author's preprint, to appear in the proceedings of ONWARD! 2015

Just-in-Time Data Structures

Mattias De Wael
Software Languages Lab, VUB (Belgium)
madewael@vub.ac.be

Stefan Marr
RMoD, INRIA Lille Nord Europe (France)
stefan.marr@inria.fr

Joeri De Koster
Software Languages Lab, VUB (Belgium)
jdekoste@vub.ac.be

Jennifer B. Sartor
VUB and Ghent University (Belgium)
jsartor@vub.ac.be

Wolfgang De Meuter
Software Languages Lab, VUB (Belgium)
wdmeuter@vub.ac.be

Abstract

Today, software engineering practices focus on finding the single "right" data representation for a program. The "right" data representation, however, might not exist: changing the representation of an object during program execution can be better in terms of performance. To this end we introduce Just-in-Time Data Structures, which enable representation changes at runtime, based on declarative input from a performance expert programmer. Just-in-Time Data Structures are an attempt to shift the focus from finding the "right" data structure to finding the "right" sequence of data representations. We present JitDS, a programming language to develop such Just-in-Time Data Structures. Further, we show two example programs that benefit from changing the representation at runtime.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords data structures, algorithms, dynamic reclassification, performance

1. Introduction

Choosing the "right" combination of a data representation and an algorithm is important for performance. Books, courses, and research papers on algorithms and data structures typically discuss the one in function of the other [7]. This makes choosing the right data representation-algorithm combination relatively easy. For instance, in the context of a linear indexable data structure, *e. g.*, `List` in Java, an algorithm that heavily relies on insertions and deletions of elements will likely benefit from a pointer-based implementation of `List`, *e. g.*, `LinkedList` in Java. On the other hand, an algorithm that heavily relies on random indexed accesses in a `List`, will benefit from an array-based implementation, *e. g.*, `ArrayList` in Java. Larger software systems, however, rarely consist of a single algorithm but rather of a complex interweaving of multiple algorithms. Finding the right data representation for a set of algorithms becomes cumbersome due to the increasing number of, possibly conflicting, requirements of these algorithms.

To facilitate the implementation of a more efficient data structure in later stages of a software development cycle, it has become best practice to program against a data interface. In this text we refer to *data interface* as the set of operations which define an abstract data type [18]. We will call a concrete implementation of a data interface a *data representation*. Together, *data interface* and *data representation* form a *classic data structure*. Above we argue that finding the "right" data representation for a data interface is much less trivial when the number of algorithms using the data interface increases because the number of requirements increases. Imagine a program that first builds a list of sorted elements, in order to heavily query the list later. Such a program consists of two phases that prefer the `LinkedList` representation and the `ArrayList` representation respectively. Choosing one in favor of the other is not trivial. Moreover, we show in the next section that a program that relies on a single data representation can be less performant compared to a program that changes data representations at runtime.

Today, data representation changes are implemented in an ad hoc way because a systematic approach does not exist. In this paper we introduce *Just-in-Time Data Structures*, a language construct where a set of data representations for a single data interface is augmented with declarative input from a performance expert programmer. This declarative

input defines which representation is to be used in which circumstances. We show that it is possible for a program with Just-in-Time Data Structures to obtain better performance than the same program that uses one single representation for the data. Moreover, Just-in-Time Data Structures allow developers to disentangle "application logic" and "performance engineering tasks".

The contributions of this paper are: 1. a *taxonomy of data representation changes*; 2. the introduction of *Just-in-Time Data Structures*; 3. the introduction of *homomorphic reclassification*, an implementation technique that we use to compile a Just-in-Time Data Structure into Java.

The remainder of this text is organized as follows: In Section 2 we elaborate on a use case where changing the representation of a data structure at runtime improves performance. In Section 3 we introduce a taxonomy of how data representation selection and data representation changes can be realized. Section 4 introduces Just-in-Time Data Structures, which generalizes the idea of changing representation at runtime, and introduces the new language constructs we propose to define such a data structure. Section 5 discusses how to compile the definition of a Just-in-Time Data Structure into Java. In Section 6 we give two examples of how a developer can implement Just-in-Time Data Structures to obtain better performance. Both the related work on data structure selection, as well as the work related to our compiler implementation techniques are discussed in Section 7. Finally, Sections 8 and 9 conclude this text and present ongoing and future work.

2. Motivating Example

In this section, we first introduce the data interface `Matrix` and two possible data representations. Then, we introduce the classic algorithm to multiply two matrices and study the effect of the chosen data representation on performance. The matrix multiplication is an example of a computation that operates on two objects with the same data interface, but which accesses the two objects with a different data access pattern. The example shows that changing the representation of the data objects to match the access patterns at runtime improves performance.

The Matrix Data Interface. The running example throughout this text is built around the mathematical concept of a two-dimensional matrix, *i. e.*, a conceptually rectangular array of numeric values. Here, we define the data interface of a `Matrix` to be: a constructor that creates a `rows` by `cols` matrix of zeroes; an accessor `get` and a mutator `set` which, based on a `row` and a `col` parameter, respectively returns or sets a value in the matrix. Listing 1 shows this data interface as a Java abstract class definition. Note that we use `abstract class` instead of `interface` in this code example. The reason thereof is twofold. First, we want to show in this example that the constructor to create an initial matrix always takes two arguments, which is not expressible in a Java `interface`.

Listing 1: Data interface for `Matrix`.

```
1  public abstract class Matrix {
2    // create a Matrix of rows by cols
3    Matrix(int rows, int cols) { ... }
4
5    // accessor to read the number of rows
6    int getRows() { ... }
7
8    // accessor to read the number of columns
9    int getCols() { ... }
10
11   // accessor to read the value of a cell
12   double get(int row, int col) { ... }
13
14   // mutator to set the value of a cell
15   void set(int row, int col, double val) { ... }
16 }
```

Listing 2: Classic matrix-matrix multiplication algorithm of two $N \times N$ matrices.

```
1 Matrix mul(Matrix A, Matrix B) {
2   Matrix C = new Matrix(N, N);
3
4   for (int i=0 ; i<N ; i++) {
5    for (int j=0 ; j<N ; j++) {
6     for (int k=0 ; k<N ; k++) {
7      temp =  C.get(i, j) +
8              (A.get(i, k) * B.get(k, j))
9      C.set( i, j, temp) }}}
```

Second, as exemplified by the first reason, the Java *interface* and our *data interface* are not identical concepts. In Java, an interface is a language construct that defines (potentially) only a part of a class's type. The *data interface* is the set of characterizing operations, *i. e.*, the complete structural type.

Two Matrix Data Representations. Let us now consider two similar data representations for the data interface defined above. Both representations store the elements of the conceptually two-dimensional data structure in a one-dimensional array. One representation, `RowMajorMatrix`, stores elements of the same row next to each other. The second data representation, `ColMajorMatrix`, stores elements of the same column next to each other.

The Matrix-Matrix Multiplication Algorithm. The classic matrix multiplication algorithm takes two matrices as input parameters (*i. e.*, `A` and `B` in Listing 2) and has a third matrix as output (*i. e.*, `C` in Listing 2).[1] For each of the elements of `C`, the dot-product of the corresponding row of `A` with the corresponding column of `B` is computed. This dot product is computed by the inner-most loop, *i. e.*, lines 6–9, which accesses `A` in row-major order and `B` in column-major order.

[1] For the ease of implementation of the example code we only consider *square* matrices of size $N \times N$.

Effect of Implementation on Performance. A simple experiment shows that the choice of data representations of the matrices `A` and `B` has a significant effect on the execution time. We executed the `mul` function with all combinations of data representations for both input matrices, while we kept the data representation of the matrix `C`, the output variable, fixed in the `RowMajorMatrix` representation. The execution times for all combinations are shown in Table 1.[2]

Data Representation		Execution Time
`A`	`B`	
Row Major Order	Col Major Order	6.33 s
Col Major Order	Col Major Order	10.29 s
Row Major Order	Row Major Order	13.72 s
Col Major Order	Row Major Order	25.83 s

Table 1: The execution time of multiplying two 1250×1250 matrices depends on the chosen data representation.

The execution time is significantly lower when the data access pattern (computation) matches the data representation, *i. e.*, `RowMajorMatrix`×`ColMajorMatrix`. Conversely, when the data access pattern and data representation conflict for *both* matrices, the execution time is significantly higher.

As a second experiment we computed the product of two row-major matrices where the representation of the second matrix `B` was changed to column-major order just before the actual multiplication. The overall execution time, thus including the cost of a transposition, is only 6.42s. Paying the extra cost of changing the representation proves to be more efficient than keeping the representation fixed for both matrices.

This matrix multiplication example shows the existence of computations that operate on a single data interface but where choosing a single data representation results in suboptimal performance. Conversely, paying the cost of a representation change at runtime results in better performance compared to relying on a single fixed data representation.

3. Taxonomy of Changing and Selecting Data Representation

Just-in-Time Data Structures, as we introduce in Section 4, change the data representation to match the computation in order to achieve better performance. The design space for techniques to change and select data representations for programs is vast. Based on the examined work (cf. Section 7), we developed a taxonomy according to which existing efforts can be categorized. Below we introduce this taxonomy and explain the axes based on the matrix multiplication example. In Section 7 we present and position the related work along the relevant axes for each approach.

[2] We gathered these numbers from a C++ implementation, compiled with $-O3$ where we multiplied two matrices of 1250×1250 elements. The resulting binary was executed on a 2.6 GHz processor with 256 KiB of L2 cache. While the presented numbers are the result of a single run only, we observed that they are representative for all runs. The code for this small experiment is available on our website.
`http://soft.vub.ac.be/~madewael/jitds/`

Listing 3: Internal transformation logic.

```
1 Matrix A = new RowMajorMatrix(rs, cs);
2 Matrix B = new TransposableMatrix(rs, cs);
3
4 // internal to TransposableMatrix
5 B.enforceColMajorOrder();
6 Matrix C = mul(A, B);
```

Listing 4: External transformation logic.

```
1 Matrix A = new RowMajorMatrix(rs, cs);
2 Matrix B = new RowMajorMatrix(rs, cs);
3
4 // external to RowMajorMatrix
5 B = new ColMajorMatrix(B);
6 Matrix C = mul(A, B);
```

Internal or External Transformation Logic. A data structure does not automagically know **how** to change its representation. Clearly, there has to be some code fragment responsible for the actual conversion from one representation to the other. The code fragment that expresses this transition is called the *transformation logic*.

We observe that the transformation logic can either be a part of the definition of a data structure (encapsulated) or not. Data structures with *internal transformation logic* encapsulate the logic that describes the representation change, within their implementation. Otherwise, we refer to them as data structures with *external transformation logic*.

By a call to `enforceColMajorOrder`, on line 5 in Listing 3, we rely on the encapsulated functionality of `TransposableMatrix` to change its internal representation. The `RowMajorMatrix` does not provide this functionality but relies on the constructor of `ColMajorMatrix` (line 5 in Listing 4) to handle the change in representation. Note that here, the internal transformation logic example keeps the object's identity intact, *e. g.*, the reference `B` in Listing 3 points to the same object before and after the call to `enforceColMajorOrder`, whereas the reference `B` in Listing 4 points to a new — `ColMajor` — object on line 5.

Internal or External Change Incentive. A data structure does not automagically know **when** to change its representation. We call the code fragment that is responsible for initiating a representation change the *representation change incentive* code. We differentiate between approaches where the representation change incentive code is encapsulated in the data structure's definition and those where it is not. Representation changes with *internal* incentive are initiated by the data structures itself, *i. e.*, as part of their implementation.

Listing 5: Choosing a data representation.

```
1 // Static Selection of Representation
2 Matrix a = new RowMajorMatrix(rs, cs);
3 Matrix b = new ColMajorMatrix(rs, cs);
4
5 // Dynamic Selection of Representation
6 Matrix c = MatrixFactory.createMatrix( ... );
```

Listing 6: Choosing a representation offline.

```
1 Matrix A = new RowMajorMatrix(rs, cs);
2 Matrix B = new ColMajorMatrix(rs, cs);
3
4
5 Matrix C = mul(A, B);
```

Conversely, when a new representation is imposed on the data structure from the outside the data structure, we say the representation change incentive is *external*.

Listings 3 and 4 are both examples of external representation incentive code, because it is the code *using* the matrix `B` that is responsible for initiating the change in representation. Prototypical examples of internal incentives can be found in the class of *self-adapting* data structures. An AVL tree, for instance, rebalances itself upon insertion.

Table 2 further clarifies the difference between *representation change incentive* and *representation transformation logic*. The example used in the code fragments deals with a list with sorted data to which elements can be added.

Online or Offline. Listings 3 and 4 are two examples of data representation changes that happen *online*, during the execution of the program (*i. e.*, at runtime). In the more classic approach, *e. g.*, on line 2 of Listing 6, the representation of a data structure does not change at runtime, but is chosen during the development of an application, *i. e.*, static data representation selection. Alternatively, the data representation selection is delayed until runtime (*e. g.*, Listing 5) but the representation remains fixed during the execution of the program, *i. e.*, dynamic data representation selection. We call data representation selection an *offline* approach.

Developer or Environment. At first sight, the choice of data representation is the responsibility of the *developer*, as is illustrated in the examples in Listings 3, 4 and 6. However, there also exist *environments* (*e. g.*, compilers, interpreters, or dynamic optimization systems) that change the physical representation of data behind the scenes. The developer using these techniques is thus not necessarily aware of them. For instance, the Javascript V8 engine does not guarantee that an `array` uses contiguous memory, but chooses the representation it sees fit (*e. g.*, sparse representation). Thus, the *Developer–Environment*-dimension stipulates the level of abstraction on which the choice of data representation takes place.

Listing 7: Maintaining multiple representations.

```
1 public class AmbiguousMatrix
2 implements TransposableMatrix {
3
4   RowMajorMatrix rm;
5   ColMajorMatrix cm;
6   boolean rowActive = true;
7   ...
8   public void enforceRowMajorOrder() {
9     rowActive=true;
10  }
11
12  public void enforceColMajorOrder() {
13    rowActive=false;
14  }
15
16  public void set(int r, int c, int v) {
17    rm.set(r,c,v);
18    cm.set(r,c,v);
19  }
20
21  public int  get(int r, int c) {
22    rowActive?rm.get(r,c):cm.get(r,c);
23  }
24 }
```

Gradual or Instant. When it is possible to unambiguously determine the current representation of a data structure at any point during the execution we say the representation change is *instant*. The matrices in Listing 4 are either in row-major representation or in col-major representation, but never in both nor in a hybrid form. Alternatively, data structures can also be implemented to (partially) maintain multiple representations simultaneously. For such data structures it is not possible to pinpoint the current representation, as it is *gradually* changing between different representations. For instance, consider `AmbiguousMatrix`, implemented as in Listing 7. While an instance of `AmbiguousMatrix` has a "principal representation" (cf. `rowActive`, a boolean that represents the active state) it uses for access, is also maintains the "other representation" during mutation.

Dedicated or General. Representation changing techniques can be deployed in two possible ways. First, we see *dedicated* techniques that are tailored towards a well defined set of use-cases, which can be deployed as-is, off the shelf. These dedicated approaches include — but are not limited to — libraries, runtimes, and self-adapting data structures. Other techniques, however, are more *general*. These techniques provide a set of concepts and insights, but leave the concrete implementation to the developer. An example of such a general concept is "transposing" data as is shown in the concrete example of matrix multiplication. On the other hand, this technique is general enough to be applied in other contexts as well. The "array-of-structs" versus "struct-of-array" discussion, for instance, applies the same technique to more heterogeneous data.

Transformation Logic

Change Incentive: Internal

Transformation Logic: Internal

```
1 // User Code:
2 myList.add(x);
3
4 // Representation Code:
5 public void add(Object x) {
6   this.data.add(x);
7   this.sort();
8 }
```

Transformation Logic: External

```
1 // User Code:
2 myList.add(x);
3
4 // Representation Code:
5 public void add(Object x) {
6   this.data.add(x);
7   Collections.sort(this);
8 }
```

Change Incentive: External

Transformation Logic: Internal

```
1 // User Code:
2 myList.add(x);
3 myList.sort();
4
5 // Representation Code:
6 public void add(Object x) {
7   this.data.add(x);
8 }
```

Transformation Logic: External

```
1 // User Code:
2 myList.add(x);
3 Collections.sort(myList);
4
5 // Representation Code:
6 public void add(Object x) {
7   this.data.add(x);
8 }
```

Table 2: A list with internal/external transition logic and internal/external incentive code to change representation.

We identified that data representation selection strategies can be categorized according to the following axes: *Internal or External Transition Logic*, *Internal or External Change Incentive*, *Online or Offline*, *Developer or Environment*, *Gradual or Instant*, and *Dedicated or General*. In Section 7 we taxonomize the work related to our Just-in-Time Data Structures according to these axes. We observe that, besides ad-hoc implementations, there does not exist a general approach that gives the developer the power to easily change the chosen data representations online. Our Just-in-Time Data Structures fill this hole. Just-in-Time Data Structures is a *general* approach that provides *developers* with the infrastructure to create data structures that can swap representation *online* using *internal transition logic*. Furthermore, our approach supports both *internal* and *external* representation change incentive code. Currently, we only support *instant* representation changes, but we foresee including *gradual* representation changes as future work.

4. Just-in-Time Data Structures

The idea of separating *data interface* from *data representation* is almost as old as computer science itself. The rationale of programming against an interface as opposed to programming with the representation directly is mainly driven by software engineering advantages such as modularity, maintainability, and evolvability. It can also be motivated by performance. For instance, in software engineering it is a tried and true approach to first implement a trivial but working representation for a data structure. Only if the initial implementation proves to be a performance bottleneck, the programmer should consider to optimize the initial implementation (*i. e.*, avoid premature optimizations).

With the advent of object-technology it became possible to have *multiple* data representations for a single data interface available at runtime. In class-based object-oriented languages, for instance, this is realized by implementing multiple classes that extend the same base class. The data representation is usually chosen statically (lines 2-3 in Listing 5) and occasionally chosen dynamically (line 6 in Listing 5). In either case, even with the aforementioned object technology, the representation chosen at *allocation* time remains *fixed* during the remainder of the data object's lifetime.

Adhering to one representation during a object's lifetime is sufficient for data structures that are used for a single role in a single algorithm. In Section 2 we showed an example of a program where relying on a single representation hampers performance. In theory, one could implement a representation that performs well in "all" situations. In practice however, such implementation are hard to find and hard the develop. Alternatively, one could implement an ad-hoc representation change. The problem with ad-hoc representation changes is that they usually do not preserve object identity. The other references to the original object still point to the original representation and multiple versions of the same mutable data are kept together in memory.

We propose Just-in-Time (JIT) Data Structures, a data structure with the intrinsic property of changing its underlying representation. To facilitate the development of Just-in-Time Data Structures, we have implemented an extension of Java called *JitDS-Java*. We use this language to informally introduce the concepts needed to implement Just-in-Time

Listing 8: The class `Matrix` combines two representations.

```
1 class Matrix
2 combines RowMajorMatrix, ColMajorMatrix  {
3
4   RowMajorMatrix to ColMajorMatrix {
5     target(source.getCols(),
6            source.getRows(),
7            source.getDataAsArray());
8     target.transpose();
9   }
10
11  ColMajorMatrix to RowMajorMatrix {
12    target(source.getCols(),
13           source.getRows(),
14           source.getDataAsArray());
15    target.transpose();
16  }
17
18  swaprule Matrix Utils.mul(Matrix a, Matrix b) {
19    if ((a.getRows()*a.getCols()) > LARGE)
20      swap a to RowMajorMatrix;
21    if ((b.getRows()*b.getCols()) > LARGE)
22      swap b to ColMajorMatrix;
23    proceed;
24  }
25 }
```

Listing 9: The class `File` combines three representations.

```
1 class File
2 combines OpenFile, ClosedFile, LockedFile  {
3   OpenFile   to ClosedFile as close { ... }
4   ClosedFile to OpenFile   as open  { ... }
5   ClosedFile to LockedFile as lock  { ... }
6 }
```

Data Structures. In Section 5 we explain how we transpile JitDS-Java to Java.

Combining Representations. Implementing a data structure in Java is realized by declaring a new class, *e. g.*, `RowMajorMatrix` or `ColMajorMatrix`. Implementing a JIT Data Structure in JitDS-Java is realized by declaring a new *JIT class* which *combines* multiple representations (lines 1 and 2 in Listing 8). The representations themselves are implemented as traditional Java classes. An instance of a JIT class can be the target of a *swap* statement (*e. g.*, lines 20 and 22 in Listing 8), which forces the data structure to adhere to the instructed representation. An instance of a JIT class implements the union of the methods implemented by its representing classes. Assume for now that all representing classes implement the same set of methods. Thus, an instance of the JIT class `Matrix` is able to respond to the methods `int getRows()`, `int getCols()`, `int get(int, int)`, and `void set(int, int, int)` (cf., Listing 1).

Listing 9 introduces a second example which models a `File` that can be in one of three *states*: open, closed, or locked (forever closed). To this end `File` combines three representation classes (lines 1 and 2).

A swap statement potentially causes the JIT Data Structure to change its representation, which is a non-trivial transformation. To express such transformations[3] we introduce a new kind of class member in the body of a Just-in-Time class: the *transition function*. A transition function defines the transition from a *source representation* to a *target representation*. For instance, the transition function on lines 4–9 in Listing 8 transforms a `RowMajorMatrix` into a `ColMajorMatrix`. The body of the transition function (between curly braces) shows much resemblance with the body of a *parameterless constructor*. Within the body of a transition function two new keywords can be used: `target` and `source`. These denote the object in the new representation and the object in the old representation respectively. Outside the body of a transition function these keywords have no meaning. The intentional semantics of a transition function are as follows: 1. before the execution of the body the original object is assigned to `source`, 2. the first statement in the body invokes a constructor of the *target representation* and assigns the resulting object to `target`, 3. during the execution of the body both `target` and `source` exist as separate objects, 4. after the execution of the body the original object replaces the object denoted by `target`. Optionally, a transition function can be named, as shown in Listing 9. The named transition function can be invoked by calling it as a parameterless method, *e. g.*, `myFile.close()`.

In the definition of the JIT class `File` (Listing 9), there are three transition functions defined. From these, it is possible to construct the finite state graph shown in Figure 1, which we call the *transition graph*. The transition graph of `File` shows that it is not possible to transition from a locked file to any other representation. When such a swap is issued at runtime, an `UnsupportedSwapException` is thrown.

An obvious critique to this approach is a potential combinatorial explosion of the number of transition functions that need to be implemented as the number of representations grows. We argue, however, that in practice this will not be an issue because of the following reasons:

- First, when we look at existing libraries, the number of different representations for a single interface is relatively small. In Java for instance there are only three implementations for the `List` interface (*i. e.*, `ArrayList`, `LinkedList`, and `Vector`). Then, the number of transition functions stays within acceptable bounds.
- Second, we conjecture that most data interfaces can be enriched such that it is possible to implement a transition function that is generic enough to transition from any representation to any other representation. An example of such a *general transition function* for the `Matrix` example is shown in Listing 10. Such a general transition function can replace all other *specialized transition functions*. Of

[3] In the work on object evolution these are called *evolvers* [6]. In [1] they are described as coercion procedures

Listing 10: A transition function that is generic enough to transition a `Matrix` from any representation to any other representation.

```
1 Matrix to Matrix {
2   target(source.getRows(), source.getCols());
3   for ( int r=0 ; r<source.getRows() ; r++ ) {
4     for ( int c=0 ; c<source.getCols() ; c++ ) {
5       target.set(r, c, source.get(r,c));
6     }
7   }
8 }
```

course, from the performance perspective, specialized transition functions are likely to be preferred. An example of a specialized transition function in the matrix example is the `transpose` function which expresses the transition from a `RowMajorMatrix` to a `ColMajorMatrix`, and vice versa. These specialized transition functions are shown in Listing 8 (lines 4–9 and 11–16).

- Third, the set of available specialized transition functions can be used transitively. In the file example we can transition from an open file to a locked file by combining two transitions, *i. e.*, `myFile.close(); myFile.lock()`. Again, a specialized and direct transition function might be preferred in terms of performance.
- A final argument to counter the "transition function explosion" is that some transitions between two representations are unlikely or even impossible to occur. Implementing a specialized transition function in such a case, serves no practical purpose. For instance, the `LockedFile` representation does not allow transitions to any other representation.

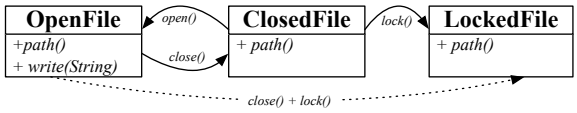


Figure 1: The states of a `File`: Open, Closed, Locked.

What we have now is a data structure that, when instructed, is able to transition between representations, given its transition graph. The remainder of this section introduces *swap rules*, the language constructs to induce a representation change; and *specialized swaps*, *i. e.*, implicit representation changes imposed by JitDS-Java.

4.1 Swap Rules

Swap rules are the constructs in our language that allow the developer of JIT Data Structures to express when a representation swap is needed. In general, a swap rule expresses what events are important to *observe* and how to *react* to them accordingly. Based on the observed usage of a JIT Data Structure a reaction can be formulated in the form of a transition from one representation into another. We identify two levels of granularity on which to make these observations. The coarsest level of granularity we consider is the level of *computation*. Observing an invocation of the matrix multiplication method `mul`, for instance, is a *computation level observation*. Based on the expert knowledge about the affinity of `mul` for the row-major × col-major representations it is beneficial for performance to impose a representation change on the arguments of `mul`. Alternatively, a more fine-grained observation is on the level of a data structure's operations, *i. e.*, invocations of the methods. In the matrix example these are for instance `get(row, col)` or `set(row, col, val)`. The observation that `set` is mostly called with `val==0` makes a sparse matrix representation a viable candidate to swap to. Note that interface invocation observations imply a reasoning from a perspective *internal* to the JIT Data Structure. The two levels of granularity coincide with the *external* versus *internal* representation change incentives introduced in Section 3. Consequently, we introduce *external swap rules* that express representation changes on the *computation* level; and we introduce *internal swap rules* that express representation changes on the interface level.

External Swap Rules are swap rules that invoke a representation change on the level of computations. In an object-oriented language, methods are a straightforward boundary of computation. Therefore we restrict ourselves to method invocations as join points[4] at which to introduce representation changes. To capture the invocation of a single method, the header of an external swap rule looks like the header of a Java method definition (*i. e.*, list of modifiers, a return type, a name, and a list of formal parameters), prepended with the keyword `swaprule` (Listing 8, line 18). Note that the name used in the swap rule should be fully qualified to capture the method in the intended class. The body of an external swap rule consists of three parts: a set of statements, a proceed statement, and again a set of statements (Listing 8, lines 18–24). All statements in the body have access to the arguments of the method invocation and can perform any necessary computation to decide whether or not to invoke a representation swap. The proceed statement represents the actual invocation of the advised method call. The relation between external swap rules and AOP is discussed in Section 5.

On lines 18–24 of Listing 8 an external swap rule is defined that captures all the invocations of the method `mul` (matrix multiplication) defined in some class named `Utils`. When the arguments, both instances of the JIT class `Matrix`, are "large enough" to benefit from a representation that is aligned with the computation, a representation swap is issued. An invocation of `doCommute` (also in `Utils`, see Listing 11), then implies potentially four representation changes.

[4] Other researchers explicitly study language constructs to express more fine grained join points [16].

Listing 11: Swapping a swappable Data Structure

```
1 boolean doCommute(Matrix a, Matrix b) {
2     return mul(a, b).equals(mul(b, a));
3 }
```

Listing 12: Internal swap rule to `RowMajorMatrix` based on the number of non-zero elements.

```
1 swaprule SparseMatrix {
2   int size = getRows()*getCols();
3   if ( getNonZeroCount() > size*0.25 ) {
4     this to RowMajorMatrix;
5   }
6 }
```

Internal Swap rules are swap rules that describe for which "state" of a JIT Data Structure it becomes opportune to issue a representation change. Conceptually, these checks are performed continuously during the execution of a program. For performance reasons, however, continuously performing these checks might not be optimal. Finding the right balance between responsiveness and performance has not yet been investigated, but is discussed in Section 8. Listing 12 shows an example of an internal swap rule. On the first line the swap rule reveals for which representations the rule is applicable, here `SparseMatrix`. The body of the internal swap rule states that the data structure should swap to the `RowMajorMatrix` representation when less than 25% of the values in the matrix are zero.

4.2 History Based and Learned Reactions

It is possible to implement more complex and expressive swap rules than the examples presented above. First, these "simple" swap rules are based on readily observable state, *e. g.*, invocation of the `mul` method, current representation and size. Second, these swap rules express a change into a developer-defined representation. Orthogonal to the choice of implementing internal or external swap rules, we also allow observations based on the *history* of the data structure's usage and we allow *learning*, to find the best target representation of a swap rule.

History Based Reactions. Because swapping comes at a certain cost, it is not always economical to change the representation eagerly. For instance, swapping from `RowMajorMatrix` to `SparseMatrix` on the first call to `set` with a zero value would be counterproductive. In such cases, it is more interesting to react to a *pattern of observations*, that was seen over time. Some representation changes should therefore be based on a *history* of observations.

To facilitate the bookkeeping of history information, external swap rules have access to statically defined member fields in the JIT class. Internal swap rules have access to instance member fields of a JIT class. Internal swap rules can also make use of a special kind of history information, in the form of *Invocation Count Expressions*.

Listing 13: Internal swap rule to `SparseMatrix` based on estimated sparsity.

```
1 #set(int row, int col, int val);
2
3 #zeroSet as set(int row, int col, int val) {
4   count-if (val == 0);
5 }
6
7 #nonZeroSet as set(int row, int col, int val) {
8   count-if (val != 0);
9 }
10
11 swaprule RowMajorMatrix {
12   if ( (#set > FREQUENT_SET) &&
13        (#zeroSet > #nonZeroSet*#nonZeroSet) ) {
14     this to SparseMatrix;
15   }
16 }
```

Invocation Count Expressions. To make counting the number of invocations of member methods easier, we introduce *invocation count expressions*. The need for similar information to decide whether or not to issue a representation change is also identified by Shacham et al. [19], *i. e.*, "opCounts", and by Xu [23], *i. e.*, "swap conditions". In its simplest form, an invocation count expression is a hash-symbol followed by a method-name and a list of formal parameters between braces (line 1 in Listing 13). Such an expression evaluates to the number of invocations of the matching method, here `set`. Adding a body to an invocation count expression allows for more complex statistics, *i. e.*, only those invocations for which at least one `count-if`-statement evaluates to `true` are counted. Optionally, an invocation count expression can be given a more revealing name. An example of invocation count expressions with names and bodies is given in Listing 13 (lines 3–5 and 7–9). The value of an invocation count expression can be used in the body of an internal swap rule by referring to it by its name preceded by a hashtag, *e. g.*, the ratio of `zeroSet` and `nonZeroSet` is used to estimate the "sparsity" of a matrix (line 13) and potentially invoke a representation change.

Learned Reactions. All example swap rules presented hitherto express transitions to a representation *defined by the developer*. Alternatively, the "right" representation to swap to can be *learned*, using machine learning techniques. In Section 6.1, we use epsilon-greedy Q-learning to find the best representation for `A` and `B` in `mul`. Assume `qLearner4Mul` to be an object that implements this learning algorithm. The swap rule in Listing 14 first asks `qLearner4Mul` for the "best" representations; then the multiplication is performed and its execution time is measured; finally, `qLearner4Mul` is informed about the time needed to execute `mul` and "learns"

Listing 14: Learned Reaction to the occurrence of a call to `mul`

```
1 static QLearner qLearner4Mul = new QLearner(2);
2
3 swaprule Matrix Utils.mul(Matrix A, Matrix B) {
4   A to qLearner4Mul.getRepresentation(0);
5   B to qLearner4Mul.getRepresentation(1);
6   long begin = System.currentTime();
7   proceed;
8   long end = System.currentTime();
9   qLearner4Mul.minimize( (end-begin) ,
10    representationOf(A), representationOf(B));
11  return C;
12 }
```

which representations for `A` and `B` it should suggest the next time `mul` is called.

Note that Listing 14 reveals two properties of JitDS-Java that where not yet discussed. On the one hand, it shows how external swap rules can access static members of a JIT class. On the other hand, it shows how representation types are first class values in JitDS-Java. They can be the result of a function call (line 4 and 5) and they can be passed as arguments to a function call (lines 9 and 10). In Section 5 we also show that they can be assigned to a variable, and we show how this is implemented in the compiler.

4.3 Specialized Swaps

If we relax the assumption that all representing classes implement the same set of methods, then we can partition the set of methods into the *core* methods, *i. e.*, those methods implemented by all representing classes, and the *specialized* methods, *i. e.*, those methods implemented by one representing class. To execute a specialized method, a data structure has to adhere to the correct representation. The *Specialized Swap* is the implicit representation change imposed by our language to allow the execution of such a specialized method.

For instance, consider the class `SparseMatrix`, a third representation for our JIT class `Matrix`. Besides the methods as defined in Listing 1, this class also provides a method `Iterator nonZeroElementsIterator()` which is not part of the core of the `Matrix` data type. When this method is invoked, as on line 3 in Listing 15, the matrix `m` is implicitly converted into a `SparseMatrix`.

Listing 15: Counting the number of non-zero elements implies an implicit representation swap.

```
1 int numberOfNonZeroElements(Matrix m) {
2   int count = 0;
3   Iterator it = m.nonZeroElementsIterator();
4   while(it.hasNext()) {
5     it.next();
6     count++;
7   }
8   return count;
9 }
```

5. Compiling Just-in-Time Data Structures into Java

We now describe the transpiler which we implemented to translate the specification of a JIT class written in JitDS-Java into Java.

Just-in-Time Class Definition. The definition of a JIT class (*e. g.*, `Matrix`) is compiled directly into a simple class definition with the same name and package. Then, we compute for each of the representation classes (*e. g.*, `RowMajorMatrix`, `ColMajorMatrix`, and `SparseMatrix`) the set of `public`, non-`static` methods. We add a (static) interface (here `Matrix.Interface`) which contains the union of the above described set of methods augmented with a `void swap(Representation)` operation and a `Representation representationOf()` operation. Then we add a non-static member class definition to the "JIT class" for each of the representations, which we call the *local representation class*. These local representation classes extend a single representation class and implement the newly defined interface. Finally, the JIT class holds a reference to an instance of the `Interface` type in a field member called `instance`. All methods of the `Interface` type are implemented by the JIT class by forwarding the call to the `instance`. For those methods implemented by the `instance`'s super class (representation class) no new implementation needs to be provided, rather Java's polymorphism takes care of those. The specialized methods need special care of the compiler. These are implemented as a call to `swap`, which changes the representation, followed by re-invocation of the intended method. Now the new `instance` does adhere to the correct representation and, by construction, knows how to respond to the invocation.

Homomorphic Reclassification. The `swap` method is implemented in each of the local representation classes as a switch statement: if `swap` is called using the current representation, nothing happens; if `swap` is called using a representation for which a transition function is defined, the instance is set to a new object with the corresponding representation using the transition function. Finally, if `swap` is called using a representation for which no matching transition function exists, an `IllegalSwapOperationException` is thrown. This functionality of changing the representation of an object at runtime while retaining its identity, is known as *dynamic object reclassification* [11]. Because the JIT Data Structure never loses properties (*i. e.*, all operations of the data interface have to be defined), the `swap` is a restricted form of dynamic reclassification called *monotonic reclassification* [6]. Moreover, a JIT Data Structure never gains properties either, which makes the `swap` an even more restricted form of dynamic reclassification which we will call *homomorphic reclassification*. Our implementation of ho-

momorphic reclassification explained above resembles the *Inheritance–Evolution* technique from [6] and is effectively a more elaborate variant of the *bridge pattern* [13].

External Swap Rules. Both the syntax and intended behavior of external swap rules have the look-and-feel of aspect-oriented programming. More precisely, our external swap rules provide a static quantification of when to execute a certain representation change [12]. It will therefore come as no surprise that our compiler translates external swap rules directly into an "around advice" with operates on a "execute pointcut" expressed in AspectJ [17]. In future work, however, we want to do the code weaving ourselves to gain more fine grained control.

Internal Swap Rules. An internal swap rule provides a dynamic quantification of when to execute a certain representation change [12]. Opposed to our implementation of external swaprules, we implemented the weaving of internal swaprules ourselves. All bodies of the internal swap rules that apply to a representation class are combined into a private method in this representation class. This method is invoked "regularly". That is, our current compiler inserts a call to this method before each core method invocation. Below and in Section 8 we hint at reducing this overhead by relying on thorough (static) analysis of the code (*e. g.*, counters being changed, combining multiple swap rules into one, or with runtime sampling).

Invocation Count Expressions. For each of the declared invocation count expressions, a private `int` member is added to the representation class. The body of each method captured by an invocation count expression is prepended with a conditional increment instruction of this member. References to these counters in the body of an internal swap rule are simply converted to the correct name.

Discussion. The compiler in its current form allows expressing all of our new constructs and compiles them to Java (*i. e.*, JIT classes, transition functions, named swap rules, internal swap rules, external swap rules, and invocation count expressions). What is currently missing is (static) analysis to aid the compiler in reducing the amount of overhead introduced in the code, and to check for anomalies in the combination of representation classes, *e. g.*, colliding method signatures. The source files needed to build the compiler are available on our website.[5]

6. Evaluation: JITMatrix and JITList

In this section we present two Just-in-Time Data Structures, that we implemented to serve as example programs. Both examples are chosen such that they exemplify the complexity of different kinds swap rules presented in Section 4. The first example is an extension of the running example of this text: the `Matrix` and its multiplication. In the `Matrix`-example we show the difference between *learned* and *developer defined* transitions. In the second example we present a Just-in-Time List which is used to find prime numbers using the traditional sieve. The `List`-example shows how to use the fine grained observations at the *interface invocation* level in combination with a *history based reaction*. The benchmarks were executed on an Ubuntu 14.04.2 server, kernel version 3.13.0-44-generic, with four AMD Opteron 6376 processors at 2.3 GHz with 64 GB of memory with NUMA (non-uniform memory access) properties. All experiments are run 30 times and the aggregated results are presented in the graphs (and text).

In the code fragments — and in the accompanying text — a lot of seemly random numbers are used, *i. e.*, problem's input sizes and the "magic numbers" in the representation swaprules. The input sizes are chosen in function of the presentation of this text, *e. g.*, to show the difference between small and large input sizes. The numbers in the swaprules, however, are the result of the tedious work performed by a performance expert. Because performance engineering is difficult [9], we consider the separation of application logic and representation change logic as a key benefit of our Just-in-Time Data Structures.

6.1 Matrices and Matrix Multiplication

Listing 16: Raising a matrix `A` to the $\mathtt{n}^{th}$ power.

```
1 public static Matrix pow(Matrix A, int n) {
2   Matrix C = makeIdentityMatrix( A.getCols() );
3   for (int i=0 ; i<n ; i++) {
4     C = mul(A, C);
5   }
6   return C;
7 }
```

As already shown, the data access pattern for the matrix multiplication algorithm, `mul(A, B)`, prefers `A` to be stored in row-major order and `B` to be stored in col-major order for better performance. Our benchmark program implements a power function `pow(Matrix m,int n)`, which raises the matrix `m` to the $\mathtt{n}^{th}$ power (Listing 16). Thus, `pow` iteratively calls `mul`. We measure the execution time of raising a 512×512 matrix to the 16^{th} power. In our experiment we compare three approaches: 1. We consider `Matrix` to be a JIT class without any swap rules and thus without any representation changes, 2. The `Matrix` from (1) with the swap rule from Listing 8 to enforce a multiplication of `RowMajor x ColMajor`, and 3. The `Matrix` from (1) with the swap rule from Listing 14, which implements the epsilon-greedy Q-learning algorithm [21] to *learn* the best representation based on execution time.[6]

Figure 2 shows a box-plot of the execution times of raising a 512×512 matrix to the 16^{th} power. The graph summarizes the executions times of 30 runs. As expected, the versions

[5] `http://soft.vub.ac.be/~madewael/jitds`

[6] The machine learning technique used here is the basic epsilon-greedy Q-learning algorithm, and serves as a prototypical implementation. Hitherto, no further research was conducted in the area of self-learning swaprules

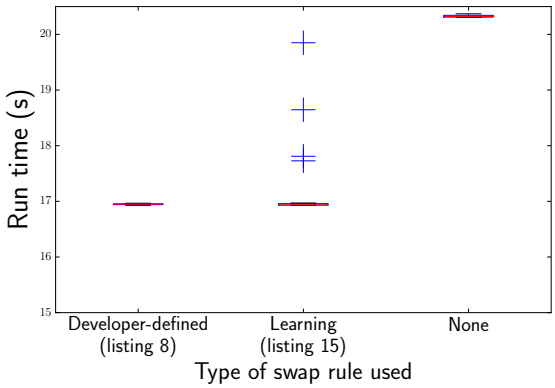


Figure 2: Raising a 512×512 matrix to the 16^{th} power.

with swap rules outperform the version where all matrices have a single representation. The outliers in the learned version are those runs where the machine learning algorithm is trying to find an optimum. As already shown in Section 2, changing the representation of the matrices yields better performance, and thus also when running the code generated by our compiler. Also the version using the learning swap rule performs clearly better than than the program without representation changes.

6.2 Search for Primes in a List

In Java the `List` interface comes with three standard implementations, *i. e.*, `LinkedList`, `ArrayList`, and `Vector`. The implementations `ArrayList` and `Vector` are roughly equivalent up to synchronization and are based on an array of `Objects`. The `LinkedList` implementation on the other hand is based on the helper-class `Entry` which holds a reference to a previous and next `Entry` and a reference to a value. Consequently, the `LinkedList` is pointer based without guarantees on the memory layout.

Intuitively, the `LinkedList` is better for dynamic lists, or those with frequent insertions and deletions of elements. The `ArrayList` is expected to outperform in random access, *e. g.*, getting and setting elements. This intuitive characterization is summarized in Table 3. As shown by other researchers (*e. g.*, [3, 23]), making a fixed choice of implementation before (or at) start-up time can be suboptimal. For instance, in programs that exhibit phased behavior [20], *e. g.*, phases of frequent inserts followed by phases of frequent selects, and vice versa. Sometimes this phase shift is lexically (*e. g.*, one specific line of code) observable in the program's code. In the following example we present a program where the phase shift is not lexically observable: in *one of the iterations* of the while loop (Listing 17, lines 19–23).

	get/set *(random position)*	**add/remove** *(current position)*
ArrayList	$O(1)$	$O(n)$
LinkedList	$O(n)$	$O(1)$

Table 3: Intuitive performance characteristics of `List`-representations in Java.

Listing 17: Implementation of the sieve of Eratosthenes.

```
1  JITList primes = new JITList();
2
3  for (int i=N ; i>=2 ; i--) {
4    primes.add(0, i);
5  }
6
7  for (int idx=0 ; idx<primes.size() ; idx++) {
8    /* Get idx'th prime */
9    int prime = primes.get(idx);
10
11   /* Advance Iterator */
12   primes.startIterator();
13   while ( primes.hasNext() &&
14           (primes.next()<=prime) ) {
15     /* do nothing */
16   }
17
18   /* Remove Multiples */
19   while ( primes.hasNext() ) {
20     if ( (primes.next()%prime) == 0) {
21       primes.remove();
22     }
23   }
24 }
```

The "sieve of Eratosthenes" is an algorithm to find all prime numbers smaller than N. The algorithm starts with a sequence of integers from 2 till N. Then, it iteratively filters all multiples of all found primes. An implementation in Java is shown in Listing 17 and was designed to play off the `ArrayList` implementation of `List` against the `LinkedList` implementation. On line 4, for instance, an element is added to the front of the list which is an increasingly expensive operation for the `ArrayList`.[7] Then, at line 7 the iterative sieving starts. Each iteration consists of a *random access* (line 9), *iterating* through the list to the wanted position (line 12–16), and finally, iterating further while potentially *removing elements* (lines 19–23).

Listing 18: A set of invocation count expressions used in the `JitList`.

```
1 #iter   as next();
2 #del    as remove();
3 #insert as add(int i, int v);
4 #get    as get(int i);
```

[7] The implementation of `add(int,Object)` in `ArrayList` requires a call to `System.arraycopy` and potentially a second call if the underlying array is too small. Conversely, `add(int,Object)` in `LinkedList` simply creates a new `Entry` and adjusts a single reference.

Listing 19: Swap rule from `ArrayList` to `LinkedList`

```
1 swaprule ArrayList {
2   if ( (size()>1000) && (#insert>10*#get)) {
3     this to LinkedList;
4   }
5 }
```

Listing 20: Swap rule from `LinkedList` to `ArrayList`

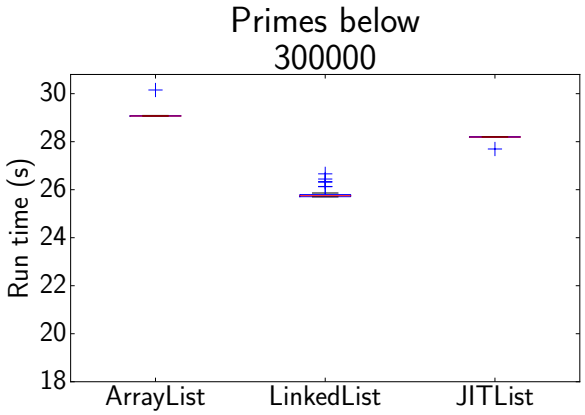
```
1 swaprule LinkedList {
2   if ( (size()>1000) && (10*#del < #iter)) {
3     this to ArrayList;
4   }
5 }
```

We implemented a `JITList` in JitDS-Java which is an JIT Data Structure that is able to swap between representations based on `ArrayList` and `LinkedList`. Listings 19 and 20 show the logic that describes when to swap from one representation to the other.

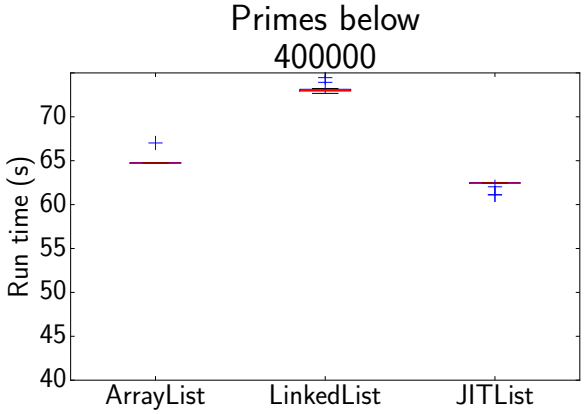
Implementation Details. Note that there are two differences between the original `List` from Java and the `JitList` as used in Listing 17. First, we do not consider generic types in our language and therefore `JitList` is *assumed* to be be like `List<Integer>`. The second difference is the use of the `JitList` *as* an `Iterator` instead of *requesting* an `Iterator` (*i. e.*, `Iterator it = primes.iterator();`). The latter difference is more fundamental and therefore further discussed in Section 8.

As an experiment we compared the execution times of running the "'sieve" application with an `ArrayList`, a `LinkedList`, and a `JITList` with the swap rules and invocation counters introduced above (Listings 18 to 20). The internal swap rule in Listing 19 is designed to change an `ArrayList` into a `LinkedList` when the number of inserts becomes too high compared to the number of random reads (`#get`). The magic constant, 10, was introduced as dampening factor to avoid premature representation changes and is determined by trial-and-error. The internal swap rule in Listing 20 is designed to change a `LinkedList` into an `ArrayList` when the list is mainly iterated over (`#iter`) compared to the number of deletions (`#del`). Again, the magic constant is a dampening factor determined by trial-and-error. Further, we vary the number of elements initially added to the list. Figure 3 summarizes the executions times of 30 runs in three box-plots, one for each input size. Comparing the `ArrayList` and the `LinkedList` implementations, we observe that for the smallest input, `LinkedList` outperforms `ArrayList`, whereas for the larger inputs the situation is reversed.

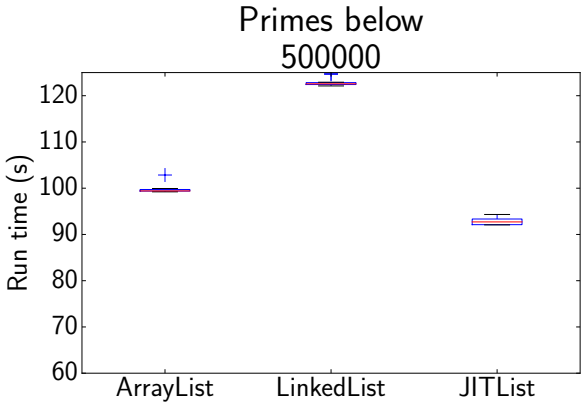
When analyzing the representation changes of the `JITList`, we observe a first transition from `ArrayList` to `LinkedList` early in the program's execution (*i. e.*, during the building of the list). A second transition, *i. e.*, from `LinkedList` to `ArrayList`, is observed during one of the early iterations of the sieve loop. We conclude, by comparing the execution time needed by the `JITList` with the others, that the representation changes allow the `JITList` to combine the best of both classic list representation, and eventually outperforms both (Figure 3c).



(a) Time to find all primes below 300,000.



(b) Time to find all primes below 400,000.



(c) Time to find all primes below 500,000.

Figure 3: Varying the initial number of elements added to the list.

7. Related Work

This section on related work is divided into two parts. In the first part we present work related to the techniques needed to implement Just-in-Time Data Structures. In a second part we discuss the work related to data structure selection in general.

7.1 Implementing a Just-in-Time Data Structure

In Section 4 we showed that changing data representation at runtime can be implemented using a restricted form of dynamic reclassification which we called homomorphic reclassification. The idea of using a restricted form of dynamic object reclassification is explored by Tal Cohen et.al. in [6]. They present three implementation techniques of which we use the inheritance-based technique. This technique effectively implements the bridge design pattern as described in [13], in a different context. Similar ideas have been explored in the context of objects and their identity in the language Gilgul by Costanza [8]. More general forms of dynamic object reclassification can be found for instance in the language Fickle [11] and in Smalltalk (cf. `become:`).

To implement JIT Data Structures, we observe the usage of the data structure and react accordingly. Our strategy to implement the observations and reactions and merge them with the application logic could also be realized through aspect-oriented programming [16], *i. e.*, we disentangle the logic for data structure selection from the rest of the application logic. While our work is not focussing on AOP as such, it is interesting to consider the work on domain-specific approaches in the context of performance. Introducing representation swaps at a non-lexical place in a program, as discussed in Section 4, implies the need for selecting a specific kind of join point. Similarly, LoopsAj introduces expressiveness dedicated to join points for loops [14], which in turn allows parallelization of the code and improves performance.

7.2 Data Structure Selection

Above we compared our implementation of JitDS-Java with other software engineering efforts. Here we focus on four approaches that have the same goal as Just-in-Time Data Structures, *i. e.*, selecting the best possible data representation for an application. We classify the approaches according to the axes presented in Section 3.

Brainy is a *general* program analysis tool that automatically selects the best data representation for a given program on a specific micro-architecture [15]. Brainy makes an *offline* decision by observing interface invocations of a sample run and feeds this information into an machine learning algorithm, which takes architecture-specific characteristics into account.

Most of the other related work is *dedicated* to collections. Chameleon is a tool that assists the *developer* in choosing the appropriate collection representation [19]. Chameleon makes its *offline* decisions based on a rule-engine and the collection's behavior gathered during sample runs of the program. The rules for the Chameleon engine are expressed in a DSL where "number of interface operation invocations" is one of the possible expressions.

CoCo on the other hand, is an *online* application-level optimization technique for collections [23]. CoCo exploits the known algorithmic characteristics of Java collections to improve performance. CoCo differs from other online approaches because it allows a *gradual* transition between representations, *i. e.*, CoCo is able to provide a "cheap transition function" to revert a representation swap. We have not yet considered this in our approach, but it is an interesting avenue of future work.

In PyPy, the homogeneity of collections in dynamic languages can be exploited by changing the strategy for storing elements [3]. Also in the V8 JavaScript interpreter the underlying representation of data is changed based on the current content and usage.[8] These ideas date back to *maps*, one of the implementation techniques of SELF's object storage model [4].

A more established form of representation changes are the offline representation changes introduced by compilers. Autoboxing and unboxing are examples thereof.[9] Typecasting and coercion are two other examples of established forms of representation changes readily available in many languages [1].

A final body of related work tackles performance from the opposite angle and changes the *computation* — as opposed to the data -£– to improve a program. The amount of work in this area (*i. e.*, compiler technology) is vast. Tang et al. [22], for instance, develop a special purpose compiler to improve the performance of stencil computation based on the target hardware and the shape of the stencil. Based on the polytope model, dependency graphs, and other theoretical properties, compilers are allowed to "rearrange" nested loops to improve performance while keeping semantics intact. From these efforts, the work of Ansel et al. [2] resembles our work the most. In their language a developer can "combine" multiple algorithms to solve the same problem (*e. g.*, insertion-sort or merge-sort) into a single algorithm (*e. g.*, sort). Much like `qsort` is currently implemented manually in the standard C headers, the PetaBricks language chooses the "right" algorithm to be used at runtime based on the data.

8. Discussion and Future Work

We presented JitDS-Java, an extension of Java, to define Just-in-Time Data Structures and discussed a straightforward compiler that translates the new constructs into Java. Moreover, in Section 6, we showed two example programs that benefit from using a JIT Data Structure. In its current form

[8] *V8 JavaScript Engine - Google Project Hosting*, V8 project authors, access date: December 21st 2014 `https://code.google.com/p/v8/source/browse/trunk/src/array.js#89`

[9] *Autoboxing and Unboxing*, Oracle, access date: April 1st 2015 `https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html`

the compiler is a prototype for fast development of JIT Data Structures (*e. g.*, `JitMatrix`, `JitList`, or `JitFile`). There are a number of points on which JitDS-Java and its compiler need to be further refined in order to allow for better usability.

Engineering: Reducing the Overhead. Currently, the compiler is implemented straightforwardly, as discussed in Section 5 and therefore would benefit from a more mature implementation which generates code with less performance overhead. First, we want to reduce the method invocation overhead introduced by the extra level of indirection of using the bridge pattern. Technically, the cheap `invokeVirtual` of a simple method call is, in our implementation, replaced by an `invokeVirtual` followed by an (expensive) `invokeInterface`. Second, in its current form, the compiler introduces a lot of checks for potential internal swap rule invocations that are not strictly necessary, *i. e.*, before every core method invocation. We plan to turn this around and invoke only those swap rules which can trigger because of newly updated counters or values, *i. e.*, push-based instead of pull-based. For this we are looking in the direction of expert systems and rule engines.

Expensive and Invalid Transitions. Writing program logic with JIT Data Structures allows the developer of the code to be oblivious of the actual representation of the data structures. For instance, using a lot of specialized methods potentially causes a lot of representation swaps invisible to the programmer. Moreover, some transitions can be invalid (cf. the file example, Figure 1). In these cases, the programmer should be warned of the potentially expensive or invalid code. We want to introduce a type system in our language that allows for static warnings when code becomes potentially expensive, and for static errors (rejected program) when the program will run into an unacceptable transition. DeLine and Fähndrich [10] present a type system which allows static reasoning about the "state", here the current representation, of objects.

Escaping Pointers. It is common in programs for member functions to return a value which holds, direct or indirect, a reference to the object itself. For instance, `Iterators` as obtained from a `List` need a reference to the list to be able to iterate. While passing references around is generally legit in OOP. In the context of JIT Data Structures, however, it raises some problems. If a reference to an internal representation is passed outside the boundaries of the JIT Data Structure, the object identity is not longer maintained.

An iterator obtained from a `JitList` before a swap will no longer be able to correctly iterate over the list after a swap. We call this the problem of *escaping pointers*. We want to introduce ownership types [5], to aid the developer in programming with JIT Data Structures and to allow the compiler to introduce guards.

Freezing and Thawing. One of the benefits of implementing JIT Data Structures is that it is possible to express rules on a *conceptual level* when it is beneficial to swap representations. In practice, however, it is possible that multiple rules should be triggered at the same time, causing a ping-pong effect of representation swaps. These cascading transitions can even be caused by a transition from one representation to another. To avoid polluting the invocation counters we are currently investigating the *Freezing and Thawing* of our JIT Data Structure's swapping capability, *i. e.*, disallowing swaps for a certain period. Currently, our compiler already "freezes" the data structure during a swap, such that a swap caused by a swap is not possible.

Interfering Swap Rules. When the number of swap rules increases, it becomes likely that they will interfere with each other. We want to give external swap rules priority over internal swap rules, *i. e.*, internal swap rules do not trigger within the execution flow of an external swap rule. In general, however, the interplay between multiple swap rules needs further investigation.

9. Conclusion

In this text we introduced *Just-in-Time Data Structures*, a general approach for developers to implement data structures that can intrinsically change their representation at runtime. This approach is beneficial for those applications where fixing the data representation before execution or at allocation time is suboptimal. We implemented JitDS-Java, a language in which it is possible to define such Just-in-Time Data Structures. Defining swap rules for a JIT Data Structure, allows developers to separate the core application logic from the crosscutting concern of data structure selection. This could ease the software engineering task by separating "engineering application logic" from "performance engineering tasks", which in turn could divide the software engineering efforts over a domain-expert developer and a performance-expert developer.

To define a Just-in-Time Data Structure, a developer *combines* multiple data representations and provides the functions to transition between them. Further, the performance-expert developer implements a set of *internal and external swap rules* to define which representation is to be used in which situations. Our compiler turns these definitions into a data structure that is *intrinsically* capable of changing its representation *during the execution* of a program.

In conclusion, this work wants to shift the focus from "trying to find the right data representation for a program" to "finding the right *sequence* of data representations for a program".

Acknowledgments

Mattias De Wael is supported by a research grant of IWT (Innovation through Science and Technology, Flanders).

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996. ISBN 0262011530.

[2] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of PLDI '09*, pages 38–49, 2009. ISBN 978-1-60558-392-1. .

[3] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. In *Proceedings of OOPLSA '13*, pages 167–182, 2013. ISBN 978-1-4503-2374-1. .

[4] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Proceedings of OOPSLA '89*, pages 49–70, 1989. ISBN 0-89791-333-7. .

[5] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of OOPSLA '98*, pages 48–64, 1998. ISBN 1-58113-005-8. .

[6] T. Cohen and J. Y. Gil. Three approaches to object evolution. In *Proceedings of PPPJ '09*, pages 57–66, 2009. ISBN 978-1-60558-598-7. .

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

[8] P. Costanza. Dynamic replacement of active objects in the gilgul programming language. In *Proceedings of CD '02*, pages 125–140, 2002. ISBN 3-540-43847-5.

[9] M. De Wael, D. Ungar, and T. Van Cutsem. When spatial and temporal locality collide: The case of the missing cache hits. In *Proceedings of ICPE '13*, pages 63–70, 2013. ISBN 978-1-4503-1636-1. .

[10] R. DeLine and M. Fähndrich. Typestates for objects. In *Proceedings of ECOOP '04*, pages 465–490, 2004. ISBN 978-3-540-22159-3. .

[11] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: FickleII. *ACM TOPLAS*, 24:153–191, 2002.

[12] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, RIACS, 2000.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

[14] B. Harbulot and J. R. Gurd. A Join Point for Loops in AspectJ. In *Proceedings of AOSD '06*, pages 63–74, 2006. ISBN 1-59593-300-X.

[15] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. In *Proceedings of PLDI '11*, pages 86–97, 2011. ISBN 978-1-4503-0663-8.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP '97*, pages 220–242, 1997. ISBN 978-3-540-63089-0. .

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Proceedings of ECOOP '01*, pages 327–354, 2001. ISBN 978-3-540-42206-8.

[18] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of Symposium on Very High Level Languages*, pages 50–59, 1974. .

[19] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of PLDI '09*, pages 408–418, 2009. ISBN 978-1-60558-392-1.

[20] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23: 84–93, 2003. ISSN 0272-1732. .

[21] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1st edition, 1998. ISBN 0262193981.

[22] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of SPAA '11*, pages 117–128, 2011.

[23] G. H. Xu. Coco: Sound and adaptive replacement of java collections. In *Proceedings of ECOOP '13*, pages 1–26, 2013.