Lazy Composition of Representations in Java

Rémi Douence, Xavier Lorca, Nicolas Loriant

Lazy Composition of Representations in Java

Rémi Douence[1], Xavier Lorca[2], and Nicolas Loriant[3]

[1] École des Mines de Nantes, INRIA, LINA UMR 6241, FR–44307 Nantes Cedex 3
[2] École des Mines de Nantes, LINA UMR 6241, FR–44307 Nantes Cedex 3
[3] INRIA, LaBRI, FR-33402 Talence Cedex

Abstract. The separation of concerns has been a core idiom of software engineering for decades. In general, software can be decomposed properly only according to a single concern, other concerns crosscut the prevailing one. This problem is well known as "the tyranny of the dominant decomposition". Similarly, at the programming level, the choice of a representation drives the implementation of the algorithms. This article explores an alternative approach with no dominant representation. Instead, each algorithm is developed in its "natural" representation and a representation is converted into another one only when it is required. To support this approach, we designed a laziness framework for Java, that performs partial conversions and dynamic optimizations while preserving the execution soundness. Performance evaluations over graph theory examples demonstrates this approach provides a practicable alternative to a naive one.

1 Introduction

The separation of concerns is a basis of software engineering. In general, each concern imposes its own decomposition to the whole software: once a primary concern has been chosen, the others become hard to modularize. This problem is well known as "the tyranny of the dominant decomposition". Similarly, at the programming level, the choice of a representation drives the implementation of algorithms. For instance, consider the case of the Cartesian or polar representations of a point. The first one fits with translations while the second with rotations, but neither efficiently implements the two operations simultaneously.

In this case, one possibility is to maintain several representations at the same time. This promotes reusability of algorithms specialized for a given representation. However, the synchronization of multiple representations has drawbacks: (1) a significant amount of computation can be required to maintain all representations up to date; (2) the space complexity increases with the number of co-existing representations; (3) the benefit in terms of expressiveness are annihilated by the complexity of the data structure manager. Another possibility is to maintain only one representation at a time and to convert the current representation into another one when an operation is simpler to program or more efficient in the other representation. This limits memory waste and avoids consistency problems. However it can be quite inefficient because of frequent conversions of the whole data structure.

In this article, we explore the last approach and we make it practical. Precisely, each algorithm is programmed in its "natural" representation. A wrapper is responsible to delegate incoming calls only to the "natural" representation and to convert a representation into another one. For the sake of efficiency, conversions are performed lazily and sequences of delayed conversions are dynamically simplified when possible. This article presents the following contributions. First, we show how to make practical the collaboration of several representations of the same data structure. Second, we provide a general framework for laziness and dynamic optimization in Java. It lets the programmer declaratively specify dependencies between computations and it makes safer the programming of subtle algorithms. These specifications can be used to automatically generate tedious infrastructure code.

The article is structured as follows. Section 2 details the motivations of the paper, through an example based on directed graph representations, and introduces the contributions of our solution. Section 3 presents an optimized scheme to implement collaborative representations. Section 4 introduces a new way to support laziness in an imperative language like Java. Laziness makes collaborative representations efficient in practice. Section 5 proposes an evaluation of the framework on a graph based example. Then, we review related work (Section 6) and conclude (Section 7).

2 Motivation and Contribution

The choice of a data structure is mainly driven by the complexity of its most frequent operations. Unfortunately, when a new operation has to be developed, such a representation may not fit with it, both theoretically (time and space complexity) and practically (architecture design). Then, an option is to maintain several appropriate representations. Nevertheless, such duplication is expensive to store and to maintain. In the following, Section 2.1 illustrates our purpose with an example based on directed graph representation, even if our work goes beyond the scope of graph representations. Next, Section 2.2 discusses the contribution.

2.1 Example: Direct Graph Based Representation

Most of the practical applications using graph based representations require various properties like connected components, path existence between two nodes, cost of a minimum spanning tree, etc. However, the time and space complexity to compute these properties differ from one graph representation to another, and no best one exists when several operations are used. The authors of [1] explain on pages 172 to 173 that several representations of graph *"are suitable for different sets of operations on graphs, and can be tuned further for maximum performance in any particular application. The edge sequence representation is good only in specialized situations. Adjacency matrices are good for rather dense graphs. Adjacency lists are good if the graph changes frequently."* Futhermore, they also observe that *"no graph representation is best for all purposes. How can*

```
1 desc(Graph G, Node s) : set of nodes
2   mark and enqueue node s;
3   while the queue is not empty do
4     dequeue a node x;
5     for y in next(G,x) do
6       if node y is not marked then
7         mark and enqueue node y;
8   return the set of marked nodes;
```

```
1 ance(Graph G, Node s) : set of nodes
2   mark and enqueue node s;
3   while the queue is not empty do
4     dequeue a node x;
5     for y in pred(G,x) do
6       if node y is not marked then
7         mark and enqueue node y;
8   return the set of marked nodes;
```

Fig. 1. Detecting the descendants (`desc(G, s)`) or the ancestors (`ance(G, s)`) of node s in a directed graph G independently of the graph representation.

on cope with the zoo of graph representation?". This observation is highlighted in applications that require graph representations to systematically compute several graph properties. This is the case for the implementation of global constraints [2]. However, the complexity of this application lead us to introduce a simpler example with only two properties and two representations.

Consider the representation of a n-nodes m-arcs directed graph as a set of adjacency lists over the successors of each node. This representation allows to easily express a search over the descendants of a given node by using the well-known *breadth first search* algorithm. Such an algorithm is depicted by Figure 1, its time complexity is related to the complexity of the `next(G, x)` operation (line 10) which provide the successors of node x in G. In the case of a representation of the directed graph by an adjacency lists over the successors, `next(G, x)` can be done in $O(1)$, and the `desc(G, s)` operation is achieved in $O(n + m)$ time. But, if the ancestors of a node have to be computed then, the `pred(G, x)` operation is achieved in $O(n)$ time and the `ance(G, s)` requires $O(n^2)$ time. The problem is similar when the directed graph is represented by an adjacency lists over the predecessors of each node. Then, the `next(G, x)` operation complexity increases to $O(n)$ time, and consequently `desc(G, s)` has an amortized complexity of $O(n^2)$.

Thus, the complexity of a given operation differs from one representation to another, and the representation choice affects the efficiency of the operations. However, practical applications are based on several operations (*e.g.,* compute both descendants and ancestors of nodes). For instance, our application uses several graph properties then, the procedures implementing the properties have

to coexist. Precisely, if the graph representations support the `reachable` procedure both for the descendants and the ancestors of node s then, each of the two implementations are efficient for one of the two procedures, but not for both.

2.2 Contributions

The duplication of the data structures must be avoided because of the related memory waste and the time complexity enhancement due to the data structure management. Thus, a collaborative representation framework has to address the following aspects: (1) the data structures related to the representations should not be duplicated (to avoid memory waste), (2) systematic total conversion from one representation to another should be avoided when the required procedure is in the wrong representation (to limit time complexity waste). In this context, providing such a solution is a tedious task. However, we propose a Java framework to conciliate these two points.

Our solution proposes a declarative language to specify the dependencies between the representations and the procedures. The dependencies are used by a general scheme that inserts the conversions when and where they are required. In order to avoid useless conversions from one representation to another, our framework introduces a lazy approach to implement a partial conversion strategy. Thus, each time a procedure is executed, only the required information is converted according to its specifications.

Most often in practical applications, the execution of some procedures can be delayed until another one requires its effective execution. Our framework addresses this point by managing lazy execution of the procedures. Thus, during the execution, a call to a "delayable" procedure results in only recording its corresponding closure in a generic trace. On the other hand, a "strict" procedure may require the execution of some delayed procedures in the trace to be safely executed. Finally, dynamic optimizations allow to limit the trace size during the execution. We provide an optimization that can detect outstanding patterns among the closures (*e.g.,* invertibility between closures, complementarity of a closure set), and operates simplifications (*e.g.,* closure removals, closure substitutions). In the following, we detail and evaluate these features of our framework.

3 Collaborative Representations

In this section we present our approach. It enables us to compose pieces of software (*i.e.,* several data representations and their corresponding algorithms) without modifying them. For the sake of simplicity, we use a toy running example but in Section 5 we evaluate a realistic example. Section 3.1 presents a simple (and inefficient) collaborative scheme. Section 3.2 and 3.3 introduce two optimizations of this scheme. Finally, Section 3.4 offers a language to define how several representations collaborate thanks to these optimizations.

3.1 Automatic Conversions

Let us consider a point in a two dimensional space. The coordinate of such a `Point` can be represented either using Cartesian coordinates or using polar coordinates. Some operations, for instance translations, are simpler to program and more efficient to execute in the Cartesian representation. On the other hand, other operations, for instance rotations, are simpler to program and more efficient to execute in the polar representation. We propose to use both representations. Each operation is programmed only in the more appropriate representation (*e.g.,* translation is implemented for Cartesian coordinates, and rotation is implemented for polar coordinates) and two conversions functions (from Cartesian to polar and from polar to Cartesian) are written once for all operations.

A wrapper is generated in order to make both representations collaborative. We arbitrarily choose the Cartesian representation to be the initial representation. When an operation implemented in Cartesian representation is called, the wrapper delegates the call to the Cartesian representation. When an operation implemented in the polar representation is called, the wrapper first converts the Cartesian coordinates into polar coordinates, then delegates the call to the polar representation, and finally converts back the polar coordinates into Cartesian coordinates. Figure 2 illustrates this behavior. In this figure, a box is a Java object and a line is a reference from an instance to another: a wrapper hides the two representations, `pc` (a `PointCart`) and `pp` (a `PointPol`). At each point in time, the representation colored gray is valid. Initially, the point is in its Cartesian representation. Upon a call to be performed in the second representation (*e.g.,* `rot(a)` at time k), the wrapper first converts `pc` into the polar representation by calling `toPointPol`, performs the operation in `pp` (time $k + 1$) and finally converts `pp` back into `pc` by calling `toPointCart`.

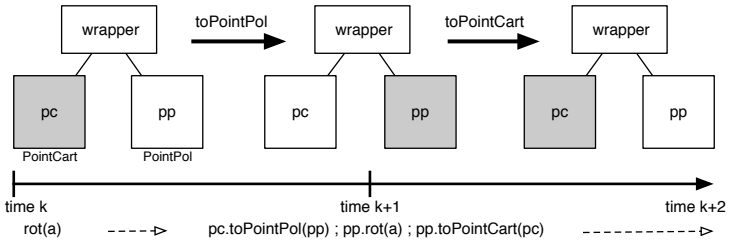

Fig. 2. Automatic conversion

3.2 Invertible Conversions

The scheme described in the previous section systematically introduces conversions when an operation in the non default representation is called. This ensures that when an operation is executed the corresponding representation is up to date. However, when we consider a sequence of operations, some conversions could be suppressed. For instance, to perform two successive rotations, there is no need to convert the coordinates (from polar to Cartesian and then from Cartesian to polar) between the two rotations. Such a scenario is depicted by Figure 3. A call to the `rot(a)` method leads to convert `pc` into `pp`. Then, the automatic conversion directly restores `pc`. When a new call to `rot(a)` is immediately performed, `pc` must be converted back into `pp`. So the sequence of inverse conversions `pp.toPointCart(pc)`; `pc.toPointPol(pp)` can be suppressed.

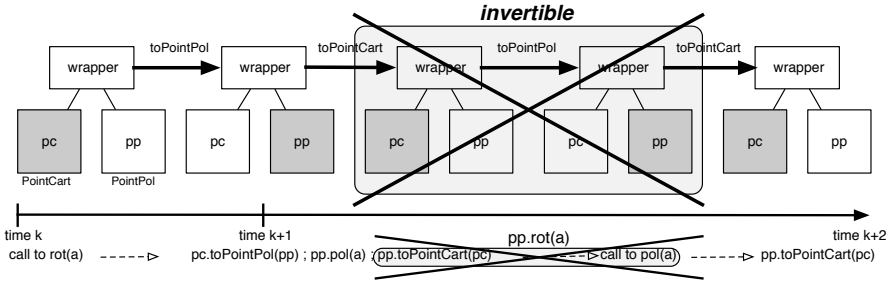

Fig. 3. Invertible conversion

Our goal is to get a general scheme that can be applied to any representation and conversion algorithm. To avoid these useless conversions, we propose the following scheme. First, we delay the conversions as late as possible. Second, we dynamically detect and suppress sequences of two inverse conversions when they occur. This technique is not as simple at it may appears. First, to delay the conversions, we have to detect when a delayed conversion must be forced (otherwise the computation would be incorrect). Second, calls to conversions can be interleaved with any other code in the program. This must be taken into account to detect sequences of two inverse conversions.

3.3 Incremental Conversions

If a representation is huge, its conversion into another one could be expensive. When a complete conversion is not required, then an optimization consists in a partial conversion of the representation. For instance, let us now consider colored points. A color can be represented either in the RGB model or in the

HSB model. We assume a colored point is represented either by a `PointCartRgb` or by a `PointPolHsb`. We can apply our collaborative technique to wrap these representations. However, note that the coordinate representation and the color representation are coupled: when a colored point coordinates are represented by Cartesian coordinates, then its color is represented by a RGB value. On the other hand, when a colored point coordinates are represented by polar coordinates, then its color is represented by a HSB value.[4] In the case of `PointCartRgb` and `PointPolHsb`, the coupling can introduce useless conversions. For instance, the sequence `rot(a1); setRed(r)` in Figure 4 requires converting a RGB color to a HSB color when `rot` is called (although `rot` does not use the color).

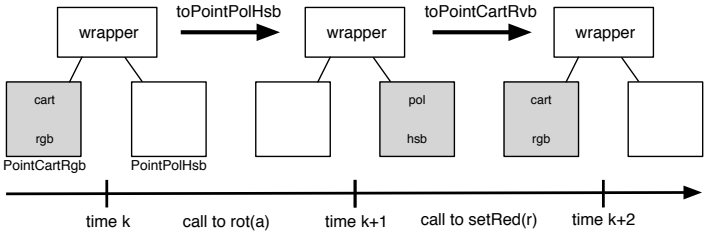

Fig. 4. Coupled representations

If the coordinate representation and the color representation were decoupled, only the required conversions would be performed. For instance, in Figure 5, when a rotation is performed, the coordinates are converted to the polar representation but the color remains a RGB value. Then, when `setRed` is called the color does not have to be converted.

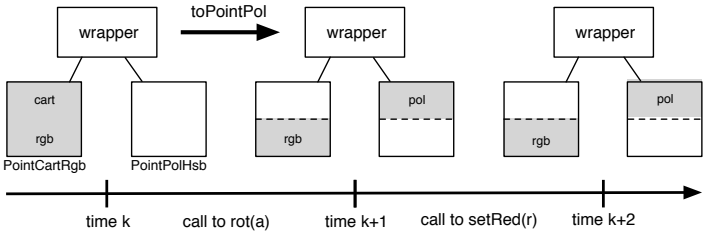

Fig. 5. Decoupled representations for incremental conversions

[4] This coupling can be easily avoided: a colored point could aggregate a coordinate and a color. In general, it is not possible to avoid such a coupling (see Section 5).

It is possible to decouple the coordinate representation and the color representation without modifying the program. Indeed, let us consider the following conversion method:

```
1void toPointPolHsb(PointCartRgb pcr) {
2  this.toPointPol(pcr);
3  this.toColorHsb(pcr);
4}
```

This method calls `toPointPol` to convert the coordinates to the polar representation, then it calls `toColorHsb` to convert the color to its HSB value. In order to decouple representations, we only have to make lazy and invertible the conversion submethods (*i.e.,* `toPointPol` and `toColorHsb`). This way, coordinates and color conversion becomes independent.

Our general scheme for representations collaboration relies on laziness and dynamic detection and suppression (*i.e.,* simplification) of sequences of inverse conversions. This requires developers to declare the dependencies between representations (and their associated methods). In the next section, we show how such dependencies can be specified by the user.

3.4 Collaboration Specification

We have defined a language to declare collaborative representations. This language provides what is required to generate the boilerplate code that implements a collaborative representation. Figure 6 presents the grammar of our language.

```
1wrapper class {
2  representation(class*);
3  conversion(method*);
4  inverse(method,method);*
5  partition {pi*};
6  lazy(method, {pi*}, {pi*});*
7  strict(method, {pi*}, {pi*});*
8}
```

Fig. 6. Specification language for automatic conversions

This language enables to specify a collaborative representation as follows. The keyword `wrapper` specifies the class identifier of the wrapper to be generated. The keyword `representation` lists the two (or more) representations to be wrapped. The keyword `conversion` lists the corresponding conversions. Note that, these are the top level conversion methods to be called by the wrapper (*e.g.,* `toPointPolHsb` and `toPointCartRgb` but not `toColorHsb` in the previous example). The keyword `inverse` lists pairs of inverse methods. The keyword `partition` introduces a set

of atoms that represent a partition of the representations. Each atom represents a part of a representation (*e.g.,* a cartesian colored point can be decomposed into an atom for the coordinate and an atom for the color, or a single atom can represent both). The finer the decomposition is, the more precise the closure dependencies are. The keyword `lazy` specifies that a method is lazy. The next two arguments specifies which atoms are read and written when the lazy method is finally forced (*i.e.,* executed). The keyword `strict` specifies that a method is strict. The next two arguments specifies which atoms are read and written when the strict method is executed. These atoms are used to compute dependencies between lazy and strict methods and to force evaluation of a delayed lazy method before a strict one reads or write the same part of the memory. Methods annotated as `strict` start by checking for delayed computation to be forced. Methods without `strict` or `lazy` annotation are plain Java (strict) methods that do not interfere with delayed computation.

The Figure 7 shows a specification of collaborative representations for colored points. We choose to partition the state in two atoms : `a1` (which represents `x`, `y`, `r`, `g` and `b`) and `a2` (which represents `rad`, `teta`, `h`, `s` and `b`). This makes the coordinates and the color representation coupled. Indeed, the accessor `getR` reads the memory part corresponding to `a2`, and the conversion method `toPointPolHsb` writes the memory part corresponding to `a2`. So, these two methods are dependent, and when the red component of the color is accessed, it first forces delayed calls to `toPointPolHsb` (that convert colors *and* coordinates).

```
1wrapper ColoredPoint {
2  representations(PointCartRgb, PointPolHsb);
3  conversions(toPointCartRgb, toPointPolHsb);
4  inverse(toPointPolHsb,toPointCartRgb);
5  partition {a1,a2};
6  lazy(toPointPolHsb, {a1}, {a1,a2});
7  lazy(toPointCartRgb, {a2}, {a1,a2});
8  strict(getX, {a1}, {}));
9  strict(setX, {}, {a1}));
10 strict(getR, {a2}, {}));
11 strict(setR, {}, {a2}));
12}
```

Fig. 7. Collaborative (coupled) representations of colored points (excerpt)

In order to decouple the coordinate representation and the color representation we only have to define a different specification based on a more precise partition. This specification would have four atoms: `a1_1` corresponding to the Cartesian coordinates, `a1_2` corresponding to the polar coordinates, `a1_2` corresponding to the RGB colors, and `a2_2` corresponding to the HSB colors. This

enables to specify, for instance, `setR`() modifies only the RGB representation (*i.e.,* `a1_2`) but not the Cartesian coordinates `a1_1`.

The finer the decomposition is, the more precise the closure dependencies are. However, the finer the decomposition is, the more expansive the closure dependencies computation are and the more memory they require. The choice of a decomposition in atoms should be guided by a tradeoff between precision and efficiency (in particular it is not worth spending more time to decide a closure evaluation can be delayed than evaluating it).

Here we summarize the steps required by our collaborative representations:

1. Programmers provides different implementations of a data structure.
2. The collaborative representations programmer implements conversion functions from a representation to another and specifies these methods are lazy as well as their side effects. He also specifies the strict methods (*e.g.,* accessors).
3. The wrapper class and the closures classes are automatically generated. They rely on our framework for safe laziness in Java.
4. Users transparently calls the resulting library. The execution of lazy methods is delayed. When a strict method is called, our framework forces the evaluation of delayed methods required to compute the result of the strict one. As the application progresses the data get scattered in the different representations.

4 A Framework for Safe Laziness in Java

In order to support our specification language introduced in the previous section, we offer a framework that deals with the most difficult and error-prone tasks of collaborative representations: laziness in an imperative language. Hence, users can focus on writing those representations. Our framework features both lazy evaluation for the Java programming language and dynamic optimizations.

4.1 Laziness in Imperative Language

While laziness is a natural feature of functional programming languages, the implementation of lazy evaluation in imperative languages such as Java is complex:

- In order to delay a method call, the method, its receiver and its arguments must be stored in data structure (*i.e.,* a closure) so that the method can actually be called later.
- The execution of a lazy method can be delayed until its result is required by a strict computation. A lazy framework is able to detect when a delayed method result is required in order to force its evaluation. In an imperative language, a method can have side effects and modify variables (*e.g.,* assign `x` with 0). When a strict method has to read the value of the variable `x`, the closure of the delayed computation (that assigns `x` with 0) must be forced. Our framework provides means to specify the side effects of methods and to automatically decide when a delayed computation must be forced.

– In an imperative language, two method calls are not in general commutative (*e.g.,* "increment x, then square it" is not equivalent to "square x, then increment it"). So, our framework uses a trace in order to maintain the order of the closures. Dependencies between them also take into account the trace: sometimes, evaluating a closure requires to evaluate another before.
– When two computations are inverse one of another (*e.g.,* "increment x" and "decrement x") and both are delayed, the two closures can be suppressed without being evaluated and without changing the result of the program.

Our framework requires the user to specify closure side effects. These specifications could be automatically generated by abstract interpretation. This would still require the user to specify a partition of the representation (*i.e.,* the corresponding abstract domain). However, such an analysis can be quite expansive in general. The side effect specifications can be used to compute closures dependencies. Next, we describe this aspect of our framework.

4.2 Closure Dependencies

In our framework, delayed computations are represented by subclasses of the class `Closure`. Closure classes can be automatically generated from their side effect specifications shown in the previous section. Closures must provide an evaluation method `eval` that actually calls the delayed method. Each closure must also provide two methods `reads` and `writes`. These methods are used to compute the dependencies between closures. They return a set of atoms implemented, in our experiment, by a `BitSet`. Let us considers two closures `c1` and `c2`, with `c1` is **older** than `c2`. When `c2` is to be evaluated, `c1` must be evaluated first if either the evaluation of `c2` reads memory written by the evaluation of `c1`, or the evaluation of `c2` writes memory read by the evaluation of `c1`, or the evaluation of `c2` writes memory written by the evaluation of `c1`.

When a method call is delayed, the corresponding closure is created and pushed on a stack `trace`. (When a closure is pushed, if the inverse of the closure is already in the stack, both closures are removed. Due to space constraints, we do not detail here this dynamic simplification phase). When a strict method is called, the method `force` in Figure 8 is called to compute dependencies and force evaluation of some closures in the trace before the strict method is evaluated.

The method `force` visits the trace from the most recent closures to the oldest ones. In order to keep track of its position in the trace this method uses a second stack `ecart` and transfers closures from one stack to another, one at a time. If the current closure is required to be evaluated (line 5) it is tagged (line 6) but it is not evaluated immediately because older closures may require to be evaluated before, so the sides effects of the current closure are taken into account (lines 7-8). The memory written by this closure `c` is added to the memory written by all the closures to be evaluated `s.writes`. The memory read by this closure `c` is added to the memory read by all the closures to be evaluated `s.reads` and the memory written by `c` is removed from `s.reads`. Then, the visit continues. When the beginning of the trace is reached, the second half of the method `force`

(lines 12-15) visits the closures from the oldest one to the most recent one. If a closure is tagged then it is evaluated, otherwise it is pushed back in the trace.

```
1 void force(AbstractState s) {
2   int i = this.ecart.size();
3   while (!this.trace.isEmpty()) {
4     Closure c = this.trace.pop();
5     if (c.isRequiredBy(s)) {
6       c.eval = true;
7       s.reads.diff(c.getWrites()).union(c.getReads());
8       s.writes.union(c.getWrites());
9     }
10    this.ecart.push(c);
11  }
12  while (this.ecart.size() != i) {
13    Closure c = this.ecart.pop();
14    if (c.eval) c.eval(); else this.trace.push(c);
15  }
16 }
```

Fig. 8. Compute closure dependencies and force their evaluation

5 Evaluations

We now evaluate our approach over a realistic example related to the graph theory in the context of global constraints implementation. Global constraints represent invaluable modeling tools for Constraint Programming (CP). They heavily rely on graph theory through properties like the connected components, the strongly connected components or the transitive closure [2]. Nevertheless, a global constraint has to combine efficiently several properties at a time. Our framework offers a promising way to efficiently and safely combines several procedures related to the graph properties.

As explained in Section 2.1 when a graph is represented by adjacency lists over the successors of each node (*i.e.,* `GraphSucc`), it is easy and efficient to express a search over the descendants of a given node. On the other hand, when the graph is represented by adjacency lists over the predecessors of each node (*i.e.,* `GraphPred`), the same search over the descendants becomes more complex. Symmetrically, it is easy and efficient to express a search over the ancestors in `GraphPred`, but it is more complex for searching the descendants. In this case, we can apply our collaborative representations scheme.

5.1 Conversion

Conversions between `GraphSucc` and `GraphPred` are based on two methods `toNodePred(n)` and `toNodeSucc(n)` which behave as depicted in Figure 9. The `toNodePred` method converts a column of the `GraphSucc` into a row of `GraphPred`. The notion of column is a logical view in `GraphSucc` (a list of lists of successor nodes) which has no corresponding direct accessor, does not exist in `GraphSucc`, so the method iterates for each node s, if the list of successors of s in `GraphSucc` contains `n`, it removes it from that list (which frees its memory so that nodes only exist in one representation at a time) and it adds `s` to the ancestors of `n` in `GraphPred`. The reverse conversion (from `GraphPred` to `GraphSucc`) is symmetric, hence invertible, that is, the back and forth conversions of a given node leaves both representations unchanged). In order to convert all the nodes of a graph, the method `toGraphSucc` (resp. `toGraphPred`) calls `toNodeSucc` (resp. `toNodePred`) for each node of the graph.

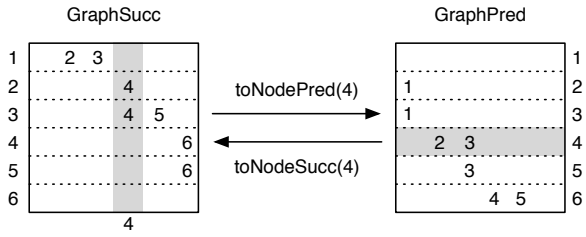

Fig. 9. Partial conversion between lines and columns

This can be specified as in Figure 10. A class `Graph` can be automatically generated to wrap an instance of `GraphSucc` and an instance of `GraphPred` (line 1 and 2). A representation is transformed into the other one by calling either `toGraphPred` or `toGraphSucc` (line 3). We partition the `toGraphSucc` representation into its columns c_i (this is a conceptual notion that is not implemented by the structure) and the `toGraphSucc` representation into its lines ℓ_j. The methods `toNodePred` and `toNodeSucc` are inverse one of another (line 5). Both are lazy, and we specify their side effect. For instance, `toNodePred(n)` reads c_n, the n^{th} column in the successor representation, it erases (*i.e.,* writes) it, and writes ℓ_n, the n^{th} line in the predecessor representation. The accessors are strict. We also specify their side effects. For instance, `getNodeSucc` reads a line (*i.e.,* all columns) in the successor representation (line 8 and 9).

```
wrapper Graph {
  representation(GraphSucc,GraphPred);
  conversion(toGraphPred,toGraphSucc);
  partition {c_1 ... c_n, ℓ_1 ... ℓ_n};
  inverse(gp.toNodeSucc(gs,n),gs.toNodePred(gp,n));
  lazy(gs.toNodePred(gp,n), {c_n}, {c_n, ℓ_n});
  lazy(gp.toNodeSucc(gs,n), {ℓ_n}, {ℓ_n, c_n});
  strict(gp.getNodePred(n), {ℓ_n}, {}));
  strict(gs.getNodeSucc(n), {c_all}, {}));
}
```

Fig. 10. Collaborative Representations `GraphSucc` and `GraphPred` (excerpt)

5.2 Conversion Strategies

Let us consider a graph with n nodes. The conversion method `toGrapPred` calls `toNodePred` for each node of the graph. If `toGrapPred` is defined as a **for** loop, its execution creates n closures (one for each call to `toNodePred`). Closure instantiation, simplification (*e.g.,* with inverse rules) and dependencies computation can be expensive. We describe two alternative definitions of `toGraphPred`.

A first alternative is for the user of our framework to define a recursive and lazy version of `toGraphPred`. In this case, a call `toGraphPred(i)` would create only two closures corresponding to `toNodePred(i)` and `toGraphPred(i+1)`. This recursive scheme can be much more efficient when nodes with low index are converted: the strict accessor call `getNodePred(m)` will require `m` closures of `toGraphPred` to be evaluated until the right closure `toNodePred(m)` is created. Unfortunately, when the last node of a graph is converted, a closure is still created for each node.

A better alternative is for the user of the framework to define a dichotomic recursive and lazy version of `toGraphPred`. When `toGraphPred(start,end)` is called it creates two closures corresponding to `toGraphPred(start,mid)` and `toGraphPred(mid+1,end)`, where `start` is the first node to convert, `end` is the last node to convert, and `mid` is the middle of `start` and `end`. This recursive scheme builds at most $2log(n)$ closures when a node has to be converted. This is the conversion scheme we use in the evaluation below.

5.3 Benchmarks

We have evaluated our approach with a scenario related to graph theory (a short example is provided by Section 2.1). We compare the execution time, on 43 200 randomly generated graphs, of the standard single representation approach (*either* list of successors `GraphSucc` *or* list of predecessors `GraphPred`) and of our dichotomic lazy conversion approach. Each graph instance is benchmarked on 8 kinds of scenarios depicted in Figure 11. It depicts the patterns used for these experiments: (`ance`{3}`desc`){6} means the method `ance` is called three times successively, then the method `desc` is called once, this is repeated 6 times with

randomly picked arguments. All patterns contains 24 method calls, but they vary in the frequency of the alternations between `ance` and `desc` calls. In this experiment, our graphs are structured as lattices and each scenario is evaluated on 5 kinds of arguments grouped by "zones": zone 1 means the arguments are chosen among all the nodes of the lattice, zone 2 means the arguments are chosen inside the bottom half of the lattice and so on. The bigger the zone, the more nodes will be converted to perform the `ance` or `desc` computation. In each case, we measured the execution time for the lists of successors representation, the lists of predecessors representation and of our approach.

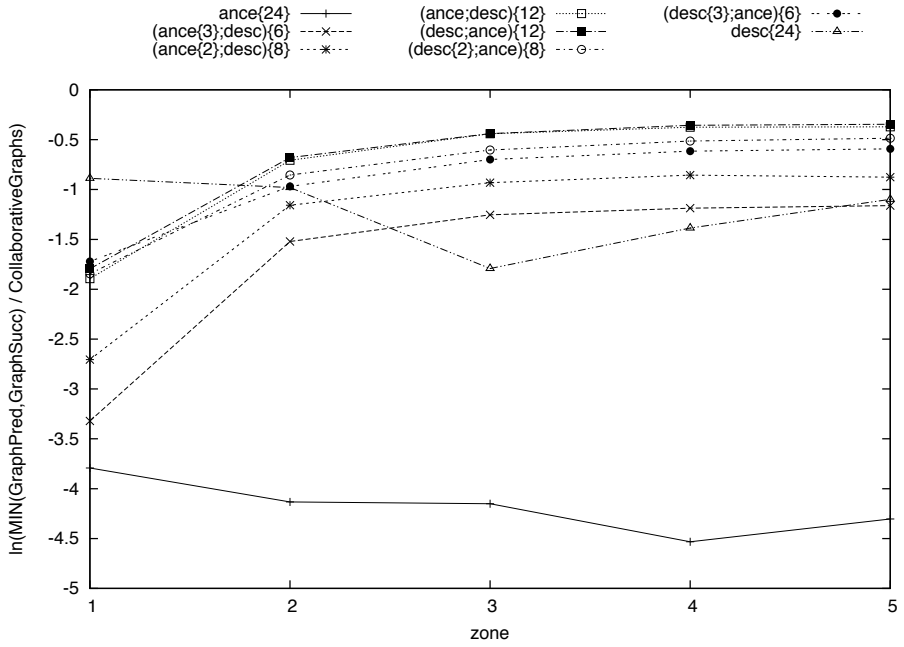

Fig. 11. Best of `GraphSucc` and `GraphPred` *versus* Collaborative Representations

For scenarios with the same number of calls of each operation (*e.g.,* (`ance`; `desc`){12}), successor and predecessor representations exhibit comparable execution time, but for non symmetric pattern (*e.g.,* `ance`{24}) one representation may be up to 150 times slower than the other. Figure 11 compares our approach with the best of the two standard representations on each scenario. It shows that for all of the patterns, our approach is less efficient than the best of the two standard representations. However, note that when fewer nodes must be converted (zones 2 to 5), our approach gets close to the best representation. Moreover, measures shows that compared to the worst of the two standard representations for each scenario, our lazy approach has comparable performance but is at best up to

90 times faster. Because the number of conversions is more likely to be higher when zone equals one, the difference of behavior of partial conversions observed, around that point may be explained by the high number of closures corresponding to delayed conversion. This makes it harder for the framework to simplify invertible conversions.

A dedicated solution to this graph scenario would outperform our approach. However, our proposal does not impose a higher space complexity and is generally more efficient than both naive approaches. Moreover, it offers a general solution: it is less error prone (*i.e.,* as long as side effects specifications are correct, no closure evaluation is forgotten), it reuses "natural" representations and it could be extended to manage more than two representations. Thus, our approach provides a good software engineering/performance trade-off, demonstrating the applicability of collaborative representations.

6 Related Work

The definition of aspect in the ASOD.Net wiki glossary [3] is based on the notion of tyranny of the dominant decomposition attributed to IBM's T. J. Watson Research center's research on Morphogenic Software. Multi-dimensional Separation of Concerns [4] shows how the software artifacts corresponding to different concerns (*a.k.a.* hyperslices) can be merged to generate a full application. This general and abstract model is instantiated by Subject-Oriented Programming [5] where hyperslices are pieces of code (*e.g.,* partial class hierarchies). Aspect-Oriented Programming [6] is quite similar but it is asymmetric: it considers the structure of a base program and it provides pointcut languages to specify where another code crosscuts the base program and the corresponding pieces (*i.e.,* advices) should be woven. All these static approaches enable to compose several pieces of code (*i.e.,* a base program and several advices). However, they focus on the code structure, and the woven program executes the base code and the advices "at the same time". This do not help to compose different data representations (and their corresponding algorithms) and to dynamically change the representation at run-time.

Early work of Don Batory on data structure compilers and data structure tailoring and optimization provide languages to describe and compose data structure representation and their corresponding algorithms. In particular, Section 3.2 of [7] introduces the notion of robust (*i.e.,* interchangeable) algorithms *versus* non robust algorithms. It should be further studied if and how our technique could be used in this context in order to interchange non robust algorithms.

Our work shares similarities with views in database management systems (DBMS) [8]. A view defines a table that is computed rather than stored. An updatable view requires the DBMS to infer a reverse mapping. However, in our framework, partial conversions dynamically scatter the data in the collaborative representations, while views duplicate the data of the underlying tables. Views have also be proposed for specification languages such as Z, but their collaboration is defined by an invariant, which is declarative but does not help to build

a concrete solution [9]. Wadler [10] has studied how to have views and pattern matching at the same time. This work is one of the main source of inspiration of our approach although it is restricted to functional languages and it focuses on pattern matching. Our dynamic simplifications of inverse closures is also inspired by a short cut to deforestation [11] that proposes such a dynamic scheme, to complement the static one, to simplify the functions `fold` and `build`.

Hughes [12] argues laziness is a powerful tool to compose programs (*i.e.,* functions). Laziness has an intuitive semantics in pure functional languages but it is tricky when sides-effects are introduced. So, lazy extensions of imperative languages are limited. For instance, LazyJ [13] extends Java's type system with lazy types. The programmer is responsible for introducing coercions that force delayed evaluations, but there are no specification or propagation of dependencies. Recent work have introduced dependencies into imperative languages. They have been used to automatically incrementalize an invariant checks when a data structure is modified [14] or to update a self-adjusting computation [15]. In both cases, laziness is not involved and caching prevents from re-evaluating a computation. It should be studied how these techniques could complement ours.

7 Conclusion and Future Work

This article proposes a new approach to compose data structure representations and their corresponding algorithms. It introduces a general scheme to replace the dominant representation by multiple collaborative representations. Each algorithm is programmed in the best representation (*i.e.,* that fits), so that the corresponding code is simpler or more efficient than in the other representations. The conversions from one representation to another are automatically inserted by a wrapper (so the original code does not have to be modified). A representation is partially converted by need: only the subparts of the representation required by a computation are converted. This technique relies on lazy evaluation and dynamic optimizations. We provide a framework to support it in Java. It let the user declaratively specify dependencies and simplification rules. The framework supports safer development (most of the code can be automatically generated and the dependencies are systematically propagated). We have validated our approach with actual experiments for different graph representations. Our evaluations shows that our approach is practical. It provides a good tradeoff between software engineering issues and performances.

Our proposal offers many opportunities for future work. We list here a few of them. First, our approach already requires the user to specify the memory read and written by methods. Such a piece of information could be used to dynamically check when a method call can be skipped (*i.e.,* when the memory `m` reads and writes has not been written since the previous call to `m`). Second, our approach relies currently on dynamic analysis only, but it could benefit from static analyses too. For instance, the automatic derivation of `reads` and `writes` from the source code could be studied. It could also be interesting to statically decide when two closures never interacts in order to have independent traces

instead of a single one (in our running example we could have a trace by graph instance). This shorter traces would make the dependencies propagation less expensive. In some cases the user could specify a conversion function and its inverse could be automatically generated [16, 17]. Finally, other applications of our framework for safe laziness in Java should be explored.

References

1. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures. Springer (2008)
2. Beldiceanu, N., Flener, P., Lorca, X.: Combining tree partitioning, precedence, and incomparability constraints. Constraints **13**(4) (2008)
3. Misc: Aosd.net wiki glossary
4. Tarr, P., Ossher, H., Harrison, W., Sutton., S.M.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE '99: Proceedings of the 21st International Conference on Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 107–119
5. Harrison, W., Ossher, H.: Subject-oriented programming: a critique of pure objects. In: OOPSLA '93: Proceedings of the eighth annual conference on Object-Oriented Programming Systems, Languages, and Applications, New York, NY, USA, ACM (1993) 411–428
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP, SpringerVerlag (1997)
7. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engineering and Methodology **1**(4) (1992) 355–398
8. Chamberlin, D.D., Boyce, R.F.: Structured query language (1974)
9. Jackson, D.: Structuring Z specifications with views. ACM Transactions on Software Engineering and Methodology **4**(4) (1995) 365–389
10. Wadler, P.: Views: a way for pattern matching to cohabit with data abstraction. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, New York, NY, USA, ACM (1987) 307–313
11. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: FPCA '93: Proceedings of the conference on Functional Programming Languages and Computer Architecture, New York, NY, USA, ACM (1993) 223–232
12. Hughes, J.: Why Functional Programming Matters. Computer Journal **32**(2) (1989) 98–107
13. Warth, A.: LazyJ: Seamless lazy evaluation in Java. In: FOOL/WOOD. (2007)
14. Shankar, A., Bodík, R.: DITTO: automatic incrementalization of data structure invariant checks (in Java). In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation, New York, NY, USA, ACM (2007) 310–319
15. Acar, U.A., Ahmed, A., Blume, M.: Imperative self-adjusting computation. SIGPLAN Not. **43**(1) (2008) 309–322
16. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM (2007) 47–58
17. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. SIGPLAN Not. **40**(1) (2005) 233–246