

# SAFE DYNAMIC MULTIPLE INHERITANCE

ERIK ERNST

*University of Aarhus, Dept. of Computer Science, Århus, Denmark*  
eernst@daimi.au.dk

**Abstract.** Multiple inheritance and similar mechanisms are usually only supported at compile time in statically typed languages. Nevertheless, dynamic multiple inheritance would be very useful in the development of complex systems, because it allows the creation of many related classes without an explosion in the size and level of redundancy in the source code. In fact, dynamic multiple inheritance *is* already available. The language `gbeta` is statically typed and has supported runtime combination of classes and methods since 1997, by means of the combination operator ‘&’. However, with certain combinations of operands the ‘&’ operator fails; as a result, dynamic creation of new classes and methods was considered a dangerous operation in all cases. This paper presents a large and useful category of combinations, and proves that combinations in this category will always succeed.

**CR Classification:** D.3 PROGRAMMING LANGUAGES

**Key words:** Multiple inheritance, dynamic mechanisms, static analysis.

## 1. Introduction

In many situations it is useful to have a large number of similar classes, providing variants of a given basic abstraction with respect to, e.g., the set of features supported, the choice of implementation, or other properties. For example, a workflow system might have many different variants of `Task`, a word processor might have many kinds of `Document`, and a family of programming tools might have many types of `AbstractSyntaxTree`.

Design patterns [17] such as ‘Decorator’, ‘Strategy’, and ‘State’ enable flexibility in the creation of such variants, but they also require a more involved and verbose style of programming, and there are semantic problems. For instance, the simple method forwarding used in these design patterns should typically have been true delegation (i.e., with `this` bound to the original message receiver, not the delegatee/forwardee). Moreover, the object identity is ill-defined when the job of one object has to be done by several. It may cause problems, e.g., if a forwardee returns `this` as the result of a method, because subsequent operations on the returned object will bypass the original message receiver, which could for instance be a decorator object.

Hence, the flexibility provided by design patterns comes at a cost, and if we do not need to change the implementation or feature set of each individual object after creation, it would be much nicer to use an ordinary class,

constructed to have the desired features and implementation. We would then have genuine late binding of methods, well-defined object identity, and a simpler programming style.

However, to support all subsets of  $N$  features by means of multiple inheritance from classes supporting one feature each, we would need  $2^N$  different classes—each class representing one subset of the features. Choosing between several alternative implementations would just aggravate this problem. We would need to write and then manage a large body of source code, spelling out every combination.

It might be possible to express specific combinations as needed in client code by means of anonymous classes. This could be more convenient, but it does not change the fact that  $2^N$  different classes (anonymous or not) would need to exist in the source code, in order to support all subsets of the features.

Dynamic multiple inheritance solves this problem by allowing us to specify just the  $N$  classes with one feature each. At run-time, a composite class with the desired set of features can then be created using dynamic multiple inheritance.

It might be claimed that a language like Java [2] already supports dynamic inheritance. This paragraph and the next one explain why this would be an overstatement. In Java it is possible to load a class dynamically, and it is also possible to programmatically generate a piece of source code and invoke a compiler on it. If Java had supported multiple inheritance of implementation, we might use this to create combinations of classes dynamically. Using interfaces and forwarding methods we might even be able to achieve much the same as we could do with class combination, still using standard Java.

However, compilation of generated source code may fail in so many and so complex ways that it cannot be considered a safe operation. Moreover, if a class *is* created and loaded, it cannot be used within the type system of a language like Java: It is necessary to use reflection in order to create instances and to determine subtyping relations. Finally, such a class accessed only via reflection cannot be used to declare types of instance variables, method arguments, etc. In other words, even though existing programming language technology in a sense enables dynamic combination of classes, the combination operation is complex and unsafe, and the resulting dynamic classes are poorly integrated into the program that created them.

The language `gbeta` [14, 13], on the other hand, provides dynamic combination of classes and other entities as an integrated facility, without compromising static type checking. The type of dynamically created classes is partially known statically, in that there is a statically known upper bound (it is known that the dynamic class is a subtype of a given, statically known class). In `gbeta`, the dynamic class can be used for creation of instances and for type declarations by means of the same syntactic expressions as those used with static classes, and the type of a dynamic class is existential (see [24] and [1, Ch.13]), i.e., the `gbeta` type system keeps track of the fact that such a type is only known by an upper bound.

This means that it is safe to *use* a dynamically created class, but it was until now unsafe to *create* it in the first place. The situation 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 a number which is just as safe to use as any other number—only the operation itself is dangerous, not the outcome.

The main contribution of this paper is to prove that the operation of combining classes dynamically is safe in a large, useful set of cases. One way to describe this set of cases is to say that at least one of the operands to the class combination operation must have been created by single inheritance. The intuitive reason why these cases are safe is that single inheritance classes consist of mixins that depend on each other in such a way that the mixins cannot exist in any other order. Since failure *only* arises when the two operands disagree on the ordering of mixins, and the single inheritance will appear in the same order everywhere, there cannot be a failure.

Combination of classes is expressed in `gbeta` by means of the pattern combination operator ‘&’. The *pattern* concept [21, 22, 13] was created as part of the design of BETA [22]. It unifies and generalizes the concepts of class, method, and more. In `gbeta` the pattern concept is generalized by being based on mixins, and pattern combination is introduced. Hence, support for dynamic pattern combination implies support for dynamic combination of classes, and of methods, etc. In this paper we concentrate on combination of classes.

The core 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 `gbeta`, and considers the applicability of the result to other languages. 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 [6]. Four years later, the OO Lisp dialect LOOPS [5] was created, supporting multiple inheritance. In 1982 the Lisp dialect Flavors [11] was created, also with multiple inheritance, and with a programmer culture that emphasized the combination of classes from various ‘flavors’—incomplete classes intended to be mixed and matched with other classes. An incomplete class is a class  $C$  that uses features which are not available in  $C$  or any of its superclasses.

It can only be used in multiple inheritance, together with classes providing the missing features.

In the CLOS [18] community this programming convention became known as *mixin classes*. A version of Smalltalk was also equipped with multiple inheritance [8]. From this point, a dichotomy emerged: Multiple inheritance was formalized [12], introduced in statically typed languages [28, 23], and problematized [20] because of thorny issues such as name clashes. Being even harder to handle, the idea that incomplete classes could be ‘mixed in’ remained exotic, mostly confined to the Lisp community.

However, Bracha and Cook [9] introduced mixins as a separate concept that generalizes several kinds of inheritance. A number of variants [27, 16] emerged, where [16] introduced the notion of an *inheritance interface*, specifying the requirements of a mixin on potential superclasses. Recently, a class-based object calculus [7] supporting statically typed mixin application seems to establish mixins as a well-understood mechanism.

Even though mixins improved on the flexibility of class combination, class combination remained a compile-time operation in statically typed languages. This paper takes one step further towards a reconciliation of dynamic class combination and static typing, because it allows for a significant category of usages of statically safe, dynamic class combination. The details of this approach were developed in context of the language `gbeta`, but in Sec. 6 we will argue that the results are also applicable in some other languages.

### 3. Informal Semantics

So what *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. In general, dynamic flexibility allows us to reuse the same piece of code with different run-time entities, thus reducing the overall redundancy and complexity of software, and at the same time expanding its capabilities. It is, however, hard to maintain safety when run-time flexibility is enhanced. Hence, the goal is *safe dynamic* multiple inheritance.

In Sect. 1 we have already discussed some benefits of being able to select features by means of dynamic class combination. Notably, we may express the equivalent of  $2^N$  static classes using just  $N$  single-feature classes and dynamic combination. Since the program using dynamic class combination is not only much smaller, but also much less redundant, it will express the core of the design in a more comprehensible manner than the program based on static class combination.

Moreover, modularity is greatly improved in other ways: We can simply include a couple of extra features in a separate module and then use them, whereas in the static approach we would have to write  $3 \times 2^N$  new classes to provide all the new combinations between two new features and  $N$  existing ones. A corresponding situation arises if we consider removal of features, again demonstrating the lack of separation of concerns in the static approach.

We think that the core benefit gained from dynamic computations with class values can be traced back to the removal of the strict connection between the program source code and the set of classes being used in the given program.

However, dynamic class combination should not only be useful, it should also be safe. The combination safety criterion introduced by this paper only restricts one operand (as mentioned, it must be created by single inheritance). This is crucial, because the other operand is then allowed to be an arbitrary pattern. Here is an example in `gbeta`:

```
Point: (# x,y: @integer #);
ColorPoint: Point(# color: @string #);
addColor:
  (# inClass,outClass: ##Point;          (* class variables *)
  enter inClass##                         (* argument list *)
  do inClass&ColorPoint## -> outClass##
  exit outClass##                         (* result *)
  #)
```

Ex.  
1

For conciseness the classes `Point` and `ColorPoint` are very simple, just providing some data members. Technically `Point` and `ColorPoint` (and `addColor`) are patterns, but we will use the words ‘class’ and ‘method’ as synonyms to ‘pattern’, as a usage hint.

The remaining code in Ex. 1 declares the method `addColor`. This method receives an argument `inClass` which is a class, constrained to be a subclass of `Point`. The body computes the combination of the argument and `ColorPoint`, and stores the result in `outClass`, which is then the result of the method. The `##` operator ensures the desired evaluation and assignment semantics, namely evaluation and assignment of pattern values. Without this operator we would have had the default, namely value assignment and value evaluation of instances of the denoted classes. It is beyond the scope of this paper to explain the entire evaluation and assignment semantics of `gbeta` so we just explain the concrete usages when they occur.

This is a good example of a kind of dynamic creation of classes that used to be unsafe in `gbeta`. With the enhancements described in this paper, it is statically known to be safe.

Assuming that `ClickablePoint` is a subclass of `Point` we could do as follows:

```

(# myClass: ##Point;      (* mutable, class reference *)
 myPoint: ^Point;        (* mutable, object reference *)
do ClickablePoint## -> addColor -> myClass##;    (* 1 *)
 myClass[] -> myPoint[]  (* 2 *)
#)

```

Ex.  
2

The [] operator specifies reference evaluation and reference assignment semantics (which is, e.g., what Java uses for data members of class types). When applied to a class (`myClass`) it implies creation of a new instance of that class, and then evaluates to a reference to the newly created object. We can now proceed to explain the example.

Example 2 declares a local attribute `myClass`, used to hold the dynamically created class, and a local attribute `myPoint`, used to hold an instance of the dynamically created class. The body of this block then proceeds (at ‘1’) to invoke the method `addColor`, giving it the class `ClickablePoint` as the argument, and storing the result in `myClass`. Finally, at ‘2’, an instance of `myClass` is created and a reference to the new object is stored in `myPoint`. This invocation of `addColor` 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`.

For comparison, here is a version of Ex. 1 in C++ that uses *static* class combination, namely multiple inheritance, to achieve a similar effect:

```

class CCPoint: public ClickablePoint, public ColorPoint
{
};

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

```

Ex.  
3

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.

Finally, we should note that the detailed semantics of the inheritance mechanisms are of course different, because `gbeta` inheritance is mixin based whereas C++ inheritance is not.

After this brief presentation of the informal semantics of pattern combination, the following sections will present and develop the formalization.

#### 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  $\mathcal{M}$  and a partial order ‘ $\preceq$ ’ on  $\mathcal{M}$ . Think of the relation ‘ $\preceq$ ’ as the inheritance relation, so we would have **ColorPoint**  $\preceq$  **Point**. 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 the **Point** class and the **ColorPoint** class is the mixin (`#color:@string#`), which adds an attribute named `color` of type `string` to its superclass. Note that in the following we use names such as **Point** and **ColorPoint** to denote the mixins, not the classes. The inheritance relation **ColorPoint**  $\preceq$  **Point** then means that the **ColorPoint** mixin specifies that it can only be attached to a superclass that includes the **Point** mixin. In other words, **ColorPoint** *requires Point*.

In this context we do not need to model the internal structure of mixins, so we consider the elements of  $\mathcal{M}$  as simple, unstructured values; for a more detailed model, see [13, Chap.12]. However, we need to model the composite entities built from mixins, namely patterns:

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:*

$$\forall x \in \text{supp}(P), y \in \mathcal{M} \quad . \quad x \preceq y \Rightarrow y \in \text{supp}(P) \quad (1)$$

$$\forall x, y \in \text{supp}(P) \quad . \quad x \preceq y \Rightarrow x \preceq_P y \quad (2)$$

$$\forall x, y \in \text{supp}(P) \quad . \quad x \preceq_P y \vee y \preceq_P x \quad (3)$$

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

We could have defined a pattern as an ordering relation on a subset of  $\mathcal{M}$ , and defined a *well-formed* pattern as a pattern which also satisfies (1)–(3), but we would then have to use the term ‘well-formed pattern’ everywhere, and the notion of a ‘pattern’ would be useless. So we decided that it would be better to define patterns as in Def. 1.

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. Returning to the **Point** example, there could never exist a class that includes **ColorPoint** without also including **Point**.

Criterion (2) says that if two mixins  $x$  and  $y$  in a pattern  $P$  are ordered according to ‘ $\preceq$ ’ then their ordering in  $P$  must coincide. In other words, a pattern is not allowed to contradict the global order on mixins; so we are not allowed to create a class where methods in **ColorPoint** are overridden by methods in **Point** (NB: this is backwards), which would be the case if we

had a class that contradicted ‘ $\preceq$ ’ for those two mixins. Another consequence is that every pattern  $P$  is a superset of the restriction of ‘ $\preceq$ ’ to  $\text{supp}(P)$ .

Criterion (3) just expresses that every pattern is a total order on its support; one way to put it is that it is always known which one of two mixins is in the most specific position, hence method overriding and similar questions always have a well-defined answer.

We will need the notion of a *rigid* pattern later. Such a pattern  $P$  has the property that the restriction of ‘ $\preceq$ ’ 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 single inheritance always produces rigid patterns, but some multiple inheritance expressions also produce a rigid pattern.

The explanation of the three criteria (1)–(3) above was kept in rather general terms. In the following we will explain the special significance they have in **gbeta**.

As mentioned, the global ordering ‘ $\preceq$ ’ expresses inheritance dependencies. In **gbeta**, we have  $x \preceq y$  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. As we know, criterion (1) says that each mixin (such as  $x$ ) can only exist in a pattern where all the mixins from which it inherits (including  $y$ ) are also present. This obviously ensures that inherited features used in  $x$  will be provided by  $y$ . But it may sound like a contradiction to the cornerstone of the mixin concept, namely the fact that a mixin is *not* tied to a particular superclass, but can be applied to many different superclasses.

However, criterion (1) does not determine the immediate super-mixin of a given mixin; it only ensures that the statically known super-mixins will be *somewhere* in the actual superclass. This means that the actual, immediate super-mixin is generally unknown. An attempt to understand the notion of pattern inheritance and combination in **gbeta** based on traditional concepts of class and (single or multiple) inheritance will fail to capture this. That is the reason why we use the term ‘mixin’, even though the mixin concept in **gbeta** is not identical to the mixin concept in, e.g., Strongtalk [10].

Criterion (2) says that the ordering of mixins in every pattern must respect the inheritance based ordering. It is only an approximation of the **gbeta** semantics to say that this means that we will respect overriding relations. In fact, **gbeta** does not use method overriding, but it uses method combination and similar phenomena arise in that context.

Finally, in context of **gbeta** we would say that criterion (3) ensures that every question about method *specificity* has a well-defined answer. Method specificity in turn determines the outcome of method combination. See [14] for details about this mechanism.

Now we can specify the meaning of the pattern combination operator ‘&’. First we present an algorithm implementing it, then we give a formalization which is used in proofs about its properties.

A simple algorithm that computes  $R_1 \& R_2$  from  $R_1$  and  $R_2$  is shown in

```

fun merge ([]: int list) (ys: int list) = ys
  | merge xs [] = xs
  | merge (xxs as x::xs) (yys as y::ys) =
    if x=y then x::(merge xs ys)
    else if not (member y xs) then y::(merge xxs ys)
    else if not (member x ys) then x::(merge xs yyn)
    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  $R_2$ .

Fig. 1. It is easy to prove that it does indeed compute ‘&’ as formalized in Def. 2 below. 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 [3]. The name C3 refers to three different consistency properties, namely monotonicity, consistency with the local precedence order, and consistency with the extended precedence graph. These concepts are associated with the inheritance graphs of languages such as Dylan [26] and CLOS [4], and we refer to [3] for details. The essence of these consistencies is that pattern combination will never create a pattern containing two mixins ordered in the opposite direction of what is specified in its operands, neither directly nor indirectly. In the form used here, the algorithm does not operate on a directed acyclic graph, it operates on two lists; this is possible because the algorithm is applied every time two patterns are combined—the multiple inheritance graph is never built, because it is reduced to a list at every combination operation.

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. In other words, this paper is all about knowing statically that this exception ‘`Inconsistent`’ will not occur.

The algorithm is suitable for implementation, and for building an intuition about the combination operator ‘&’. However, we need a formalization of the combination operator in order to be able to prove something about its properties. The formalization is as follows:

DEFINITION 2. (‘&’) *Given two total order relations  $R_1$  and  $R_2$ , the combination of them is defined as follows:*

$$R_1 \& R_2 \triangleq R \cup (R_{21} \setminus R^{-1}) \quad (4)$$

where  $R \triangleq (R_1 \cup R_2)^*$ , i.e., the transitive closure of the union of  $R_1$  and  $R_2$ ,  $R_{21} \triangleq \text{supp}(R_2) \times \text{supp}(R_1)$ , 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.,  $\{(y, x) \mid (x, y) \in R\}$ .

For example, if we combine two total orders  $R_1 = \{(a, a), (b, a), (b, b)\}$  and  $R_2 = \{(a, a), (c, a), (c, c)\}$  then

$$\begin{aligned} R &= \{(a, a), (b, a), (b, b), (c, a), (c, c)\} \\ R_{21} \setminus R^{-1} &= \{(c, a), (c, b)\} \\ R_1 \&R_2 &= \{(a, a), (b, b), (c, c), (b, a), (c, a), (c, b)\} \end{aligned}$$

Using the isomorphism between total order relations and lists (holding the elements in decreasing order), we may express this result more concisely as  $R_1 = [a, b]$ ,  $R_2 = [a, c]$ , and  $R_1 \&R_2 = [a, b, c]$ .

In [14] the operator ‘&’ was defined on total pre-orders,<sup>1</sup> 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 [14]), 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 a partial order even though both  $R_1$  and  $R_2$  are total orders. For example, with  $R_1 = [a, b]$  and  $R_2 = [b, a]$  we get  $R_1 \&R_2 = \{(a, a), (a, b), (b, a), (b, b)\}$  which is not a partial order. The exception ‘Inconsistent’ in Fig. 1 is raised for a given pair of operands  $R_1$  and  $R_2$  if and only if the outcome of using Def. 2 is a total pre-order which is *not* a total order.

The results in [14] seem to make the entire goal of this paper irrelevant—just use total pre-orders to define the structure of patterns, and class combinations are provably safe. In fact, we tried to integrate patterns as total pre-orders into `gbeta` for about two years; but the inherent lack of ordering<sup>2</sup> 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,

<sup>1</sup> A pre-order is a relation which is reflexive and transitive.

<sup>2</sup> A total order is isomorphic to a list, whereas a total pre-order is isomorphic to a list of sets of mutually unordered elements.

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.

As it was proved in [14, Prop. 2],  $R_1 \& R_2$  is a total order—i.e., criterion (3) is satisfied—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 combining 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 combination 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:

**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 both patterns.*

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

$$P \text{ is-subpattern-of } Q \quad \stackrel{\Delta}{\Leftrightarrow} \quad P \supseteq Q$$

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 combination, compared to main-stream multiple inheritance.

## 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 combination at compile time, thus checking statically that they can be combined. This is similar to ordinary static 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 [14]), 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 ‘ $\preceq$ ’ to  $\text{supp}(P)$  is a total order, any reordering will produce a non-pattern, and ‘rigid’ refers to this lack of reordering flexibility. The consequence is that there can only be *one* pattern with support  $\text{supp}(P)$ ; if we know  $\text{supp}(P)$ , we also know  $P$ . Single inheritance (and some special cases of multiple inheritance) produces a rigid class: 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) *Assume  $P$  is a rigid pattern and  $Q$  an arbitrary pattern. If  $x, y \in \text{supp}(P) \cap \text{supp}(Q)$ , then*

$$(x \preceq_P y \wedge x \preceq_Q y) \quad \vee \quad (y \preceq_P x \wedge y \preceq_Q x)$$

The proof of this lemma can be found in App. Appendix 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. Appendix 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 useful 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 `inClass` in the method `addColor` in Ex. 1. This allows us to build a complex pattern dynamically 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 Real Languages

First we consider the applicability of the formal result in context of the language `gbeta`, then we turn to other languages and discuss how hard it would be to use the result there.

### 6.1 In Context of *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: Each pattern is a list of mixins, so it is indeed a total order. Pattern combination ('&') produces a new list of mixins, or it raises an exception, so dynamic creation of patterns will preserve the invariant that all patterns are total orders.

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 is used to *define* the ordering ' $\preceq$ ', and because

*The value of a pattern expression is always exactly  
the locally, statically known value, or a subpattern  
thereof.* (5)

As mentioned, a *subpattern*  $Q$  of a pattern  $P$  is a superlist of mixins, i.e., we can produce  $P$  by deleting zero or more mixins from  $Q$ . Hence, the locally statically known mixins in (5) will also be available in any subpattern.

Property (5) is similar to a property that holds for 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 (assuming the declaration `Point p;`,  $p$  will be NULL or an instance of `Point` or of a subclass such as `ColorPoint`). Such a property also holds in *gbeta*, but (5) is a higher-level version of the main-stream property because it is concerned with classes, not objects.

The value of a pattern expression often corresponds to a class, so 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. Assuming the pseudo-Java declaration `SomePoint p;` where `SomePoint` is dynamic, but 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 type analysis of *assignments* to  $p$  than what is required when all class denotations are compile-time constants.

Finally, the word 'locally' is used in (5) because type analysis of one given occurrence of an 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 when seen from 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 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 thereafter 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 ‘ $\preceq$ ’, 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 single inheritance pattern expression will satisfy (2), because ‘ $\preceq$ ’ is derived from the locally, statically known ordering of mixins. A multiple inheritance pattern expression consists of a number of single inheritance pattern expressions combined using ‘ $\&$ ’. For the induction step, assume that  $P\&Q$  is a pattern, containing a pair of mixins  $x$  and  $y$  with  $x \preceq_{P\&Q} y$ , but  $y \preceq x$ . If  $y \in \text{supp}(P)$  then also  $x \in \text{supp}(P)$  by property (1), and  $y \preceq_P x$  by the induction hypothesis—which is a contradiction because  $P\&Q$  should then also contain  $y$  and  $x$  in that order. Similarly if  $y \in \text{supp}(Q)$ .

A very important observation is that the safety of pattern combination with a rigid pattern applies recursively: When pattern combination propagates, as described in [14], it may happen that one of the *derived* pattern combinations fails. For instance, we may be able to create a class by combining two existing classes, but the derived combination of the virtual methods in those two classes may cause a combination failure. But Th. 1 applies 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 combined with an arbitrary pattern, also when the propagation is taken into consideration.

In context of `gbeta` it is necessary to require that some of the mixins used for pattern combination are defined at the global level: The “well-behaved” argument to ‘ $\&$ ’—the rigid one—must also consist of mixins defined globally. There is no such restriction on the mixins in the other argument to ‘ $\&$ ’, they can be global mixins or inner mixins in any combination. This is crucial, because it means that we can still use a compile-time check to ascertain that there is at least one argument which is well-behaved (rigid and global), without having to know *anything* about the other argument. This globality

restriction is hard to avoid, but it also seems to be a reasonably acceptable restriction for programmers in their daily work.

One problem remains, however. A *final* binding on a virtual pattern specifies that it cannot be further specialized in subpatterns, and violations of this restriction in a dynamic class combination cannot always be detected statically. 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 type-safe to call certain methods, etc. However, many years of practical experience with BETA (which has both virtual patterns and final bindings) seem 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 [15], where final bindings do not suffice. If we decide to keep final bindings in the language, 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.

The approach has been implemented in context of **gbeta**. The test according to Th. 1 was implemented in a preliminary (incorrect) version in the autumn of 2001, and corrected in the summer of 2002. The problem was that the first version did not exclude inner mixins in the well-behaved argument. The basic **gbeta** framework of mixins, pattern combination, and compile-time analysis has been implemented years ago and used ever since.

## 6.2 In Context of Other Languages

In languages such as CLOS [4] and Dylan [26] where multiple inheritance is available—including multiple inheritance of implementation—and is based on linearization, it would be quite realistic to take advantage of the main result of this paper. The main issue is that the linearization algorithm must be changed to the one given in Fig. 1. It would then be safe to use multiple inheritance in expressions where the denotations of classes are not compile time constants. Setting up class objects with class precedence lists should proceed exactly as it does now, but there may be a need to change the compiler and/or the run-time system in order to support dynamic multiple inheritance *efficiently*.

Note that the lack of static type-checking in these languages would entirely bypass the need to adjust a type checking algorithm to handle existential types. Also note that it is still useful to know that a particular operation will succeed at run-time, even in a context where there is no static type-checking in general. Moreover, since these languages do not have inner classes or virtual classes, not to mention final bindings of them, the restrictions and problems in **gbeta** associated with these two concepts simply do not apply here. In other words, the result seems to fit even more smoothly into these languages than into **gbeta**.

In contrast, it would require very deep changes to the semantics (hence also the implementation) of traditional, statically typed languages such as C++, Java, or Eiffel to use the results of this paper directly, because inheritance in these languages is not based on linearization. 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, but this would also require a deep redesign of the type checking algorithms.

## 7. Related Work

Section 2 already mentioned a number of related approaches over a long period of time. 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 algorithm [3], but the formalization (Def. 2) and proofs of properties in terms thereof are our contributions. The mixin ordering ‘ $\preceq$ ’ ensures correctness in a similar way as the inheritance interface of [16] and the mixin application type check in [7], 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* [29, 25, 19]. In particular, LAVA [19] gives a solution to the problem of how to support static and dynamic delegation in a type safe manner. 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, 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. For instance, 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). This might immediately produce wrong behavior, and since it might break invariants in the `Color` part of the network the behavior could also be wrong at a much later point in time. With dynamic class combination each logical object is just an object, 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 `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, this paper gives a formalization and a proof that an important, comprehensible, and useful category of combination operations is indeed safe: At least one of the operands must be a global, rigid pattern, i.e., it must be created by means of single inheritance without using inner mixins. For safety, final bindings must be statically visible, but this problem seems to be solvable.

Safe dynamic pattern combination allows the gradual construction of a complex pattern by means of repeated enhancements with conceptually clear and wholesome single-inheritance constructs, so we claim that safe, dynamic combination of classes and methods is now available in a form that is useful for practical programming.

The result seems to be rather easy to apply to some dynamic languages such as `CLOS` and `Dylan`, but there is still much to do in order to integrate this result into more main-stream languages.

## Appendix A. Proofs

PROOF. [of Lemma 2] Assume that  $P$ ,  $Q$ ,  $x$ , and  $y$  are as specified in the lemma.  $P$  is a total order, so we have either  $x \preceq_P y$  or  $y \preceq_P x$ . Assume that  $x \preceq_P y$ .  $P$  is a pattern, so it must respect the global ordering ‘ $\preceq$ ’ as specified in (2). Moreover,  $P$  is rigid so the restriction of ‘ $\preceq$ ’ 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  $y \preceq_P x$ .  $\square$

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  $x_1, x_2 \dots x_k$  of at least two distinct elements such that

$$(x_i \preceq_P x_j) \quad \vee \quad (x_i \preceq_Q x_j)$$

whenever  $i \in \{1 \dots k - 1\} \wedge j = i + 1$  or  $i = k \wedge j = 1$ .

Consider the sequence as a circular graph with vertices  $x_1 \dots x_k$  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  $x_a \dots x_b$  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  $y_1 \dots y_m$  such that

$$(y_i \preceq_P y_{i+1}) \quad \wedge \quad (y_{i+1} \preceq_Q y_j)$$

whenever  $i \in \{1, 3, 5 \dots m - 3\} \wedge j = i + 2$  or  $i = m - 1 \wedge j = 1$ .

Note that  $y_i \in \text{supp}(P) \cap \text{supp}(Q)$  for all  $i \in \{1 \dots m\}$ , 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  $y_1, y_3, \dots, y_{m-1}$ . We cannot have  $y_i \preceq_P y_j$  for all  $i \in \{1, 3 \dots m - 3\} \wedge j = i + 2$  and  $i = m - 1 \wedge j = 1$ , because then we would have a cycle in  $P$ . So we have  $y_j \preceq_P y_i$  for some  $i \in \{1, 3 \dots m - 3\} \wedge j = i + 2$  or  $i = m - 1 \wedge j = 1$ . With a possible renaming we can assume that  $y_3 \preceq_P y_1$ . Then we have  $y_3 \preceq_P y_1 \preceq_P y_2$  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.  $\square$

## References

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
- [2] Ken Arnold and James Gosling. *The Java<sup>TM</sup> Programming Language*. The Java<sup>TM</sup> Series. Addison-Wesley, Reading, MA, USA, 1998.
- [3] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, Andrew L. M. Shalit, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *Proceedings OOPSLA'96*, pages 69–82, New York, October 1996. ACM Press.
- [4] B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. *Common Lisp Object System Specification*. Document 88-002R. X3J13, June 1988.
- [5] D. G. Bobrow and M. S. Stefik. *The LOOPS Manual*. Xerox PARC, 1981.
- [6] D. G. Bobrow and T. Winograd. An overview of KRL, a knowledge representation language. *Cognitive Sci 1:1*, 1977.
- [7] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In Rachid Guerraoui, editor, *Proceedings ECOOP'99*, LNCS 1628, pages 43–66. Springer-Verlag, Lisboa, Portugal, June 1999.
- [8] Alan H. Borning and Daniel H. H. Ingalls. Multiple inheritance in smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237. American Association for Artificial Intelligence, 1982. Also Univ. of Washington Tech. Rep. 82-06-02.
- [9] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90*, pages 303–311, October 1990.

- [10] Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *Proceedings OOPSLA'93*, pages 215–230, October 1993.
- [11] Howard I. Cannon. *Flavors: A non-hierarchical approach to object-oriented programming*. Symbolics Inc., 1982.
- [12] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, International Symposium Sophia-Antipolis Proceedings*, LNCS 173, pages 51–67. Springer-Verlag, Berlin, Germany, June 1984.
- [13] Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.
- [14] Erik Ernst. Propagating class and method combination. In Rachid Guerraoui, editor, *Proceedings ECOOP'99*, LNCS 1628, pages 67–91, Lisboa, Portugal, June 1999. Springer-Verlag.
- [15] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP'01*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.
- [16] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 January 1998.
- [17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.
- [18] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, USA, 1989.
- [19] Günter Kniesel. *Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, Computer Science Department III, University of Bonn, July 2000.
- [20] Jørgen Lindskov Knudsen. Name collision in multiple classification hierarchies. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP'88*, LNCS 322, pages 93–109, Oslo, August 15-17 1988. Springer-Verlag.
- [21] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. Strong typing of object-oriented languages revisited. In *Proceedings OOPSLA/ECOOP'90*, pages 140–150, October 1990.
- [22] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.
- [23] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.
- [24] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 37–51. ACM, ACM, January 1985.
- [25] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings ECOOP'02*, LNCS 2374, pages 89–110, Malaga, Spain, 2002. Springer-Verlag.
- [26] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.
- [27] Patrick Steyaert, Wim Codenie, Theo D'Hondt, Koen De Hondt, Carine Lucas, and Marc Van Limberghen. Nested Mixin-Methods in Agora. In Oscar M. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 197–219. Springer-Verlag, 1993.
- [28] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring '87 EUUG Conference*, Helsinki, May 1987.
- [29] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA'87*, pages 227–242, Orlando, FL, October 1987.