# A Monotonic Superclass Linearization for Dylan

Kim Barrett <kab@camellia.org>
Bob Cassels <Cassels@harlequin.com>
Paul Haahr <haahr@netcom.com>
David A. Moon <Moon@mc.lcs.mit.edu>
Keith Playford <keith@harlequin.co.uk>
P. Tucker Withington <ptw@harlequin.com>†

*28 June 1996*

## Abstract

Object-oriented languages with multiple inheritance and automatic conflict resolution typically use a linearization of superclasses to determine which version of a property to inherit when several superclasses provide definitions. Recent work has defined several desirable characteristics for linearizations, the most important being monotonicity, which prohibits inherited properties from skipping over direct superclasses. Combined with Dylan's sealing mechanism, a monotonic linearization enables some compile-time method selection that would otherwise be impossible in the absence of a closed-world assumption.

The Dylan linearization is monotonic, easily described, strictly observes local precedence order, and produces the same ordering as CLOS when that is monotonic. We present an implementation based on merging and a survey of class heterarchies from several large programs, analyzing where commonly used linearizations differ.

# Introduction

### Why linearizations?

In a class-based object-oriented language, objects are instances of *classes*. The *properties* of an object – what slots or instance variables it has, which methods are applicable to it – are determined by its class. A new class is defined as the *subclass* of some pre-existing classes (its *superclasses* – in a single-inheritance language, only one direct superclass is allowed), and it *inherits* the properties of the superclasses, unless those properties are overridden in the new class. Typically, circular superclass relationships are prohibited, so a hierarchy (or heterarchy, in the case of multiple inheritance) of classes may be modeled as a directed acyclic graph with ordered edges. Nodes correspond to classes, and edges point to superclasses. Languages that use this model include Ada 95, C++, CLOS, Dylan, Eiffel, Java, Oberon-2, Sather, and Smalltalk.

In object-oriented systems with multiple inheritance, some mechanism must be used for resolving conflicts when inheriting different definitions of the same property from multiple superclasses. Some languages require manual resolution by the programmer, with mechanisms such as explicit delegation in C++ [ES 90] and feature renaming in Eiffel. [Meyer 88]

Dylan, [Apple 92] [Apple 94] [Shalit 96] like other object-oriented descendants of Lisp (e.g., Flavors [Moon 86], LOOPS [SB 86], and CLOS [Steele 90] [KdRB 91]), automatically resolves conflicts occurring in method dispatch. This resolution is implemented using a *linearization*. When a class is created, a linearization of its superclasses, including itself, (also known as the "class precedence list" or CPL) is determined, ordered from

† The work reported here was started when Barrett, Cassels, and Moon were at Apple Computer, and Haahr, Playford, and Withington were at Harlequin. At the time of writing, Cassels, Playford, and Withington are with Harlequin, Barrett is at IS Robotics, and Haahr and Moon are unaffiliated.

most specific to least specific. When several methods are applicable for a given call, the one defined on the most specific class, according to the linearization, is selected.

Dylan, like CLOS, uses generic functions with multimethods, that is, methods specialized on more than one parameter. The use of the class precedence list generalizes to multimethods without difficulty, but for purposes of presentation, we will not consider multimethods further in this paper.

Most object-oriented languages implicitly use a rule similar to linearization for method dispatching in single inheritance: a class is more specific than any of its superclasses, so methods defined for subclasses override methods defined for superclasses. The problem with generalizing this to multiple inheritance is that the simple rule does not make clear which of two superclasses with no subclass/superclass relationship between them is more specific.

For example, consider the simple use of multiple inheritance in example 1a. (For details on Dylan language constructs, see the *Dylan Reference Manual*. [Shalit 96]) The question multiple inheritance raises is "What is the starting-edge for an <hv-grid>?" If it is more like a horizontal than a vertical grid, it is the left edge, but if it is more like a vertical grid, it is the top edge. In an explicit resolution system, the author of the <hv-grid> class would have to write a declaration or method to choose which superclass to obtain the starting-edge behavior from. In contrast, when a linearization is used, the default behavior is determined by which of <horizontal-grid> or <vertical-grid> appears first in the linearization.

Following CLOS, Dylan uses the *local precedence order* – the order of the direct superclasses given in the class definition – in computing the linearization, with earlier superclasses considered more specific than later ones. Therefore, since <horizontal-grid> precedes <vertical-grid> in the definition of <hv-grid> it will also precede it in the linearization. The full linearization for <hv-grid> is

```
<hv-grid>, <horizontal-grid>,
<vertical-grid>, <grid-layout>, <object>
```

On the other hand, to create a combined horizontal and vertical grid which is more like a vertical grid than a horizontal one, the only change necessary to the definitions above would be to reverse the order of the direct superclasses in the class that combines the two grids; see example 1b.

It is possible that an inheritance graph is *inconsistent* under a given linearization mechanism. This means that the linearization is over-constrained and thus does not exist for the given inheritance structure. An example of an inconsistent inheritance relationship appears in example 1c. <confused-grid> is inconsistent because it attempts to create a linearization that has

```
define class <grid-layout> (<object>) … end;
define class <horizontal-grid> (<grid-layout>) … end;
define class <vertical-grid> (<grid-layout>) … end;
define class <hv-grid> (<horizontal-grid>, <vertical-grid>) … end;

define method starting-edge (grid :: <horizontal-grid>)
  #"left"
end method starting-edge;

define method starting-edge (grid :: <vertical-grid>)
  #"top"
end method starting-edge;
```

*Example 1a: A simple use of multiple inheritance*

```
define class <vh-grid> (<vertical-grid>, <horizontal-grid>) … end;
```

*Example 1b: Reversing classes in the linearization*

```
define class <confused-grid> (<hv-grid>, <vh-grid>) … end;
```

*Example 1c: An inconsistent class definition*

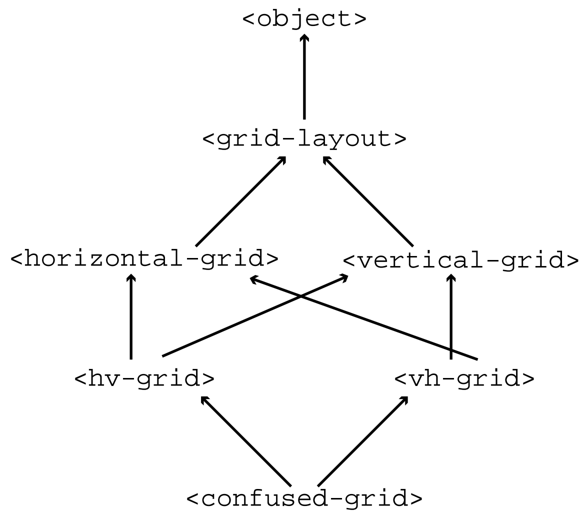


*Figure 1: An inconsistent class heterarchy*

`<horizontal-grid>` before `<vertical-grid>`, because it subclasses `<hv-grid>`, and `<vertical-grid>` before `<horizontal-grid>`, because it subclasses `<vh-grid>`. Clearly, both of these constraints cannot be obeyed in the same class.

**Characteristics of linearizations**

A number of characteristics have been identified as being desirable for linearizations. Two of these characteristics – acceptability and monotonicity – were advocated by Ducournau, Habib, *et. al.* [DHHM 92] [DHHM 94]. In addition, the CLOS linearization strictly observes local precedence order, and Ducournau, Habib, *et. al.* discuss the local precedence order (under the name $prec_C$).

An *acceptable* linearization is one in which only the shape of a class's inheritance graph may be used in determining the linearization. All the linearizations considered here are acceptable, but one based on classes outside the inheritance graph (as in the global linearization proposed in [Baker 91]) or on the names of classes would not be.

A linearization that *observes local precedence order* will not produce a linearization which is inconsistent with the local precedence orders of any of the superclasses. That is, if class A precedes class B in the local precedence order of class C, the linearizations of C and all of its subclasses should have A before B.

A *monotonic* linearization is one in which every property inherited by a class is inherited or defined by one of its direct superclasses; that is, an inherited property cannot skip over all of the direct superclasses. This means that the linearization of a class must be an extension without reordering of the linearizations of all of its superclasses.

[DHHM 94] gives the class heterarchy pictured in figure 2 as an example for monotonicity. The linearizations for `<pedalo>` and its direct superclasses using the CLOS mechanism are:

```
<pedal-wheel-boat>:
  <pedal-wheel-boat>, <engineless>,
  <day-boat>, <wheel-boat>, <boat>,
  <object>

<small-catamaran>:
  <small-catamaran>, <small-multihull>,
  <day-boat>, <boat>, <object>

<pedalo>:
  <pedalo>, <pedal-wheel-boat>,
  <engineless>, <wheel-boat>,
  <small-catamaran>, <small-multihull>,
  <day-boat>, <boat>, <object>
```

Consider a method defined on `<day-boat>` and `<wheel-boat>`. For both direct superclasses of `<pedalo>`, the method on `<day-boat>` is the most specific, yet the method on `<wheel-boat>` is the most specific for `<pedalo>`. With a monotonic linearization, this surprising result cannot occur.

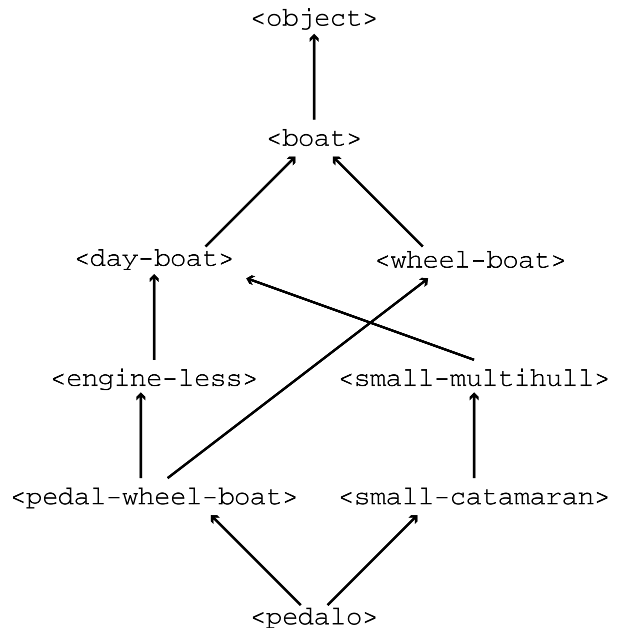


*Figure 2: For CLOS, a non-monotonic class heterachy, from [DHHM 94]*

## Computing linearizations

Typically, a linearization is computed by merging a set of constraints or, equivalently, topologically sorting a relation on a graph, though other mechanisms are possible. The linearizations we considered can be expressed as the merging of a set of sequences and a selection rule for ambiguous cases.

The merge of several sequences is a sequence that contains each of the elements of the input sequences. An element that appears in more than one of the input sequences appears only once in the output sequence. If two elements appear in the same input sequence, their order in the output sequence is the same as their order in that input sequence. (Note that a class cannot appear twice in the same input sequence.)

The linearization used in Dylan merges the local precedence order of the class being defined with the linearizations of its direct superclasses. The CLOS linearization merges the local precedence orders of the class and of all its superclasses.

If there is no possible merged output sequence which is consistent with all the input sequences, the class heterarchy is inconsistent.

It is possible that there are several valid merges for some set of input sequences. For example, consider the class heterarchy in example 2 (pictured in figure 3). Using the Dylan mechanism, computing the linearization for <popup-menu> involves merging the sequences:

```
<popup-menu>, <menu>, <popup-mixin>
<menu>, <choice-widget>, <object>
<popup-mixin>, <object>
```

where the first is the local precedence order for <popup-menu> and the second and third are, respectively, the linearizations of <menu> and <popup-mixin>.

The first two elements of the merged result are <popup-menu> and <menu>, but the third element of the linearization is not unambiguously determined by the input sequences. After <popup-menu> and <menu> have been placed in the output, either <choice-widget> or <popup-mixin> could appear next, since both have all their predecessors already in the merged sequence.

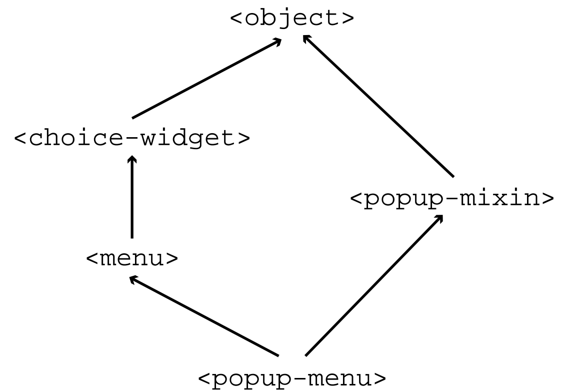


*Figure 3: The heterarchy from Example 2*

A mechanism is needed for choosing between <choice-widget> and <popup-mixin>. Dylan and CLOS use the same rule, which is to select the class that has a direct subclass closest to the end of the output sequence, as currently computed. The result so far is (<popup-menu>, <menu>), and <menu> is a direct subclass of <choice-widget> so <choice-widget> is selected. After that, no ambiguities remain, and the full linearization is the sequence (<popup-menu>, <menu>, <choice-widget>, <popup-mixin>, <object>).

[DHHM 94] introduces the $\mathcal{L}^*_{\text{LOOPS}}$ linearization, a variation on the linearization used in LOOPS, which was designed to be monotonic. The original construction of $\mathcal{L}^*_{\text{LOOPS}}$ is in terms of a depth-first topological sort on the *linearization graph*, which is a graph where the vertices are the classes in the heterarchy and the arcs are determined by the linearizations of the direct superclasses. To facilitate comparisons with the other linearizations, we will describe $\mathcal{L}^*_{\text{LOOPS}}$ in terms of merging.

$\mathcal{L}^*_{\text{LOOPS}}$ merges the linearizations of the direct superclasses, but, unlike Dylan and CLOS, does not include the local precedence order in the merge. When

```
define class <choice-widget> (<object>) … end;
define class <menu> (<choice-widget>) … end;
define class <popup-mixin> (<object>) … end;
define class <popup-menu> (<menu>, <popup-mixin>) … end;
```

*Example 2: A class heterarchy with an ambiguity in constraints*

selecting the next class from several alternatives for which all the predecessors are already in the output sequence, it uses the first unselected class in a depth-first ordering of the linearization graph.

# The Dylan Linearization

### Why Monotonicity?

Dylan [Apple 92] [Apple 94] [Shalit 96] originally specified a linearization equivalent to the one used in CLOS. In light of the research done by Ducournau, Habib, *et. al.*, and the definition of the monotonicity criterion, it was decided that the CLOS linearization should be replaced by a monotonic linearization.

The first reason for the change is that a monotonic linearization seems a better match for users' intuitions about how inheritance and linearizations work. With a monotonic linearization, it is easier to understand the behavior of classes when multiple inheritance is used, largely because behavior of instances of a class can be explained in terms of the direct superclasses.

A second reason for picking a monotonic linearization for Dylan is that it enables more compile-time method selection. Dylan has an innovative mechanism, known as *sealing*, for describing what extensions to a library are possible after it is compiled. A *sealed class* may not be directly subclassed outside of the library where it is defined, whereas an *open* class may be. Similarly, methods may be added outside the defining library to an *open generic function*, but not to a *sealed* one. A *domain* of a generic function may be sealed, which prohibits the definition of new methods or classes which would change the applicability of methods for the types specified by the domain. (For details on sealing, see the *Dylan Reference Manual*. [Shalit 96]) The restrictions imposed by sealing permit a Dylan compiler to select methods for generic function calls at compile-time without imposing a closed-word assumption on the program.

Consider the pedalo example from above in the context of sealing. Suppose that all the classes are open (and thus can be extended), but there is a sealed function `max-distance` with methods defined on `<day-boat>` and `<wheel-boat>`. Now consider the method defined in example 3. If the linearization is known to be monotonic, the compiler can choose to dispatch the call to `max-distance` directly to the method defined on `<day-boat>`. This is known statically because no new methods can be defined on `max-distance` – it is sealed – and `<day-boat>` is always more specific than `<wheel-boat>` for instances of `<pedal-wheel-boat>`.

### The Algorithm

As described above, the Dylan linearization merges the local precedence order of a class with the linearizations of its direct superclasses. When there are several possible choices for the next element of the linearization, the class that has a direct subclass closest to the end of the output sequence is selected.

It should be clear that the Dylan linearization is monotonic, because the merge procedure never reorders the linearizations of superclasses when producing the linearization. Similarly, it obeys local precedence order because the merge explicitly takes local precedence into account, and the local precedence orders of superclasses are propagated by the linearizations.

An implementation of the Dylan linearization appears in Appendix A.

# Empirical results

When we decided to adopt a monotonic linearization for Dylan, we had initially considered using $\mathcal{L}^*_{\text{LOOPS}}$. The first problem we encountered was that the presentation of the algorithm in [DHHM 94] makes it hard to see what the differences with the existing (CLOS) approach were. We were concerned that existing class heterarchies would be reordered by $\mathcal{L}^*_{\text{LOOPS}}$, causing mysterious bugs. We wanted to ensure that all of the differences between our new linearization and the previous one (CLOS) could be justified as part of the desired new properties (e.g., monotonicity) and were not due to gratuitous

```
define method max-pedal-rotations (pwb :: <pedal-wheel-boat>)
  max-distance(pwb) / distance-per-pedal-rotation(pwb)
end method max-pedal-rotations;
```

*Example 3: A method which may permit sealing optimizations*

| *heterarchy* | *classes* | *MI joins* | *different CPLs* | *Dylan vs. CLOS* | *Dylan vs.* $\mathcal{L}^*_{LOOPS}$ | *CLOS vs.* $\mathcal{L}^*_{LOOPS}$ | $\mathcal{L}^*_{LOOPS}$ *inconsistent* |
|---|---|---|---|---|---|---|---|
| LispWorks | 507 | 70 | 0 | 0 | 0 | 0 | 0 |
| CLIM | 842 | 184 | 31 (21) | 31 (21) | 5 | 31 (21) | 0 |
| database | 38 | 4 | 0 | 0 | 0 | 0 | 0 |
| emulator | 571 | 205 | 8 (4) | 8 (4) | 0 | 8 (4) | 0 |
| proprietary | 665 | 124 | 81 (12) | 2 | 81 (12) | 81 (12) | 19 (8) |
| Watson | 673 | 114 | 0 | 0 | 0 | 0 | 0 |
| total | 3296 | 701 | 120 (37) | 41 (27) | 86 (17) | 120 (37) | 19 (8) |

*Table 1: Comparison of the Dylan, CLOS, and* $\mathcal{L}^*_{LOOPS}$ *linearizations*

incompatibilities. Compatibility with CLOS was considered important both because of an existing body of Dylan code that assumed that linearization and because of the substantial amount of real experience with it in the Common Lisp community indicating that it could be used successfully.

In order to understand the scope of the change we were making to Dylan, we surveyed the classes defined in six class heterarchies from existing large CLOS programs and compared the linearizations computed for them by the Dylan, CLOS, and $\mathcal{L}^*_{LOOPS}$ mechanisms. The results of this survey are summarized in table 1.

The first column contains the program which the heterarchies came from. The programs we studied were:

LispWorks – The implementation of CLOS in Harlequin's LispWorks development environment and the set of classes used in its user interface. (Note that all these class heterarchies were built on top of the basic LispWorks heterarchy, but those classes have only been counted once, in the totals for LispWorks.)

CLIM – The Common Lisp Interface Manager, a Lisp-based programming interface that provides a layered set of portable facilities for constructing user interfaces. [MY 94]

database – An interface to SQL databases.

emulator – A Dylan emulator on top of CLOS and some basic class libraries (collections, streams) written in Dylan. (The CLOS linearization was used for all classes when originally written.)

proprietary – A large proprietary application written by a Harlequin customer which makes heavy use of CLOS.

Watson™ – Harlequin's Watson product, a data analysis tool.

The next two columns give the number of classes and the number of multiply inheriting classes in the heterarchy, respectively. It is only the multiply inheriting classes for which there is the possibility of a difference in linearizations. The next four columns summarize the differences that appeared in the linearizations. First, we show the number of classes for which any two linearizations differed, and the following columns give the results for pairwise comparisons between the different linearizations.

The final column reports the number of classes in each heterarchy which were inconsistent under the $\mathcal{L}^*_{LOOPS}$ linearizations; no inconsistent classes were found when using Dylan or CLOS. We do not claim that $\mathcal{L}^*_{LOOPS}$ inherently makes inconsistent classes more common: this data was taken from CLOS programs, and any inconsistencies with respect to the CLOS linearization would have been eliminated as part of the normal development process before the class heterarchies were surveyed. The Dylan linearization is sufficiently similar to CLOS that definitions consistent with one are typically consistent with the other.

It is important to recognize that if two linearizations differ for a given class, they will differ for its subclasses; similarly, if a class is inconsistent under one linearization, all its subclasses will be, too. The difference or inconsistency even appears in subclasses which are constructed by single inheritance, despite the fact that all linearizations use the same mechanism for single inheritance: prefixing the linearization of the single superclass with the class being defined. Therefore, in the table above, we have reported in parentheses the number of classes for each category which were not subclasses of

classes that were already counted in the category, when that number differed from the total number. The large number of differences in the linearizations encountered from the "proprietary" data set can be explained by observing that a few classes, with a large number of subclasses, account for most of the differences; in fact, a single class accounts for more than half of the differences between the Dylan and $\mathcal{L}^*_{\text{LOOPS}}$ linearizations.

### Comparison with $\mathcal{L}^*_{\text{LOOPS}}$

An important difference between CLOS and $\mathcal{L}^*_{\text{LOOPS}}$ that our survey made clear is that $\mathcal{L}^*_{\text{LOOPS}}$ does not observe local precedence order for some inheritance graphs. Notably, *transitivity edges* in the inheritance graph – where a class has a direct superclass that is also an indirect superclass – are ignored by $\mathcal{L}^*_{\text{LOOPS}}$, leading to cases where it disobeys local precedence order.

To see the effect of transitivity edges, consider the alternate definition in example 4 of the popup menu class from example 2.

The $\mathcal{L}^*_{\text{LOOPS}}$ linearization for `<new-popup-menu>` is (`<new-popup-menu>`, `<menu>`, `<choice-widget>`, `<popup-mixin>`, `<object>`), which violates the local precedence order in `<new-popup-menu>` that `<popup-mixin>` is supposed to appear before `<choice-widget>`. The linearization produced by both Dylan and CLOS is (`<new-popup-menu>`, `<menu>`, `<popup-mixin>`, `<choice-widget>`, `<object>`).

Uses of inheritance with transitivity edges may seem, at first glance, odd. Why should `<new-popup-menu>` list both `<menu>` and `<choice-widget>` as direct superclasses, when `<menu>` is a superclass of `<choice-widget>`? We hypothesize two reasons for such inheritance graphs. The first reason is that one can use such a technique to exercise fine control over the linearization. That is, the selection rule used in the linearization may not do what the programmer wants in some cases, and adding transitivity edges can constrain the merge so that the intended result is obtained. In this example, the programmer's intention could be to ensure that the behavior of `<new-popup-menu>` follows `<popup-mixin>` and not `<choice-widget>` for some particular method. However, this form of tuning is fragile, because it requires deep knowledge of the heterarchy and the linearization on behalf of the programmer; subtle changes in early parts of the class heterarchy could cause such uses to create inconsistencies. We suspect that this technique is rarely used, but an author of one program in our survey reports adjusting local precedence orders until method dispatch matched his intuition, and that process may have introduced transitivity edges.

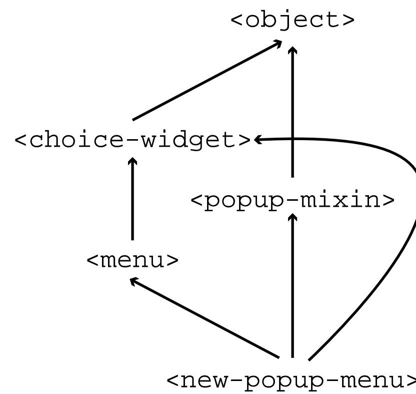


*Figure 4: The heterarchy from example 4*

The second reason comes from just the opposite cause: the programmer knows little about the class heterarchy, and intends to mix in some behavior she wasn't aware was already in one of the classes being used. This commonly occurs during development if classes near the root of the heterarchy are redefined with extra superclasses that might already be inherited by some subclasses.

We believe that observing local precedence order is an important characteristic for a linearization, and our experience with Dylan and CLOS confirms this. If local precedence is not obeyed, the order in a linearization of the direct superclasses cannot be understand only in terms of the class declaration and the ability to exercise fine control over the linearization process is lost. Ducournau, Habib, *et. al.*, [DHHM 94] disagree, noting that "The fact that LOOPS does not always respect the local order – here *prec'* – is not a problem … [because] the part of *prec'* which is not respected by LOOPS is always a contradictory part of" the extended precedence graph. [Section 2.4.4] We note that the "contradiction" caused

```
define class <new-popup-menu> (<menu>, <popup-mixin>, <choice-widget>) … end;
```

*Example 4: A variation on the popup menu class, with an extra constraint*

by transitivity edges is only with edges that are added to the class graph to produce the extended precedence graph, and the contradiction takes the form of cycles in the EPG. (See below for details on the extended precedence graph.)

**Comparison with CLOS**

The Dylan linearization is an extension of the one used in CLOS, where the central difference is that Dylan uses the linearizations of superclasses to preserve monotonicity.

The Dylan and CLOS linearizations, when considered as merge operations, have identical structures. Further, the rule used in selecting the next class when there are several for which all predecessors have already been removed from the input sequences is the same. The only difference is in the sequences being merged: CLOS uses the local precedence orders of all superclasses (including the class itself), whereas Dylan uses the linearizations of the direct superclasses and the local precedence order of the class.

Note that the sequences merged in the Dylan linearization strictly contain those merged by CLOS; that is, Dylan imposes a superset of the constraints used in CLOS on linearizations, and these extra constraints – ordering requirements from the linearizations of superclasses – are exactly those needed to enforce monotonicity. The only cases where the linearizations can be different are those in which CLOS selects among several classes where all the predecessors have been placed in the output sequence, but at least one of those classes has a predecessor in the Dylan linearization, and therefore cannot be selected next. That extra predecessor enforces monotonicity; if it were not present, the result would be non-monotonic. Thus, the Dylan and CLOS linearizations produce the same results when CLOS is monotonic.

Consider the `<pedalo>` example from above. The Dylan linearizations of the direct superclasses of `<pedalo>` are the same as those found in CLOS:

```
<pedal-wheel-boat>:
  <pedal-wheel-boat>, <engineless>,
  <day-boat>, <wheel-boat>, <boat>,
  <object>

<small-catamaran>:
  <small-catamaran>, <small-multihull>,
  <day-boat>, <boat>, <object>
```

Thus the Dylan linearization for `<pedalo>` is the result of merging those two sequences with the following, which is the local precedence order for `<pedalo>`:

```
<pedalo>, <pedal-wheel-boat>,
  <small-catamaran>
```

On the other hand, the CLOS linearization is produced by merging

```
<boat>, <object>

<day-boat>, <boat>

<wheel-boat>, <boat>

<engineless>, <day-boat>

<pedal-wheel-boat>, <engineless>,
  <wheel-boat>

<small-multihull>, <day-boat>

<small-catamaran>, <small-multihull>

<pedalo>, <pedal-wheel-boat>,
  <small-multihull>
```

Note that the sequences used in the Dylan linearization require that `<day-boat>` precede `<wheel-boat>` in `<pedalo>`, due to the effect of the linearization of `<pedal-wheel-boat>`. No such requirement exists for CLOS, thus it is able to select `<wheel-boat>` before `<day-boat>`, with a non-monotonic result.

Ducournau, Habib, *et. al.*, derived $\mathcal{L}^*_{\text{LOOPS}}$ from the LOOPS linearization by constructing a linearization graph for a class and applying LOOPS to that rather than to the inheritance graph defining the class. The Dylan linearization can be considered an application of the CLOS mechanism to the linearization graph; using the terminology of [DHHM 94], it might be named $\mathcal{L}^*_{\text{CLOS}}$.

# The extended precedence graph and the C3 Linearization

The *extended precedence graph* (or EPG), described in [DHHM 92] and [DHHM 94], is an extension of a class heterarchy graph to include the transitive effects of local precedence order.

Consider the class heterarchy as a directed graph with nodes corresponding to classes and a directed edge leading from each subclass to its superclasses. The EPG for a class *C* is constructed by augmenting the heterarchy

graph for *C* with edges connecting each pair of classes (e.g., *A* and *B*) that do not have a subclass/superclass relationship.

The direction of the edge from *A* to *B* is determined by finding the maximal common subclasses of A and B among the superclasses of *C,* that is, classes which are subclasses of both *A* and *B* but do not have any superclasses that are subclasses of both. Since *C* is a subclass of *A* and *B*, there exists at least one such class. For each such class M, if A or a subclass of *A* precedes *B* or a subclass of *B* in the local precedence order of *M*, there is a directed edge from *A* to *B*. Similarly, if *B* or one of its subclasses precedes *A* or one of its subclasses in *M*'s local precedence order, there is an edge from *B* to *A*.

Note that the EPG may contain cycles. Ducournau, Habib, *et. al.* [DHHM 92] prove that if the EPG is acyclic, the CLOS and LOOPS linearizations produce the same results and are monotonic. (This is true also of the Dylan and $\mathcal{L}^*_{\text{LOOPS}}$ linearizations.)

A linearization is consistent with the extended precedence graph if and only if there is a path in the EPG from every class in the linearization to all of its successors in the linearization. Since there is a path from every node within a cycle to every other node in the cycle, consistency with the EPG imposes no ordering among classes found within a cycle in the EPG, but does imply an ordering for the acyclic portions of the graph.

Unfortunately, the Dylan linearization, like CLOS but unlike $\mathcal{L}^*_{\text{LOOPS}}$, isn't always consistent with the extended precedence graph, and this can lead to counter-intuitive linearizations. Consider the class heterarchy in figure 5.

The Dylan (and CLOS) linearizations order the superclasses of `<editable-scrollable-pane>` as

```
<editable-scrollable-pane>,
  <scrollable-pane>, <editable-pane>,
  <pane>, <editing-mixin>,
  <scrolling-mixin>, <object>
```

which may have surprising consequences for the user, in that `<editing-mixin>` precedes `<scrolling-mixin>` in the linearization of `<editable-scrollable-pane>`, despite the fact that `<scrollable-pane>`, from where `<editable-scrollable-pane>` inherits `<scrolling-mixin>`, precedes `<editable-pane>`, where it gets `<editing-mixin>` from, in `<editable-scrollable-pane>`'s local precedence order. If `<editable-scrollable-pane>` inherits a property defined differently by both `<scrolling-mixin>` and `<editing-mixin>`, `<editable-scrollable-pane>` will behave like `<editable-pane>` rather than `<scrollable-pane>`, contradicting a programmer's reasonable expectation about the interaction of inheritance and local precedence order.

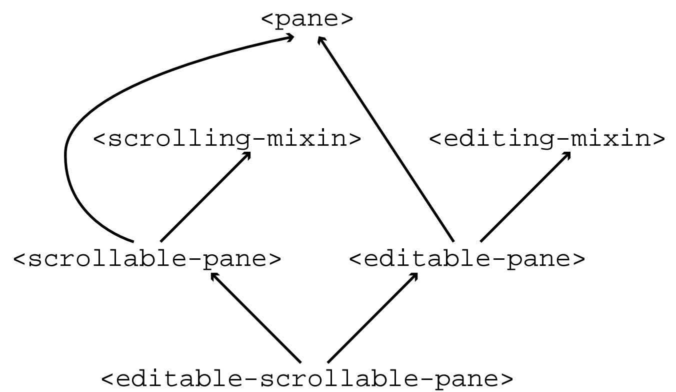

*Figure 5: A heterarchy where Dylan's linearization does not observe the extended precedence graph*

### C3 – A linearization consistent with the EPG

If consistency with the extended precedence graph, local precedence order, and monotonicity are desired, a hybrid of the Dylan and $\mathcal{L}^*_{\text{LOOPS}}$ linearizations can be used. We call this linearization C3, because it is consistent with these three properties. C3 uses the constraints of the Dylan linearization with a selection rule modeled on the one used in $\mathcal{L}^*_{\text{LOOPS}}$.

Our implementation of C3 is similar to the merging algorithm used for the Dylan linearization, differing only in the selection rule. When choosing among several classes for the next element of the linearization, when the input sequences alone do not determine the selection, the C3 linearization chooses the class which appears in the linearization of the earliest direct superclass of the class being defined, in local precedence order. Note that at most one candidate class is in the linearization of each of the direct superclasses, because if two were to appear, one would precede the other due to the constraints and thus the order between them would be determined by the monotonicity requirement.

Examining the `<popup-menu>` class from example 2 again, after `<popup-menu>` and `<menu>` have been

picked as the first two elements of the linearization, a decision has to be made between <choice-widget> and <popup-mixin> as the next element. Since <choice-widget> is a superclass of <menu>, and <menu> precedes <popup-mixin> in the local precedence order of <popup-menu>, the C3 algorithm selects <choice-widget> next. This is the same result as Dylan, though for a different reason.

On the other hand, given the <editable-scrollable-pane> example above, where Dylan is not consistent with the EPG, the selection rule of C3 diverges from Dylan's when selecting between <scrolling-mixin> and <editing-mixin>. Because <scrolling-mixin> is a superclass of <scrollable-pane>, <editing-mixin> is a superclass of <editable-pane>, and <scrollable-pane> precedes <editable-pane> in the local precedence order of <editable-scrollable-pane>, <scrolling-mixin> is selected as the next class.

To show that C3 is consistent with the extended precedence graph, we will demonstrate that C3 cannot violate consistency with the EPG because every time it selects a class it is following arcs that appear in the EPG. First, note that the edges of the class heterarchy graph, from which the EPG is constructed, are included in the input sequences for the C3 merge: the local precedence order and the linearizations of superclasses.

Next, observe that the C3 selection rule, when presented with a choice of two classes where the maximal common subclass of the classes is the class being defined (say *C*), will pick, by construction, the one that is itself or has a subclass earliest in the local precedence order; that is, the one consistent with the extended precedence graph.

What about the case where the selection rule must choose between classes where the maximal common subclass is a superclass of *C*? That cannot occur with the C3 algorithm, because, if there were such a choice to make, it would have been made in the linearization of that maximal common superclass, and thus would be reflected in the input sequences used by the C3 merge. Finally, we observe that every predecessor relationship encoded in the linearizations of superclasses used in the merging process comes from one of three sources: a subclass/superclass relationship, the local precedence order of a superclass of *C*, or one of the augmenting edges from the EPG for a superclass of *C*.

The general effect of the selection rule used in C3, as in $\mathcal{L}^*_{\text{LOOPS}}$, is to produce a depth-first ordering of the class heterarchy, constrained by superclass relationships and local precedence orders. In contrast, the selection rule used by Dylan and CLOS leads to depth-first behavior locally within the graph, but somewhat arbitrary behavior when considering the graph as a whole.

An implementation of the C3 linearization appears in Appendix B.

### Comparison with other linearizations

We compared the C3 linearization with the Dylan and $\mathcal{L}^*_{\text{LOOPS}}$ linearizations on the same set of class heterarchies we used above. The results of the comparison are summarized in table 2.

The first three columns report the same information as in the table above. The fourth and fifth columns report the number of linearizations for which C3 differed from Dylan and $\mathcal{L}^*_{\text{LOOPS}}$, respectively. The final column gives the number of classes for which C3 was unable to produce

| *heterarchy* | *classes* | *MI joins* | *C3 vs. Dylan* | *C3 vs.* $\mathcal{L}^*_{LOOPS}$ | *C3 inconsistent* |
|---|---|---|---|---|---|
| LispWorks | 507 | 70 | 0 | 0 | 0 |
| CLIM | 842 | 184 | 5 | 1 | 0 |
| database | 38 | 4 | 0 | 0 | 0 |
| emulator | 571 | 205 | 0 | 0 | 0 |
| proprietary | 665 | 124 | 80 (11) | 62 (4) | 19 (8) |
| Watson | 673 | 114 | 0 | 0 | 0 |
| total | 3296 | 701 | 85 (16) | 63 (5) | 19 (8) |

*Table 2: Comparison of the C3, Dylan, and* $\mathcal{L}^*_{LOOPS}$ *linearizations*

a consistent linearization. Again, parenthesized entries indicate the number of relevant classes where the difference or inconsistency was not a result of a similar problem in a superclass.

We observe that, most of the time, C3, Dylan, and $\mathcal{L}^*_{\text{LOOPS}}$ produce the same result. When they differ, C3 is more commonly the same as $\mathcal{L}^*_{\text{LOOPS}}$ than it is the same as Dylan, though it is often different from both in such cases.

Further, among the classes we surveyed, C3 finds inconsistent class definitions in exactly the same classes which $\mathcal{L}^*_{\text{LOOPS}}$ does; in general, this indicates that those classes multiply inherit from classes where the extended precedence graphs led to contradictory linearizations. Again, we do not consider the lack of inconsistencies in the Dylan linearization a general property of the mechanism; because of the similarity of the Dylan and CLOS linearizations, this data, coming from large CLOS programs, is unlikely to contain inconsistencies under the Dylan linearization.

Since C3 differs from Dylan only in the selection rule, and C3's selection rule is used to enforce consistency with the extended precedence graph, C3 and Dylan only produce different results when Dylan's selection rule would lead it to be inconsistent with the EPG.

Similarly, since the linearization graph used in $\mathcal{L}^*_{\text{LOOPS}}$ contains precisely the edges corresponding to the linearizations of superclasses used as input to C3's merge and the same selection rule in both algorithms (though presented differently in [DHHM 94]), but local precedence order is not used in the linearization graph, the only cases where the results of those algorithms differ is where the presence of local precedence order forces C3 to make a different decision from $\mathcal{L}^*_{\text{LOOPS}}$.

Note that if $\mathcal{L}^*_{\text{LOOPS}}$ and Dylan produce the same result (which is inherently consistent with the extended precedence graph and local precedence order), it is the same as C3.

# Results

We have presented two new linearizations, Dylan and C3, that are monotonic and obey local precedence order; C3 is also consistent with the extended precedence graph. We've contrasted those linearizations with two existing linearizations, CLOS and $\mathcal{L}^*_{\text{LOOPS}}$, in terms of their structure and how they behave on existing class heterarchies.

The differences among the linearizations can be summarized by examining which kinds of class topologies they differ on. Dylan is the same as CLOS except where CLOS is non-monotonic. C3 is the same as Dylan except where Dylan is not consistent with the extended precedence graph, and the same as $\mathcal{L}^*_{\text{LOOPS}}$ except where that violates local precedence order.

In the abstract, C3 is the "best" of the linearizations we considered. However, C3 diverges from CLOS in more ways than the Dylan linearization does. Consistency with the extended precedence graph is not a necessary precondition for doing Dylan's sealing optimizations and C3 makes a significant number of classes from existing CLOS heterarchies inconsistent; thus, using it for Dylan would have been a more radical shift, late in the language design process, than using the monotonic variation on CLOS that was chosen.

# Acknowledgments

Roland Ducournau, Michel Habib, Marianne Huchard, and M.L. Mugnier formalized the characteristics of linearizations which underlie this work and offered valuable clarifications of their research.

Joseph Wilson implemented $\mathcal{L}^*_{\text{LOOPS}}$ in the Marlais interpreter and proposed changing the linearization used in Dylan. Glenn S. Burke studied the $\mathcal{L}^*_{\text{LOOPS}}$ algorithm when we first considered switching linearizations. Andrew L. M. Shalit participated in the discussions about changing the linearization. Judy Anderson, John Aspinall, Nicolas Graube, Kevin Males, Scott McKay, and Martin Simmons, all of Harlequin, provided us with the class heterarchies and related information for our survey of linearizations.

Susan Karp commented on and proofread drafts of this paper. James Nicholson assisted in the production of a camera-ready version.

Apple Computer, Inc., and Harlequin, Inc. and Ltd., supported the authors during the design of the Dylan language, when this work was undertaken.

# Appendix A: Implementation of the Dylan Linearization

```
define constant compute-class-linearization =
  method (c :: <class>) => (cpl :: <list>)
    local method merge-lists (reversed-partial-result :: <list>,
                              remaining-inputs :: <sequence>)

            if (every?(empty?, remaining-inputs))
              reverse!(reversed-partial-result)
            else
              // start of selection rule
              local method candidate (c :: <class>)
                      // returns c if it can go in the result now, otherwise false

                      local method head? (l :: <list>)
                              c == head(l)
                            end method head?,

                            method tail? (l :: <list>)
                              member?(c, tail(l))
                            end method tail?;
                      any?(head?, remaining-inputs)
                        & ~any?(tail?, remaining-inputs)
                        & c
                    end method candidate,

                    method candidate-direct-superclass (c :: <class>)
                      any?(candidate, direct-superclasses(c))
                    end method candidate-direct-superclass;

              let next = any?(candidate-direct-superclass,
                              reversed-partial-result);
              // end of selection rule

              if (next)
                local method remove-next (l :: <list>)
                        if (head(l) == next) tail(l) else l end
                      end method remove-next;
                merge-lists(pair(next, reversed-partial-result),
                            map(remove-next, remaining-inputs))
              else
                error("Inconsistent precedence graph");
              end if
            end if
          end method merge-lists;

    let c-direct-superclasses = direct-superclasses(c);
    local method cpl-list (c)
            as(<list>, all-superclasses(c))
          end method cpl-list;
    merge-lists(list(c),
                concatenate(map(cpl-list, c-direct-superclasses),
                            list(as(<list>, c-direct-superclasses))));

  end method; // compute-class-linearization
```

A few aspects of this program may need to be explained. The selection rule from above is enforced because, when choosing the next class, `any?` searches the reversed partially computed CPL in order and returns the first true value it encounters.

The function `all-superclasses` is defined to return the linearization for a class. It is called for each of the direct superclasses, and it in turn calls compute-class-linearization recursively, potentially storing the results to avoid recomputing the linearizations. There is no possibility of infinite recursion because circularities are prohibited in the inheritance graph; the recursive calls bottom out at `<object>`, the only class in Dylan which has no superclasses.

The function `direct-superclasses` returns the direct superclasses of its argument, in local precedence order.

For other details on Dylan, see the *Dylan Reference Manual*. [Shalit 96]

# Appendix B: Implementation of the C3 Linearization

The C3 linearization can be obtained by replacing the implementation of the selection rule from Dylan program above (that is, the definitions of the local methods `candidate` and `candidate-direct-superclass` and the binding of the local variable `next`) with the following:

```
local method candidate (c :: <class>)
        // returns c if it can go in the result now,
        // otherwise false

        local method tail? (l :: <list>)
                member?(c, tail(l))
              end method tail?;

        ~any?(tail?, remaining-inputs)
          & c
      end method candidate,

      method candidate-at-head (l :: <list>)
        ~empty?(l) & candidate(head(l))
      end candidate-at-head;

let next = any?(candidate-at-head, remaining-inputs);
```

Again, the property that `any?` returns the first matching result enforces the selection rule, because lists of remaining input sequences are maintained according to the local precedence order of the classes from which they are obtained.

For this to be a correct implementation of C3, the call to `all-superclasses` in the local function `cpl-list` must return the C3 linearization rather than the built-in Dylan linearization.

# References


[Apple 92] Apple Computer, Inc. *Dylan: an object-oriented dynamic language*. 1992.

[Apple 94] Apple Computer, Inc. *Dylan Interim Reference Manual*. 1994.

[Baker 91] Henry G. Baker. *CLOStrophobia: Its Etiology and Treatment*. ACM OOPS Messenger 2(4), October 1991.

[DHHM 92] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. *Monotonic Conflict Resolution Mechanisms for Inheritance*. OOPSLA '92 Proceedings, October 1992.

[DHHM 94] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. *Proposal for a Monotonic Multiple Inheritance Linearization*. OOPSLA '94 Proceedings, October 1994.

[ES 90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[KdRB 91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[MY 94] Scott McKay, William York, *et. al*. *Common Lisp Interface Manager (CLIM II) Specification*. 1994.

[Meyer 88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[Moon 86] David A. Moon. *Object-oriented Programming with Flavors*. OOPSLA '86 Proceedings, November 1986.

[SB 86] Mark Stefik and Daniel G. Bobrow. *Object-Oriented Programming: Themes and Variations*. AI Magazine 6(4), 1986.

[Shalit 96] Andrew L.M. Shalit, *Dylan Reference Manual*. Addison-Wesley, 1996. Available as `http://www.cambridge.apple.com/dylan/drm/drm-1.html` on the World Wide Web.

[Steele 90] Guy L. Steele, Jr. *Common Lisp: the Language (2nd edition)*. Digital Press, 1990.