc{MethodsOncePerClass}(P)$ **iff**
(**class** c **extends** c' $\{\cdots\ t_1\ md_1\ \cdots\ t_2\ md_2\ \cdots\}$ is in P) implies $md_1 \neq md_2$

$\textsc{ClassesDefined}(P)$ **iff** (c is in P) implies $c =$ **Object** or (**class** c is in P)

$c \prec_P c'$ **iff** **class** c **extends** c' is in P

$\langle m, kind, (t_1 \ldots\ t_n \to t_0), (var_1, \ldots var_n), e\rangle \notin_P c$ **iff**
class c **extends** c' $\{\cdots\ kind\ t_0\ md(t_1\ var_1, \ldots t_n\ var_n)\{\ e\ \}\ \cdots\}$ is in P

$\leq_P$ = transitive–reflexive closure of $\prec_P$ plus $c \leq_P \top$ and $\bot \leq_P c$

$<_P$ = irreflexive restriction of $\leq_P$

$\textsc{WellFoundedClasses}(P)$ **iff** $\leq_P$ is antisymmetric

$\textsc{ExtensionsConsistent}(P)$ **iff**
($\langle m, kind, (t_1 \ldots\ t_n \to t_0), (var_1, \ldots var_n), e\rangle \notin_P c_1$)
and ($\langle m, kind', (t'_1 \ldots\ t'_n \to t'_0), (var'_1, \ldots var'_n), e'\rangle \notin_P c_2$)
implies ($c_1 \not\leq_P c_2$ or $(t_1 \ldots\ t_n \to t_0) = (t'_1 \ldots\ t'_n \to t'_0)$)

$\langle m, kind, (t_1 \ldots\ t_n \to t_0), (var_1, \ldots var_n), e\rangle \in_P c$ **iff**
$c = \min_{\leq_P}(\{c \mid c' \leq_P c$ and $\langle m, kind, (t'_1 \ldots\ t'_n \to t'_0), (var'_1, \ldots var'_n), e''\rangle \notin_P c\})$
and $\langle m, kind, (t_1 \ldots\ t_n \to t_0), (var_1, \ldots var_n), e\rangle \notin_P c$

Figure 10. Predicates and relations for BETAJAVA

$$\frac{P \vdash_{\mathsf{d}} defn_1 \ldots \quad P \vdash_{\mathsf{d}} defn_n \quad P, \emptyset \vdash_{\mathsf{e}} e : t}{\vdash_{\mathsf{p}} P}$$

where $P = defn_1 \ldots\ defn_n\ e$, $\textsc{ClassesOnce}(P)$, $\textsc{ObjectBuiltin}(P)$, $\textsc{MethodsOncePerClass}(P)$, $\textsc{ClassesDefined}(P)$, $\textsc{WellFoundedClasses}(P)$, and $\textsc{ExtensionsConsistent}(P)$

$$\frac{P, \textbf{this}{:}c, var_1{:}t_1, \ldots var_n{:}t_n \vdash_{\mathsf{s}} e : t_0}{P, c \vdash_{\mathsf{m}} kind\ t_0\ md(t_1\ var_1, \ldots t_n\ var_n)\{\ e\ \}} \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e : c'}{P, \Gamma \vdash_{\mathsf{s}} e : c} \text{ where } c' \leq_P c$$

$$P, \Gamma \vdash_{\mathsf{e}} \textbf{new}\ c : c \qquad P, \Gamma \vdash_{\mathsf{e}} \textbf{null} : c \qquad P, \Gamma \vdash_{\mathsf{e}} var : t \text{ where } \Gamma(var) = t$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e : c \quad P, \Gamma \vdash_{\mathsf{s}} e_1 : t_1 \ \ldots \quad P, \Gamma \vdash_{\mathsf{s}} e_n : t_n}{P, \Gamma \vdash_{\mathsf{e}} e.md(e_1, \ldots e_n) : t_0} \text{ where } \langle md, kind, (t_1 \ldots\ t_n \to t_0), (var_1, \ldots var_n), e_0\rangle \in_P c$$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e_1 : t_1 \ \ldots \quad P, \Gamma \vdash_{\mathsf{s}} e_n : t_n}{P, \Gamma \vdash_{\mathsf{e}} \textbf{super}\ var{:}c.md(e_1, \ldots e_n) : t_0}$$
where $\Gamma(var) = c'$, $c' \prec_P c$ and $\langle md, kind, (t_1 \ldots\ t_n \to t_0), (var_1, \ldots var_n), e\rangle \in_P c$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e_1 : t_1 \ \ldots \quad P, \Gamma \vdash_{\mathsf{s}} e_n : t_n \quad P, \Gamma \vdash_{\mathsf{s}} e_0 : t_0}{P, \Gamma \vdash_{\mathsf{e}} \textbf{inner}\ var{:}c.md(e_1, \ldots e_n)\ \textbf{else}\ e_0 : t_0}$$
where $\Gamma(var) = c$ and $\langle md, \textbf{beta}, (t_1 \ldots\ t_n \to t_0), (var_1, \ldots var_n), e\rangle \in_P c$

Figure 11. Type-checking rules for BETAJAVA

$\langle m, var_1, \ldots var_n, \check{e}\rangle \in^{\mathsf{j}}_P [c', c''](\check{e}')$ **iff**
$\hat{c} = \min_{\leq_P}(\{\top\} \cup \{c \mid c' \leq_P c \text{ and } c <_P c'' \text{ and } \langle m, kind, (t'_1 \ldots\ t'_n \to t'_0), (var'_1, \ldots var'_n), e''\rangle \notin_P c\})$
and $(\langle m, kind, (t_1 \ldots\ t_n \to t_0), (var_1, \ldots var_n), \check{e}\rangle \notin_P \hat{c}$
or $(\hat{c} = \top, \check{e} = \check{e}',$ and $var_1, \ldots var_n$ not in $e))$

$\langle m, var_1, \ldots var_n, \check{e}\rangle \in^{\mathsf{b}}_P [c', c''](\check{e}')$ **iff**
$\check{c} = \max_{\leq_P}(\{\bot\} \cup \{c \mid c' \leq_P c \text{ and } c <_P c'' \text{ and } \langle m, \textbf{beta}, (t'_1 \ldots\ t'_n \to t'_0), (var'_1, \ldots var'_n), e''\rangle \notin_P c\})$
and $(\langle m, \textbf{beta}, (t_1 \ldots\ t_n \to t_0), (var_1, \ldots var_n), \check{e}\rangle \notin_P \check{c}$
or $(\check{c} = \bot$ and $\langle m, var_1, \ldots var_n, \check{e}\rangle \in^{\mathsf{j}}_P [c', c''](\check{e}'))$

Figure 12. Relations for BETAJAVA evaluation

$$
\begin{array}{rcl@{\qquad}rcl}
v & = & \textbf{null} & E & = & [\,] \\
 & | & var & & | & E.md(e, \ldots e) \\
 & & & & | & v.md(v, \ldots v, E, e, \ldots e) \\
 & & & & | & \textbf{super}\ v{:}c.md(v, \ldots v, E, e, \ldots e) \\
 & & & & | & \textbf{inner}\ v{:}c.md(v, \ldots v, E, e, \ldots e)\ \textbf{else}\ e
\end{array}
$$

$P \vdash \langle E[\textbf{new}\ c], \Sigma\rangle \longmapsto_{\mathsf{bj}} \langle E[var], \Sigma[var \leftarrow \langle c\rangle]\rangle$
where $var \notin \mathrm{dom}(\Sigma)$

$P \vdash \langle E[var.md(v_1, \ldots v_n)], \Sigma\rangle \longmapsto_{\mathsf{bj}} \langle E[e[var_n \leftarrow v_n] \ldots [var_1 \leftarrow v_1][\textbf{this} \leftarrow var]], \Sigma\rangle$
where $\Sigma(var) = \langle c\rangle$ and $\langle md, var_1, \ldots var_n, e\rangle \in^{\mathsf{b}}_P [c, \textbf{Object}](\bot)$

$P \vdash \langle E[\textbf{super}\ var{:}c.md(v_1, \ldots v_n)], \Sigma\rangle \longmapsto_{\mathsf{bj}} \langle E[e[var_n \leftarrow v_n] \ldots [var_1 \leftarrow v_1][\textbf{this} \leftarrow var]], \Sigma\rangle$
where $\langle md, \textbf{java}, (t_1 \ldots\ t_n \to t_0), (var_1, \ldots var_n), e\rangle \in_P c$

$P \vdash \langle E[\textbf{inner}\ var{:}c.md(v_1, \ldots v_n)\ \textbf{else}\ e'], \Sigma\rangle \longmapsto_{\mathsf{bj}} \langle E[e[var_n \leftarrow v_n] \ldots [var_1 \leftarrow v_1][\textbf{this} \leftarrow var]], \Sigma\rangle$
where $\Sigma(var) = \langle c_0\rangle$ and $\langle md, var_1, \ldots var_n, e\rangle \in^{\mathsf{b}}_P [c_0, c](e')$

Figure 13. Evaluation rules for BETAJAVA

Theorem 1 (Type Soundness): If $\vdash_{\mathsf{p}} P$ where $P = defn_1 \ldots\ defn_n\ e$, then either

- $P \vdash \langle e, \emptyset\rangle \longmapsto\!\!\to_{\mathsf{bj}} \langle v, \Sigma\rangle$ for some v and Σ;
- $P \vdash \langle e, \emptyset\rangle \longmapsto\!\!\to_{\mathsf{bj}} \langle e', \Sigma\rangle$ implies $P \vdash \langle e', \Sigma\rangle \longmapsto_{\mathsf{bj}} \langle e'', \Sigma'\rangle$ for some e'' and Σ'; or
- $P \vdash \langle e, \emptyset\rangle \longmapsto\!\!\to_{\mathsf{bj}} \langle E[\textbf{null}.md(v_1, \ldots v_n)], \Sigma\rangle$ for some E, md, $v_1, \ldots v_n$, and Σ.

The main lemma in support of this theorem states that each step taken in the evaluation preserves the type correctness of the expression-store pair (relative to the program) [26]. Specifically, for a configuration on the left-hand side of an evaluation step, there exists a type environment that establishes the expression's type as some t. This environment must be consistent with the store.

The soundness proof for CLASSICJAVA [13] is easily adapted to BETAJAVA. The **super** rule is unchanged, so the the proof that a method is found is also unchanged. The normal- and **inner**-call forms use the new method-finding relation $\in^{\mathsf{b}}_P$, but $\in^{\mathsf{b}}_P$ finds a method anytime that $\in_P$ finds one, and if different implementations are found, then EXTENSIONSCONSISTENT(P) ensures that the types are consistent.

5 Implementation and Experience

MzScheme is the base language for the PLT Scheme programming suite, which includes DrScheme [11]. MzScheme extends the standard Scheme language [19] with numerous constructs, including a Java-like object system.[2] The object system is used primarily to implement DrScheme's graphical interface.

5.1 Base Implementation

Classes and objects in MzScheme are dynamically typed, which means that a method call from outside an object typically requires a dynamic method-name lookup. Self and **super** calls within an object, however, are always resolved at class-construction time. A self call uses a virtual method table indirection, and a **super** call is a direct function call. In short, these calls are implemented as in a statically typed object-oriented language, such as Java.

Classes are values in MzScheme, and the superclass position in a class declaration can be an arbitrary expression. Consequently, a *mixin* can be defined by placing a **class** expression within a procedure that accepts a superclass argument. DrScheme uses this form of mixin extensively. For example, the "autosave" behavior for a text editor is implemented as a mixin, so that autosaving can be added to any class that implements the text-editor interface. Add-on *tools* for DrScheme introduce new mixins to extend DrScheme's behavior.

The autosave mixin must extend the editor's *on-close* method, which is called when the editor's window is closed, so that the autosave timer is disabled. In previous versions of DrScheme, only Java-style methods were available, so a mixin that overrides *on-close* was obligated to call the superclass method. Failing to call the superclass method in a tool-introduced extension would create a

[2] Technically, the object system is an external library.

	allow override	*allow augment*
new method	**public**	**pubment**
override existing	**override**	**overment**
augment existing	**augride**	**augment**

Figure 14. Method keywords in MzScheme

bug or resource leak in DrScheme's core, and such a leak appeared in practice. Fixing the bug was trivial, but discovering the bug was difficult, because the mixin implementor naturally concentrated on testing the mixin's own *on-close* behavior. Many other *on-...* methods in DrScheme have the same protocol, with the same danger of errors.

5.2 Adding Inner

An **inner** method call is implemented in MzScheme using an auxillary method table, as described in Section 3.2. Overall, to implement a prototype combination of **beta** and **inner**, we added or changed roughly 100 lines of Scheme macro code in the 2800-line implementation of MzScheme's object system. Our production version added another 150 lines of code.

Before adding **inner** to MzScheme, each method was declared as either **public** or **override**. A **public** declaration indicates that the method is new in the class, whereas **override** indicates that the method should be declared already in the superclass (in which case **super** can be used). This distinction is statically apparent in Java, but not in MzScheme, due to MzScheme's form of classes as values.

After adding **inner** to MzScheme, a method declaration must declare whether the method is new, overriding, or augmenting, and also whether subclasses are allowed to override or augment the method. We defined a different keyword for each combination, as shown in Figure 14. For example, the **pubment** keyword means "new *pub*lic method, allow aug*ment* only."

We are converting many of DrScheme's *on-...* methods from **public** to **pubment**, thus eliminating the potential for bugs in DrScheme's core due to a missing call to a superclass method in an add-on tool. We expect also to simplify the set of methods in our classes, much as **beta** methods eliminated the need for a *paintInner* method in the example of Section 2.

6 Related Work

Smalltalk [16] was the first language to popularize extension as overriding behavior. This branch of extension has inspired many languages, including C++ [25] and Java [17]. Although many of these languages added additional features such as multiple inheritance and mixins, they all maintained overriding as the only form of method refinement.

CLOS [18] is another language on the Smalltalk branch of method extension, but it supports an **inner**-like call through an *:around* qualifier and *call-next-method*. Normally, *call-next-method* acts like **super** in Java. In the case where *call-next-method* is used in the least specific *:around* method, the most specific method without a qualifier will be called. This pattern simulates a single **inner** call, but there is no way to simulate multiple **inner** calls that move down a class hierarchy, nor is there a way to use *call-next-method* as a **super**-like call and an **inner**-like call in the same method.

Beta [20] inspired gbeta [9]. In gbeta, methods are treated as a sequence of method bodies (which gbeta calls *mixins*); an **inner** statement goes from the current method body to the next one in the sequence. By default, the methods are ordered from the first declaration of the method to the last augmentation, but a programmer can control the order through specific merging operators [10]. For example, the programmer can name an individual method body and later add a new body immediately before or after the named one. A programmer can also place a newly declared body at the beginning of the method's sequence. Clearly, our **beta** and **inner** declarations cannot emulate such general merging operations, but gbeta's merging operations also cannot implement our semantics. In particular, gbeta offers no way to ensure that a behavior is never overridden (as guaranteed by our **beta**).

As far we know, no one has created a simple extension to a Beta-like language that allows method overriding and **super** without allowing Beta method to be overridden. We also have found no work adding a simple **inner** extension to a Java-like language.

Cook, in his Ph.D. thesis [7], develops a semantic model of inheritance and uses it in the analysis of various programming languages, including Smalltalk and Beta. He observes that the underlying inheritance mechanisms of Beta and Smalltalk are the same. The difference is in the combination of inherited structure with local definitions: Smalltalk and Beta have inverted inheritance hierarchies, with Beta's superpatterns acting as subclasses and subpatterns as superclasses. Cook's model can express method refinement as either overriding and augmenting, but not both behaviors combined.

Clark [6] describes a functional language with primitives for object-oriented programming. In his language, an extension of a class is a function of the shadowed definitions available in **super** plus the shadowing definitions available to the superclass in **inner**. He briefly addresses the issue of whether a subclass definition should shadow a superclass definition or vice versa, but only to define the choice as one or the other.

Bracha and Cook [5] propose mixins as a method of combining the inheritance mechanisms of Java and Beta. By choosing the correct composition of mixins, a programmer can achieve either Java-like or Beta-like behavior from methods. Ancona and Zucca [3, 2] similarly demonstrate formally how overriding operators can be expressed in a mixin-based framework, but these systems do not allow both accessing behavior from a previously composed mixin's method and accessing behavior from a successively composed mixin's method; the mixin composition operator determines which will occur. Furthermore, a tedious programming pattern is required to simulate Java-style and Beta-style extension, which is only a slight improvement over the programming pattern required in Java to achieve Beta-style refinement.

Duggan [8] describes a language with mixins that includes **super** and **inner** constructs, but the **inner** construct does not correspond directly to Beta's **inner**. Duggan's system includes an operation to combine two mixins, and in the case that the mixins both define a particular method, one definition will override the other. The **inner** keyword allows the overriding definition to make use of the overridden one. Thus, **inner** provides access to a method of a combined mixin, rather than a method of a subclass. (The **super** construct, meanwhile, has the traditional meaning, referring to the base mixin from which a mixin inherits.)

Bettini et al. [4] present an extension to Java based on the *decorator* design pattern. By relying explicitly on a delegation-style of method refinement instead of a more specific method-refinement construct, a class can retain primary control of a method. Their system is not a simple extension of Java, but rather requires a programmer to adopt a drastically different way of thinking about classes in order to use their extension. Also, it is unclear how their extension would work for methods whose return type is not **void**.

The GNU EDMA system [24] supports object-oriented design through a C library of functions to mimic class and object construction. The library provides specific functions to simulate both Java-style and Beta-style method refinement, and no doubt a combination can be implemented. It appears that no such combination has been worked out to date.

7 Conclusion and Future Work

Programmers benefit by having both Java-style method overriding and Beta-style method augmentation within a single programming language. We have shown why such a combination is beneficial, we have demonstrated that our formulation of the combination is sound, and we have shown that it can be implemented with relatively minor changes to an existing Java-like language.

Although we have mainly considered the combination in the context of class-based languages, we have also noted that mixins magnify our interest in the combination. From a mixin perspective, a default expression for **super** may be as useful as a default expression for **inner**, allowing a programmer to apply a mixin to a class without a base method for a **super** call. We see no immediate uses for this generalization, but we intend to watch for places in DrScheme's implementation that could benefit from **super** defaults.

The MzScheme changes described in Section 5 are included in version 299.10 and later. An implementation of Section 4's model of BETAJAVA (using the PLT reduction semantics tool [23]) is available at the following web site:

`http://www.cs.utah.edu/plt/super+inner/`

Acknowledgments

We would like to thank Yang Liu for work on a prototype combination of **beta** and **java** dispatching, Richard Cobbe for his CLASSICJAVA reduction semantics, Erik Ernst for comments and comparisons to gbeta's merging operators, and the anonymous reviewers for their suggestions.

8 References

[1] P. America. Designing an object-oriented programming language with behavioural subtyping. In *Proc. ACM International Workshop on Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[2] D. Ancona and E. Zucca. Overriding operators in a mixin-based framework. In *Proc. Symposium on Programming Language Implementation and Logic Programming*, pages 47–61, 1997.

[3] D. Ancona and E. Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.

[4] L. Bettini, S. Capecchi, and B. Venneri. Extending Java to dynamic object behaviors. In *Proc. Workshop on Object-Oriented Developments*, volume 82. Elsevier, 2003.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, Oct. 1990.

[6] A. Clark. A layered object-oriented programming language. *GEC Journal of Research*, 11(2):173–180, 1994.

[7] W. R. Cook. *A Denotational Semantics of Inheritance*. Ph.D. thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.

[8] D. Duggan. A mixin-based, semantics-based approach to reusing domain-specific programming languages. In *Proc. European Conference on Object-Oriented Programming*, volume 1850 of *LNCS*, pages 179–200, Berlin Heidelberg, 2000. Springer-Verlag.

[9] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[10] E. Ernst. Propagating class and method combination. In *Proc. European Conference on Object-Oriented Programming*, LNCS 1628, pages 67–91. Springer-Verlag, June 1999.

[11] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Proc. Symposium on Programming Language Implementation and Logic Programming*, pages 369–388, Sept. 1997.

[12] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[13] M. Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, 1999.

[14] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 171–183, Jan. 1998.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[16] A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley, Reading, 1989.

[17] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[18] S. E. Keene. *Object-Oriented Programming in Common LISP, a Programmer's Guide to CLOS*. Addison-Wesley, 1988.

[19] R. Kelsey, W. Clinger, and J. Rees (Eds.). The revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), Sept. 1998.

[20] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley, 1993.

[21] B. H. Liskov and J. Wing. Behavioral subtyping using invariants and constraints. Technical Report CMU CS-99-156, School of Computer Science, Carnegie Mellon University, July 1999.

[22] B. H. Liskov and J. M. Wing. A behavioral notion of subtyp-

ing. *ACM Transactions on Computing Systems*, November 1994.

[23] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proc. International Conference on Rewriting Techniques and Applications*, June 2004.

[24] D. M. Oliveira. GNU EDMA. http://www.gnu.org/software/edma/edma.html.

[25] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.

[26] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report 160, Rice University, 1991. *Information and Computation*, volume 115(1), 1994, pp. 38–94.