Just-in-Time Data Structures

Towards Declarative Swap Rules

Mattias De Wael

Software Languages Lab
Vrije Universiteit Brussel (Belgium)
madewael@vub.ac.be

Abstract

Just-in-Time Data Structures are an attempt to vulgarise the idea that changing the representation (i.e., implementation) of a data structure at runtime can improve the performance of a program compared to its counter part that relies on a single representation. In previous work, we developed a language to develop such Just-in-Time Data Structures. To express "when" to change between representations, a dedicated language construct was introduced: the swap rule. A swap rule analyses the state and usage of a just-in-time data structure and reacts as defined by a developer. Opposed to what the name suggest, swap rules are currently implemented as imperative statements woven into the codebase. Their intend, however, is declarative and therefore we think that swap rules should become real declarative rules.

This extended abstract presents Just-in-Time Data Structures as a case for applying state-of-the-art in low overhead dynamic analysis. Changing from an imperative to a declarative implementation of swap rules will allow for more efficient execution of our programs by reducing the overhead of continuous analysis.

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

Keywords Data structures, algorithms, dynamic reclassification, performance

1. Introduction

Today, software engineering practices focus on finding the single "right" data representation for a program. The "right" data representation, however, might not exist: relying on a single representation of the data for the lifetime of the program can be suboptimal in terms of performance. Just-in-Time Data Structures are an attempt to vulgarise the idea that changing the representation of a data structure at runtime can improve the performance of a program over its counter part that relies on a single representation only. Or in other words: 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.

A Just-in-Time Data Structure enables online representation changes based on declarative input from a performance expert programmer, expressed in the form a dedicated language construct: the *internal swap rule*. Classically, such an internal swap rule describes for which *state* of the data structure which representation should be used. Their intend is clearly declarative, for instance: "if the matrix **is** sparse, change to a sparse representation.". In the current implementation, however, internal swap rules are imperative statements woven into the codebase.

2. Matrices and Representation Changes

In previous work we present an elaborate and convincing motivation for the need of Just-in-Time Data Structures [1]. Here, we merely introduce an example without any evaluation: Imagine a simple data interface which allows the user to `get(row,col)` and `set(row,col,val)` values from a 2 dimensional `Matrix`. Further, a `Matrix` implements the operations `getRows` and `getCols`. For this interface, many possible implementations (representations) are possible. Here, consider both `ArrayMatrix` and `SparseMatrix` to implement `Matrix`. `ArrayMatrix` stores all values in an 2d array, whereas `SparseMatrix` uses a more clever storage scheme that only stores non-zero values, e.g., Compressed Row Storage (CRS[1]). Because there exist specialised algorithms for sparse matrices, it would be interesting for an `ArrayMatrix` to "automagically" change its representation to `SparseMatrix` when it actually *becomes* a sparse matrix, i.e., when the number of zero values greatly exceeds the number of non-zero values.

[1] See for instance http://netlib.org/linalg/html_templates/node91.html

WODA'15, October 26, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3909-4/15/10
http://dx.doi.org/10.1145/2823363.2823371

Listing 1. A class that combines two implementations.

```
1  JitMatrix combines ArrayMatrix, SparseMatrix {
2   #set(row,col,val);
3   #zeroSet    as set(row,col,val){ count-if(val==0); }
4   #nonZeroSet as set(row,col,val){ count-if(val!=0); }
5
6   swaprule ArrayMatrix {
7    int size = this.getRows() * this.getCols();
8    boolean frequentSet = (#set > (0.3*size));
9    boolean sparseSet = (#zeroSet > #nonZeroSet*#nonZeroSet );
10   if ( frequentSet && sparseSet )
11    this to SparseMatrix;
12  }
13 }
```

Swap Rules and Invocation Counters. Of course data structures do not "automagically" change their representation. Therefore our language allows developers to express *declarative statements* about their data structures and when they need to change representation. An example thereof is given in Listing 1.

The two language constructs demonstrated in Listing 1 are *swap rules* and *invocation counters*. Lines 2-4 show definitions of invocation counters. These, as the name suggest, count the number of invocations of a given operation. For non-static operations these counters are allocated per instance. For instance, `#set(row,col,val)`, the simplest invocation counter of the three, creates a new "counter" addressable by the name of the "instrumented" method, here `#set`. More complex invocation counters, such as `#zeroSet` and `#nonZeroSet`, allow to specify a name for the counter and allows the developer to put conditions on when to increment the counter. `#zeroSet`, for instance, only counts those invocations of `set` where `val` is 0.

The swap rule definition (see line 6) describes when an `ArrayMatrix` should change its representation. Here, the developer expressed that when the number of "zero sets" vastly exceeds the number of "non-zero sets", while the number of overall "sets" is significantly large enough (see line 10), the `ArrayMatrix` should become a `SparseMatrix` (see line 11).

Listing 1 shows further how `JitMatrix` combines the representations `ArrayMatrix` and `SparseMatrix`. To define a working program also *transition functions* are needed, an thorough explanation of of those can be found in [1].

3. Imperative Implementation

The focus in [1] lays in the *design of a language* to facilitate a shift in focus from finding the "right" data structure to finding the "right" sequence of data structures. As a result, the language is merely given a prototype implementation, just powerful enough to show the performance benefit of changing representation in a handful examples. Currently swap rules are implemented in a straightforward and unoptimised manner. Because swap rules react on the "change of state" of a data structure, the current implementation simply "executes the body of the swap rule" after each invocation of each operation, regardless of the actual occurrence of a state change. While the current implementation allows virtually any expression available in the base language (a Java subset) to occur in the body of a swap rule, it is wise to keep its computational complexity to a minimum. Moreover, all examples in [1], show swap rules which exhibit an declarative nature. Therefore, we conjecture that a more mature implementation of the dynamic analysis as expressed by a swap rule should take benefit of the intended declarative nature to avoid the overhead of excessive checks at runtime.

4. Declarative Implementation

One problem with online dynamic analysis is the inherent overhead introduced by instrumenting the code base. In our next implementation of Just-in-Time Data Structures we want to incorporate established techniques from the area of "runtime monitoring" with low overhead. Our current direction of research is in the reduction in expressiveness in swap rule bodies in order to allow for the construction of dependency graphs. Moreover, replacing the current implementations with a *incremental computation* based on this graph, should greatly reduce the overhead.

Moreover, a transition from imperative to declarative changes the triggering mechanism of swap rule from a pull-based to a push-based mechanism: the current implementation opportunistically checks the need for a swap after each operation (pull-based), whereas the incremental approach triggers the intended swap when a swap-condition is met (push-based).

5. Related Work and Future Work

The idea of Just-in-Time Data Structures is largely inspired by the work of Xu [3] and Shacham et al. [2]. While both efforts focus on Java's collection framework, Just-in-Time Data Structures wants to generalise the approach by providing language support to the average software engineer. In this way we want to facilitate a shift in focus from finding the "right" data structure to finding the "right" sequence of data structures. An more thorough overview, by means of a taxonomy of related work, is given in [1]. As this extended abstract suggest, our future and ongoing work wants to focus on a more mature implementation of the Just-in-Time Data Structure language by making swap rules truly declarative. To this end we seek for experts in the field of "runtime monitoring with low overhead" to help us with understanding the costs and benefits of the aforementioned shift towards *declarative swap rules*.

References

[1] M. De Wael, S. Marr, J. De Koster, J. B. Sartor, and W. De Meuter. Just-in-time data structures. In *Onward!*, 2015.

[2] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *proceedings of PLDI*, 2009.

[3] G. H. Xu. Coco: Sound and adaptive replacement of java collections. In *proceedings of ECOOP*, 2013.

Mattias De Wael is supported by a research grant of IWT, Flanders.