Safe Dynamic Multiple Inheritance

Erik Ernst

Dept. of Computer Science, University of Aarhus, Denmark

**Abstract.** Combination of descriptive entities—i.e. multiple inheritance and related mechanisms—is usually only supported at compile time in statically typed languages. The language gbeta is statically typed and has supported run-time creation of classes and methods since 1997, by means of the pattern combination operator ‘\&’. However, with certain combinations of operands the ‘\&’ operator fails; as a result, creation of new classes and methods at run-time had to be considered a dangerous operation. This paper presents a large and useful class of combinations, and proves that combinations in this class will always succeed.

1 Introduction

Descriptive entities—such as classes and methods—can be combined in some languages. Multiple inheritance is an example of such a mechanism, combining classes. The pattern combination operator ‘\&’ in gbeta [11, 10] is another such mechanism, capable of combining classes as well as methods since the pattern concept [16, 17, 10] unifies and generalizes the concepts of class and method.

Conventionally, combination of descriptive entities is confined to compile time in languages with static type checking. Pattern combination in gbeta has been supported as a run-time facility since 1997. The resulting patterns could be used safely, subject to the same kind of type checking as all other patterns known only by an upper bound, e.g., open virtual patterns. But the combination operation itself could fail for certain pairs of operands. It is in a sense similar to a simple arithmetic operation like division: The expression \(x/y\) may cause an exception if \(y\) is zero, otherwise the result will be just as safe to use as any other number—only the operation itself is dangerous, not the outcome.

This paper describes a large and useful set of operand pairs to the gbeta operator ‘\&’ where the combination will succeed, thereby enabling programmers to use run-time combination of classes and methods without worrying about failure. One way to describe this set of operand pairs is to say that at least one operand must have been created by single inheritance.

The rest of this paper concentrates on the treatment of a formalization of the pattern combination mechanism and its domain. To set the scene, Sec. 2 describes a larger historic picture in which this work strives to make a small mark. Next, the informal semantics of dynamic pattern combination is introduced in Sec. 3. Section 4 presents the basic domains and operations, such as mixins, patterns, and pattern combination. Section 5 describes a criterion on the operands of a pattern combination operation that ensures successful combination, and proves
the correctness of this claim. Section 6 discusses this result in context of the language \textit{gbeta}. Finally, Sec. 7 presents related work, and Sec. 8 concludes.

2 The Historic Context

Long ago in 1977, a need arose in AI research to handle multiple classification, realized in the knowledge representation language KRL [4]. Four years later, the OO Lisp dialect LOOPS [3] was created, supporting multiple inheritance. In 1982 the Lisp dialect Flavors [8] was created, also with multiple inheritance, and with a programmer culture that emphasized the composition of classes from various ‘flavors’—incomplete classes intended to be mixed and matched with other classes. In CLOS [13], this programming convention became known as \textit{mixin classes}. A version of Smalltalk was also equipped with multiple inheritance [6]. From this point, a dichotomy emerged: Multiple inheritance was formalized [9], introduced in statically typed languages [22, 18], and problematized [15] because of thorny issues such as name clashes. Being even harder to handle, the idea that incomplete classes could be ‘mixed’ remained exotic, mostly confined to the Lisp community.

However, Bracha and Cook [7] introduced mixins as a separate concept that generalizes several kinds of inheritance. A number of variants [21, 12] emerged, where [12] introduced the notion of an \textit{inheritance interface}, specifying the requirements of a mixin on potential superclasses. Recently, a class-based object calculus [5] supporting statically typed mixin application seems to establish mixins as a well-understood mechanism.

Even though mixins improve on the flexibility of class combination, it remained a second class operation in statically typed languages, because it could only happen at compile time. This paper takes one step further towards a reconciliation of dynamic class combination and static typing, because it allows for a significant class of usages of statically safe, dynamic class combination. The details of this approach fit into the language \textit{gbeta}, but we will return to a discussion about the applicability of the results in other languages in the section about related work.

3 Informal Semantics

So what \textit{is} dynamic class combination, and why would it be useful? Dynamic class combination is simply a group of mechanisms similar to static class combination (including various kinds of multiple inheritance). Semantically it differs only by being available at run-time. There are many reasons why this is useful, just as there are many reasons why it is useful to do other things dynamically, e.g., dynamically choosing the length of a string, dynamically choosing the implementation of a method based on the class of the receiver, etc.

The core benefit gained from dynamic, first-class computations over class values is the removal of the strict connection between the program source code and the set of classes being used in the given program. Traditionally, a program
would have to be large in order to describe many different entities—2000 different things would require 2000 class declarations. Even if most of those class declarations were imported from libraries, programmers would need to read and understand the class interfaces in order to use them correctly. With dynamic class combination it is possible to describe a large number of entities using a few mixins, each describing a particular aspect, and then combine the mixins as needed—with only 11 independent extensions of a given interface we would have over 2000 possible variants. Hence, it would often be possible to drastically cut down on the number of classes or other descriptive entities needed in order to cover a certain range of variants. Since this abstracts over a redundant expression of a lot of combinations, the resulting program will express the core of the design much more concisely than the program based on static class combination.

However, dynamic class combination should be both safe and useful. It is crucial that the combination safety criterion introduced by this paper only restricts one operand (as mentioned, it must be created by single inheritance). The other operand is allowed to be an arbitrary pattern. Here is an example in \texttt{gbeta}:

\begin{verbatim}
addColor:
  (# inClass,outClass: ##Point;
   enter inClass## (* argument list *)
   do inClass & ColorPoint ## -> outClass## (* method body *)
   exit outClass## (* result *)
#)
\end{verbatim}

Box 1

The classes \texttt{Point} and \texttt{ColorPoint} are unsurprising and assumed to be defined elsewhere—technically they are patterns, but we will use the words ‘class’ and ‘method’ as synonyms to ‘pattern’, as a usage hint.

The example in box 1 declares the method \texttt{addColor}; this method receives an argument \texttt{inClass} which is a class, constrained to be a subclass of \texttt{Point}. The body computes the combination of the argument and \texttt{ColorPoint}, and stores the result in \texttt{outClass}, which is then the result of the method. This is a good example of a kind of dynamic creation of classes that used to be unsafe in \texttt{gbeta}. With the enhancements described in this paper, it is statically known to be safe.

Assuming that \texttt{ClickablePoint} is a subclass of \texttt{Point} we could do this:

\begin{verbatim}
(# myClass: ##Point; (* mutable, class-valued reference *)
  myPoint: ^Point; (* mutable, object-valued reference *)
  do ClickablePoint## -> addColor -> myClass##; (* 1 *)
  &myClass[] -> myPoint[] (* 2 *)
#)
\end{verbatim}

Box 2

This expression declares a local attribute \texttt{myClass}, used to hold the dynamically created class, and a local attribute \texttt{myPoint}, used to hold an instance of the dynamically created class. The body of this block then (at ‘1’) proceeds to invoke the method \texttt{addColor}, giving it the class \texttt{ClickablePoint} as the argument and storing the result in \texttt{myClass}. Finally, at ‘2’, an instance of \texttt{myClass} is created and a reference to the new object is stored in \texttt{myPoint}. This invocation of
addClass is type correct because the argument ClickablePoint is a subclass of Point—the declared type bound of the argument of addColor—and because the returned result is known to be some subclass of Point and myClass is allowed to contain any subclass of Point. Finally, the new instance of myClass can be assigned to myPoint because myClass is declared to be some subclass of Point, and myPoint is allowed to refer to instances of any subclass of Point.

To illustrate the effect of this, here is a version in C++ that uses static class combination, namely multiple inheritance, to achieve a similar effect:

```cpp
class CCPoint: public ClickablePoint, public ColorPoint {};

// in some function body:
{
    Point *myPoint = new CCPoint();
}
```

Here we create a new class CCPoint as a combination of the existing classes ClickablePoint and ColorPoint. The combined class is then used to create an instance, myPoint. Of course, the C++ example is less flexible than the gbeta example, because the C++ example is expressed in terms of compile-time classes and compile-time class combination operations, whereas the gbeta method addColor can create a combination of ColorPoint and any subclass of Point, possibly a subclass that is not known until run-time or even one that is created dynamically.

4 Basic Entities

This section describes the basic domains and operations in a formalization of the gbeta pattern combination mechanism. First, we have a countable set $\mathcal{M}$ whose members are known as mixins. $\mathcal{M}$ is partially ordered by the relation ‘$\preceq$’. In this context we do not need to model the internal structure of mixins, so we consider the elements of $\mathcal{M}$ as primitive; for a more detailed model, see [10, Chap.12].

**Definition 1 (Pattern).** The set of patterns, $\mathcal{P}$, is defined to be the set of ordering relations on subsets of $\mathcal{M}$ such that each pattern $P \in \mathcal{P}$ satisfies the following criteria:

\begin{align*}
\forall x &\in \text{supp}(P), \ y \in \mathcal{M}. \ x \preceq y \Rightarrow y \in \text{supp}(P) \quad (1) \\
\forall x, y &\in \text{supp}(P). \ x \preceq y \Rightarrow x \preceq_P y \quad (2) \\
\forall x, y &\in \text{supp}(P). \ x \preceq_P y \lor y \preceq_P x \quad (3)
\end{align*}

If $R$ is an order relation on a set $A$ then $\text{supp}(R)$ is the support of $R$, i.e., $\{x \in A \mid \exists y \in A. \ (x, y) \in R \lor (y, x) \in R\}$, and $\preceq_n$ is $R$ itself used in binary expressions, i.e., $x \preceq_n y \iff (x, y) \in R$.

Criterion (1) says that for every mixin $x$ in a pattern $P$, all elements larger than $x$ are also in $P$. In other words, the support of a pattern is upwards closed. Criterion (2) says that if two mixins $x$ and $y$ in a pattern $P$ are ordered according
to ‘≤’ then their ordering in \( P \) must coincide. In other words, a pattern is not allowed to contradict the global ordering of mixins. As a consequence, every pattern \( P \) must be a superset of the restriction of ‘≤’ to \( \text{supp}(P) \). Criterion (3) just expresses that every pattern is a total order on its support.

We will need the notion of a rigid pattern later. Such a pattern \( P \) has the property that the restriction of ‘≤’ to \( \text{supp}(P) \) is a total order. The word ‘rigid’ refers to the lack of reordering flexibility of such a pattern: Any reordering of mixins within \( P \) will produce a non-pattern. Note that rigid patterns are produced by single inheritance.

Intuitively, an element of \( \mathcal{M} \) is a mixin, i.e., an increment that differentiates a given class from its immediate super-class. For example, the difference between Point and ColorPoint could be the mixin \((\#\text{color: @string#})\), which adds an attribute named color to its superclass.

The intuition behind the global ordering ‘≤’ is that it expresses inheritance dependencies. We have \( x \leq y \) if and only if the mixin \( x \) statically depends on the mixin \( y \), i.e., if code inside \( x \) has been type checked under the assumption that attributes in \( y \) are available. Now, criterion (1) says that each mixin can only exist in a pattern where all the mixins from which it inherits are also present. Criterion (2) says that the ordering of mixins in every pattern must respect the inheritance based ordering. It is a useful approximation of the \text{gbeta} semantics to say that this means that we will respect overriding relations: If, e.g., a mixin ColorPoint overrides a method that it inherits from Point then there can be no pattern that includes these two mixins and lets the Point implementation override the ColorPoint implementation—if both are present then the ColorPoint implementation must dominate. Finally, criterion (3) ensures that every question about overriding has a well-defined answer.

Now we can specify the meaning of the pattern combination operator:

\textbf{Definition 2 (‘&’).} Given two total order relations \( R_1 \) and \( R_2 \), the combination of them is defined as follows:

\[
R_1 \& R_2 \triangleq R \cup (R_{21} \setminus R^{-1})
\]

where \( R \triangleq (R_1 \cup R_2)^* \), i.e., the transitive closure of the union of \( R_1 \) and \( R_2 \), \( R_{21} = \text{supp}(R_2) \times \text{supp}(R_1) \), i.e., all edges from an element in \( R_2 \) to an element in \( R_1 \), and \( R^{-1} \) is the inverse relation of \( R \), i.e., \( \{(y, x) \mid (x, y) \in R\} \).

In [11] this operator was defined on total pre-orders, so the present definition is slightly less general. When \( R_1 \) and \( R_2 \) are total pre-orders, \( R_1 \& R_2 \) is also a total pre-order (a proof appears in [11]), so in that case there is no potential failure to deal with. Note that a total pre-order need not even be a partial order, and there are cases where \( R_1 \& R_2 \) is not even a partial order even though both \( R_1 \) and \( R_2 \) are total orders. This seems to make the entire goal of this paper irrelevant—just use total pre-orders, and class combinations are provably safe.

\(^1\) Note that overriding is used here as a useful approximation to the actual \text{gbeta} semantics which is based on pattern combination and not overriding.
In fact, we tried to integrate patterns as total pre-orders into gbeta for about two years; but the inherent lack of ordering\textsuperscript{2} creates profound problems with combination of behavior: It is simply not appropriate to deny the programmer explicit control over the ordering of imperative actions. We believe that it will again and again give rise to unexpected and wrong semantics, because programmers will find it hard to reason about the correctness of all the possible reorderings of actions. Just imagine debugging a program if you knew that the compiler or runtime system would arbitrarily swap a few statements here and there. The new developments starting with the results in this paper do not have this problem, and seem much more promising.

Note that ‘&’ is not a commutative operator. Formally, this is easy to see. Intuitively, it reflects the fact that ‘&’ is a linearization algorithm, and it is well-known that multiple inheritance based on linearization delivers automatic conflict resolution for name clashes. Swapping the arguments to ‘&’ corresponds to reversing the basis for conflict resolution. Moreover, automatic conflict resolution is obviously a necessity with dynamic class combination.

Intuitively, $R_1 \& R_2$ is computed by putting $R_1$ and $R_2$ together (which produces $R_1 \cup R_2$), then adding ordering edges such that the relation is again transitive (yielding $R$), and finally adding all edges from $R_2$ to $R_1$ that do not contradict $R$—effectively making $R_2$ elements smaller than $R_1$ elements, unless something is known to the contrary. Even though it is possible to establish an intuition about ‘&’ based on this explanation, we feel that an algorithmic approach is much more useful when actually computing $A \& B$ for some concrete patterns $A$ and $B$. On the other hand, Def. 2 is the right tool to use when proving that ‘&’ has specific properties.

A simple algorithm that computes $R_1 \& R_2$ from $R_1$ and $R_2$ is shown in Fig. 1. It is easy to prove that it does indeed compute ‘&’. It is expressed as a function in Standard ML, and for notational simplicity it operates on lists of integers where the real algorithm operates on lists of mixins. The algorithm is a formulation of an algorithm known as the ‘C3’ linearization [2]. The situation where the exception Inconsistent is thrown is the situation where the two argument patterns disagree on the ordering of some pair of mixins, and that is exactly the situation that Thm. 1 below allows us to rule out statically when at least one argument is rigid.

As it was proved in [11, Prop. 2], $R_1 \& R_2$ is a total order whenever $R_1$ and $R_2$ are total orders and $R_1 \cup R_2$ does not have a cycle. Hence, in these cases the result of merging two patterns $P \& Q$ is a totally ordered set of mixins. Moreover, it is easy to prove that $R_i \subseteq R_1 \& R_2$ for $i \in \{1, 2\}$ for all total orders $R_1$ and $R_2$, i.e., that merging does not change the ordering of any two mixins in $R_1$ or in $R_2$. Similarly, it is easy to show that $\text{supp}(R_1) \cup \text{supp}(R_2) = \text{supp}(R_1 \& R_2)$. This shows that criterion (1) and (2) are also satisfied for $P \& Q$. We conclude:

\textbf{Lemma 1 (Partial closure property of ‘&’).} Given two patterns $P$ and $Q$. If $P \cup Q$ does not contain cycles then $P \& Q$ and $Q \& P$ are patterns.

\textsuperscript{2} A total order is isomorphic to a list, whereas a total pre-order is isomorphic to a list of \textit{sets} of mutually unordered elements.
fun merge ([]: int list) (ys: int list) = ys
 | merge (xxs as x::xs) [] = xxs
 | merge (xxs as x::xs) (yys as y::ys) =
   if x=y then x::(merge xs ys)
   else if not (member x ys) then x::(merge xs yys)
   else if not (member y xs) then y::(merge xxs ys)
   else raise Inconsistent;
fun member x [] = false
 | member x (y::ys) =
   if x=y then true else member x ys;

Fig. 1. Algorithm that produces $R_1\&R_2$ from arguments $R_1$ and $R_2$

The subpattern (e.g., subclass) relation in gbeta is defined as follows:

$$P \text{ is-subpattern-of } Q \iff P \supseteq Q$$

Since $R_1\&R_2 \supseteq R_i$, $i \in \{1, 2\}$, it follows that ‘&’ produces subpatterns of its arguments. Noting that a subpattern has all the mixins, and hence all the features, of any of its superpatterns, this demonstrates that it is reasonable to describe ‘&’ as a mechanism that is similar to multiple inheritance—it combines “classes” and produces a “subclass” of each operand. The fact that it also combines “methods” and several other kinds of entities—because they are all patterns—just broadens the scope of multiple inheritance, compared to mainstream languages.

5 A Safety Criterion

As we saw in the previous section, two patterns $P$ and $Q$ can be combined to a new pattern $P\&Q$ if $P \cup Q$ does not contain cycles.

A problem arises if $P \cup Q$ does contain cycles, corresponding to the raise Inconsistent case in the algorithm in Fig. 1. In context of the static analysis of gbeta, from 1997 until recently, we considered it impossible to ascertain that two patterns $P$ and $Q$ would satisfy this no-cycles criterion unless both $P$ and $Q$ were compile time constant expressions. If $P$ and $Q$ are compile time constants it is of course possible to perform the merging at compile time, thus checking statically that they can be merged. This corresponds to ordinary multiple inheritance.

Consequently, any combination operation applied to a pattern that is not completely known at compile time would until recently be flagged by the gbeta compiler as a dangerous operation. A new possibility arises with the introduction of the global ordering ‘$\preceq$’ (not considered in [11]), and the requirement that patterns respect this ordering. Consider a special kind of patterns, namely the rigid ones:

Definition 3 (Rigid patterns). A pattern $P$ is rigid iff the restriction of the global ordering ‘$\preceq$’ to $\text{supp}(P)$ is a total order.
When the restriction of ‘\(\leq\)’ to \(\text{supp}(P)\) is a total order, there can only be one pattern with support \(\text{supp}(P)\). Hence, any reordering of mixins within \(P\) will produce a non-pattern. ‘Rigid’ refers to this lack of reordering flexibility. Intuitively, a rigid class is a class that is produced by single inheritance. Using only single inheritance, the most specific mixin inherits from all the other mixins, the second most specific inherits from all other except the most specific one, etc.

**Lemma 2 (Everybody agrees with a rigid pattern).** Let \(P\) be a rigid pattern and \(Q\) an arbitrary pattern. If \(x, y \in \text{supp}(P) \cap \text{supp}(Q)\), then
\[
(x \leq_P y \land x \leq_Q y) \lor (y \leq_P x \land y \leq_Q x)
\]

The proof of this lemma can be found in App. A. Now we can state and prove the main result of this paper:

**Theorem 1 (It is safe to merge with a rigid pattern).** Let \(P\) be a rigid pattern and \(Q\) an arbitrary pattern. Then \(P\&Q\) and \(Q\&P\) are both patterns.

Again, the proof is in App. A. As a consequence of this theorem, it is sufficient to verify that at least one of the two operands to ‘\(&\)’ is rigid, then the operation will succeed. This is most fortunate, because rigid patterns are a very natural choice for an incremental enhancement of a given, arbitrary pattern.

The rigid pattern would be a conceptually coherent and focused descriptive entity, created by single inheritance, referred to by means of a compile-time constant denotation, and meaningful as a unit of enhancement for a complex pattern. The other operand can be an arbitrary pattern, e.g., a mutable pattern-valued attribute like \(\text{inClass}\) in the method \(\text{addColor}\) in box 1. This allows us to build a complex pattern by repeatedly adding conceptually focused aspects, i.e. rigid patterns, with no need for exact static knowledge about the complex pattern under creation, and without worrying about failure of the combination operation.

6 In Context of the Language gbeta

How do we know that the language gbeta actually invariantly maintains the properties (1), (2), and (3) for all patterns?

Property (3) is easy: Since each pattern (in the current implementation) is represented as a list of representations of mixins, it is indeed a total order.

The property (1), that the support is upwards closed, is maintained because no operation can ever remove a mixin from a pattern, and because every mixin is initially created by evaluation of a pattern expression whose locally, statically known structure was used to define the ordering ‘\(\leq\)’, and because

\[
\text{The value of a pattern expression is always exactly the locally, statically known value, or a subpattern thereof. (5)}
\]

A subpattern of a pattern \(P\) is a pattern that is a superlist of mixins, i.e., a list of mixins that can produce \(P\) by deleting zero or more mixins. Hence, the locally statically known mixins will also be available in a subpattern.
This criterion is similar to a criterion that is applicable to statically typed, main-stream, object-oriented languages, namely that every object reference at run-time will be NULL or will refer to an object which is an instance of the declared class or a subclass thereof (given the declaration `Point p;`, `p` will be a `Point` or a `ColorPoint`, or...). Such a criterion also applies to gbeta, but (5) is a higher-level version of the same thing. First, the value of a pattern expression often corresponds to a class; we need to consider the run-time value of classes—e.g., when used to declare the type of an object reference—because they are not necessarily known at compile-time, as opposed to the situation in main-stream languages (given the pseudo-Java declaration `SomePoint p;` where `SomePoint` is statically known as `Point`. `SomePoint` will be `Point` or some subclass thereof; `p` will then be an instance of `SomePoint` or a subclass thereof again—this of course requires a more sophisticated treatment of the safety of assigning to `p`). Finally, the word ‘locally’ is used in (5) because type analysis of one given expression in a gbeta program may result in different types as seen from different locations in the program. For instance, if a gbeta method returns a reference to the current object (pseudo-Java: `foo() { return this; }`) then the inferred return type of that method would depend on the knowledge about the receiver object, i.e., the type of that occurrence of ‘this’ would differ in different locations of the program. This example deals with objects (‘this’ denotes an object), but similar phenomena emerge in connection with denotations of classes. Even though this is complex and non-main-stream, we hope that (5) makes sense by now.

Property (5) has been a fundamental element in the gbeta static analysis for about 5 years, and the several hundred experimental or testing programs and many thousands of experiments have not produced a counter-example. Since our gbeta implementation performs extensive (double)checking of properties established by static analysis, most errors in the static analysis would immediately give rise to an error-stop along with a diagnostic message. A counter-example would be such an error-stop related to (5), and that has only occurred for brief periods during the development when a bug was introduced into the static analysis, and shortly after fixed. Nevertheless, it would of course be highly useful to formalize the entire gbeta analysis and establish a proof of this property.

Finally property (2), that patterns respect ‘⪯’, is ensured by property (5) together with the fact that pattern combination preserves the ordering of mixins in each operand. Let us sketch a proof by induction in the number of pattern combination operations: Property (5) ensures that a pattern that is not created by pattern combination will satisfy (2), because ‘⪯’ is derived from the locally, statically known ordering of mixins at each pattern declaration. For the induction step, assume that `P&Q` is a pattern, containing a pair of mixins `x` and `y` with `x ⪯_{P&Q} y`, but `y ⪯ x`. If `y ∈ supp(P)` then also `x ∈ supp(P)` by property (1), and `y ⪯_{x} x` by the induction hypothesis—which is a contradiction because `P&Q` should then also contain `y` and `x` in that order. Similarly if `y ∈ supp(Q)`.

A very important observation is that the safety of pattern combination with a rigid pattern applies recursively: When pattern combination propagates, as described in [11], it may happen that one of the derived pattern combinations
fails. For instance, we may be able to create a class by merging two existing classes, but the derived combination of the virtual methods in those two classes may cause a combination failure. But Th. 1 saves us here just as well as it did in the one-level case. We just need to check that every virtual pattern syntactically nested in the rigid pattern is itself rigid. Every derived pattern combination will then have at least one rigid operand, and hence it will succeed. So all we need to do is to extend the rigidity test to be applied recursively to syntactically nested virtuals: a recursively rigid pattern can be safely merged with an arbitrary pattern, also when the propagation is taken into consideration.

There is one requirement that the implementation of gbeta has not satisfied until recently. The problem is that gbeta must allow all patterns in the set $P$ to exist. It has been a deeply built-in restriction in gbeta that no pattern would be allowed to contain two mixins associated with the same syntactic expression (on the form $(#...#)$), but with two different environments. It would take too much space to explain the details of this problem here; suffice it to say that the difficulties inherent in the required generalization of gbeta seem to be solved by now, as the implementation of support for multiple mixins with the same syntax in a pattern is progressing. The test according to Th. 1 was implemented in the autumn of 2001. The basic framework of mixins, pattern combination, and compile-time analysis has been implemented and used for years.

One snake remains, however, in the paradise. A final binding on a virtual pattern specifies that it cannot be further specialized in subpatterns. In a version of gbeta without final bindings, the problem does of course not arise. Final bindings are a well-known means to remove covariance, thereby making it typesafe to call certain methods etc. However, many years of practical experience with Beta (which has both virtual patterns and final bindings) seems to indicate that immutable object references are a more useful tool to this end—it does not constrain the set of patterns that may exist; and it is crucial for family polymorphism, where final bindings do not suffice. If we decide to keep final bindings, one solution would be to define that a mixin with a final binding does not create a subtype—in other words, the subsumption test which now checks whether we can create a given pattern $Q$ by deleting zero or more mixins from a pattern $P$ must then refuse to delete mixins containing a final binding.

7 Applicability Elsewhere, and Related Work

As mentioned in Sec. 2, gbeta is based on mixins, and mixins improve on the flexibility of class combination. Hence, in order to use this result in another language, it would be necessary for that language to be based on mixins, and to use C3 linearization. The proofs of properties depend on the details of C3 as formalized here, and we do not think that other known linearization algorithms are similarly well-behaved—see [2] for a description of known problems with other linearization algorithms. It would probably be a rather mild change to introduce C3 into CLOS, so the results should be applicable here. Similarly for
Dylan [20]. However, since these languages do not support static type checking, this would only affect the run-time semantics.

On the other hand, it would require very deep changes to the semantics of statically typed languages such as C++, Java, or Eiffel to use the results of this paper directly. Moreover, since the whole point is to make dynamic class combination safe, such languages should have their type analysis adapted in order to deal with class expressions which are not compile-time constants. Probably the most useful approach would be to introduce existential types [1, Ch.13], but it would require a deep redesign of the type checking algorithms to handle existential types.

Let us compare gbeta some more with other approaches. The mixin concept is fundamental to gbeta, but mixins cannot be manipulated individually, only via the ‘&’ operator; in this sense it is similar to CLOS and the other early class-based approaches mentioned in Sec. 2. It also uses linearization, specifically the C3 [2] algorithm, but the formalization (Def. 2) and proofs of properties in terms thereof are our contributions. The mixin ordering ‘≤’ ensures correctness in a similar way as the inheritance interface of [12] and the mixin application type check in [5], but gbeta mixin application is more dynamic because it allows combination of patterns at run-time. As this paper shows, dynamic mixin application is safe under certain conditions.

Finally, dynamic mixin application creates similarities between gbeta and languages with delegation. In particular, LAVA [14] supports static and dynamic delegation in a type safe manner; moreover, [14] made it clear for me that final declarations are incompatible with subtyping when they occur in a delegatee, corresponding to a mixin’s superclass. Delegation allows dynamic construction of multi-object structures with shared methods and state, and this gives a similar freedom to construct variants as dynamic class combination. There is a trade-off here, where delegation provides even more flexibility than dynamic class combination, but dynamic class combination provides a more clear and robust object model. An example in LAVA is that, for type safety, certain methods in a dynamic delegatee cannot be called on this object (i.e., the delegation network), but must be called on the so-called holder object (which is not the delegation network, but only the single object that happens to execute the current method); this tends to split the network into the individual objects that it was constructed from. Another example is the confusion about the several potential this pointers in a delegation network—is it a bug if a client obtains a pointer to an arbitrary object in a delegation network? It can certainly break a simulation of inheritance, where all pointers should point to the most specific atomic object.³ With dynamic class combination each logical object is just an object, ³ If we build a ColorPoint delegation network out of a Color object that delegates to a Point object then all client pointers must point to the Color object; otherwise methods defined in Point and redefined in Color would execute as in a simple Point object (the Color redefinition is ignored), and this might produce wrong semantics as well as breaking invariants in the Color part of the network.
not a network of objects. We believe that this removes a lot of confusion and improves the robustness of programs.

8 Conclusion

We have presented a formalization of the core of the pattern algebra that allows the language \texttt{gbeta} to combine classes and methods dynamically, and then use the outcome in a statically safe manner. It has been a long-standing problem in this context that the combination operation itself could not be guaranteed to succeed, unless applied to compile-time constant pattern expressions—essentially reducing pattern combination to either a static or an unsafe operation. However, the formalization was used to prove that an important, comprehensible, and useful category of combination operations is indeed safe, namely when at least one of the operands is a rigid pattern, i.e., it has been created by means of single inheritance. For safety, final bindings must be statically visible, but this problem seems to be solvable. Since this allows the gradual construction of a complex pattern by means of repeated enhancements with conceptually clear and wholesome single-inheritance constructs, we believe that safe, dynamic combination of classes and methods is now available in a form that is useful for practical programming. There is still much to do, however, in order to integrate this result into more main-stream languages.

References


A Proofs

Proof (of Lemma 2). Assume that \( x \preceq_P y \). \( P \) is a pattern, so it must respect the global ordering ‘\( \succeq \)’ as specified in (2). Moreover, \( P \) is rigid so the restriction of ‘\( \succeq \)’ to \( \text{supp}(P) \) is a total order. But then every pair of elements in \( P \) is ordered in accordance with ‘\( \preceq \)’, hence \( x \preceq y \). Finally, \( x \preceq y \) implies that \( x \preceq_Q y \), because \( Q \) must also satisfy (2). The argument is similar under the assumption \( y \preceq_P x \). \( \Box \)

Proof (of Theorem 1). Assume that \( P \cup Q \) contains a cycle. We must show that this leads to a contradiction, and then the result follows from Lemma 1.
First note that $P \cup Q$ is not an order relation, so we cannot assume transitivity etc. Now, a cycle in $P \cup Q$ is a finite sequence $x_1, x_2 \ldots x_k$ of at least two distinct elements such that
\[
(x_i \preceq_P x_j) \lor (x_i \preceq_Q x_j)
\]
for $i \in \{1 \ldots k - 1\} \land j = i + 1$ and $i = k \land j = 1$.

Consider the sequence as a circular graph with vertices $x_1 \ldots x_k$ and colored, directed edges between them—a green edge from $x_i$ to $x_j$ if $x_i \preceq_P x_j$ and a red edge same place if $x_i \preceq_Q x_j$ (and both a red and a green edge if both $x_i \preceq_P x_j$ and $x_i \preceq_Q x_j$).

Note that we cannot have a complete red cycle or a complete green cycle, since $P$ and $Q$ are both patterns and hence total orders. However, we can replace every green path $x_a \ldots x_b$ with a green edge from $x_a$ to $x_b$, thus deleting all elements between $x_a$ and $x_b$, and similarly for red paths. This is because $P$ is transitive and $Q$ is transitive, and a single-color path refers to only one of $P$ and $Q$. Any red edges on a green path are just deleted in the process, and similarly for green edges on a red path. Assume that we have performed this transformation as often as possible. The graph will now have alternating red and green edges, corresponding to a cycle that can be described as follows: It is a sequence of at least two distinct elements $y_1 \ldots y_m$ such that
\[
(y_i \preceq_P y_{i+1}) \land (y_{i+1} \preceq_Q y_i)
\]
for $i \in \{1, 3, 5 \ldots m - 3\} \land j = i + 2$, and $i = m - 1 \land j = 1$.

Note that $y_i \in \text{supp}(P) \cap \text{supp}(Q)$ for all $i \in \{1 \ldots m\}$, because every $y_i$ is adjacent to both a red and a green edge. Since $P$ is total this means that all elements on the cycle are related in $P$. Now consider the sequence $y_1, y_3, \ldots y_{m-1}$. We cannot have $y_i \preceq_P y_j$ for all $i \in \{1, 3 \ldots m - 3\} \land j = i + 2$ and $i = m - 1 \land j = 1$, because then we would have a cycle in $P$. So we have $y_i \preceq_P y_j$, for some $i \in \{1, 3 \ldots m - 3\} \land j = i + 2$ or $i = m - 1 \land j = 1$. With a possible renaming we can assume that $y_3 \preceq_P y_1$. Then we have $y_1 \preceq_P y_3 \preceq_P y_2$ and by transitivity of $P$, $y_3 \preceq_P y_2$. But we also had $y_2 \preceq_Q y_3$, which establishes the required contradiction since Lemma 2 tells us that the relation $y_2 \preceq_Q y_3$ cannot exist in any pattern when $y_3 \preceq_P y_2$ exists in a rigid pattern. $\qed$