Safe Dynamic Multiple Inheritance

Erik Ernst[1]

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 [?,?] is another such mechanism, capable of combining classes as well as methods since the pattern concept [?,?,?] 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 class of pairs of operands 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 class is to say that at least one operand must have been created by single inheritance.

It is crucial that this criterion only restricts one operand, the other operand is allowed to be an arbitrary pattern. Because of this, it is possible for programmers to employ dynamic combination of patterns which are not known at compile-time, as in the following gbeta example:

```
addColor:
  (# inClass,outClass: ##Point;
  enter inClass##                              (* argument list *)
  do inClass&ColorPoint## -> outClass##        (* method body *)
  exit outClass##                              (* result *)
  #)
```
Box 1

The classes `Point` and `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 **??** 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. 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 this:

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

This expression 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 (at '1') proceeds 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`.

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:

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

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

The rest of this paper concentrates on the treatment of a formalization of the core of the pattern combination mechanism and its domain. Section **??** presents the basic domains and operations, such as mixins, patterns, and pattern combination. Section **??** describes a criterion on the operands of a pattern combination operation that ensures successful combination, and proves the correctness of this claim. Section **??** discusses this result in context of the language gbeta. Finally, Sec. **??** presents related work and Sec. **??** concludes.

2 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 [**?**, 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:*

$$\forall x \in \mathrm{dom}(P),\ y \in \mathcal{M}\ .\ \ x \preceq y \Rightarrow y \in \mathrm{dom}(P) \tag{1}$$

$$\forall x, y \in \mathrm{dom}(P)\ .\ \ x \preceq y \Rightarrow x \preceq_P y \tag{2}$$

$$\forall x, y \in \mathrm{dom}(P)\ .\ \ x \preceq_P y\ \vee\ y \preceq_P x \tag{3}$$

If R is an order relation on a set A then $\mathrm{dom}(R)$ is the *domain* 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., $x \preceq_R y \Leftrightarrow (x,y) \in R$.

Criterion (**??**) says that for every mixin x in a pattern P, all elements larger than x are also in P. In other words, the domain of a pattern is upwards closed. Criterion (**??**) 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 ordering of mixins. As a consequence, every pattern P must be a superset of the restriction of '$\preceq$' to $\mathrm{dom}(P)$. Criterion (**??**) just expresses that every pattern is a total order on its domain.

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 `(# color: @string #)`, which adds an attribute named `color` to its superclass.

The intuition behind the global ordering '$\preceq$' is that it expresses inheritance dependencies. 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. Now, criterion (**??**) says that each mixin can only exist in a pattern where all the mixins from which it inherits are also present. Criterion (**??**) says that the ordering of mixins in every pattern must respect the inheritance based ordering. It is a useful approximation of the 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 (**??**) ensures that every question about overriding has a well-defined answer.[1] Now we can specify the meaning of the pattern combination operator:

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

$$R_1 \,\&\, R_2 \;\triangleq\; R \cup (\mathrm{dom}(R_2) \times \mathrm{dom}(R_1) \setminus \overline{R}) \tag{4}$$

where $R \triangleq (R_1 \cup R_2)^*$, i.e., the transitive closure of the union of R_1 and R_2, and $\overline{R}$ is the inverse relation of R, i.e. $\{(y,x)|(x,y) \in R\}$.

In [**?**] this operator was defined on total pre-orders, so the present definition is slightly less general. In fact, we tried to integrate patterns as total pre-orders into gbeta for about two years; but the inherent lack of ordering[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. The new developments starting with the results in this paper seem much more promising.

Intuitively, $R_1 \& R_2$ is computed by adding R_1 and R_2 (that is $R_1 \cup R_2$), then adding ordering-elements such that the relation is again transitive (producing R), and finally adding all elements from R_2 to R_1 that do not contradict R—effectively making R_2 elements smaller than R_1 elements, unless anything is known to the contrary. A simple algorithm that computes $R_1 \& R_2$ from R_1 and R_2 is given in [**?**, Fig. 4]. It is known as the 'C3' linearization algorithm [**?**].

As it was proved in [**?**, Prop. 2], this operation will produce a total order whenever $R_1 \cup R_2$ 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 $\mathrm{dom}(R_1) \cup \mathrm{dom}(R_2) = \mathrm{dom}(R_1 \& R_2)$. This shows that criterion (**??**) and (**??**) 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 patterns.*

[1] Note that overriding is used here as a useful approximation to the actual gbeta semantics which is based on pattern combination and not overriding.

[2] A total order is isomorphic to a list, whereas a total pre-order is isomorphic to a list of *sets* of mutually unordered elements

3 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. 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. Hence, any combination operation applied to a pattern that was only known by an upper bound at compile-time would be flagged by the `gbeta` compiler as a dangerous operation.

However, a new possibility arises with the introduction of the global ordering '$\preceq$' (not considered in [**?**]), 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* $\mathrm{dom}(P)$ *is a total order.*

When the restriction of '$\preceq$' to $\mathrm{dom}(P)$ is a total order, there can only be *one* pattern with domain $\mathrm{dom}(P)$. 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 $x, y \in \mathrm{dom}(P) \cap \mathrm{dom}(Q)$, 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. **??**. 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. **??**. 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 `inClass` in the method `addColor`. 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.

4 In Context of the Language

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

Property (**??**) 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 (**??**), that the domain 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 '$\preceq$', and because

> *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.

Property (**??**) 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. It would be highly useful though hardly trivial to formalize the entire gbeta analysis and establish a proof of this property.

Finally property (**??**), that patterns respect '$\preceq$', is ensured by property (**??**) 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 (**??**) ensures that a pattern that is not created by pattern combination will satisfy (**??**), because '$\preceq$' 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 $x \preceq_{P\&Q} y$, but $y \preceq x$. If $y \in \mathrm{dom}(P)$ then also $x \in \mathrm{dom}(P)$ by property (**??**), 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 $y \in \mathrm{dom}(Q)$.

A very important observation is that the safety of pattern combination with a rigid pattern applies recursively: When pattern combination propagates, as described in [**?**], 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. **??** 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 $\mathcal{P}$ 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 stabilizing. The test according to Th. **??** has been implemented for about two months. 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 type-safe 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.

5 Related Work

Long ago in 1977, a need arose in AI research to handle multiple classification, realized in the knowledge representation language KRL [**?**]. Four years later, the OO Lisp dialect LOOPS [**?**] was created, supporting multiple inheritance. In 1982 the Lisp dialect Flavors [**?**] 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 [**?**], this programming convention became known as *mixin classes*. A version of Smalltalk was also equipped with multiple inheritance [**?**]. From this point, a dichotomy emerged: Multiple inheritance was formalized [**?**], introduced in statically typed languages [**?**,**?**], and problematized [**?**] 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 [**?**] introduced mixins as a separate concept that generalizes several kinds of inheritance. A number of variants [**?**,**?**] emerged, where [**?**] introduced the notion of an *inheritance interface*, specifying the requirements of a mixin on potential superclasses. Recently, a class-based object calculus [**?**] supporting statically typed mixin application seems to establish mixins as a well-understood mechanism, thus bridging the long-standing gap between the statically and the dynamically typed side of the dichotomy.

The mixin concept is fundamental to gbeta, but mixins cannot be manipulated individually, only via '&'; in this sense it is similar to CLOS and the other early class-based approaches. It also uses linearization, specifically the C3 [?] algorithm, but the formalization (Def. ??) 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 [?] and the mixin application type check in [?], but gbeta mixin application is more dynamic because it allows combination of patterns not known before run-time. As this paper shows, dynamic mixin application is safe under certain conditions.

Finally, dynamic mixin application connects gbeta and languages with *delegation*. In particular, LAVA [?] supports static and dynamic delegation; [?] made it clear that `final` declarations are incompatible with subtyping when they occur in a delegatee, corresponding to a mixin's superclass.

6 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 two 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 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.

References

1. 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, ACM SIGPLAN Notices*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 69–82, New York, October6–10 1996. ACM Press.
2. D. G. Bobrow and M. S. Stefik. *The LOOPS Manual*. Xerox PARC, 1981.
3. D. G. Bobrow and T. Winograd. An overview of krl, a knowledge representation language. *Cognitive Sci 1:1*, 1977.
4. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, New York, NY, June 1999.

5. 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.
6. Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, volume 25, 10, pages 303–311, October 1990.
7. Howard I. Cannon. Flavors: A non-hierarchical approach to object-oriented programming. Symbolics Inc., 1982.
8. L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, International Symposium Sophia-Antipolis Proceedings*, volume 173 of *LNCS*, pages 51–67. Springer-Verlag, Berlin, Germany, June 1984.
9. 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.
10. 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.
11. 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.
12. Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, USA, 1989.
13. Günter Kniesel. *Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, Computer Science Department III, University of Bonn, July 2000.
14. 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.
15. Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. Strong typing of object-oriented languages revisited. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, volume 25, 10, pages 140–150, October 1990.
16. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.
17. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science, C.A.R. Hoare Series Editor. Prentice Hall International, Hemel Hempstead, UK, 1988. Nouvelle édition révisée : [**?**].
18. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.
19. 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.
20. Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring '87 EUUG Conference*, Helsinki, May 1987.

A Proofs

Proof (of Lemma **??***).* Assume that $x \preceq_P y$. P is a pattern, so it must respect the global ordering '$\preceq$' as specified in (**??**). Moreover, P is rigid so the restriction of '$\preceq$' to

$\mathrm{dom}(P)$ 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 (**??**). The argument is similar under the assumption $y \preceq_P x$. □

Proof (of Theorem **??***).* Assume that $P \cup Q$ contains a cycle. We must show that this leads to a contradiction, and then the result follows from Lemma **??**.

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

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

for $i \in \{1 \dots k-1\} \wedge j = i+1$ and $i = k \wedge j = 1$.

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 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

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

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

Note that $y_i \in \mathrm{dom}(P) \cap \mathrm{dom}(Q)$ 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 **??** 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. □